Deep Reinforcement Learning Hands-On

Apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more

Second edition — Includes multi-agent methods and advanced exploration techniques

Maxim Lapan

Packt>

Deep Reinforcement Learning Hands-On

Second Edition

Apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more

Maxim Lapan

Deep Reinforcement Learning Hands-On
Second Edition

Contributors

About the authors

Maxim Lapan is a deep learning enthusiast and independent researcher. His background and 15 years' work expertise as a software developer and a systems architect covers everything from low-level Linux kernel driver development to performance optimization and the design of distributed applications working on thousands of servers. With extensive work experience in big data, machine learning, and large parallel distributed HPC and non-HPC systems, he has the ability to explain complicated things using simple words and vivid examples. His current areas of interest surround the practical applications of deep learning, such as deep natural language processing and deep reinforcement learning.

Maxim lives in Moscow, Russia, with his family.

About the reviewers

Mikhail Yurushkin holds a PhD. His areas of research are high-performance computing and optimizing compiler development. Mikhail is a senior lecturer at SFEDU university, Rostov-on-Don, Russia. He teaches advanced deep learning courses on computer vision and NLP. Mikhail has worked for over eight years in cross-platform native C++ development, machine learning, and deep learning. He is an entrepreneur and founder of several technological start-ups, including BroutonLab – Data Science Company, which specializes in the development of AI-powered software products.

Per-Arne Andersen is a PhD student in deep reinforcement learning at the University of Agder, Norway. He has authored several technical papers on reinforcement learning for games and received the best student award from the British Computer Society for his research into model-based reinforcement learning. Per-Arne is also an expert on network security, having worked in the field since 2012. His current research interests include machine learning, deep learning, network security, and reinforcement learning.

Sergey Kolesnikov is an industrial and academic research engineer with over five years' experience in machine learning, deep learning, and reinforcement learning. He's currently working on industrial applications that deal with CV, NLP, and RecSys, and is involved in reinforcement learning academic research. He is also interested in sequential decision making and psychology. Sergey is a NeurIPS competition winner and an open source evangelist. He is also the creator of Catalyst – a high-level PyTorch ecosystem for accelerated deep learning/reinforcement learning research and development.

Table of Contents

Preface

The topic of this book is reinforcement learning (RL), which is a subfield of machine learning (ML); it focuses on the general and challenging problem of learning optimal behavior in a complex environment. The learning process is driven only by the reward value and observations obtained from the environment. This model is very general and can be applied to many practical situations, from playing games to optimizing complex manufacturing processes.

Due to its flexibility and generality, the field of RL is developing very quickly and attracting lots of attention, both from researchers who are trying to improve existing methods or create new methods and from practitioners interested in solving their problems in the most efficient way.

Why I wrote this book

This book was written as an attempt to fill the obvious gap in practical and structured information about RL methods and approaches. On the one hand, there is lots of research activity all around the world. New research papers are being published almost every day, and a large portion of deep learning (DL) conferences, such as Neural Information Processing Systems (NeurIPS) or the International Conference on Learning Representations (ICLR), are dedicated to RL methods. There are also several large research groups focusing on the application of RL methods to robotics, medicine, multi-agent systems, and others.

Information about the recent research is widely available, but it is too specialized and abstract to be easily understandable. Even worse is the situation surrounding the practical aspect of RL, as it is not always obvious how to make the step from an abstract method described in its mathematical-heavy form in a research paper to a working implementation solving an actual problem.

This makes it hard for somebody interested in the field to get a clear understanding of the methods and ideas behind papers and conference talks. There are some very good blog posts about various RL aspects that are illustrated with working examples, but the limited format of a blog post allows authors to describe only one or two methods, without building a complete structured picture and showing how different methods are related to each other. This book is my attempt to address this issue.

The approach

Another aspect of the book is its orientation to practice. Every method is implemented for various environments, from the very trivial to the quite complex. I've tried to make the examples clean and easy to understand, which was made possible by the expressiveness and power of PyTorch. On the other hand, the complexity and requirements of the examples are oriented to RL hobbyists without access to very large computational resources, such as clusters of graphics processing units (GPUs) or very powerful workstations. This, I believe, will make the fun-filled and exciting RL domain accessible to a much wider audience than just research groups or large artificial intelligence companies. This is still deep RL, so access to a GPU is highly recommended. Approximately half of the examples in the book will benefit from being run on a GPU.

In addition to traditional medium-sized examples of environments used in RL, such as Atari games or continuous control problems, the book contains several chapters (10, 14, 15, 16, and 18) that contain larger projects, illustrating how RL methods can be applied to more complicated environments and tasks. These examples are still not full-sized, real-life projects (they would occupy a separate book on their own), but just larger problems illustrating how the RL paradigm can be applied to domains beyond the well-established benchmarks.

Another thing to note about the examples in the first three parts of the book is that I've tried to make them self-contained, with the source code shown in full. Sometimes this has led to the repetition of code pieces (for example, the training loop is very similar in most of the methods), but I believe that giving you the freedom to jump directly into the method you want to learn is more important than avoiding a few repetitions. All examples in the book are available on GitHub: `https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition`, and you're welcome to fork them, experiment, and contribute.

Who this book is for

The main target audience is people who have some knowledge of ML, but want to get a practical understanding of the RL domain. The reader should be familiar with Python and the basics of DL and ML. An understanding of statistics and probability is an advantage, but is not absolutely essential for understanding most of the book's material.

What this book covers

Chapter 1, What Is Reinforcement Learning?, contains an introduction to RL ideas and the main formal models.

Chapter 2, OpenAI Gym, introduces the practical aspects of RL, using the open source library Gym.

Chapter 3, Deep Learning with PyTorch, gives a quick overview of the PyTorch library.

Chapter 4, The Cross-Entropy Method, introduces one of the simplest methods in RL to give you an impression of RL methods and problems.

Chapter 5, Tabular Learning and the Bellman Equation, introduces the value-based family of RL methods.

Chapter 6, Deep Q-Networks, describes deep Q-networks (DQNs), an extension of the basic value-based methods, allowing us to solve a complicated environment.

Chapter 7, Higher-Level RL Libraries, describes the library PTAN, which we will use in the book to simplify the implementations of RL methods.

Chapter 8, DQN Extensions, gives a detailed overview of a modern extension to the DQN method, to improve its stability and convergence in complex environments.

Chapter 9, Ways to Speed up RL Methods, provides an overview of ways to make the execution of RL code faster.

Chapter 10, Stocks Trading Using RL, is the first practical project and focuses on applying the DQN method to stock trading.

Chapter 11, Policy Gradients – an Alternative, introduces another family of RL methods that is based on policy learning.

Chapter 12, The Actor-Critic Method, describes one of the most widely used methods in RL.

Chapter 13, Asynchronous Advantage Actor-Critic, extends the actor-critic method with parallel environment communication, which improves stability and convergence.

Chapter 14, Training Chatbots with RL, is the second project and shows how to apply RL methods to natural language processing problems.

Chapter 15, The TextWorld Environment, covers the application of RL methods to interactive fiction games.

Chapter 16, Web Navigation, is another long project that applies RL to web page navigation using the MiniWoB set of tasks.

Chapter 17, Continuous Action Space, describes the specifics of environments using continuous action spaces and various methods.

Chapter 18, RL in Robotics, covers the application of RL methods to robotics problems. In this chapter, I describe the process of building and training a small hardware robot with RL methods.

Chapter 19, Trust Regions – PPO, TRPO, ACKTR, and SAC, is yet another chapter about continuous action spaces describing the trust region set of methods.

Chapter 20, Black-Box Optimization in RL, shows another set of methods that don't use gradients in their explicit form.

Chapter 21, Advanced Exploration, covers different approaches that can be used for better exploration of the environment.

Chapter 22, Beyond Model-Free – Imagination, introduces the model-based approach to RL and uses recent research results about imagination in RL.

Chapter 23, AlphaGo Zero, describes the AlphaGo Zero method and applies it to the game Connect 4.

Chapter 24, RL in Discrete Optimization, describes the application of RL methods to the domain of discrete optimization, using the Rubik's Cube as an environment.

Chapter 25, Multi-agent RL, introduces a relatively new direction of RL methods for situations with multiple agents.

To get the most out of this book

All the chapters in this book describing RL methods have the same structure: in the beginning, we discuss the motivation of the method, its theoretical foundation, and the idea behind it. Then, we follow several examples of the method applied to different environments with the full source code.

You can use the book in different ways:

1. To quickly become familiar with some method, you can read only the introductory part of the relevant chapter

2. To get a deeper understanding of the way the method is implemented, you can read the code and the comments around it

3. To gain a deep familiarity with the method (the best way to learn, I believe) you can try to reimplement the method and make it work, using the provided source code as a reference point

In any case, I hope the book will be useful for you!

Download the example code files

You can download the example code files for this book from your account at `www.packt.com/`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `http://www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the on-screen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781838826994_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example; "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
def grads_func(proc_name, net, device, train_queue):
    envs = [make_env() for _ in range(NUM_ENVS)]

    agent = ptan.agent.PolicyAgent(
        lambda x: net(x)[0], device=device, apply_softmax=True)
    exp_source = ptan.experience.ExperienceSourceFirstLast(
        envs, agent, gamma=GAMMA, steps_count=REWARD_STEPS)

    batch = []
    frame_idx = 0
    writer = SummaryWriter(comment=proc_name)
```

Any command-line input or output is written as follows:

```
rl_book_samples/Chapter11$ ./02_a3c_grad.py --cuda -n final
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this. For example: "Select **System info** from the **Administration** panel."

> Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit, `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
What Is Reinforcement Learning?

Reinforcement learning (**RL**) is a subfield of **machine learning** (**ML**) that addresses the problem of the automatic learning of optimal decisions over time. This is a general and common problem that has been studied in many scientific and engineering fields.

In our changing world, even problems that look like static input-output problems can become dynamic if time is taken into account. For example, imagine that you want to solve the simple supervised learning problem of pet image classification with two target classes—dog and cat. You gather the training dataset and implement the classifier using your favorite **deep learning** (**DL**) toolkit. After a while, the model that has converged demonstrates excellent performance. Great! You deploy it and leave it running for a while. However, after a vacation at some seaside resort, you return to discover that dog grooming fashions have changed and a significant portion of your queries are now misclassified, so you need to update your training images and repeat the process again. Not so great!

The preceding example is intended to show that even simple ML problems have a hidden time dimension. This is frequently overlooked, but it might become an issue in a production system. RL is an approach that natively incorporates an extra dimension (which is usually time, but not necessarily) into learning equations. This places RL much closer to how people understand **artificial intelligence** (**AI**).

In this chapter, we will discuss RL in more detail and you will become familiar with the following:

- How RL is related to and differs from other ML disciplines: **supervised and unsupervised learning**

- What the main RL formalisms are and how they are related to each other
- Theoretical foundations of RL—the Markov decision processes

Supervised learning

You may be familiar with the notion of supervised learning, which is the most studied and well-known machine learning problem. Its basic question is, how do you automatically build a function that maps some input into some output when given a set of example pairs? It sounds simple in those terms, but the problem includes many tricky questions that computers have only recently started to address with some success. There are lots of examples of supervised learning problems, including the following:

- **Text classification**: Is this email message spam or not?
- **Image classification and object location**: Does this image contain a picture of a cat, dog, or something else?
- **Regression problems**: Given the information from weather sensors, what will be the weather tomorrow?
- **Sentiment analysis**: What is the customer satisfaction level of this review?

These questions may look different, but they share the same idea—we have many examples of input and desired output, and we want to learn how to generate the output for some future, currently unseen input. The name *supervised* comes from the fact that we learn from known answers provided by a "ground truth" data source.

Unsupervised learning

At the other extreme, we have the so-called unsupervised learning, which assumes no supervision and has no known labels assigned to our data. The main objective is to learn some hidden structure of the dataset at hand. One common example of such an approach to learning is the clustering of data. This happens when our algorithm tries to combine data items into a set of clusters, which can reveal relationships in data. For instance, you might want to find similar images or clients with common behaviors.

Another unsupervised learning method that is becoming more and more popular is **generative adversarial networks** (**GANs**). When we have two competing neural networks, the first network is trying to generate fake data to fool the second network, while the second network is trying to discriminate artificially generated data from data sampled from our dataset. Over time, both networks become more and more skillful in their tasks by capturing subtle specific patterns in the dataset.

Reinforcement learning

RL is the third camp and lies somewhere in between full supervision and a complete lack of predefined labels. On the one hand, it uses many well-established methods of supervised learning, such as **deep neural networks** for function approximation, stochastic gradient descent, and backpropagation, to learn data representation. On the other hand, it usually applies them in a different way.

In the next two sections of the chapter, we will explore specific details of the RL approach, including assumptions and abstractions in its strict mathematical form. For now, to compare RL with supervised and unsupervised learning, we will take a less formal, but more easily understood, path.

Imagine that you have an agent that needs to take actions in some environment. (Both "agent" and "environment" will be defined in detail later in this chapter.) A robot mouse in a maze is a good example, but you can also imagine an automatic helicopter trying to perform a roll, or a chess program learning how to beat a grandmaster. Let's go with the robot mouse for simplicity.
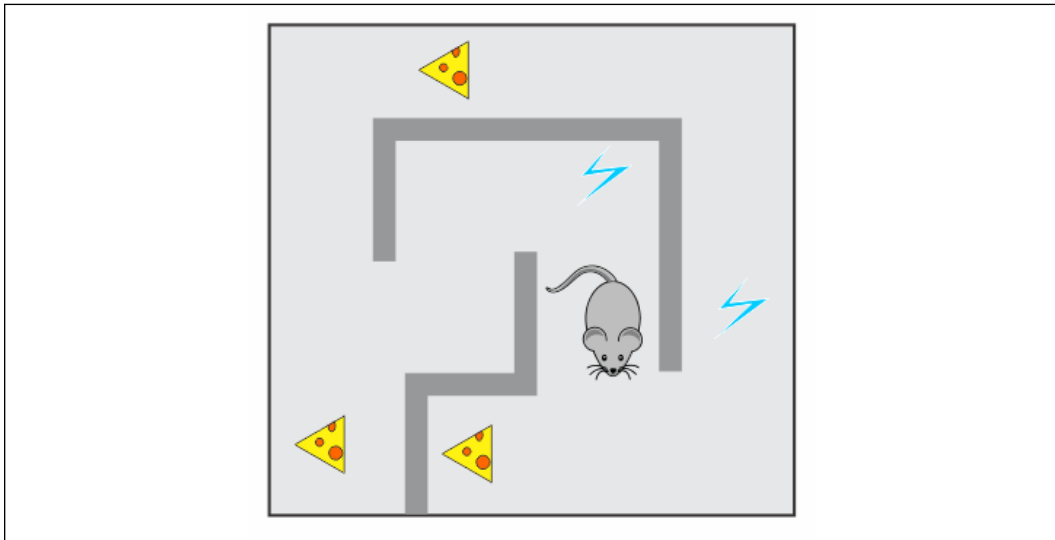
Figure 1.1: The robot mouse maze world

In this case, the environment is a maze with food at some points and electricity at others. The robot mouse can take actions, such as turn left/right and move forward. At each moment, it can observe the full state of the maze to make a decision about the actions to take. The robot mouse tries to find as much food as possible while avoiding getting an electric shock whenever possible. These food and electricity signals stand as the reward that is given to the agent (robot mouse) by the environment as additional feedback about the agent's actions. The reward is a very important concept in RL, and we will talk about it later in the chapter. For now, it is enough for you to know that the final goal of the agent is to get as much total reward as possible. In our particular example, the robot mouse could suffer a slight electric shock to get to a place with plenty of food—this would be a better result for the robot mouse than just standing still and gaining nothing.

We don't want to hard-code knowledge about the environment and the best actions to take in every specific situation into the robot mouse—it will take too much effort and may become useless even with a slight maze change. What we want is to have some magic set of methods that will allow our robot mouse to learn on its own how to avoid electricity and gather as much food as possible. RL is exactly this magic toolbox and it behaves differently from supervised and unsupervised learning methods; it doesn't work with predefined labels in the way that supervised learning does. Nobody labels all the images that the robot sees as *good* or *bad*, or gives it the best direction to turn in.

However, we're not completely blind as in an unsupervised learning setup—we have a reward system. The reward can be positive from gathering the food, negative from electric shocks, or neutral when nothing special happens. By observing the reward and relating it to the actions taken, our agent learns how to perform an action better, gather more food, and get fewer electric shocks. Of course, RL generality and flexibility comes with a price. RL is considered to be a much more challenging area than supervised or unsupervised learning. Let's quickly discuss what makes RL tricky.

RL's complications

The first thing to note is that observation in RL depends on an agent's behavior and, to some extent, it is the *result* of this behavior. If your agent decides to do inefficient things, then the observations will tell you nothing about what it has done wrong and what should be done to improve the outcome (the agent will just get negative feedback all the time). If the agent is stubborn and keeps making mistakes, then the observations will give the false impression that there is no way to get a larger reward—*life is suffering*—which could be totally wrong.

In ML terms, this can be rephrased as *having non-i.i.d. data*. The abbreviation **i.i.d.** stands for **independent and identically distributed**, a requirement for most supervised learning methods.

The second thing that complicates our agent's life is that it needs to not only *exploit* the knowledge it has learned, but actively *explore* the environment, because maybe doing things differently will significantly improve the outcome. The problem is that too much exploration may also seriously decrease the reward (not to mention the agent can actually *forget* what it has learned before), so we need to find a balance between these two activities somehow. This exploration/exploitation dilemma is one of the open fundamental questions in RL. People face this choice all the time—should I go to an already known place for dinner or try this fancy new restaurant? How frequently should I change jobs? Should I study a new field or keep working in my area? There are no universal answers to these questions.

The third complication factor lies in the fact that reward can be seriously delayed after actions. In chess, for example, one single strong move in the middle of the game can shift the balance. During learning, we need to discover such causalities, which can be tricky to discern during the flow of time and our actions.

However, despite all these obstacles and complications, RL has seen huge improvements in recent years and is becoming more and more active as a field of research and practical application.

Interested in learning more? Let's dive into the details and look at RL formalisms and play rules.

RL formalisms

Every scientific and engineering field has its own assumptions and limitations. In the previous section, we discussed supervised learning, in which such assumptions are the knowledge of input-output pairs. You have no labels for your data? You need to figure out how to obtain labels or try to use some other theory. This doesn't make supervised learning *good* or *bad*; it just makes it inapplicable to your problem.

There are many historical examples of practical and theoretical breakthroughs that have occurred when somebody tried to challenge rules in a creative way. However, we also must understand our limitations. It's important to know and understand game rules for various methods, as it can save you tons of time in advance. Of course, such formalisms exist for RL, and we will spend the rest of this book analyzing them from various angles.

The following diagram shows two major RL entities—**agent** and **environment**—and their communication channels—**actions**, **reward**, and **observations**. We will discuss them in detail in the next few sections:
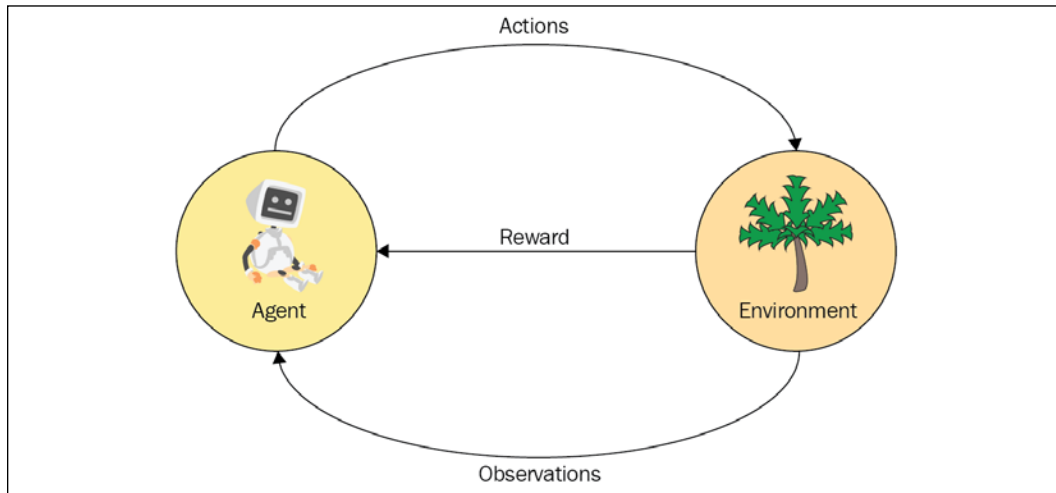
Figure 1.2: RL entities and their communication channels

Reward

Let's return to the notion of reward. In RL, it's just a scalar value we obtain periodically from the environment. As mentioned, reward can be positive or negative, large or small, but it's just a number. The purpose of reward is to tell our agent how well it has behaved. We don't define how frequently the agent receives this reward; it can be every second or once in an agent's lifetime, although it's common practice to receive rewards every fixed timestamp or at every environment interaction, just for convenience. In the case of once-in-a-lifetime reward systems, all rewards except the last one will be zero.

As I stated, the purpose of reward is to give an agent feedback about its success, and it's a central thing in RL. Basically, the term *reinforcement* comes from the fact that reward obtained by an agent should reinforce its behavior in a positive or negative way. Reward is *local*, meaning that it reflects the success of the agent's recent activity and not all the successes achieved by the agent so far. Of course, getting a large reward for some action doesn't mean that a second later you won't face dramatic consequences as a result of your previous decisions. It's like robbing a bank—it could look like a good idea until you think about the consequences.

What an agent is trying to achieve is the largest accumulated reward over its sequence of actions. To give you a better understanding of reward, here is a list of some concrete examples with their rewards:

- **Financial trading**: An amount of profit is a reward for a trader buying and selling stocks.

- **Chess**: Reward is obtained at the end of the game as a win, lose, or draw. Of course, it's up to interpretation. For me, for example, achieving a draw in a match against a chess grandmaster would be a huge reward. In practice, we need to specify the exact reward value, but it could be a fairly complicated expression. For instance, in the case of chess, the reward could be proportional to the opponent's strength.

- **Dopamine system in the brain**: There is a part of the brain (limbic system) that produces dopamine every time it needs to send a positive signal to the rest of the brain. Higher concentrations of dopamine lead to a sense of pleasure, which reinforces activities considered by this system to be *good*. Unfortunately, the limbic system is *ancient* in terms of the things it considers good—food, reproduction, and dominance—but that is a totally different story!

- **Computer games**: They usually give obvious feedback to the player, which is either the number of enemies killed or a score gathered. Note in this example that reward is already accumulated, so the RL reward for arcade games should be the derivative of the score, that is, +1 every time a new enemy is killed and 0 at all other time steps.

- **Web navigation**: There are problems, with high practical value, that require the automated extraction of information available on the web. Search engines are trying to solve this task in general, but sometimes, to get to the data you're looking for, you need to fill in some forms or navigate through a series of links, or complete CAPTCHAs, which can be difficult for search engines to do. There is an RL-based approach to those tasks in which the reward is the information or the outcome that you need to get.

- **Neural network (NN) architecture search**: RL has been successfully applied to the domain of NN architecture optimization, where the aim is to get the best performance metric on some dataset by tweaking the number of layers or their parameters, adding extra bypass connections, or making other changes to the NN architecture. The reward in this case is the performance (accuracy or another measure showing how accurate the NN predictions are).

- **Dog training**: If you have ever tried to train a dog, you know that you need to give it something tasty (but not too much) every time it does the thing you've asked. It's also common to punish your pet a bit (negative reward) when it doesn't follow your orders, although recent studies have shown that this isn't as effective as a positive reward.
- **School marks**: We all have experience here! School marks are a reward system designed to give pupils feedback about their studying.

As you can see from the preceding examples, the notion of reward is a very general indication of the agent's performance, and it can be found or artificially injected into lots of practical problems around us.

The agent

An agent is somebody or something who/that interacts with the environment by executing certain actions, making observations, and receiving eventual rewards for this. In most practical RL scenarios, the agent is our piece of software that is supposed to solve some problem in a more-or-less efficient way. For our initial set of six examples, the agents will be as follows:

- **Financial trading**: A trading system or a trader making decisions about order execution
- **Chess**: A player or a computer program
- **Dopamine system**: The brain itself, which, according to sensory data, decides whether it was a good experience
- **Computer games**: The player who enjoys the game or the computer program. (Andrej Karpathy once tweeted that "we were supposed to make AI do all the work and we play games but we do all the work and the AI is playing games!")
- **Web navigation**: The software that tells the browser which links to click on, where to move the mouse, or which text to enter
- **NN architecture search**: The software that controls the concrete architecture of the NN being evaluated
- **Dog training**: You make decisions about the actions (feeding/punishing), so, the agent is you
- **School**: Student/pupil

The environment

The environment is everything outside of an agent. In the most general sense, it's the rest of the universe, but this goes slightly overboard and exceeds the capacity of even tomorrow's computers, so we usually follow the general sense here.

The agent's communication with the environment is limited to reward (obtained from the environment), actions (executed by the agent and given to the environment), and observations (some information besides the reward that the agent receives from the environment). We have discussed reward already, so let's talk about actions and observations next.

Actions

Actions are things that an agent can do in the environment. Actions can, for example, be moves allowed by the rules of play (if it's a game), or doing homework (in the case of school). They can be as simple as *move pawn one space forward* or as complicated as *fill the tax form in for tomorrow morning*.

In RL, we distinguish between two types of actions—discrete or continuous. Discrete actions form the finite set of mutually exclusive things an agent can do, such as move left or right. Continuous actions have some value attached to them, such as a car's action *turn the wheel* having an angle and direction of steering. Different angles could lead to a different scenario a second later, so just *turn the wheel* is definitely not enough.

Observations

Observations of the environment form the second information channel for an agent, with the first being reward. You may be wondering why we need a separate data source. The answer is convenience. Observations are pieces of information that the environment provides the agent with that say what's going on around the agent.

Observations may be relevant to the upcoming reward (such as seeing a bank notification about being paid) or may not be. Observations can even include reward information in some vague or obfuscated form, such as score numbers on a computer game's screen. Score numbers are just pixels, but potentially we could convert them into reward values; it's not a big deal with modern DL at hand.

On the other hand, reward shouldn't be seen as a secondary or unimportant thing—reward is the main force that drives the agent's learning process. If a reward is wrong, noisy, or just slightly off course from the primary objective, then there is a chance that training will go in a wrong direction.

It's also important to distinguish between an environment's state and observations. The state of an environment potentially includes every atom in the universe, which makes it impossible to measure everything about the environment. Even if we limit the environment's state to be small enough, most of the time, it will be either not possible to get full information about it or our measurements will contain noise. This is completely fine, though, and RL was created to support such cases natively. Once again, let's return to our set of examples to capture the difference:

- **Financial trading**: Here, the environment is the whole financial market and everything that influences it. This is a huge list of things, such as the latest news, economic and political conditions, weather, food supplies, and Twitter trends. Even your decision to stay home today can potentially indirectly influence the world's financial system (if you believe in the "butterfly effect"). However, our observations are limited to stock prices, news, and so on. We don't have access to most of the environment's state, which makes trading such a nontrivial thing.

- **Chess**: The environment here is your board *plus* your opponent, which includes their chess skills, mood, brain state, chosen tactics, and so on. Observations are what you see (your current chess position), but, at some levels of play, knowledge of psychology and the ability to read an opponent's mood could increase your chances.

- **Dopamine system**: The environment here is your brain *plus* your nervous system and your organs' states *plus* the whole world you can perceive. Observations are the inner brain state and signals coming from your senses.

- **Computer game**: Here, the environment is your computer's state, including all memory and disk data. For networked games, you need to include other computers *plus* all Internet infrastructure between them and your machine. Observations are a screen's pixels and sound only. These pixels are not a tiny amount of information (somebody calculated that the total number of possible moderate-size images (1024×768) is significantly larger than the number of atoms in our galaxy), but the whole environment state is definitely larger.

- **Web navigation**: The environment here is the Internet, including all the network infrastructure between the computer on which our agent works and the web server, which is a really huge system that includes millions and millions of different components. The observation is normally the web page that is loaded at the current navigation step.

- **NN architecture search**: In this example, the environment is fairly simple and includes the NN toolkit that performs the particular NN evaluation and the dataset that is used to obtain the performance metric. In comparison to the Internet, this looks like a tiny toy environment.

Observations might be different and include some information about testing, such as loss convergence dynamics or other metrics obtained from the evaluation step.

- **Dog training**: Here, the environment is your dog (including its hardly observable inner reactions, mood, and life experiences) and everything around it, including other dogs and even a cat hiding in a bush. Observations are signals from your senses and memory.

- **School**: The environment here is the school itself, the education system of the country, society, and the cultural legacy. Observations are the same as for the dog training example—the student's senses and memory.

This is our mise en scène and we will play around with it in the rest of this book. You will have already noticed that the RL model is extremely flexible and general, and it can be applied to a variety of scenarios. Let's now look at how RL is related to other disciplines, before diving into the details of the RL model.

There are many other areas that contribute or relate to RL. The most significant are shown in the following diagram, which includes six large domains heavily overlapping each other on the methods and specific topics related to decision-making (shown inside the inner gray circle).
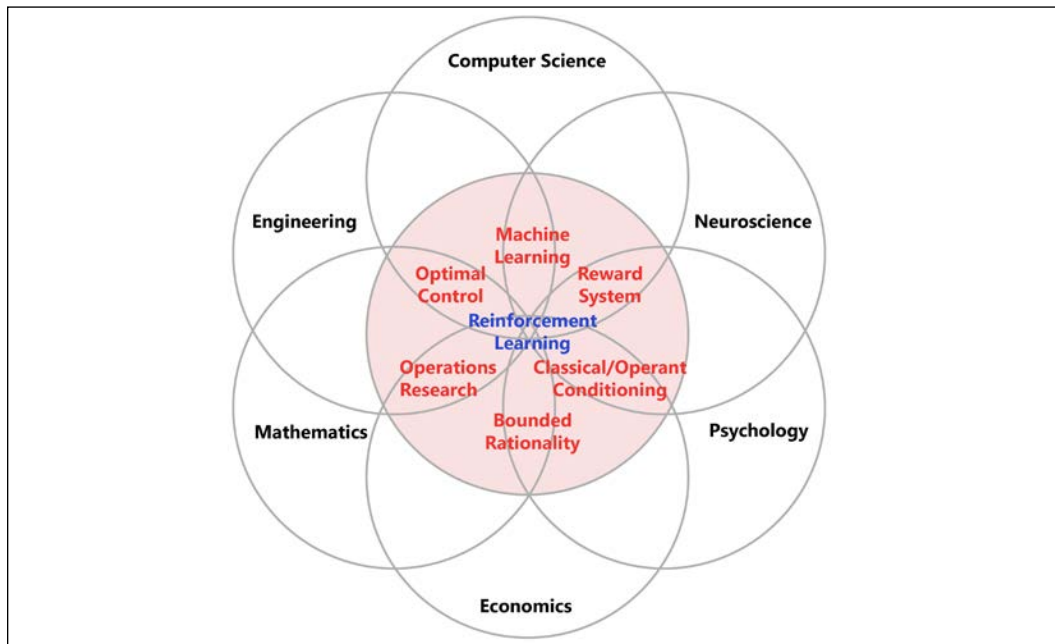
Figure 1.3: Various domains in RL

At the intersection of all those related, but still different, scientific areas sits RL, which is so general and flexible that it can take the best available information from these varying domains:

- **ML**: RL, being a subfield of ML, borrows lots of its machinery, tricks, and techniques from ML. Basically, the goal of RL is to learn how an agent should behave when it is given imperfect observational data.

- **Engineering (especially optimal control)**: This helps with taking a sequence of optimal actions to get the best result.

- **Neuroscience**: We used the dopamine system as our example, and it has been shown that the human brain acts similarly to the RL model.

- **Psychology**: This studies behavior in various conditions, such as how people react and adapt, which is close to the RL topic.

- **Economics**: One of the important topics is how to maximize reward in terms of imperfect knowledge and the changing conditions of the real world.

- **Mathematics**: This works with idealized systems and also devotes significant attention to finding and reaching the optimal conditions in the field of operations research.

In the next part of the chapter, you will become familiar with the theoretical foundations of RL, which will make it possible to start moving toward the methods used to solve the RL problem. The upcoming section is important for understanding the rest of the book.

The theoretical foundations of RL

In this section, I will introduce you to the mathematical representation and notation of the formalisms (reward, agent, actions, observations, and environment) that we just discussed. Then, using this as a knowledge base, we will explore the second-order notions of the RL language, including state, episode, history, value, and gain, which will be used repeatedly to describe different methods later in the book.

Markov decision processes

Before that, we will cover **Markov decision processes** (**MDPs**), which will be described like a Russian matryoshka doll: we will start from the simplest case of a **Markov process** (**MP**), then extend that with rewards, which will turn it into a **Markov reward process**. Then, we will put this idea into an extra envelope by adding actions, which will lead us to an MDP.

MPs and MDPs are widely used in computer science and other engineering fields. So, reading this chapter will be useful for you not only for RL contexts, but also for a much wider range of topics. If you're already familiar with MDPs, then you can quickly skim this chapter, paying attention only to the terminology definitions, as we will use them later on.

The Markov process

Let's start with the simplest child of the Markov family: the MP, which is also known as the **Markov chain**. Imagine that you have some system in front of you that you can only observe. What you observe is called **states**, and the system can switch between states according to some laws of dynamics. Again, you cannot influence the system, but can only watch the states changing.

All possible states for a system form a set called the **state space**. For MPs, we require this set of states to be finite (but it can be extremely large to compensate for this limitation). Your observations form a sequence of states or a **chain** (that's why MPs are also called Markov chains). For example, looking at the simplest model of the weather in some city, we can observe the current day as *sunny* or *rainy*, which is our state space. A sequence of observations over time forms a chain of states, such as [*sunny*, *sunny*, *rainy*, *sunny*, …], and this is called **history**.

To call such a system an MP, it needs to fulfill the **Markov property**, which means that the future system dynamics from any state have to depend on this state only. The main point of the Markov property is to make every observable state self-contained to describe the future of the system. In other words, the Markov property requires the states of the system to be distinguishable from each other and unique. In this case, only one state is required to model the future dynamics of the system and not the whole history or, say, the last *N* states.

In the case of our toy weather example, the Markov property limits our model to represent only the cases when a sunny day can be followed by a rainy one with the same probability, regardless of the amount of sunny days we've seen in the past. It's not a very realistic model, as from common sense we know that the chance of rain tomorrow depends not only on the current conditions but on a large number of other factors, such as the season, our latitude, and the presence of mountains and sea nearby. It was recently proven that even solar activity has a major influence on the weather. So, our example is really naïve, but it's important to understand the limitations and make conscious decisions about them.

Of course, if we want to make our model more complex, we can always do this by extending our state space, which will allow us to capture more dependencies in the model at the cost of a larger state space. For example, if you want to capture separately the probability of rainy days during summer and winter, then you can include the season in your state.

In this case, your state space will be [*sunny+summer, sunny+winter, rainy+summer, rainy+winter*] and so on.

As your system model complies with the Markov property, you can capture transition probabilities with a **transition matrix**, which is a square matrix of the size $N{\times}N$, where N is the number of states in our model. Every cell in a row, i, and a column, j, in the matrix contains the probability of the system to transition from state i to state j.

For example, in our sunny/rainy example, the transition matrix could be as follows:

	Sunny	**Rainy**
Sunny	0.8	0.2
Rainy	0.1	0.9

In this case, if we have a sunny day, then there is an 80% chance that the next day will be sunny and a 20% chance that the next day will be rainy. If we observe a rainy day, then there is a 10% probability that the weather will become better and a 90% probability of the next day being rainy.

So, that's it. The formal definition of an MP is as follows:

- A set of states (S) that a system can be in
- A transition matrix (T), with transition probabilities, which defines the system dynamics

A useful visual representation of an MP is a graph with nodes corresponding to system states and edges, labeled with probabilities representing a possible transition from state to state. If the probability of a transition is 0, we don't draw an edge (there is no way to go from one state to another). This kind of representation is also widely used in finite state machine representation, which is studied in automata theory.
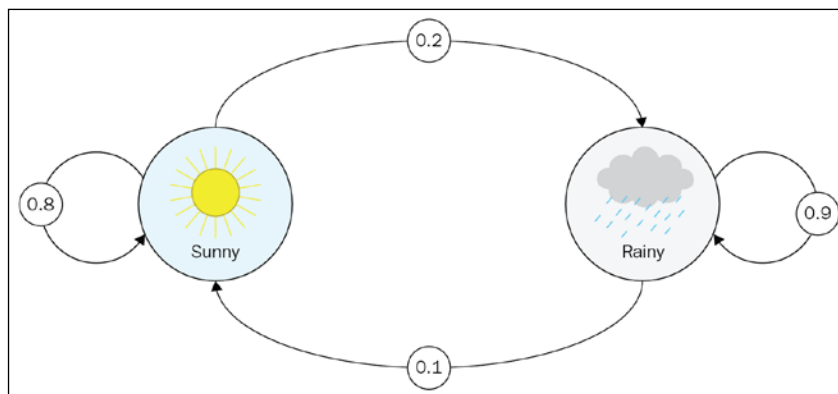
Figure 1.4: The sunny/rainy weather model

Again, we're talking about observation only. There is no way for us to influence the weather, so we just observe it and record our observations.

To give you a more complicated example, let's consider another model called *office worker* (Dilbert, the main character in Scott Adams' famous cartoons, is a good example). His state space in our example has the following states:

- **Home**: He's not at the office
- **Computer**: He's working on his computer at the office
- **Coffee**: He's drinking coffee at the office
- **Chatting**: He's discussing something with colleagues at the office

The state transition graph looks like this:
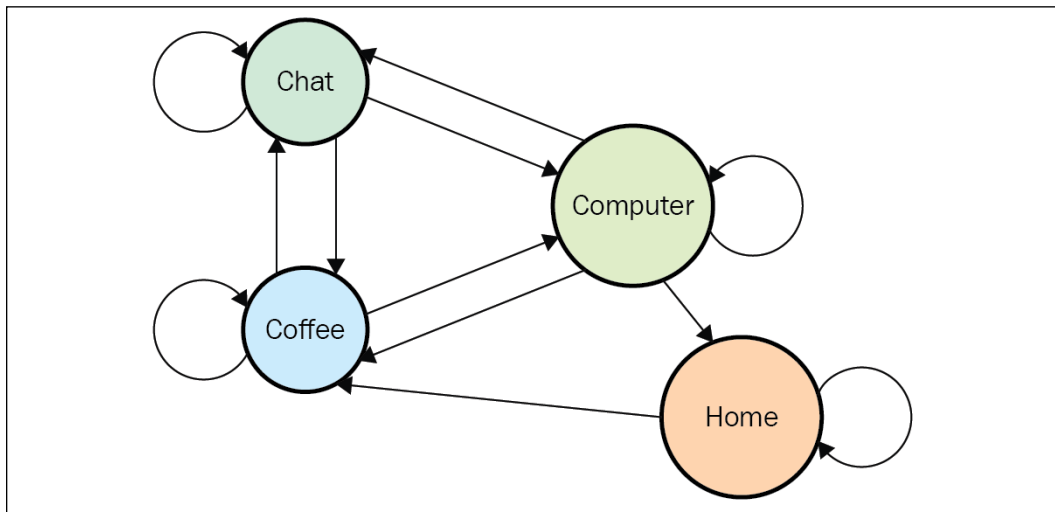
Figure 1.5: The state transition graph for our office worker

We assume that our office worker's weekday usually starts from the **Home** state and that he starts his day with **Coffee** without exception (no **Home** → **Computer** edge and no **Home** → **Chatting** edge). The preceding diagram also shows that workdays always end (that is, going to the **Home** state) from the **Computer** state.

The transition matrix for the preceding diagram is as follows:

	Home	Coffee	Chat	Computer
Home	60%	40%	0%	0%
Coffee	0%	10%	70%	20%

Chat	0%	20%	50%	30%
Computer	20%	20%	10%	50%

The transition probabilities could be placed directly on the state transition graph, as shown here:
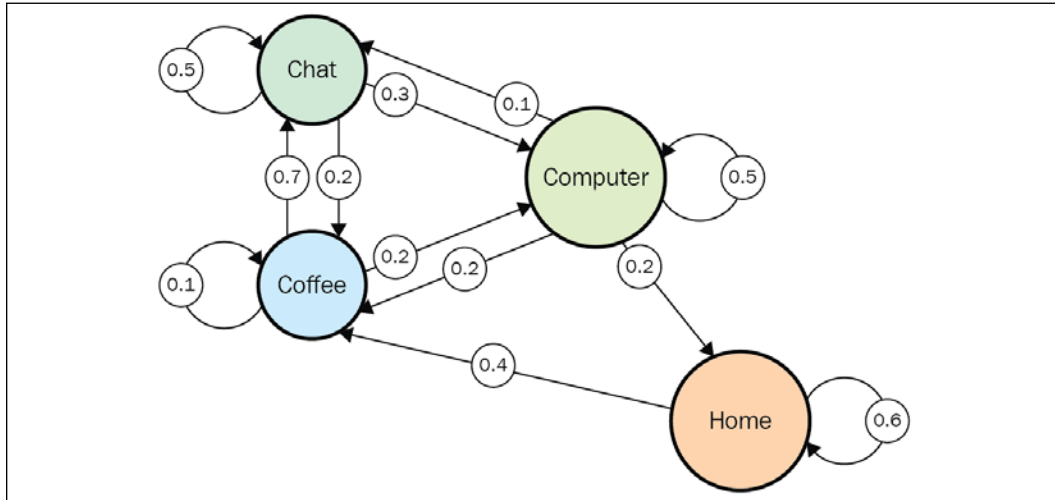
Figure 1.6: The state transition graph with transition probabilities

In practice, we rarely have the luxury of knowing the exact transition matrix. A much more real-world situation is when we only have observations of our system's states, which are also called **episodes**:

- **Home → Coffee → Coffee → Chat → Chat → Coffee → Computer → Computer → Home**
- **Computer → Computer → Chat → Chat → Coffee → Computer → Computer → Computer**
- **Home → Home → Coffee → Chat → Computer → Coffee → Coffee**

It's not complicated to estimate the transition matrix from our observations — we just count all the transitions from every state and normalize them to a sum of 1. The more observation data we have, the closer our estimation will be to the true underlying model.

It's also worth noting that the Markov property implies stationarity (that is, the underlying transition distribution for any state does not change over time). Nonstationarity means that there is some hidden factor that influences our system dynamics, and this factor is not included in observations. However, this contradicts the Markov property, which requires the underlying probability distribution to be the same for the same state regardless of the transition history.

It's important to understand the difference between the actual transitions observed in an episode and the underlying distribution given in the transition matrix. Concrete episodes that we observe are randomly sampled from the distribution of the model, so they can differ from episode to episode. However, the probability of the concrete transition to be sampled remains the same. If this is not the case, Markov chain formalism becomes nonapplicable.

Now we can go further and extend the MP model to make it closer to our RL problems. Let's add rewards to the picture!

Markov reward processes

To introduce reward, we need to extend our MP model a bit. First, we need to add value to our transition from state to state. We already have probability, but probability is being used to capture the dynamics of the system, so now we have an extra scalar number without extra burden.

Reward can be represented in various forms. The most general way is to have another square matrix, similar to the transition matrix, with reward given for transitioning from state i to state j, which reside in row i and column j.

As mentioned, reward can be positive or negative, large or small. In some cases, this representation is redundant and can be simplified. For example, if reward is given for reaching the state regardless of the previous state, we can keep only **state → reward** pairs, which are a more compact representation. However, this is applicable only if the reward value depends solely on the target state, which is not always the case.

The second thing we're adding to the model is the discount factor γ (gamma), which is a single number from 0 to 1 (inclusive). The meaning of this will be explained after the extra characteristics of our Markov reward process have been defined.

As you will remember, we observe a chain of state transitions in an MP. This is still the case for a Markov reward process, but for every transition, we have our extra quantity — reward. So now, all our observations have a reward value attached to every transition of the system.

For every episode, we define **return** at the time, t, as this quantity:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Let's try to understand what this means. For every time point, we calculate return as a sum of subsequent rewards, but more distant rewards are multiplied by the discount factor raised to the power of the number of steps we are away from the starting point at t. The discount factor stands for the foresightedness of the agent. If gamma equals 1, then return, G_t, just equals a sum of all subsequent rewards and corresponds to the agent that has perfect visibility of any subsequent rewards. If gamma equals 0, G_t will be just immediate reward without any subsequent state and will correspond to absolute short-sightedness.

These extreme values are useful only in corner cases, and most of the time, gamma is set to something in between, such as 0.9 or 0.99. In this case, we will look into future rewards, but not too far. The value of $\gamma = 1$ might be applicable in situations of short finite episodes.

This gamma parameter is important in RL, and we will meet it a lot in the subsequent chapters. For now, think about it as a measure of how far into the future we look to estimate the future return. The closer it is to 1, the more steps ahead of us we will take into account.

This return quantity is not very useful in practice, as it was defined for every specific chain we observed from our Markov reward process, so it can vary widely, even for the same state. However, if we go to the extreme and calculate the mathematical expectation of return for any state (by averaging a large number of chains), we will get a much more useful quantity, which is called the **value of the state**:

$$V(s) = \mathbb{E}[G|S_t = s]$$

This interpretation is simple—for every state, s, the value, $V(s)$, is the average (or expected) return we get by following the Markov reward process.

To show this theoretical stuff in practice, let's extend our office worker (Dilbert) process with reward and turn it into a **Dilbert reward process** (**DRP**). Our reward values will be as follows:

- **Home** → **Home** : 1 (as it's good to be home)
- **Home** → **Coffee** : 1
- **Computer** → **Computer** : 5 (working hard is a good thing)
- **Computer** → **Chat** : −3 (it's not good to be distracted)
- **Chat** → **Computer** : 2
- **Computer** → **Coffee** : 1
- **Coffee** → **Computer** : 3

- **Coffee → Coffee** : 1
- **Coffee → Chat** : 2
- **Chat → Coffee** : 1
- **Chat → Chat** : −1   (long conversations become boring)

A diagram of this is shown here:
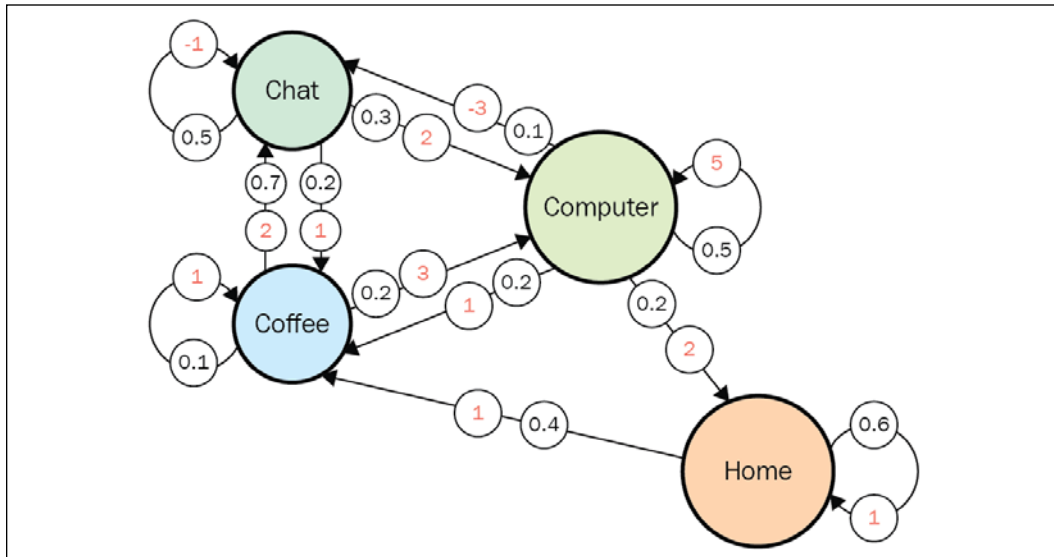
Figure 1.7: A state transition graph with transition probabilities (dark) and rewards (light)

Let's return to our gamma parameter and think about the values of states with different values of gamma. We will start with a simple case: gamma = 0. How do you calculate the values of states here? To answer this question, let's fix our state to **Chat**. What could the subsequent transition be? The answer is that it depends on chance. According to our transition matrix for the Dilbert process, there is a 50% probability that the next state will be **Chat** again, 20% that it will be **Coffee**, and 30% that it will be **Computer**. When gamma = 0, our return is equal only to a value of the next immediate state. So, if we want to calculate the value of the **Chat** state, then we need to sum all transition values and multiply that by their probabilities:

V(chat) = −1 * 0.5 + 2 * 0.3 + 1 * 0.2 = 0.3

V(coffee) = 2 * 0.7 + 1 * 0.1 + 3 * 0.2 = 2.1

V(home) = 1 * 0.6 + 1 * 0.4 = 1.0

V(computer) = 5 * 0.5 + (−3) * 0.1 + 1 * 0.2 + 2 * 0.2 = 2.8

So, **Computer** is the most valuable state to be in (if we care only about immediate reward), which is not surprising as **Computer → Computer** is frequent, has a large reward, and the ratio of interruptions is not too high.

Now a trickier question—what's the value when gamma = 1? Think about this carefully. The answer is that the value is infinite for all states. Our diagram doesn't contain *sink* states (states without outgoing transitions), and when our discount equals 1, we care about a potentially infinite number of transitions in the future. As you've seen in the case of gamma = 0, all our values are positive in the short term, so the sum of the infinite number of positive values will give us an infinite value, regardless of the starting state.

This infinite result shows us one of the reasons to introduce gamma into a Markov reward process instead of just summing all future rewards. In most cases, the process can have an infinite (or large) amount of transitions. As it is not very practical to deal with infinite values, we would like to limit the horizon we calculate values for. Gamma with a value less than 1 provides such a limitation, and we will discuss this later in this book. On the other hand, if you're dealing with finite-horizon environments (for example, the tic-tac-toe game, which is limited by at most nine steps), then it will be fine to use gamma = 1. As another example, there is an important class of environments with only one step called the **multi-armed bandit MDP**. This means that on every step, you need to make a selection of one alternative action, which provides you with some reward and the episode ends.

As I already mentioned about the Markov reward process, gamma is usually set to a value between 0 and 1. However, with such values, it becomes almost impossible to calculate them accurately by hand, even for Markov reward processes as small as our Dilbert example, because it will require summing hundreds of values. Computers are good at tedious tasks such as this, and there are several simple methods that can quickly calculate values for Markov reward processes for given transition and reward matrices. We will see and even implement one such method in *Chapter 5*, *Tabular Learning and the Bellman Equation*, when we will start looking at Q-learning methods.

For now, let's put another layer of complexity around our Markov reward processes and introduce the final missing piece: actions.

Adding actions

You may already have ideas about how to extend our Markov reward process to include actions. Firstly, we must add a set of actions (*A*), which has to be finite. This is our agent's **action space**. Secondly, we need to condition our transition matrix with actions, which basically means that our matrix needs an extra action dimension, which turns it into a cube.

If you remember, in the case of MPs and Markov reward processes, the transition matrix had a square form, with the source state in rows and target state in columns. So, every row, *i*, contained a list of probabilities to jump to every state:
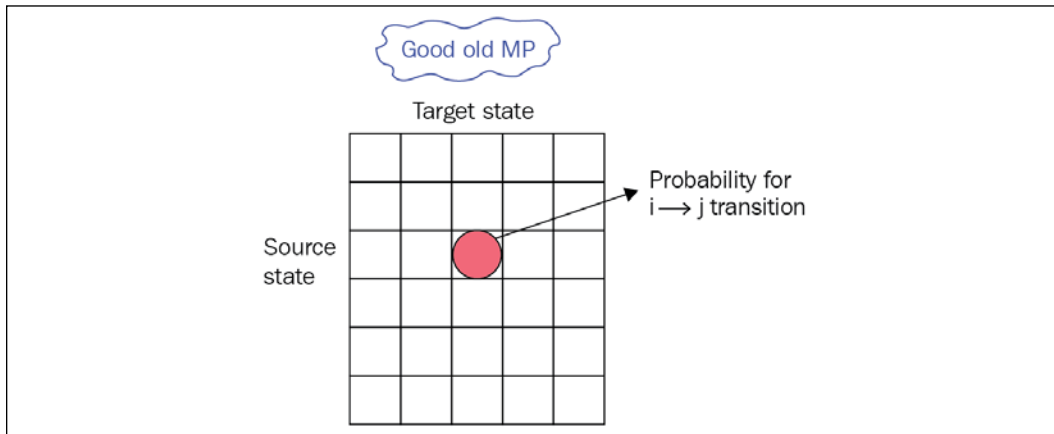
Figure 1.8: The transition matrix in square form

Now the agent no longer passively observes state transitions, but can actively choose an action to take at every state transition. So, for every source state, we don't have a list of numbers, but we have a matrix, where the **depth dimension** contains actions that the agent can take, and the other dimension is what the target state system will jump to after actions are performed by the agent. The following diagram shows our new transition table, which became a cube with the source state as the height dimension (indexed by **i**), the target state as the width (**j**), and the action the agent can take as the depth (**k**) of the transition table:
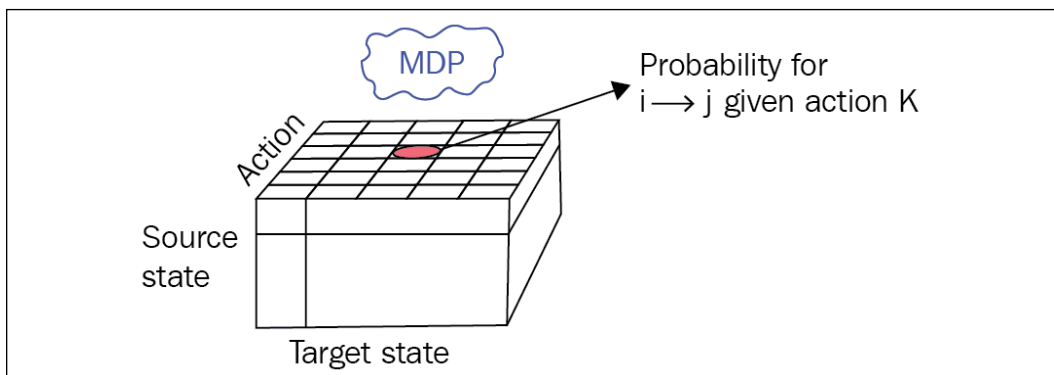
Figure 1.9: Transition probabilities for the MDP

So, in general, by choosing an action, the agent can affect the probabilities of the target states, which is a useful ability.

To give you an idea of why we need so many complications, let's imagine a small robot that lives in a 3×3 grid and can execute the actions *turn left*, *turn right*, and *go forward*. The state of the world is the robot's position plus orientation (up, down, left, and right), which gives us 3 × 3 × 4 = 36 states (the robot can be at any location in any orientation).

Also, imagine that the robot has imperfect motors (which is frequently the case in the real world), and when it executes *turn left* or *turn right*, there is a 90% chance that the desired turn happens, but sometimes, with a 10% probability, the wheel slips and the robot's position stays the same. The same happens with *go forward*—in 90% of cases it works, but for the rest (10%) the robot stays at the same position.

In the following illustration, a small part of a transition diagram is shown, displaying the possible transitions from the state (1, 1, up), when the robot is in the center of the grid and facing up. If the robot tries to move forward, there is a 90% chance that it will end up in the state (0, 1, up), but there is a 10% probability that the wheels will slip and the target position will remain (1, 1, up).

To properly capture all these details about the environment and possible reactions to the agent's actions, the general MDP has a 3D transition matrix with the dimensions source state, action, and target state.
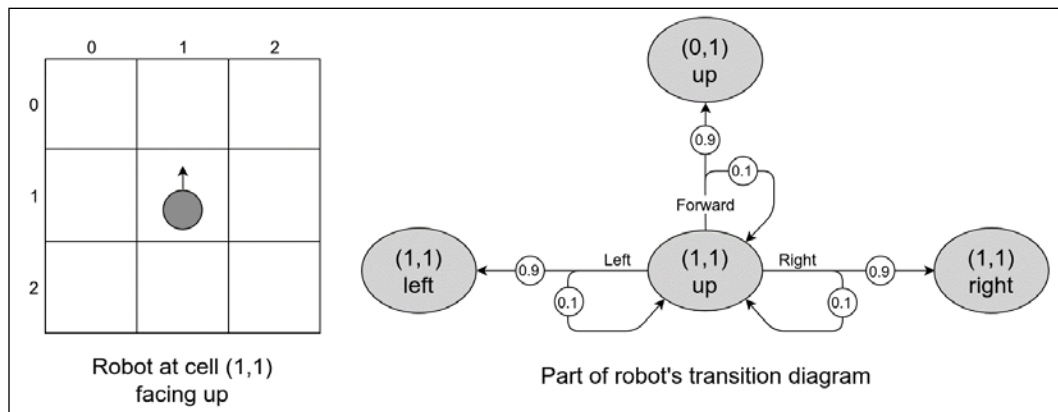
Figure 1.10: A grid world environment

Finally, to turn our Markov reward process into an MDP, we need to add actions to our reward matrix in the same way that we did with the transition matrix. Our reward matrix will depend not only on the state but also on the action. In other words, the reward the agent obtains will now depend not only on the state it ends up in but also on the action that leads to this state.

This is similar to when you put effort into something—you're usually gaining skills and knowledge, even if the result of your efforts wasn't too successful. So, the reward could be better if you're doing something rather than not doing something, even if the final result is the same.

Now, with a formally defined MDP, we're finally ready to cover the most important thing for MDPs and RL: **policy**.

Policy

The simple definition of policy is that it is some set of rules that controls the agent's behavior. Even for fairly simple environments, we can have a variety of policies. For example, in the preceding example with the robot in the grid world, the agent can have different policies, which will lead to different sets of visited states. For example, the robot can perform the following actions:

- Blindly move forward regardless of anything
- Try to go around obstacles by checking whether that previous *forward* action failed
- Funnily spin around to entertain its creator
- Choose an action by randomly modeling a drunk robot in the grid world scenario

You may remember that the main objective of the agent in RL is to gather as much return as possible. So, again, different policies can give us different amounts of return, which makes it important to find a good policy. This is why the notion of policy is important.

Formally, policy is defined as the probability distribution over actions for every possible state:

$$\pi(a|s) = P[A_t = a|S_t = s]$$

This is defined as probability and not as a concrete action to introduce randomness into an agent's behavior. We will talk later in the book about why this is important and useful. Deterministic policy is a special case of probabilistics with the needed action having 1 as its probability.

Another useful notion is that if our policy is fixed and not changing, then our MDP becomes a Markov reward process, as we can reduce the transition and reward matrices with a policy's probabilities and get rid of the action dimensions.

Congratulations on getting to this stage! This chapter was challenging, but it was important for understanding subsequent practical material. After two more introductory chapters about OpenAI Gym and deep learning, we will finally start tackling this question—how do we teach agents to solve practical tasks?

Summary

In this chapter, you started your journey into the RL world by learning what makes RL special and how it relates to the supervised and unsupervised learning paradigms. We then learned about the basic RL formalisms and how they interact with each other, after which we covered MPs, Markov reward processes, and MDPs. This knowledge will be the foundation for the material that we will cover in the rest of the book.

In the next chapter, we will move away from the formal theory to the practice of RL. We will cover the setup required and libraries, and then you will write your first agent.

2
OpenAI Gym

After talking so much about the theoretical concepts of reinforcement learning (RL) in *Chapter 1*, *What Is Reinforcement Learning?*, let's start doing something practical! In this chapter, you will learn the basics of OpenAI Gym, a library used to provide a uniform API for an RL agent and lots of RL environments. This removes the need to write boilerplate code.

You will also write your first randomly behaving agent and become more familiar with the basic concepts of RL that we have covered so far. By the end of the chapter, you will have an understanding of:

- The high-level requirements that need to be implemented to plug the agent into the RL framework
- A basic, pure-Python implementation of the random RL agent
- OpenAI Gym

The anatomy of the agent

As you learned in the previous chapter, there are several entities in RL's view of the world:

- **The agent**: A thing, or person, that takes an active role. In practice, the agent is some piece of code that implements some policy. Basically, this policy decides what action is needed at every time step, given our observations.
- **The environment**: Some model of the world that is external to the agent and has the responsibility of providing observations and giving rewards. The environment changes its state based on the agent's actions.

Let's explore how both can be implemented in Python for a simple situation. We will define an environment that will give the agent random rewards for a limited number of steps, regardless of the agent's actions. This scenario is not very useful, but it will allow us to focus on specific methods in both the environment and agent classes. Let's start with the environment:

```python
class Environment:
    def __init__(self):
        self.steps_left = 10
```

In the preceding code, we allowed the environment to initialize its internal state. In our case, the state is just a counter that limits the number of time steps that the agent is allowed to take to interact with the environment.

```python
def get_observation(self) -> List[float]:
    return [0.0, 0.0, 0.0]
```

The `get_observation()` method is supposed to return the current environment's observation to the agent. It is usually implemented as some function of the internal state of the environment. If you're curious about what is meant by `-> List[float]`, that's an example of Python type annotations, which were introduced in Python 3.5. You can find out more in the documentation at `https://docs.python.org/3/library/typing.html`. In our example, the observation vector is always zero, as the environment basically has no internal state.

```python
def get_actions(self) -> List[int]:
    return [0, 1]
```

The `get_actions()` method allows the agent to query the set of actions it can execute. Normally, the set of actions that the agent can execute does not change over time, but some actions can become impossible in different states (for example, not every move is possible in any position of the tic-tac-toe game). In our simplistic example, there are only two actions that the agent can carry out, which are encoded with the integers 0 and 1.

```python
def is_done(self) -> bool:
    return self.steps_left == 0
```

The preceding method signaled the end of the episode to the agent. As you saw in *Chapter 1*, *What Is Reinforcement Learning?*, the series of environment-agent interactions is divided into a sequence of steps called episodes. Episodes can be finite, like in a game of chess, or infinite, like the Voyager 2 mission (a famous space probe that was launched over 40 years ago and has traveled beyond our solar system). To cover both scenarios, the environment provides us with a way to detect when an episode is over and there is no way to communicate with it anymore.

```
    def action(self, action: int) -> float:
        if self.is_done():
            raise Exception("Game is over")
        self.steps_left -= 1
        return random.random()
```

The `action()` method is the central piece in the environment's functionality. It does two things – handles an agent's action and returns the reward for this action. In our example, the reward is random and its action is discarded. Additionally, we update the count of steps and refuse to continue the episodes that are over.

Now when looking at the agent's part, it is much simpler and includes only two methods: the constructor and the method that performs one step in the environment:

```
class Agent:
    def __init__(self):
        self.total_reward = 0.0
```

In the constructor, we initialize the counter that will keep the total reward accumulated by the agent during the episode.

```
    def step(self, env: Environment):
        current_obs = env.get_observation()
        actions = env.get_actions()
        reward = env.action(random.choice(actions))
        self.total_reward += reward
```

The `step` function accepts the environment instance as an argument and allows the agent to perform the following actions:

- Observe the environment
- Make a decision about the action to take based on the observations
- Submit the action to the environment
- Get the reward for the current step

For our example, the agent is dull and ignores the observations obtained during the decision-making process about which action to take. Instead, every action is selected randomly. The final piece is the glue code, which creates both classes and runs one episode:

```
if __name__ == "__main__":
    env = Environment()
    agent = Agent()
```

```
    while not env.is_done():
        agent.step(env)

    print("Total reward got: %.4f" % agent.total_reward)
```

You can find the preceding code in this book's GitHub repository at `https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition` in the `Chapter02/01_agent_anatomy.py` file. It has no external dependencies and should work with any more-or-less modern Python version. By running it several times, you'll get different amounts of reward gathered by the agent.

The simplicity of the preceding code illustrates the important basic concepts that come from the RL model. The environment could be an extremely complicated physics model, and an agent could easily be a large neural network (NN) that implements the latest RL algorithm, but the basic pattern will stay the same – on every step, the agent will take some observations from the environment, do its calculations, and select the action to take. The result of this action will be a reward and a new observation.

You may ask, if the pattern is the same, why do we need to write it from scratch? What if it is already implemented by somebody and could be used as a library? Of course, such frameworks exist, but before we spend some time discussing them, let's prepare your development environment.

Hardware and software requirements

The examples in this book were implemented and tested using Python version 3.7. I assume that you're already familiar with the language and common concepts such as virtual environments, so I won't cover in detail how to install the package and how to do this in an isolated way. The examples will use the previously mentioned Python type annotations, which will allow us to provide type signatures for functions and class methods.

The external libraries that we will use in this book are open source software, and they include the following:

- **NumPy**: This is a library for scientific computing, and implementing matrix operations and common functions.
- **OpenCV Python bindings**: This is a computer vision library and provides many functions for image processing.

- **Gym**: This is an RL framework that has various environments that can be communicated with in a unified way.

- **PyTorch**: This is a flexible and expressive **deep learning** (**DL**) library. A short crash course on it will be given in *Chapter 3, Deep Learning with PyTorch.*

- **PyTorch Ignite**: This is a set of high-level tools on top of PyTorch used to reduce boilerplate code. It will be covered briefly in *Chapter 3*. The full documentation is available here: `https://pytorch.org/ignite/`.

- **PTAN** (`https://github.com/Shmuma/ptan`): This is an open source extension to Gym that I created to support the modern deep RL methods and building blocks. All classes used will be described in detail together with the source code.

Other libraries will be used for specific chapters; for example, we will use Microsoft TextWorld for solving text-based games, PyBullet for robotic simulations, OpenAI Universe for browser-based automation problems, and so on. Those specialized chapters will include installation instructions for those libraries.

A significant portion of this book (parts two, three, and four) is focused on the modern deep RL methods that have been developed over the past few years. The word "deep" in this context means that DL is heavily used. You may be aware that DL methods are computationally hungry. One modern graphics processing unit (GPU) can be 10 to 100 times faster than even the fastest multiple central processing unit (CPU) systems. In practice, this means that the same code that takes one hour to train on a system with a GPU could take from half a day to one week even on the fastest CPU system. It doesn't mean that you can't try the examples from this book without having access to a GPU, but it will take longer. To experiment with the code on your own (the most useful way to learn anything), it is better to get access to a machine with a GPU. This can be done in various ways:

- Buying a modern GPU suitable for CUDA

- Using cloud instances. Both Amazon Web Services and Google Cloud can provide you with GPU-powered instances

- Google Colab offers free GPU access to its Jupyter notebooks

The instructions on how to set up the system are beyond the scope of this book, but there are plenty of manuals available on the Internet. In terms of an operating system (OS), you should use Linux or macOS. Windows is supported by PyTorch and Gym, but the examples in the book were not fully tested under Windows OS.

To give you the exact versions of the external dependencies that we will use throughout the book, here is an output of the `pip freeze` command (it may be useful for the potential troubleshooting of examples in the book, as open source software and DL toolkits are evolving extremely quickly):

```
atari-py==0.2.6
gym==0.15.3
numpy==1.17.2
opencv-python==4.1.1.26
tensorboard==2.0.1
torch==1.3.0
torchvision==0.4.1
pytorch-ignite==0.2.1
tensorboardX==1.9
tensorflow==2.0.0
ptan==0.6
```

All the examples in the book were written and tested with PyTorch 1.3, which can be installed by following the instructions on the `http://pytorch.org` website (normally, that's just the `conda install pytorch torchvision -c pytorch` command).

Now, let's go into the details of the OpenAI Gym API, which provides us with tons of environments, from trivial to challenging ones.

The OpenAI Gym API

The Python library called Gym was developed and has been maintained by OpenAI (`www.openai.com`). The main goal of Gym is to provide a rich collection of environments for RL experiments using a unified interface. So, it is not surprising that the central class in the library is an environment, which is called `Env`. Instances of this class expose several methods and fields that provide the required information about its capabilities. At a high level, every environment provides these pieces of information and functionality:

- A set of actions that is allowed to be executed in the environment. Gym supports both discrete and continuous actions, as well as their combination

- The shape and boundaries of the observations that the environment provides the agent with

- A method called `step` to execute an action, which returns the current observation, the reward, and the indication that the episode is over

- A method called `reset`, which returns the environment to its initial state and obtains the first observation

Let's now talk about these components of the environment in detail.

The action space

As mentioned, the actions that an agent can execute can be discrete, continuous, or a combination of the two. Discrete actions are a fixed set of things that an agent can do, for example, directions in a grid like left, right, up, or down. Another example is a push button, which could be either pressed or released. Both states are mutually exclusive, because a main characteristic of a discrete action space is that only one action from a finite set of actions is possible.

A continuous action has a value attached to it, for example, a steering wheel, which can be turned at a specific angle, or an accelerator pedal, which can be pressed with different levels of force. A description of a continuous action includes the boundaries of the value that the action could have. In the case of a steering wheel, it could be from −720 degrees to 720 degrees. For an accelerator pedal, it's usually from 0 to 1.

Of course, we are not limited to a single action; the environment could take multiple actions, such as pushing multiple buttons simultaneously or steering the wheel and pressing two pedals (the brake and the accelerator). To support such cases, Gym defines a special container class that allows the nesting of several action spaces into one unified action.

The observation space

As mentioned in *Chapter 1*, *What Is Reinforcement Learning?*, observations are pieces of information that an environment provides the agent with, on every timestamp, besides the reward. Observations can be as simple as a bunch of numbers or as complex as several multidimensional tensors containing color images from several cameras. An observation can even be discrete, much like action spaces. An example of a discrete observation space is a lightbulb, which could be in two states – on or off, given to us as a Boolean value.

So, you can see the similarity between actions and observations, and how they have found their representation in Gym's classes. Let's look at a class diagram:
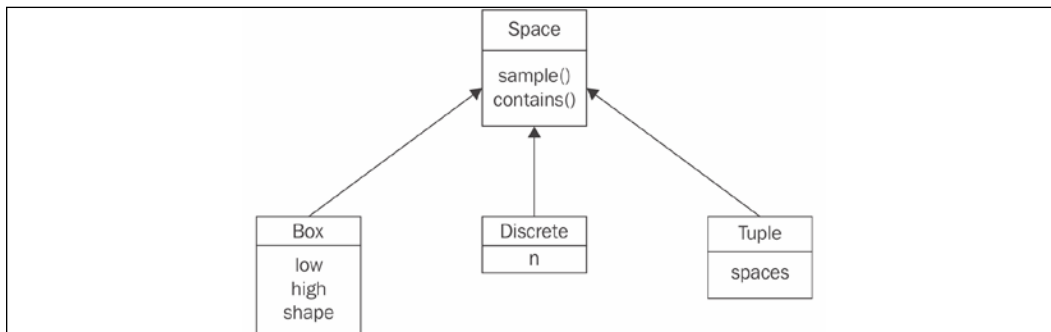
Figure 2.1: The hierarchy of the Space class in Gym

The basic abstract class `Space` includes two methods that are relevant to us:

- `sample()`: This returns a random sample from the space
- `contains(x)`: This checks whether the argument, x, belongs to the space's domain

Both of these methods are abstract and reimplemented in each of the `Space` subclasses:

- The `Discrete` class represents a mutually exclusive set of items, numbered from 0 to $n - 1$. Its only field, n, is a count of the items it describes. For example, `Discrete(n=4)` can be used for an action space of four directions to move in [left, right, up, or down].

- The `Box` class represents an n-dimensional tensor of rational numbers with intervals [low, high]. For instance, this could be an accelerator pedal with one single value between `0.0` and `1.0`, which could be encoded by `Box(low=0.0, high=1.0, shape=(1,), dtype=np.float32)` (the `shape` argument is assigned a tuple of length `1` with a single value of `1`, which gives us a one-dimensional tensor with a single value). The `dtype` parameter specifies the space's value type and here we specify it as a `NumPy 32-bit float`. Another example of `Box` could be an Atari screen observation (we will cover lots of Atari environments later), which is an RGB (red, green, and blue) image of size 210×160: `Box(low=0, high=255, shape=(210, 160, 3), dtype=np.uint8)`. In this case, the `shape` argument is a tuple of three elements: the first dimension is the height of the image, the second is the width, and the third equals `3`, which all correspond to three color planes for red, green, and blue, respectively. So, in total, every observation is a three-dimensional tensor with 100,800 bytes.

- The final child of `Space` is a `Tuple` class, which allows us to combine several `Space` class instances together. This enables us to create action and observation spaces of any complexity that we want. For example, imagine we want to create an action space specification for a car. The car has several controls that can be changed at every timestamp, including the steering wheel angle, brake pedal position, and accelerator pedal position. These three controls can be specified by three float values in one single `Box` instance. Besides these essential controls, the car has extra discrete controls, like a turn signal (which could be off, right, or left) or horn (on or off). To combine all of this into one action space specification class, we can create `Tuple(spaces=(Box(low=-1.0, high=1.0, shape=(3,), dtype=np.float32), Discrete(n=3),Discrete(n=2)))`. This flexibility is rarely used; for example, in this book, you will see only the `Box` and `Discrete` actions and observation spaces, but the `Tuple` class can be useful in some cases.

There are other `Space` subclasses defined in Gym, but the preceding three are the most useful ones. All subclasses implement the `sample()` and `contains()` methods. The `sample()` function performs a random sample corresponding to the `Space` class and parameters. This is mostly useful for action spaces, when we need to choose the random action. The `contains()` method verifies that the given arguments comply with the `Space` parameters, and it is used in the internals of Gym to check an agent's actions for sanity. For example, `Discrete.sample()` returns a random element from a discrete range, and `Box.sample()` will be a random tensor with proper dimensions and values lying inside the given range.

Every environment has two members of type `Space`: the `action_space` and `observation_space`. This allows us to create generic code that could work with any environment. Of course, dealing with the pixels of the screen is different from handling discrete observations (as in the former case, we may want to preprocess images with convolutional layers or with other methods from the computer vision toolbox); so, most of the time, this means optimizing the code for a particular environment or group of environments, but Gym doesn't prevent us from writing generic code.

The environment

The environment is represented in Gym by the `Env` class, as mentioned earlier, which has the following members:

- `action_space`: This is the field of the `Space` class and provides a specification for allowed actions in the environment.

- observation_space: This field has the same Space class, but specifies the observations provided by the environment.
- reset(): This resets the environment to its initial state, returning the initial observation vector.
- step(): This method allows the agent to take the action and returns information about the outcome of the action – the next observation, the local reward, and the end-of-episode flag. This method is a bit complicated and we we will look at it in detail later in this section.

There are extra utility methods in the Env class, such as render(), which allows us to obtain the observation in a human-friendly form, but we won't use them. You can find the full list in Gym's documentation, but let's focus on the core Env methods: reset() and step().

So far, you have seen how our code can get information about the environment's actions and observations, so now you need to get familiar with actioning itself. Communications with the environment are performed via step and reset.

As reset is much simpler, we will start with it. The reset() method has no arguments; it instructs an environment to reset into its initial state and obtain the initial observation. Note that you have to call reset() after the creation of the environment. As you may remember from *Chapter 1*, *What Is Reinforcement Learning?*, the agent's communication with the environment may have an end (like a "Game Over" screen). Such sessions are called episodes, and after the end of the episode, an agent needs to start over. The value returned by this method is the first observation of the environment.

The step() method is the central piece in the environment's functionality. It does several things in one call, which are as follows:

- Telling the environment which action we will execute on the next step
- Getting the new observation from the environment after this action
- Getting the reward the agent gained with this step
- Getting the indication that the episode is over

The first item (action) is passed as the only argument to this method, and the rest are returned by the step() method. Precisely, this is a tuple (Python tuple and not the Tuple class we discussed in the previous section) of four elements (observation, reward, done, and info). They have these types and meanings:

- observation: This is a NumPy vector or a matrix with observation data.

- `reward`: This is the float value of the reward.

- `done`: This is a Boolean indicator, which is `True` when the episode is over.

- `info`: This could be anything environment-specific with extra information about the environment. The usual practice is to ignore this value in general RL methods (not taking into account the specific details of the particular environment).

You may have already got the idea of environment usage in an agent's code – in a loop, we call the `step()` method with an action to perform until this method's `done` flag becomes `True`. Then we can call `reset()` to start over. There is only one piece missing – how we create `Env` objects in the first place.

Creating an environment

Every environment has a unique name of the `EnvironmentName-vN` form, where *N* is the number used to distinguish between different versions of the same environment (when, for example, some bugs get fixed or some other major changes are made). To create an environment, the `gym` package provides the `make(env_name)` function, whose only argument is the environment's name in string form.

At the time of writing, Gym version 0.13.1 contains 859 environments with different names. Of course, all of those are not unique environments, as this list includes all versions of an environment. Additionally, the same environment can have different variations in the settings and observations spaces. For example, the Atari game Breakout has these environment names:

- **Breakout-v0, Breakout-v4**: The original Breakout with a random initial position and direction of the ball

- **BreakoutDeterministic-v0, BreakoutDeterministic-v4**: Breakout with the same initial placement and speed vector of the ball

- **BreakoutNoFrameskip-v0, BreakoutNoFrameskip-v4**: Breakout with every frame displayed to the agent

- **Breakout-ram-v0, Breakout-ram-v4**: Breakout with the observation of the full Atari emulation memory (128 bytes) instead of screen pixels

- **Breakout-ramDeterministic-v0**, **Breakout-ramDeterministic-v4**

- **Breakout-ramNoFrameskip-v0**, **Breakout-ramNoFrameskip-v4**

In total, there are 12 environments for good old Breakout. In case you've never seen it before, here is a screenshot of its gameplay:

Figure 2.2: The gameplay of Breakout

Even after the removal of such duplicates, Gym 0.13.1 comes with an impressive list of 154 unique environments, which can be divided into several groups:

- **Classic control problems**: These are toy tasks that are used in optimal control theory and RL papers as benchmarks or demonstrations. They are usually simple, with low-dimension observation and action spaces, but they are useful as quick checks when implementing algorithms. Think about them as the "MNIST for RL" (MNIST is a handwriting digit recognition dataset from *Yann LeCun*, which you can find at http://yann.lecun.com/exdb/mnist/).

- **Atari 2600**: These are games from the classic game platform from the 1970s. There are 63 unique games.

- **Algorithmic**: These are problems that aim to perform small computation tasks, such as copying the observed sequence or adding numbers.

- **Board games**: These are the games of Go and Hex.

- **Box2D**: These are environments that use the Box2D physics simulator to learn walking or car control.

- **MuJoCo**: This is another physics simulator used for several continuous control problems.

- **Parameter tuning**: This is RL being used to optimize NN parameters.

- **Toy text**: These are simple grid world text environments.

- **PyGame**: These are several environments implemented using the PyGame engine.

- **Doom**: These are nine mini-games implemented on top of ViZDoom.

The full list of environments can be found at `https://gym.openai.com/envs` or on the wiki page in the project's GitHub repository. An even larger set of environments is available in OpenAI Universe (currently discontinued by OpenAI), which provides general connectors to virtual machines while running Flash and native games, web browsers, and other real-world applications. OpenAI Universe extends the Gym API, but it follows the same design principles and paradigm. You can check it out at `https://github.com/openai/universe`. We will deal with Universe more closely in *Chapter 13*, *Asynchronous Advantage Actor-Critic*, in terms of MiniWoB and browser automation.

Enough theory! Let's now look at a Python session working with one of Gym's environments.

The CartPole session

Let's apply our knowledge and explore one of the simplest RL environments that Gym provides.

```
$ python
Python 3.7.5 |Anaconda, Inc.| (default, Mar 29 2018, 18:21:58)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import gym
>>> e = gym.make('CartPole-v0')
```

Here, we have imported the `gym` package and created an environment called `CartPole`. This environment is from the classic control group and its gist is to control the platform with a stick attached by its bottom part (see the following figure).

The trickiness is that this stick tends to fall right or left and you need to balance it by moving the platform to the right or left on every step.
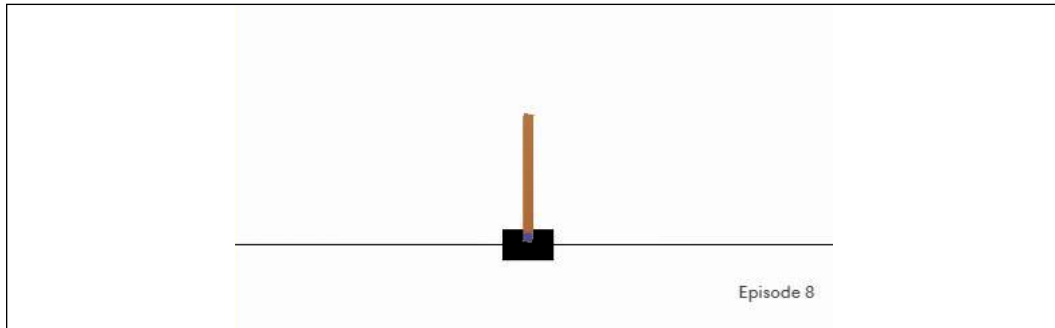
Figure 2.3: The CartPole environment

The observation of this environment is four floating-point numbers containing information about the *x* coordinate of the stick's center of mass, its speed, its angle to the platform, and its angular speed. Of course, by applying some math and physics knowledge, it won't be complicated to convert these numbers into actions when we need to balance the stick, but our problem is this – how do we learn to balance this system *without knowing* the exact meaning of the observed numbers and only by getting the reward? The reward in this environment is 1, and it is given on every time step. The episode continues until the stick falls, so to get a more accumulated reward, we need to balance the platform in a way to avoid the stick falling.

This problem may look difficult, but in just two chapters, we will write the algorithm that will easily solve CartPole in minutes, without any idea about what the observed numbers mean. We will do it only by trial and error and using a bit of RL magic.

Let's continue with our session.

```
>>> obs = e.reset()
>>> obs
array([-0.04937814, -0.0266909 , -0.03681807, -0.00468688])
```

Here, we reset the environment and obtained the first observation (we always need to reset the newly created environment). As I said, the observation is four numbers, so let's check how we can know this in advance.

```
>>> e.action_space
Discrete(2)
>>> e.observation_space
Box(4,)
```

The `action_space` field is of the `Discrete` type, so our actions will be just 0 or 1, where 0 means pushing the platform to the left and 1 means to the right. The observation space is of `Box(4,)`, which means a vector of size `4` with values inside the `[-inf, inf]` interval.

```
>>> e.step(0)
(array([-0.04991196, -0.22126602, -0.03691181, 0.27615592]), 1.0,
False, {})
```

Here, we pushed our platform to the left by executing the action `0` and got the tuple of four elements:

- A new observation, which is a new vector of four numbers
- A reward of `1.0`
- The `done` flag with value `False`, which means that the episode is not over yet and we are more or less okay
- Extra information about the environment, which is an empty dictionary

Next, we will use the `sample()` method of the `Space` class on the `action_space` and `observation_space`.

```
>>> e.action_space.sample()
0
>>> e.action_space.sample()
1
>>> e.observation_space.sample()
array([2.06581792e+00, 6.99371255e+37, 3.76012475e-02,
-5.19578481e+37])
>>> e.observation_space.sample()
array([4.6860966e-01, 1.4645028e+38, 8.6090848e-02, 3.0545910e+37])
```

This method returned a random sample from the underlying space, which in the case of our `Discrete` action space means a random number of 0 or 1, and for the observation space means a random vector of four numbers. The random sample of the observation space may not look useful, and this is true, but the sample from the action space could be used when we are not sure how to perform an action. This feature is especially handy because you don't know any RL methods yet, but we still want to play around with the Gym environment. Now that you know enough to implement your first randomly behaving agent for CartPole, let's do it.

The random CartPole agent

Although the environment is much more complex than our first example in *The anatomy of the agent* section, the code of the agent is much shorter. This is the power of reusability, abstractions, and third-party libraries!

So, here is the code (you can find it in `Chapter02/02_cartpole_random.py`).

```
import gym


if __name__ == "__main__":
    env = gym.make("CartPole-v0")
    total_reward = 0.0
    total_steps = 0
    obs = env.reset()
```

Here, we created the environment and initialized the counter of steps and the reward accumulator. On the last line, we reset the environment to obtain the first observation (which we will not use, as our agent is stochastic).

```
    while True:
        action = env.action_space.sample()
        obs, reward, done, _ = env.step(action)
        total_reward += reward
        total_steps += 1
        if done:
            break

    print("Episode done in %d steps, total reward %.2f" % (
        total_steps, total_reward))
```

In this loop, we sampled a random action, then asked the environment to execute it and return to us the next observation (`obs`), the `reward`, and the `done` flag. If the episode is over, we stop the loop and show how many steps we have taken and how much reward has been accumulated. If you start this example, you will see something like this (not exactly, though, due to the agent's randomness):

```
rl_book_samples/Chapter02$ python 02_cartpole_random.py
Episode done in 12 steps, total reward 12.00
```

As with the interactive session, the warning is not related to our code, but to Gym's internals. On average, our random agent takes 12 to 15 steps before the pole falls and the episode ends. Most of the environments in Gym have a "reward boundary," which is the average reward that the agent should gain during 100 consecutive episodes to "solve" the environment. For CartPole, this boundary is 195, which means that, on average, the agent must hold the stick for 195 time steps or longer. Using this perspective, our random agent's performance looks poor. However, don't be disappointed; we are just at the beginning, and soon you will solve CartPole and many other much more interesting and challenging environments.

Extra Gym functionality – wrappers and monitors

What we have discussed so far covers two-thirds of the Gym core API and the essential functions required to start writing agents. The rest of the API you can live without, but it will make your life easier and your code cleaner. So, let's briefly cover the rest of the API.

Wrappers

Very frequently, you will want to extend the environment's functionality in some generic way. For example, imagine an environment gives you some observations, but you want to accumulate them in some buffer and provide to the agent the *N* last observations. This is a common scenario for dynamic computer games, when one single frame is just not enough to get the full information about the game state. Another example is when you want to be able to crop or preprocess an image's pixels to make it more convenient for the agent to digest, or if you want to normalize reward scores somehow. There are many such situations that have the same structure – you want to "wrap" the existing environment and add some extra logic for doing something. Gym provides a convenient framework for these situations – the `Wrapper` class.

The class structure is shown in the following diagram.
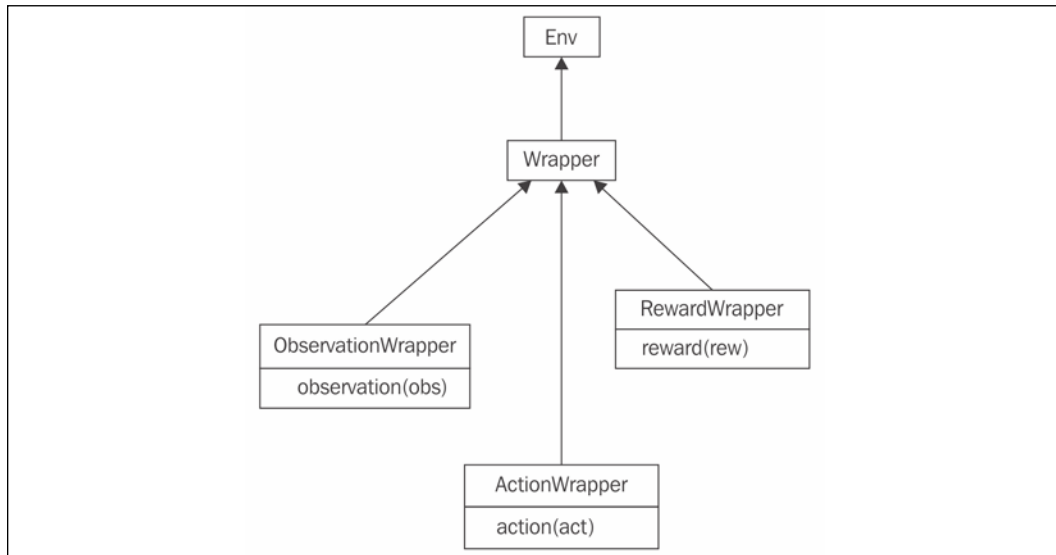
Figure 2.4: The hierarchy of Wrapper classes in Gym

The `Wrapper` class inherits the `Env` class. Its constructor accepts the only argument – the instance of the `Env` class to be "wrapped." To add extra functionality, you need to redefine the methods you want to extend, such as `step()` or `reset()`. The only requirement is to call the original method of the superclass.

To handle more specific requirements, such as a `Wrapper` class that wants to process only observations from the environment, or only actions, there are subclasses of `Wrapper` that allow the filtering of only a specific portion of information. They are as follows:

- `ObservationWrapper`: You need to redefine the observation (`obs`) method of the parent. The `obs` argument is an observation from the wrapped environment, and this method should return the observation that will be given to the agent.

- `RewardWrapper`: This exposes the reward (`rew`) method, which can modify the reward value given to the agent.

- `ActionWrapper`: You need to override the action (`act`) method, which can tweak the action passed to the wrapped environment by the agent.

To make it slightly more practical, let's imagine a situation where we want to intervene in the stream of actions sent by the agent and, with a probability of 10%, replace the current action with a random one. It might look like an unwise thing to do, but this simple trick is one of the most practical and powerful methods for solving the exploration/exploitation problem that I mentioned briefly in *Chapter 1*, *What Is Reinforcement Learning?*. By issuing the random actions, we make our agent explore the environment and from time to time drift away from the beaten track of its policy. This is an easy thing to do using the `ActionWrapper` class (a full example is in `Chapter02/03_random_action_wrapper.py`).

```python
import gym
from typing import TypeVar
import random


Action = TypeVar('Action')


class RandomActionWrapper(gym.ActionWrapper):
    def __init__(self, env, epsilon=0.1):
        super(RandomActionWrapper, self).__init__(env)
        self.epsilon = epsilon
```

Here, we initialized our wrapper by calling a parent's __init__ method and saving epsilon (the probability of a random action).

```python
    def action(self, action: Action) -> Action:
        if random.random() < self.epsilon:
            print("Random!")
            return self.env.action_space.sample()
        return action
```

This is a method that we need to override from a parent's class to tweak the agent's actions. Every time we roll the die, and with the probability of epsilon, we sample a random action from the action space and return it instead of the action the agent has sent to us. Note that using `action_space` and wrapper abstractions, we were able to write abstract code, which will work with any environment from Gym. Additionally, we must print the message every time we replace the action, just to verify that our wrapper is working. In the production code, of course, this won't be necessary.

```python
if __name__ == "__main__":
    env = RandomActionWrapper(gym.make("CartPole-v0"))
```

Now it's time to apply our wrapper. We will create a normal CartPole environment and pass it to our `Wrapper` constructor. From here on, we will use our wrapper as a normal `Env` instance, instead of the original CartPole. As the `Wrapper` class inherits the `Env` class and exposes the same interface, we can nest our wrappers in any combination we want. This is a powerful, elegant, and generic solution.

```
obs = env.reset()
total_reward = 0.0

while True:
    obs, reward, done, _ = env.step(0)
    total_reward += reward
    if done:
        break

print("Reward got: %.2f" % total_reward)
```

Here is almost the same code, except that every time we issue the same action, `0`, our agent is dull and does the same thing. By running the code, you should see that the wrapper is indeed working.

```
rl_book_samples/Chapter02$ python 03_random_actionwrapper.py
Random!
Random!
Random!
Random!
Reward got: 12.00
```

If you want, you can play with the epsilon parameter on the wrapper's creation and verify that randomness improves the agent's score on average.

We should move on now and look at another interesting gem that is hidden inside Gym: `Monitor`.

Monitor

Another class that you should be aware of is `Monitor`. It is implemented like `Wrapper` and can write information about your agent's performance in a file, with an optional video recording of your agent in action. Some time ago, it was possible to upload the result of the `Monitor` class' recording to the `https://gym.openai.com` website and see your agent's position in comparison to other people's results (see the following screenshot), but, unfortunately, at the end of August 2017, OpenAI decided to shut down this upload functionality and froze all the results. There are several alternatives to the original website, but they are not ready yet. I hope this situation will be resolved soon, but at the time of writing, it is not possible to check your results against those of others.

Just to give you an idea of how the Gym web interface looked, here is the CartPole environment leaderboard:

CartPole-v0

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

CartPole-v0 defines "solving" as getting average reward of 195.0 over 100 consecutive trials.

This environment corresponds to the version of the cart-pole problem described by Barto, Sutton, and Anderson [Barto83].

Episode 0

karpathy's algorithm, Took 211 episodes to solve the environment. 195.27 ± 1.57 a year ago

CartPole-v0 Evaluations

ALGORITHM	EPISODES BEFORE SOLVE	SUBMITTED
n1try's algorithm writeup	85.0	11 days ago
mbalunovic's algorithm writeup	306.0	11 days ago
ruippeixotog's algorithm writeup	933.0	12 days ago
ruippeixotog's algorithm writeup	961.0	13 days ago

Figure 2.5: The Gym web interface with CartPole submissions

Every submission in the web interface had details about training dynamics. For example, the following is my solution for one of Doom's mini-games:

Figure 2.6: Submission dynamics for the DoomDefendLine environment

Despite this, `Monitor` is still useful, as you can take a look at your agent's life inside the environment. So, here is how we add `Monitor` to our random CartPole agent, which is the only difference (the entire code is in `Chapter02/04_cartpole_random_monitor.py`).

```
if __name__ == "__main__":
    env = gym.make("CartPole-v0")
    env = gym.wrappers.Monitor(env, "recording")
```

The second argument that we pass to `Monitor` is the name of the directory that it will write the results to. This directory shouldn't exist, otherwise your program will fail with an exception (to overcome this, you could either remove the existing directory or pass the `force=True` argument to the `Monitor` class' constructor).

The `Monitor` class requires the `FFmpeg` utility to be present on the system, which is used to convert captured observations into an output video file. This utility must be available, otherwise `Monitor` will raise an exception. The easiest way to install `FFmpeg` is using your system's package manager, which is OS distribution-specific.

To start this example, one of these three extra prerequisites should be met:

- The code should be run in an X11 session with the **OpenGL extension** (**GLX**)
- The code should be started in an `Xvfb` virtual display
- You can use X11 forwarding in an SSH connection

The reason for this is video recording, which is done by taking screenshots of the window drawn by the environment. Some of the environment uses OpenGL to draw its picture, so the graphical mode with OpenGL needs to be present. This could be a problem for a virtual machine in the cloud, which physically doesn't have a monitor and graphical interface running. To overcome this, there is a special "virtual" graphical display, called **Xvfb** (**X11 virtual framebuffer**), which basically starts a virtual graphical display on the server and forces the program to draw inside it. This would be enough to make `Monitor` happily create the desired videos.

To start your program in the Xvfb environment, you need to have it installed on your machine (this usually requires installing the `xvfb` package) and run the special script, `xvfb-run`:

```
$ xvfb-run -s "-screen 0 640x480x24" python 04_cartpole_random_monitor.py
[2017-09-22 12:22:23,446] Making new env: CartPole-v0
[2017-09-22 12:22:23,451] Creating monitor directory recording
[2017-09-22 12:22:23,570] Starting new video recorder writing to
recording/openaigym.video.0.31179.video000000.mp4
Episode done in 14 steps, total reward 14.00
[2017-09-22 12:22:26,290] Finished writing results. You can upload them
to the scoreboard via gym.upload('recording')
```

As you may see from the preceding log, the video has been written successfully, so you can peek inside one of your agent's sections by playing it.

Another way to record your agent's actions is to use SSH X11 forwarding, which uses the SSH ability to tunnel X11 communications between the X11 client (Python code that wants to display some graphical information) and X11 server (software that knows how to display this information and has access to your physical display).

In X11 architecture, the client and the server are separated and can work on different machines. To use this approach, you need the following:

1. An X11 server running on your local machine. Linux comes with an X11 server as a standard component (all desktop environments use X11). On a Windows machine, you can set up third-party X11 implementations, such as open source VcXsrv (available in `https://sourceforge.net/projects/vcxsrv/`).

2. The ability to log into your remote machine via SSH, passing the `-X` command-line option: `ssh -X servername`. This enables X11 tunneling and allows all processes started in this session to use your local display for graphics output.

Then, you can start a program that uses the `Monitor` class and it will display the agent's actions, capturing the images in a video file.

Summary

You have started to learn about the practical side of RL! In this chapter, we installed OpenAI Gym, with its tons of environments to play with. We studied its basic API and created a randomly behaving agent.

You also learned how to extend the functionality of existing environments in a modular way and became familiar with a way to record our agent's activity using the `Monitor` class. This will be heavily used in the upcoming chapters.

In the next chapter, we will do a quick DL recap using PyTorch, which is a favorite library among DL researchers. Stay tuned.

Deep Learning with PyTorch

3

In the previous chapter, you became familiar with open source libraries, which provided you with a collection of reinforcement learning (RL) environments. However, recent developments in RL, and especially its combination with deep learning (DL), now make it possible to solve much more challenging problems than ever before. This is partly due to the development of DL methods and tools. This chapter is dedicated to one such tool, PyTorch, which enables us to implement complex DL models with just a bunch of lines of Python code.

The chapter doesn't pretend to be a complete DL manual, as the field is very wide and dynamic; however, we will cover:

- The PyTorch library specifics and implementation details (assuming that you are already familiar with DL fundamentals)
- Higher-level libraries on top of PyTorch, with the aim of simplifying common DL problems
- The library PyTorch ignite, which will be used in some examples

Compatibility note

All of the examples in this chapter were updated for the latest PyTorch 1.3.0, which has some minor changes in comparison to 0.4.0, which was used in the first edition of this book. If you are using the old PyTorch, consider upgrading. Throughout this chapter, we will discuss the differences that are present in the latest version.

Tensors

A tensor is the fundamental building block of all DL toolkits. The name sounds rather mystical, but the underlying idea is that a tensor is a multi-dimensional array. Using the analogy of school math, one single number is like a point, which is zero-dimensional, while a vector is one-dimensional like a line segment, and a matrix is a two-dimensional object. Three-dimensional number collections can be represented by a parallelepiped of numbers, but they don't have a separate name in the same way as a *matrix*. We can keep the term "tensor" for collections of higher dimensions.

Another thing to note about tensors used in DL is that they are only partially related to tensors used in *tensor calculus* or *tensor algebra*. In DL, a tensor is any multi-dimensional array, but in mathematics, a tensor is a mapping between vector spaces, which might be represented as a multi-dimensional array in some cases, but has much more semantical payload behind it. Mathematicians usually frown at anybody who uses well-established mathematical terms to name different things, so be warned!
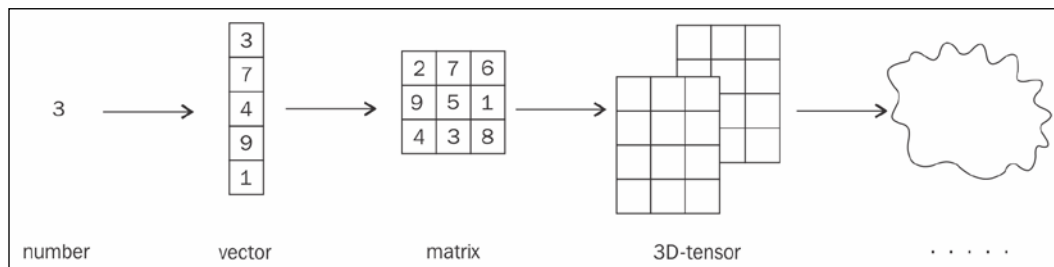
Figure 3.1: Going from a single number to an *n*-dimensional tensor

The creation of tensors

If you are familiar with the NumPy library (and you should be), then you already know that its central purpose is the handling of multi-dimensional arrays in a generic way. In NumPy, such arrays aren't called tensors, but they are, in fact, tensors. Tensors are used very widely in scientific computations as generic storage for data. For example, a color image could be encoded as a 3D tensor with the dimensions of width, height, and color plane.

Apart from dimensions, a tensor is characterized by the type of its elements. There are eight types supported by PyTorch: three float types (16-bit, 32-bit, and 64-bit) and five integer types (8-bit signed, 8-bit unsigned, 16-bit, 32-bit, and 64-bit). Tensors of different types are represented by different classes, with the most commonly used being `torch.FloatTensor` (corresponding to a 32-bit float), `torch.ByteTensor` (an 8-bit unsigned integer), and `torch.LongTensor` (a 64-bit signed integer). The rest can be found in the PyTorch documentation.

There are three ways to create a tensor in PyTorch:

- By calling a constructor of the required type.
- By converting a NumPy array or a Python list into a tensor. In this case, the type will be taken from the array's type.
- By asking PyTorch to create a tensor with specific data for you. For example, you can use the `torch.zeros()` function to create a tensor filled with zero values.

To give you examples of these methods, let's look at a simple session:

```
>>> import torch
>>> import numpy as np
>>> a = torch.FloatTensor(3, 2)
>>> a
tensor([[4.1521e+09,  4.5796e-41],
        [ 1.9949e-20, 3.0774e-41],
        [ 4.4842e-44, 0.0000e+00]])
```

Here, we imported both PyTorch and NumPy and created an uninitialized tensor of size 3×2. By default, PyTorch allocates memory for the tensor, but doesn't initialize it with anything. To clear the tensor's content, we need to use its operation:

```
>>> a.zero_()
tensor([[ 0., 0.],
        [ 0., 0.],
        [ 0., 0.]])
```

There are two types of operation for tensors: inplace and functional. Inplace operations have an underscore appended to their name and operate on the tensor's content. After this, the object itself is returned. The functional equivalent creates a *copy* of the tensor with the performed modification, leaving the original tensor untouched. Inplace operations are usually more efficient from a performance and memory point of view.

Another way to create a tensor by its constructor is to provide a Python iterable (for example, a list or tuple), which will be used as the contents of the newly created tensor:

```
>>> torch.FloatTensor([[1,2,3],[3,2,1]])
tensor([[ 1., 2., 3.],
        [ 3., 2., 1.]])
```

Here, we are creating the same tensor with zeros using NumPy:

```
>>> n = np.zeros(shape=(3, 2))
```

```
>>> n
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
>>> b = torch.tensor(n)
>>> b
tensor([[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]], dtype=torch.float64)
```

The `torch.tensor` method accepts the NumPy array as an argument and creates a tensor of appropriate shape from it. In the preceding example, we created a NumPy array initialized by zeros, which created a double (64-bit float) array by default. So, the resulting tensor has the `DoubleTensor` type (which is shown in the preceding example with the `dtype` value). Usually, in DL, double precision is not required and it adds an extra memory and performance overhead. Common practice is to use the 32-bit float type, or even the 16-bit float type, which is more than enough. To create such a tensor, you need to specify explicitly the type of NumPy array:

```
>>> n = np.zeros(shape=(3, 2), dtype=np.float32)
>>> torch.tensor(n)
tensor([[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]])
```

As an option, the type of the desired tensor could be provided to the `torch.tensor` function in the `dtype` argument. However, be careful, since this argument expects to get a PyTorch type specification and not the NumPy one. PyTorch types are kept in the `torch` package, for example, `torch.float32`, `torch.uint8`.

```
>>> n = np.zeros(shape=(3,2))
>>> torch.tensor(n, dtype=torch.float32)
tensor([[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]])
```
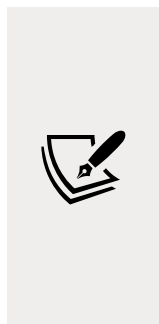
Compatibility note

The `torch.tensor()` method and explicit PyTorch type specification were added in the 0.4.0 release, and this is a step toward the simplification of tensor creation. In previous versions, the `torch.from_numpy()` function was a recommended way to convert NumPy arrays, but it had issues with handling the combination of the Python list and NumPy arrays. This `from_numpy()` function is still present for backward compatibility, but it is deprecated in favor of the more flexible `torch.tensor()` method.

Scalar tensors

Since the 0.4.0 release, PyTorch has supported zero-dimensional tensors that correspond to scalar values (on the left of *Figure 1*). Such tensors can be the result of some operations, such as summing all values in a tensor. Previously, such cases were handled by the creation of a one-dimensional (vector) tensor with a single dimension equal to one.

This solution worked, but it wasn't very simple, as extra indexation was needed to access the value. Now, zero-dimensional tensors are natively supported and returned by the appropriate functions, and they can be created by the `torch.tensor()` function. For accessing the actual Python value of such a tensor, there is the special `item()` method:

```
>>> a = torch.tensor([1,2,3])
>>> a
tensor([ 1, 2, 3])
>>> s = a.sum()
>>> s
tensor(6)
>>> s.item()
6
>>> torch.tensor(1)
tensor(1)
```

Tensor operations

There are lots of operations that you can perform on tensors, and there are too many to list them all. Usually, it's enough to search in the PyTorch documentation at `http://pytorch.org/docs/`. I need to mention that besides the inplace and functional variants that we already discussed (that is, with and without an underscore, like `abs()` and `abs_()`), there are two places to look for operations: the `torch` package and the tensor class. In the first case, the function usually accepts the tensor as an argument. In the second, it operates on the called tensor.

Most of the time, tensor operations are trying to correspond to their NumPy equivalent, so if there is some not-very-specialized function in NumPy, then there is a good chance that PyTorch will also have it. Examples are `torch.stack()`, `torch.transpose()`, and `torch.cat()`.

GPU tensors

PyTorch transparently supports CUDA GPUs, which means that all operations have two versions—CPU and GPU—that are automatically selected. The decision is made based on the type of tensors that you are operating on.

Every tensor type that I mentioned is for CPU and has its GPU equivalent. The only difference is that GPU tensors reside in the `torch.cuda` package, instead of just `torch`. For example, `torch.FloatTensor` is a 32-bit float tensor that resides in CPU memory, but `torch.cuda.FloatTensor` is its GPU counterpart.

To convert from CPU to GPU, there is a tensor method, `to(device)`, that creates a copy of the tensor to a specified device (this could be CPU or GPU). If the tensor is already on the device, nothing happens and the original tensor will be returned. The device type can be specified in different ways. First of all, you can just pass a string name of the device, which is `"cpu"` for CPU memory or `"cuda"` for GPU. A GPU device could have an optional device index specified after the colon; for example, the second GPU card in the system could be addressed by `"cuda:1"` (index is zero-based).

Another slightly more efficient way to specify a device in the `to()` method is by using the `torch.device` class, which accepts the device name and optional index. To access the device that your tensor is currently residing in, it has a `device` property.

```
>>> a = torch.FloatTensor([2,3])
>>> a
tensor([ 2., 3.])
>>> ca = a.to('cuda'); ca
tensor([ 2.,3.], device='cuda:0')
```

Here, we created a tensor on CPU, then copied it to GPU memory. Both copies can be used in computations and all GPU-specific machinery is transparent to the user:

```
>>> a + 1
tensor([ 3., 4.])
>>> ca + 1
tensor([ 3., 4.], device='cuda:0')
>>> ca.device
device(type='cuda', index=0)
```
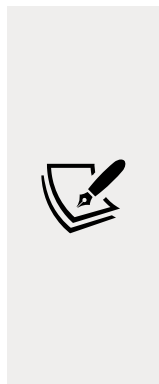
Compatibility note

The `to()` method and `torch.device` class were introduced in 0.4.0. In previous versions, copying between CPU and GPU was performed by separate tensor methods, `cpu()` and `cuda()`, respectively, which required adding the extra lines of code to explicitly convert tensors into their CUDA versions. In the latest version, you can create a desired `torch.device` object at the beginning of the program and use `to(device)` on every tensor that you're creating. The old methods in the tensor, `cpu()` and `cuda()`, are still present and might be handy if you want to ensure that a tensor is in CPU or GPU memory regardless of its original location.

Gradients

Even with transparent GPU support, all of this dancing with tensors isn't worth bothering with without one "killer feature"—the automatic computation of gradients. This functionality was originally implemented in the Caffe toolkit and then became the de facto standard in DL libraries.

Computing gradients manually was extremely painful to implement and debug, even for the simplest neural network (NN). You had to calculate derivatives for all your functions, apply the chain rule, and then implement the result of the calculations, praying that everything was done right. This could be a very useful exercise for understanding the nuts and bolts of DL, but it wasn't something that you wanted to repeat over and over again by experimenting with different NN architectures.

Luckily, those days have gone now, much like programming your hardware using a soldering iron and vacuum tubes! Now, defining an NN of hundreds of layers requires nothing more than assembling it from predefined building blocks or, in the extreme case of you doing something fancy, defining the transformation expression manually.

All gradients will be carefully calculated for you, backpropagated, and applied to the network. To be able to achieve this, you need to define your network architecture in terms of the DL library used, which can be different in the details, but generally must be the same—you must define the order in which your network will transform inputs to outputs.
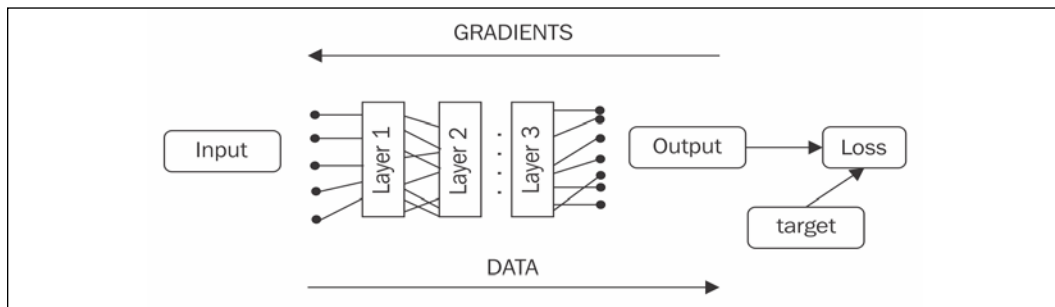
Figure 3.2: Data and gradients flowing through the NN

What can make a fundamental difference is *how* your gradients are calculated. There are two approaches:

- **Static graph**: In this method, you need to define your calculations in advance and it won't be possible to change them later. The graph will be processed and optimized by the DL library before any computation is made. This model is implemented in TensorFlow (<2), Theano, and many other DL toolkits.

- **Dynamic graph**: You don't need to define your graph in advance exactly as it will be executed; you just need to execute operations that you want to use for data transformation on your actual data. During this, the library will record the order of the operations performed, and when you ask it to calculate gradients, it will unroll its history of operations, accumulating the gradients of the network parameters. This method is also called notebook gradients and it is implemented in PyTorch, Chainer, and some others.

Both methods have their strengths and weaknesses. For example, static graph is usually faster, as all computations can be moved to the GPU, minimizing the data transfer overhead. Additionally, in static graph, the library has much more freedom in optimizing the order that computations are performed in or even removing parts of the graph.

On the other hand, although dynamic graph has a higher computation overhead, it gives a developer much more freedom. For example, they can say, "For this piece of data, I can apply this network two times, and for this piece of data, I'll use a completely different model with gradients clipped by the batch mean." Another very appealing strength of the dynamic graph model is that it allows you to express your transformation more naturally and in a more "Pythonic" way. In the end, it's just a Python library with a bunch of functions, so just call them and let the library do the magic.

> **Compatibility note**
>
> Since version 1.0, PyTorch has supported the **just-in-time** (**JIT**) compiler, which takes PyTorch code and exports it into so-called TorchScript. This is intermediate representation that can be executed faster and without Python dependency in a production environment.

Tensors and gradients

PyTorch tensors have a built-in gradient calculation and tracking machinery, so all you need to do is convert the data into tensors and perform computations using the tensor methods and functions provided by `torch`. Of course, if you need to access underlying low-level details, you always can, but most of the time, PyTorch does what you're expecting.

There are several attributes related to gradients that every tensor has:

- `grad`: A property that holds a tensor of the same shape containing computed gradients.

- `is_leaf`: `True` if this tensor was constructed by the user and `False` if the object is a result of function transformation.

- `requires_grad`: `True` if this tensor requires gradients to be calculated. This property is inherited from leaf tensors, which get this value from the tensor construction step (`torch.zeros()` or `torch.tensor()` and so on). By default, the constructor has `requires_grad=False`, so if you want gradients to be calculated for your tensor, then you need to explicitly say so.

To make all of this gradient-leaf machinery clearer, let's consider this session:

```
>>> v1 = torch.tensor([1.0, 1.0], requires_grad=True)
>>> v2 = torch.tensor([2.0, 2.0])
```

In the preceding code, we created two tensors. The first requires gradients to be calculated and the second doesn't.

```
>>> v_sum = v1 + v2
>>> v_res = (v_sum*2).sum()
>>> v_res

tensor(12., grad_fn=<SumBackward0>)
```

So, now we have added both vectors element-wise (which is vector `[3, 3]`), doubled every element, and summed them together. The result is a zero-dimensional tensor with the value `12`. Okay, so this is simple math so far. Now let's look at the underlying graph that our expressions created:
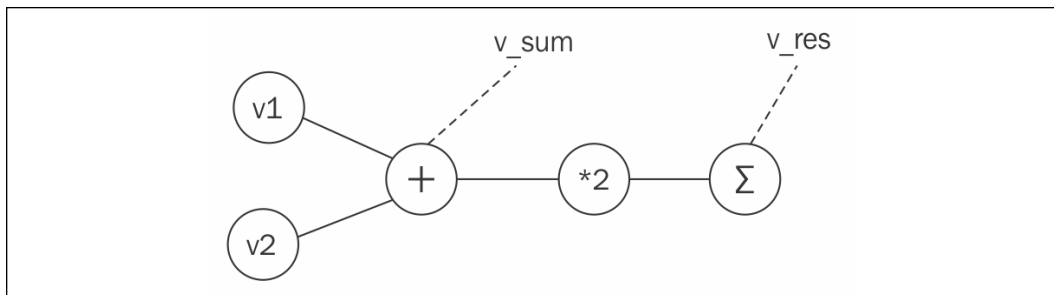
Figure 3.3: Graph representation of the expression

If we check the attributes of our tensors, then we will find that **v1** and **v2** are the only leaf nodes and every variable, except **v2**, requires gradients to be calculated:

```
>>> v1.is_leaf, v2.is_leaf
(True, True)
>>> v_sum.is_leaf, v_res.is_leaf
(False, False)
>>> v1.requires_grad
```

```
True
>>> v2.requires_grad
False
>>> v_sum.requires_grad
True
>>> v_res.requires_grad
True
```

Now, let's tell PyTorch to calculate the gradients of our graph:

```
>>> v_res.backward()
>>> v1.grad

tensor([ 2., 2.])
```

By calling the `backward` function, we asked PyTorch to calculate the numerical derivative of the `v_res` variable with respect to any variable that our graph has. In other words, what influence do small changes to the `v_res` variable have on the rest of the graph? In our particular example, the value of two in the gradients of `v1` means that by increasing any element of `v1` by one, the resulting value of `v_res` will grow by two.

As mentioned, PyTorch calculates gradients only for leaf tensors with `requires_grad=True`. Indeed, if we try to check the gradients of `v2`, we get nothing:

```
>>> v2.grad
```

The reason for that is efficiency in terms of computations and memory. In real life, our network can have millions of optimized parameters, with hundreds of intermediate operations performed on them. During gradient descent optimization, we are not interested in gradients of any intermediate matrix multiplication; the only things we want to adjust in the model are gradients of loss with respect to model parameters (weights). Of course, if you want to calculate the gradients of input data (it could be useful if you want to generate some adversarial examples to fool the existing NN or adjust pretrained word embeddings), then you can easily do so by passing `requires_grad=True` on tensor creation.

Basically, you now have everything needed to implement your own NN optimizer. The rest of this chapter is about extra, convenient functions, which will provide you with higher-level building blocks of NN architectures, popular optimization algorithms, and common loss functions. However, don't forget that you can easily reimplement all of these bells and whistles in any way that you like. This is why PyTorch is so popular among DL researchers—for its elegance and flexibility.

Compatibility note

Support of gradients calculation in tensors is one of the major changes in PyTorch 0.4.0. In previous versions, graph tracking and gradients accumulation were done in a separate, very thin class, `Variable`. This worked as a wrapper around the tensor and automatically saved the history of computations in order to be able to backpropagate. This class is still present in 0.4.0, but it is deprecated and will go away soon, so new code should avoid using it. From my perspective, this change is great, as the `Variable` logic was really thin, but it still required extra code and the developer's attention to wrap and unwrap tensors. Now, gradients are a built-in tensor property, which makes the API much cleaner.

NN building blocks

In the `torch.nn` package, you will find tons of predefined classes providing you with the basic functionality blocks. All of them are designed with practice in mind (for example, they support mini-batches, they have sane default values, and the weights are properly initialized). All modules follow the convention of *callable*, which means that the instance of any class can act as a function when applied to its arguments. For example, the `Linear` class implements a feed-forward layer with optional bias:

```
>>> import torch.nn as nn
>>> l = nn.Linear(2, 5)
>>> v = torch.FloatTensor([1, 2])
>>> l(v)
tensor([ 1.0532,  0.6573, -0.3134,  1.1104, -0.4065], grad_
fn=<AddBackward0>)
```

Here, we created a randomly initialized feed-forward layer, with two inputs and five outputs, and applied it to our float tensor. All classes in the `torch.nn` packages inherit from the `nn.Module` base class, which you can use to implement your own higher-level NN blocks. You will see how you can do this in the next section, but, for now, let's look at useful methods that all `nn.Module` children provide. They are as follows:

- `parameters()`: This function returns an iterator of all variables that require gradient computation (that is, module weights).
- `zero_grad()`: This function initializes all gradients of all parameters to zero.
- `to(device)`: This function moves all module parameters to a given device (CPU or GPU).

- `state_dict()`: This function returns the dictionary with all module parameters and is useful for model serialization.
- `load_state_dict()`: This function initializes the module with the state dictionary.

The whole list of available classes can be found in the documentation at `http://pytorch.org/docs`.

Now, I should mention one very convenient class that allows you to combine other layers into the pipe: `Sequential`. The best way to demonstrate `Sequential` is through an example:

```
>>> s = nn.Sequential(
... nn.Linear(2, 5),
... nn.ReLU(),
... nn.Linear(5, 20),
... nn.ReLU(),
... nn.Linear(20, 10),
... nn.Dropout(p=0.3),
... nn.Softmax(dim=1))
>>> s
Sequential(
  (0): Linear(in_features=2, out_features=5, bias=True)
  (1): ReLU()
  (2): Linear(in_features=5, out_features=20, bias=True)
  (3): ReLU()
  (4): Linear(in_features=20, out_features=10, bias=True)
  (5): Dropout(p=0.3)
  (6): Softmax()
)
```

Here, we defined a three-layer NN with softmax on output, applied along dimension 1 (dimension 0 is batch samples), **rectified linear unit (ReLU)** nonlinearities, and dropout. Let's push something through it:

```
>>> s(torch.FloatTensor([[1,2]]))
tensor([[0.1115, 0.0702, 0.1115, 0.0870, 0.1115, 0.1115, 0.0908,
0.0974, 0.0974, 0.1115]], grad_fn=<SoftmaxBackward>)
```

So, our mini-batch is one example successfully traversed through the network!

Custom layers

In the previous section, I briefly mentioned the `nn.Module` class as a base parent for all NN building blocks exposed by PyTorch. It's not just a unifying parent for the existing layers—it's much more than that. By subclassing the `nn.Module` class, you can create your own building blocks, which can be stacked together, reused later, and integrated into the PyTorch framework flawlessly.

At its core, the `nn.Module` provides quite rich functionality to its children:

- It tracks all submodules that the current module includes. For example, your building block can have two feed-forward layers used somehow to perform the block's transformation.

- It provides functions to deal with all parameters of the registered submodules. You can obtain a full list of the module's parameters (`parameters()` method), zero its gradients (`zero_grads()` method), move to CPU or GPU (`to(device)` method), serialize and deserialize the module (`state_dict()` and `load_state_dict()`), and even perform generic transformations using your own callable (`apply()` method).

- It establishes the convention of `Module` application to data. Every module needs to perform its data transformation in the `forward()` method by overriding it.

- There are some more functions, such as the ability to register a hook function to tweak module transformation or gradients flow, but they are more for advanced use cases.

These functionalities allow us to nest our submodels into higher-level models in a unified way, which is extremely useful when dealing with complexity. It could be a simple one-layer linear transformation or a 1001-layer **residual NN** (**ResNet**) monster, but if they follow the conventions of `nn.Module`, then both of them could be handled in the same way. This is very handy for code simplicity and reusability.

To make our life simpler, when following the preceding convention, PyTorch authors simplified the creation of modules through careful design and a good dose of Python magic. So, to create a custom module, we usually have to do only two things—register submodules and implement the `forward()` method.

Let's look at how this can be done for our `Sequential` example from the previous section, but in a more generic and reusable way (full sample is `Chapter03/01_ modules.py`):

```
class OurModule(nn.Module):
    def __init__(self, num_inputs, num_classes, dropout_prob=0.3):
        super(OurModule, self).__init__()
        self.pipe = nn.Sequential(
            nn.Linear(num_inputs, 5),
            nn.ReLU(),
            nn.Linear(5, 20),
            nn.ReLU(),
            nn.Linear(20, num_classes),
            nn.Dropout(p=dropout_prob),
            nn.Softmax(dim=1)
        )
```

This is our module class that inherits `nn.Module`. In the constructor, we pass three parameters: the input size, the output size, and the optional dropout probability. The first thing we need to do is call the parent's constructor to let it initialize itself.

In the second step, we need to create an already familiar `nn.Sequential` with a bunch of layers and assign it to our class field named `pipe`. By assigning a `Sequential` instance to our field, we will automatically register this module (`nn. Sequential` inherits from `nn.Module`, as does everything in the `nn` package). To register it, we don't need to call anything, we just need to assign our submodules to fields. After the constructor finishes, all those fields will be registered automatically (if you really want to, there is a function in `nn.Module` to register submodules).

```
    def forward(self, x):
        return self.pipe(x)
```

Here, we must override the `forward` function with our implementation of data transformation. As our module is a very simple wrapper around other layers, we just need to ask them to transform the data. Note that to apply a module to the data, we need to call the module as callable (that is, pretend that the module instance is a function and call it with the arguments) and *not* use the `forward()` function of the `nn.Module` class. This is because `nn.Module` overrides the `__call__()` method, which is being used when we treat an instance as callable. This method does some `nn.Module` magic stuff and calls our `forward()` method. If we call `forward()` directly, we will intervene with the `nn.Module` duty, which can give wrong results.

So, that's what we need to do to define our own module. Now, let's use it:

```python
if __name__ == "__main__":
    net = OurModule(num_inputs=2, num_classes=3)
    v = torch.FloatTensor([[2, 3]])
    out = net(v)
    print(net)
    print(out)
```

We create our module, providing it with the desired number of inputs and outputs, then we create a tensor and ask our module to transform it, following the same convention of using it as callable. After that, we print our network's structure (`nn.Module` overrides `__str__()` and `__repr__()`) to represent the inner structure in a nice way. The last thing we show is the result of the network's transformation.

The output of our code should look like this:

```
rl_book_samples/Chapter03$ python 01_modules.py
OurModule(
  (pipe): Sequential(
    (0): Linear(in_features=2, out_features=5, bias=True)
    (1): ReLU()
    (2): Linear(in_features=5, out_features=20, bias=True)
    (3): ReLU()
    (4): Linear(in_features=20, out_features=3, bias=True)
    (5): Dropout(p=0.3, inplace=False)
    (6): Softmax(dim=1)
  )
)
tensor([[0.5436, 0.3243, 0.1322]], grad_fn=<SoftmaxBackward>)
Cuda's availability is True
Data from cuda: tensor([[0.5436, 0.3243, 0.1322]], device='cuda:0', grad_
fn=<CopyBackwards>)
```

Of course, everything that was said about the dynamic nature of PyTorch is still true. The `forward()` method is called for every batch of data, so if you want to do some complex transformations based on the data you need to process, like hierarchical softmax or a random choice of network to apply, then nothing can stop you from doing so. The count of arguments to your module is also not limited by one parameter. So, if you want, you can write a module with multiple required parameters and dozens of optional arguments, and it will be fine.

Next, we need to get familiar with two important pieces of the PyTorch library that will simplify our lives: loss functions and optimizers.

The final glue – loss functions and optimizers

The network that transforms input data into output is not the only thing we need for training. We need to define our learning objective, which is to have a function that accepts two arguments—the network's output and the desired output. Its responsibility is to return to us a single number—how close the network's prediction is from the desired result. This function is called the **loss function**, and its output is the **loss value**. Using the loss value, we calculate gradients of network parameters and adjust them to decrease this loss value, which pushes our model to better results in the future. Both the loss function and the method of tweaking a network's parameters by gradient are so common and exist in so many forms that both of them form a significant part of the PyTorch library. Let's start with loss functions.

Loss functions

Loss functions reside in the `nn` package and are implemented as an `nn.Module` subclass. Usually, they accept two arguments: output from the network (prediction) and desired output (ground-truth data, which is also called the label of the data sample). At the time of writing, PyTorch 1.3.0 contains 20 different loss functions and, of course, nothing stops you from writing any function you want to optimize in the explicit form.

The most commonly used standard loss functions are:

- `nn.MSELoss`: The mean square error between arguments, which is the standard loss for regression problems.

- `nn.BCELoss` and `nn.BCEWithLogits`: Binary cross-entropy loss. The first version expects a single probability value (usually it's the output of the `Sigmoid` layer), while the second version assumes raw scores as input and applies `Sigmoid` itself. The second way is usually more numerically stable and efficient. These losses (as their names suggest) are frequently used in binary classification problems.

- `nn.CrossEntropyLoss` and `nn.NLLLoss`: Famous "maximum likelihood" criteria that are used in multi-class classification problems. The first version expects raw scores for each class and applies `LogSoftmax` internally, while the second expects to have log probabilities as the input.

There are other loss functions available and you are always free to write your own `Module` subclass to compare the output and target. Now, let's look at the second piece of the optimization process.

Optimizers

The responsibility of the basic optimizer is to take the gradients of model parameters and change these parameters in order to decrease the loss value. By decreasing the loss value, we are pushing our model toward the desired output, which can give us hope for better model performance in the future. Changing parameters may sound simple, but there are lots of details here and the optimizer procedure is still a hot research topic. In the `torch.optim` package, PyTorch provides lots of popular optimizer implementations and the most widely known are as follows:

- `SGD`: A vanilla stochastic gradient descent algorithm with an optional momentum extension
- `RMSprop`: An optimizer proposed by Geoffrey Hinton
- `Adagrad`: An adaptive gradients optimizer
- `Adam`: A quite successful and popular combination of both `RMSprop` and `Adagrad`

All optimizers expose the unified interface, which makes it easy to experiment with different optimization methods (sometimes the optimization method can really make a difference in convergence dynamics and the final result). On construction, you need to pass an iterable of tensors, which will be modified during the optimization process. The usual practice is to pass the result of the `params()` call of the upper-level `nn.Module` instance, which will return an iterable of all leaf tensors with gradients.

Now, let's discuss the common blueprint of a training loop.

```
for batch_x, batch_y in iterate_batches(data, batch_size=32):   #1
    batch_x_t = torch.tensor(batch_x)                           #2
    batch_y_t = torch.tensor(batch_y)                           #3
    out_t = net(batch_x_t)                                      #4
    loss_t = loss_function(out_t, batch_y_t).                   #5
    loss_t.backward()                                          #6
    optimizer.step()                                          #7
    optimizer.zero_grad()                                    #8
```

Usually, you iterate over your data over and over again (one iteration over a full set of examples is called an *epoch*). Data is usually too large to fit into CPU or GPU memory at once, so it is split into batches of equal size. Every batch includes data samples and target labels, and both of them have to be tensors (lines 2 and 3).

You pass data samples to your network (line 4) and feed its output and target labels to the loss function (line 5). The result of the loss function shows the "badness" of the network result relative to the target labels. As input to the network and the network's weights are tensors, all transformations of your network are nothing more than a graph of operations with intermediate tensor instances. The same is true for the loss function—its result is also a tensor of one single loss value.

Every tensor in this computation graph remembers its parent, so to calculate gradients for the whole network, all you need to do is call the `backward()` function on a loss function result (line 6). The result of this call will be the unrolling of the graph of the performed computations and the calculating of gradients for every leaf tensor with `require_grad=True`. Usually, such tensors are our model's parameters, such as the weights and biases of feed-forward networks, and convolution filters. Every time a gradient is calculated, it is accumulated in the `tensor.grad` field, so one tensor can participate in a transformation multiple times and its gradients will be properly summed together. For example, one single **recurrent neural network** (**RNN**) cell could be applied to multiple input items.

After the `loss.backward()` call is finished, we have the gradients accumulated, and now it's time for the optimizer to do its job—it takes all gradients from the parameters we have passed to it on construction and applies them. All this is done with the method `step()` (line 7).

The last, but not least, piece of the training loop is our responsibility to zero gradients of parameters. This can be done by calling `zero_grad()` on our network, but, for our convenience, the optimizer also exposes such a call, which does the same thing (line 8). Sometimes `zero_grad()` is placed at the beginning of the training loop, but it doesn't matter much.

The preceding scheme is a very flexible way to perform optimization and it can fulfill the requirements even in sophisticated research. For example, you can have two optimizers tweaking the options of different models on the same data (and this is a real-life scenario from **generative adversarial network** (**GAN**) training).

So, we are done with the essential functionality of PyTorch required to train NNs. This chapter ends with a practical medium-size example to demonstrate all the concepts covered, but before we get to it, we need to discuss one important topic that is essential for an NN practitioner—monitoring the learning process.

Monitoring with TensorBoard

If you have ever tried to train an NN on your own, then you will know how painful and uncertain it can be. I'm not talking about following the existing tutorials and demos, when all the hyperparameters are already tuned for you, but about taking some data and creating something from scratch. Even with modern DL high-level toolkits, where all best practices, such as proper weights initialization; optimizers' betas, gammas, and other options set to sane defaults; and tons of other stuff hidden under the hood, there are still lots of decisions that you can make, hence lots of things that could go wrong. As a result, your network almost never works from the first run and this is something that you should get used to.

Of course, with practice and experience, you will develop a strong intuition about the possible causes of problems, but intuition needs input data about what's going on inside your network. So, you need to be able to peek inside your training process somehow and observe its dynamics. Even small networks (such as tiny MNIST tutorial networks) could have hundreds of thousands of parameters with quite nonlinear training dynamics.

DL practitioners have developed a list of things that you should observe during your training, which usually includes the following:

- Loss value, which normally consists of several components like base loss and regularization losses. You should monitor both the total loss and the individual components over time.

- Results of validation on training and test datasets.

- Statistics about gradients and weights.

- Values produced by the network. For example, if you are solving a classification problem, you definitely want to measure the entropy of predicted class probabilities. In the case of a regression problem, raw predicted values can give tons of data about the training.

- Learning rates and other hyperparameters, if they are adjusted over time.

The list could be much longer and include domain-specific metrics, such as word embedding projections, audio samples, and images generated by GANs. You also may want to monitor values related to training speed, like how long an epoch takes, to see the effect of your optimizations or problems with hardware.

To make a long story short, you need a generic solution to track lots of values over time and represent them for analysis, preferably developed especially for DL (just imagine looking at such statistics in an Excel spreadsheet). Luckily, such tools exist and we will explore them next.

TensorBoard 101

When the first edition of this book was written, there wasn't too much choice for NN monitoring. As time has passed by and new people and companies have become involved with the pursuit of machine learning (ML) and DL, more new tools have appeared. In this book, we will still focus on the TensorBoard utility from TensorFlow, but you might consider trying other utilities around that.

From the first public version, TensorFlow included a special tool called TensorBoard, which was developed to solve the problem we are talking about—how to observe and analyze various NN characteristics over training. TensorBoard is a powerful, generic solution with a large community and it looks quite pretty:
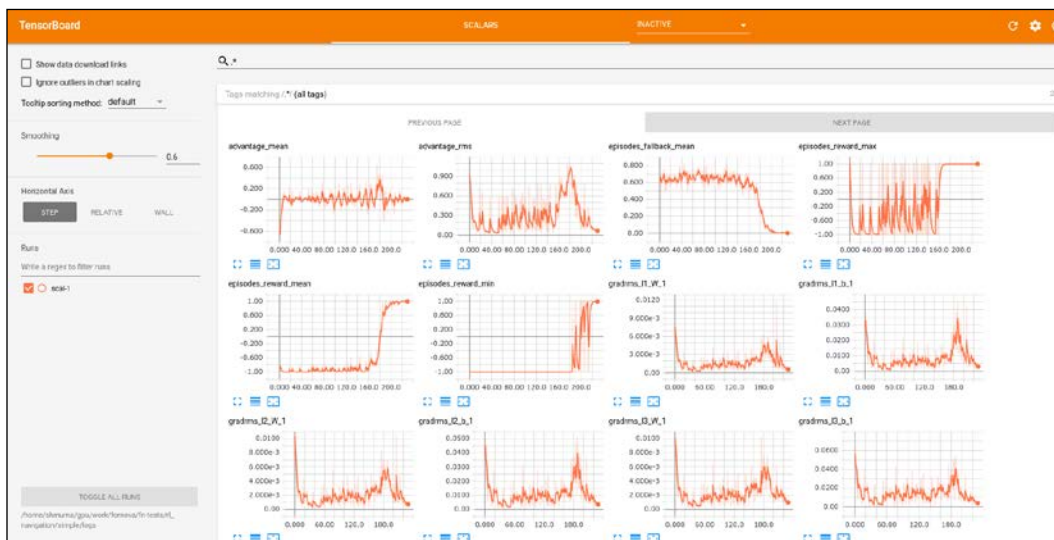
Figure 3.4: The TensorBoard web interface

From the architecture point of view, TensorBoard is a Python web service that you can start on your computer, passing it the directory where your training process will save values to be analyzed. Then, you point your browser to TensorBoard's port (usually 6006), and it shows you an interactive web interface with values updated in real time. It's nice and convenient, especially when your training is performed on a remote machine somewhere in the cloud.

Originally, TensorBoard was deployed as a part of TensorFlow, but recently, it has been moved to a separate project (it's still being maintained by Google) and it has its own package name. However, TensorBoard still uses the TensorFlow data format, so to be able to write training statistics from PyTorch optimization, you will need both the tensorboard and tensorflow packages installed.

In theory, this is all you need to start monitoring your networks, as the `tensorflow` package provides you with classes to write the data that TensorBoard will be able to read. However, it's not very practical, as those classes are very low level. To overcome this, there are several third-party open source libraries that provide a convenient high-level interface. One of my favorites, which is used in this book, is `tensorboardX` (https://github.com/lanpa/tensorboardX). It can be installed with `pip install tensorboardX`.

> **Compatibility note**
>
> In PyTorch 1.1, experimental support of the TensorBoard format was implemented, which makes it unnecessary to have `tensorboardX` installed (for details, check https://pytorch.org/docs/stable/tensorboard.html). But we will still use this third-party package, as PyTorch Ignite depends on it.

Plotting stuff

To give you an impression of how simple `tensorboardX` is, let's consider a small example that is not related to NNs, but is just about writing stuff into TensorBoard (the full example code is in `Chapter03/02_tensorboard.py`).

```
import math
from tensorboardX import SummaryWriter

if __name__ == "__main__":
    writer = SummaryWriter()
    funcs = {"sin": math.sin, "cos": math.cos, "tan": math.tan}
```

We import the required packages, create a writer of data, and define functions that we are going to visualize. By default, `SummaryWriter` will create a unique directory under the `runs` directory for every launch, to be able to compare different launches of training. The names of the new directory include the current date and time, and the hostname. To override this, you can pass the `log_dir` argument to `SummaryWriter`. You can also add a suffix to the name of the directory by passing a comment option, for example to capture different experiments' semantics, such as `dropout=0.3` or `strong_regularisation`.

```
        for angle in range(-360, 360):
            angle_rad = angle * math.pi / 180
            for name, fun in funcs.items():
                val = fun(angle_rad)
                writer.add_scalar(name, val, angle)

        writer.close()
```

Here, we loop over angle ranges in degrees, convert them into radians, and calculate our functions' values. Every value is added to the writer using the `add_scalar` function, which takes three arguments: the name of the parameter, its value, and the current iteration (which has to be an integer).

The last thing we need to do after the loop is close the writer. Note that the writer does a periodical flush (by default, every two minutes), so even in the case of a lengthy optimization process, you will still see your values.

The result of running this will be zero output on the console, but you will see a new directory created inside the `runs` directory with a single file. To look at the result, we need to start TensorBoard:

```
rl_book_samples/Chapter03$ tensorboard --logdir runs
TensorBoard 2.0.1 at http://127.0.0.1:6006/ (Press CTRL+C to quit)
```

If you are running TensorBoard on a remote server, you will need to add the `--bind_all` command-line option to make it accessible from the outside. Now you can open `http://localhost:6006` in your browser to see something like this:
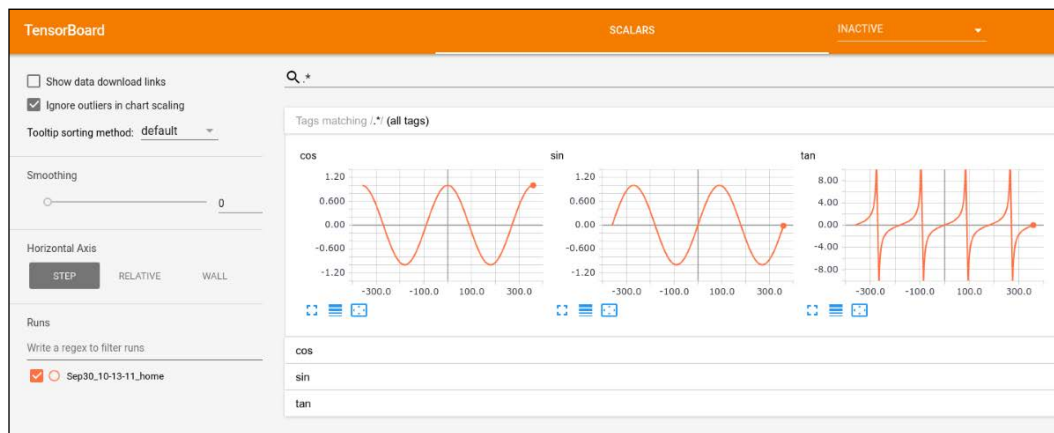
Figure 3.5: Plots produced by the example

The graphs are interactive, so you can hover over them with your mouse to see the actual values and select regions to zoom in and look at details. To zoom out, double-click inside the graph. If you run your program several times, then you will see several items in the **Runs** list on the left, which can be enabled and disabled in any combination, allowing you to compare the dynamics of several optimizations. TensorBoard allows you to analyze not only scalar values but also images, audio, text data, and embeddings, and it can even show you the structure of your network. Refer to the documentation of `tensorboardX` and `tensorboard` for all those features.

Now, it's time to combine everything you learned in this chapter and look at a real NN optimization problem using PyTorch.

Example – GAN on Atari images

Almost every book about DL uses the MNIST dataset to show you the power of DL, which, over the years, has made this dataset extremely boring, like a fruit fly for genetic researchers. To break this tradition, and add a bit more fun to the book, I've tried to avoid well-beaten paths and illustrate PyTorch using something different. I briefly referred to GANs earlier in the chapter. They were invented and popularized by *Ian Goodfellow*. In this example, we will train a GAN to generate screenshots of various Atari games.

The simplest GAN architecture is this: we have two networks and the first works as a "cheater" (it is also called the generator), and the other is a "detective" (another name is the discriminator). Both networks compete with each other—the generator tries to generate fake data, which will be hard for the discriminator to distinguish from your dataset, and the discriminator tries to detect the generated data samples. Over time, both networks improve their skills—the generator produces more and more realistic data samples, and the discriminator invents more sophisticated ways to distinguish the fake items.

Practical usage of GANs includes image quality improvement, realistic image generation, and feature learning. In our example, practical usefulness is almost zero, but it will be a good example of how clean and short PyTorch code can be for quite complex models.

So, let's get started. The whole example code is in the file `Chapter03/03_atari_gan.py`. Here, we will look at only significant pieces of code, without the `import` section and constants declaration:

```
class InputWrapper(gym.ObservationWrapper):
    def __init__(self, *args):
        super(InputWrapper, self).__init__(*args)
        assert isinstance(self.observation_space, gym.spaces.Box)
        old_space = self.observation_space
        self.observation_space = gym.spaces.Box(
            self.observation(old_space.low),
            self.observation(old_space.high),
            dtype=np.float32)

    def observation(self, observation):
        new_obs = cv2.resize(
            observation, (IMAGE_SIZE, IMAGE_SIZE))
```

```
                  # transform (210, 160, 3) -> (3, 210, 160)
                  new_obs = np.moveaxis(new_obs, 2, 0)
                  return new_obs.astype(np.float32)
```

This class is a wrapper around a Gym game, which includes several transformations:

- Resize the input image from 210×160 (the standard Atari resolution) to a square size 64×64

- Move the color plane of the image from the last position to the first, to meet the PyTorch convention of convolution layers that input a tensor with the shape of the channels, height, and width

- Cast the image from `bytes` to `float`

Then, we define two `nn.Module` classes: `Discriminator` and `Generator`. The first takes our scaled color image as input and, by applying five layers of convolutions, converts it into a single number passed through a `Sigmoid` nonlinearity. The output from `Sigmoid` is interpreted as the probability that `Discriminator` thinks our input image is from the real dataset.

`Generator` takes as input a vector of random numbers (latent vector) and using the "transposed convolution" operation (it is also known as `deconvolution`) converts this vector into a color image of the original resolution. We will not look at those classes here as they are lengthy and not very relevant to our example; you can find them in the complete example file.
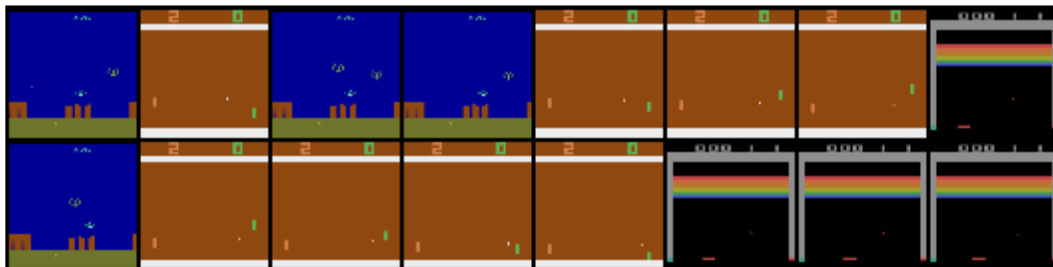
Figure 3.6: Sample screenshots from three Atari games

As input, we will use screenshots from several Atari games played simultaneously by a random agent. *Figure 3.6* is an example of what the input data looks like and it is generated by the following function:

```
def iterate_batches(envs, batch_size=BATCH_SIZE):
    batch = [e.reset() for e in envs]
    env_gen = iter(lambda: random.choice(envs), None)

    while True:
```

```
e = next(env_gen)
obs, reward, is_done, _ = e.step(e.action_space.sample())
if np.mean(obs) > 0.01:
    batch.append(obs)
if len(batch) == batch_size:
    # Normalising input between -1 to 1
    batch_np = np.array(batch, dtype=np.float32)
    batch_np *= 2.0 / 255.0 - 1.0
    yield torch.tensor(batch_np)
    batch.clear()
if is_done:
    e.reset()
```

This infinitely samples the environment from the provided array, issues random actions, and remembers observations in the `batch` list. When the batch becomes of the required size, we normalize the image, convert it to a tensor, and `yield` from the generator. The check for the non-zero mean of the observation is required due to a bug in one of the games to prevent the flickering of an image.

Now, let's look at our main function, which prepares models and runs the training loop.

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--cuda", default=False, action='store_true',
        help="Enable cuda computation")
    args = parser.parse_args()

    device = torch.device("cuda" if args.cuda else "cpu")
    envs = [
        InputWrapper(gym.make(name))
        for name in ('Breakout-v0', 'AirRaid-v0', 'Pong-v0')
    ]
    input_shape = envs[0].observation_space.shape
```

Here, we process the command-line arguments (which could be only one optional argument, `--cuda`, enabling the GPU computation mode) and create our environment pool with a wrapper applied. This environment array will be passed to the `iterate_batches` function to generate training data.

```
net_discr = Discriminator(input_shape=input_shape).to(device)
net_gener = Generator(output_shape=input_shape).to(device)

objective = nn.BCELoss()
```

```
gen_optimizer = optim.Adam(
    params=net_gener.parameters(), lr=LEARNING_RATE,
    betas=(0.5, 0.999))
dis_optimizer = optim.Adam(
    params=net_discr.parameters(), lr=LEARNING_RATE,
    betas=(0.5, 0.999))
writer = SummaryWriter()
```

In this piece, we create our classes—a summary writer, both networks, a loss function, and two optimizers. Why two? It's because that's the way that GANs get trained: to train the discriminator, we need to show it both real and fake data samples with appropriate labels (1 for real and 0 for fake). During this pass, we update only the discriminator's parameters.

After that, we pass both real and fake samples through the discriminator again, but this time, the labels are 1s for all samples and we update only the generator's weights. The second pass teaches the generator how to fool the discriminator and confuse real samples with the generated ones.

```
gen_losses = []
dis_losses = []
iter_no = 0

true_labels_v = torch.ones(BATCH_SIZE, device=device)
fake_labels_v = torch.zeros(BATCH_SIZE, device=device)
```

Here, we define arrays, which will be used to accumulate losses, iterator counters, and variables with the true and fake labels.

```
for batch_v in iterate_batches(envs):
    # fake samples, input is 4D: batch, filters, x, y
    gen_input_v = torch.FloatTensor(
        BATCH_SIZE, LATENT_VECTOR_SIZE, 1, 1)
    gen_input_v.normal_(0, 1).to(device)
    batch_v = batch_v.to(device)
    gen_output_v = net_gener(gen_input_v)
```

At the beginning of the training loop, we generate a random vector and pass it to the `Generator` network.

```
dis_optimizer.zero_grad()
dis_output_true_v = net_discr(batch_v)
dis_output_fake_v = net_discr(gen_output_v.detach())
dis_loss = objective(dis_output_true_v, true_labels_v) + \
           objective(dis_output_fake_v, fake_labels_v)
dis_loss.backward()
dis_optimizer.step()
dis_losses.append(dis_loss.item())
```

At first, we train the discriminator by applying it two times: to the true data samples in our batch and to the generated ones. We need to call the `detach()` function on the generator's output to prevent gradients of this training pass from flowing into the generator (`detach()` is a method of `tensor`, which makes a copy of it without connection to the parent's operation).

```
gen_optimizer.zero_grad()
dis_output_v = net_discr(gen_output_v)
gen_loss_v = objective(dis_output_v, true_labels_v)
gen_loss_v.backward()
gen_optimizer.step()
gen_losses.append(gen_loss_v.item())
```

Now it's the generator's training time. We pass the generator's output to the discriminator, but now we don't stop the gradients. Instead, we apply the objective function with `True` labels. It will push our generator in the direction where the samples that it generates make the discriminator confuse them with the real data.

That was the code related to training, and the next couple of lines report losses and feed image samples to TensorBoard:

```
iter_no += 1
if iter_no % REPORT_EVERY_ITER == 0:
    log.info("Iter %d: gen_loss=%.3e, dis_loss=%.3e",
             iter_no, np.mean(gen_losses),
             np.mean(dis_losses))
    writer.add_scalar(
        "gen_loss", np.mean(gen_losses), iter_no)
    writer.add_scalar(
        "dis_loss", np.mean(dis_losses), iter_no)
    gen_losses = []
    dis_losses = []
if iter_no % SAVE_IMAGE_EVERY_ITER == 0:
    writer.add_image("fake", vutils.make_grid(
        gen_output_v.data[:64], normalize=True), iter_no)
    writer.add_image("real", vutils.make_grid(
        batch_v.data[:64], normalize=True), iter_no)
```

The training of this example is quite a lengthy process. On a GTX 1080 GPU, 100 iterations take about 40 seconds. At the beginning, the generated images are completely random noise, but after 10k-20k iterations, the generator becomes more and more proficient at its job and the generated images become more and more similar to the real game screenshots.

My experiments gave the following images after 40k-50k of training iterations (several hours on a GPU):
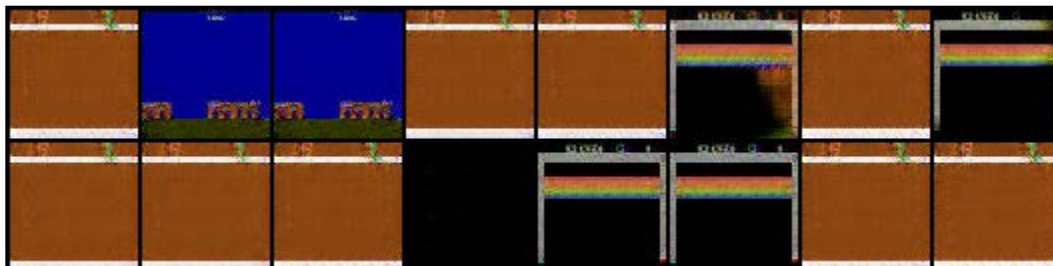
Figure 3.7: Sample images produced by the generator network

PyTorch Ignite

PyTorch is an elegant and flexible library, which makes it a favorite choice for thousands of researchers, DL enthusiasts, industry developers, and others. But flexibility has its own price: too much code to be written to solve your problem. Sometimes, this is very beneficial, such as when implementing some new optimization method or DL trick that hasn't been included in the standard library yet. Then you just implement the formulas using Python and PyTorch magic will do all the gradients and backpropagation machinery for you. Another example is in situations when you have to work on a very low level, fiddling with gradients, optimizer details, and the way your data is transformed by the NN.

However, sometimes you don't need this flexibility, which happens when you work on routine tasks, like the simple supervised training of an image classifier. For such tasks, standard PyTorch might be at too low a level when you need to deal with the same code over and over again. The following is a non-exhaustive list of topics that are an essential part of any DL training procedure, but require some code to be written:

- Data preparation and transformation, and the generation of batches
- Calculation of training metrics, like loss values, accuracy, and F1-scores
- Periodical testing of the model being trained on the test and validation datasets
- Model checkpointing after some number of iterations or when a new best metric is achieved
- Sending metrics into a monitoring tool like TensorBoard
- Hyperparameters change over time, like a learning rate decrease/increase schedule
- Writing training progress messages on the console

They are all doable using only PyTorch, of course, but it might require you to write a significant amount of code. As those tasks occur in any DL project, it quickly becomes cumbersome to write the same code over and over again. The normal approach to solving the issue is to write the functionality once, wrap it into a library, and reuse it later. If the library is open source and of good quality (easy to use, provides a good level of flexibility, written properly, and so on), it will become popular as more and more people use it in their projects. This process is not DL-specific; it happens everywhere in the software industry.

There are several libraries for PyTorch that simplify the solving of common tasks: `ptlearn`, `fastai`, `ignite` and some others. The current list of "PyTorch ecosystem projects" can be found here: `https://pytorch.org/ecosystem`.

It might be appealing to start using those high-level libraries from the beginning, as they allow you to solve common problems with just a couple of lines of code, but there is some danger here. If you only know how to use high-level libraries without understanding low-level details, you might get stuck on problems that can't be solved solely by standard methods. In the very dynamic field of ML, this happens very often.

The main focus of this book is to ensure that you understand RL methods, their implementation, and their applicability, so, we will use an incremental approach. In the beginning, we will implement methods using only PyTorch code, but with more progress, examples will be implemented using high-level libraries. For RL, this will be the small library written by me: PTAN (`https://github.com/Shmuma/ptan/`), and it will be introduced in *Chapter 7*, *Higher-Level RL Libraries*.

To reduce the amount of DL boilerplate code, we will use a library called PyTorch Ignite: `https://pytorch.org/ignite/`. In this section, a small overview of Ignite will be given, then we will check the Atari GAN example once it has been rewritten using Ignite.

Ignite concepts

At a high level, Ignite simplifies the writing of the training loop in PyTorch DL. Earlier in this chapter (in the section *Optimizers*), you saw that the minimal training loop consists of:

- Sampling a batch of training data

- Applying an NN to this batch to calculate the loss function—the single value we want to minimize

- Running backpropagation of the loss to get gradients on the network's parameters in respect to the loss

- Asking the optimizer to apply the gradients to the network
- Repeating until we are happy or bored of waiting

The central piece of Ignite is the `Engine` class, which loops over the data source, applying the processing function to the data batch. In addition to that, Ignite offers the ability to provide functions to be called at specific conditions of the training loop. Those conditions are called `Events` and could be at the:

- Beginning/end of the whole training process
- Beginning/end of a training epoch (iteration over the data)
- Beginning/end of a single batch processing

In addition to that, custom events exist and allow you to specify your function to be called every *N* events, for example, if you want to do some calculations every 100 batches or every second epoch.

A very simplistic example of Ignite is shown in the following code block:

```
from ignite.engine import Engine, Events

def training(engine, batch):
    optimizer.zero_grad()
    x, y = prepare_batch()
    y_out = model(x)
    loss = loss_fn(y_out, y)
    loss.backward()
    optimizer.step()
    return loss.item()

engine = Engine(training)
engine.run(data)
```

This code is not runnable, as it misses lots of stuff, like the data source, model, and optimizer creation, but it shows the basic idea of Ignite usage. The main benefit of Ignite is in the ability it provides to extend the training loop with existing functionality. You want the loss value to be smoothed and written in TensorBoard every 100 batches? No problem! Add two lines and it will be done. You want to run model validation every 10 epochs? Okay, write a function to run a test and attach it to `engine`, and it will be called.

A description of the full Ignite functionality is beyond the scope of the book, but you can read the documentation on the official website: `https://pytorch.org/ignite`.

To give you an illustration of Ignite, let's change the example of GAN training on Atari images. The full example code is available in `Chapter03/04_atari_gan_ignite.py`, so in the following code snippets, I'll show only the parts that differ.

```
from ignite.engine import Engine, Events
from ignite.metrics import RunningAverage
from ignite.contrib.handlers import tensorboard_logger as tb_logger
```

First, we import several Ignite classes, `Engine` and `Events`, which have already been outlined. The package `ignite.metrics` contains classes related to working with the performance metrics of the training process, such as confusion matrices, precision, and recall. In our example, we will use the class `RunningAverage`, which provides a way to smooth time series values. In the previous example, we did this by calling `np.mean()` on an array of losses, but `RunningAverage` provides a more convenient (and mathematically more correct) way of doing this. In addition, we import TensorBoard logger from the Ignite `contrib` package (the functionality of which is contributed by others).

```
def process_batch(trainer, batch):
    gen_input_v = torch.FloatTensor(
        BATCH_SIZE, LATENT_VECTOR_SIZE, 1, 1)
    gen_input_v.normal_(0, 1).to(device)
    batch_v = batch.to(device)
    gen_output_v = net_gener(gen_input_v)

    dis_optimizer.zero_grad()
    dis_output_true_v = net_discr(batch_v)
    dis_output_fake_v = net_discr(gen_output_v.detach())
    dis_loss = objective(dis_output_true_v, true_labels_v) + \
               objective(dis_output_fake_v, fake_labels_v)
    dis_loss.backward()
    dis_optimizer.step()

    gen_optimizer.zero_grad()
    dis_output_v = net_discr(gen_output_v)
    gen_loss = objective(dis_output_v, true_labels_v)
    gen_loss.backward()
    gen_optimizer.step()

    if trainer.state.iteration % SAVE_IMAGE_EVERY_ITER == 0:
        fake_img = vutils.make_grid(
            gen_output_v.data[:64], normalize=True)
        trainer.tb.writer.add_image(
            "fake", fake_img, trainer.state.iteration)
        real_img = vutils.make_grid(
```

```
                    batch_v.data[:64], normalize=True)
                trainer.tb.writer.add_image(
                    "real", real_img, trainer.state.iteration)
                trainer.tb.writer.flush()
            return dis_loss.item(), gen_loss.item()
```

As a next step, we need to define our processing function, which takes the data batch and does an update of both the discriminator and generator models on this batch. This function can return any data to be tracked during the training process; in our case, it will be two loss values for both models. In this function, we can also save images to be displayed in TensorBoard.

After this is done, all we need to do is create an `Engine` instance, attach the required handlers, and run the training process.

```
        engine = Engine(process_batch)
        tb = tb_logger.TensorboardLogger(log_dir=None)
        engine.tb = tb
        RunningAverage(output_transform=lambda out: out[0]).\
            attach(engine, "avg_loss_gen")
        RunningAverage(output_transform=lambda out: out[1]).\
            attach(engine, "avg_loss_dis")

        handler = tb_logger.OutputHandler(tag="train",
            metric_names=['avg_loss_gen', 'avg_loss_dis'])
        tb.attach(engine, log_handler=handler,
                    event_name=Events.ITERATION_COMPLETED)
```

In the preceding code, we create our engine, passing our processing function and attaching two `RunningAverage` transformations for our two loss values. Being attached, every `RunningAverage` produces a so-called "metric"—a derived value kept around during the training process. The names of our smoothed metrics are `avg_loss_gen` for smoothed loss from the generator and `avg_loss_dis` for smoothed loss from the discriminator. Those two values will be written in TensorBoard after every iteration.

```
        @engine.on(Events.ITERATION_COMPLETED)
        def log_losses(trainer):
            if trainer.state.iteration % REPORT_EVERY_ITER == 0:
                log.info("%d: gen_loss=%f, dis_loss=%f",
                        trainer.state.iteration,
                        trainer.state.metrics['avg_loss_gen'],
                        trainer.state.metrics['avg_loss_dis'])

        engine.run(data=iterate_batches(envs))
```

The last piece of code attaches another event handler, which will be our function, and is called by the `Engine` on every iteration completion. It will write a log line with an iteration index and values of smoothed metrics. The final line starts our engine, passing the already defined function as the data source (function `iterate_batches` is a generator, returning the normal iterator over batches, so, it will be perfectly fine to pass its output as a `data` argument).

And that's it. If you run the example `Chapter03/04_atari_gan_ignite.py`, it will work the same way as our previous example, which might not be very impressive for such a small example, but in real projects, Ignite usage normally pays off by making your code cleaner and more extensible.

Summary

In this chapter, you saw a quick overview of PyTorch's functionality and features. We talked about basic fundamental pieces, such as tensors and gradients, and you saw how an NN can be made from the basic building blocks, before learning how to implement those blocks yourself.

We discussed loss functions and optimizers, as well as the monitoring of training dynamics. Finally, you were introduced to PyTorch Ignite, a library used to provide a higher-level interface for training loops. The goal of the chapter was to give a very quick introduction to PyTorch, which will be used later in the book.

In the next chapter, we are ready to start dealing with the main subject of this book: RL methods.

4
The Cross-Entropy Method

In the last chapter, you got to know PyTorch. In this chapter, we will wrap up part one of this book and you will become familiar with one of the reinforcement learning (RL) methods: cross-entropy.

Despite the fact that it is much less famous than other tools in the RL practitioner's toolbox, such as **deep Q-network** (**DQN**) or advantage actor-critic, the cross-entropy method has its own strengths. Firstly, the cross-entropy method is really simple, which makes it an easy method to follow. For example, its implementation on PyTorch is less than 100 lines of code.

Secondly, the method has good convergence. In simple environments that don't require complex, multistep policies to be learned and discovered, and that have short episodes with frequent rewards, the cross-entropy method usually works very well. Of course, lots of practical problems don't fall into this category, but sometimes they do. In such cases, the cross-entropy method (on its own or as part of a larger system) can be the perfect fit.

In this chapter, we will cover:

- The practical side of the cross-entropy method
- How the cross-entropy method works in two environments in Gym (the familiar CartPole and the grid world of FrozenLake)
- The theoretical background of the cross-entropy method. This section is optional and requires a bit more knowledge of probability and statistics, but if you want to understand why the method works, then you can delve into it

The taxonomy of RL methods

The cross-entropy method falls into the **model-free** and **policy-based** category of methods. These notions are new, so let's spend some time exploring them. All the methods in RL can be classified into various aspects:

- Model-free or model-based
- Value-based or policy-based
- On-policy or off-policy

There are other ways that you can taxonomize RL methods, but, for now, we are interested in the preceding three. Let's define them, as your problem specifics can influence your decision on a particular method.

The term "model-free" means that the method doesn't build a model of the environment or reward; it just directly connects observations to actions (or values that are related to actions). In other words, the agent takes current observations and does some computations on them, and the result is the action that it should take. In contrast, **model-based** methods try to predict what the next observation and/or reward will be. Based on this prediction, the agent tries to choose the best possible action to take, very often making such predictions multiple times to look more and more steps into the future.

Both classes of methods have strong and weak sides, but usually pure model-based methods are used in deterministic environments, such as board games with strict rules. On the other hand, model-free methods are usually easier to train as it's hard to build good models of complex environments with rich observations. All of the methods described in this book are from the model-free category, as those methods have been the most active area of research for the past few years. Only recently have researchers started to mix the benefits from both worlds (for example, refer to DeepMind's papers on imagination in agents. This approach will be described in *Chapter 22*, *Beyond Model-Free – Imagination*).

By looking from another angle, policy-based methods directly approximate the policy of the agent, that is, what actions the agent should carry out at every step. The policy is usually represented by a probability distribution over the available actions.

In contrast, the method could be **value-based**. In this case, instead of the probability of actions, the agent calculates the value of every possible action and chooses the action with the best value. Both of those families of methods are equally popular and we will discuss value-based methods in the next part of the book. Policy methods will be the topic of part three.

The third important classification of methods is **on-policy** versus **off-policy**. We will discuss this distinction more in parts two and three of the book, but, for now, it will be enough to explain off-policy as the ability of the method to learn on historical data (obtained by a previous version of the agent, recorded by human demonstration, or just seen by the same agent several episodes ago).

So, our cross-entropy method is model-free, policy-based, and on-policy, which means the following:

- It doesn't build any model of the environment; it just says to the agent what to do at every step

- It approximates the policy of the agent

- It requires fresh data obtained from the environment

The cross-entropy method in practice

The cross-entropy method's description is split into two unequal parts: practical and theoretical. The practical part is intuitive in its nature, while the theoretical explanation of *why* the cross-entropy method works, and what's happening, is more sophisticated.

You may remember that the central and trickiest thing in RL is the agent, which is trying to accumulate as much total reward as possible by communicating with the environment. In practice, we follow a common machine learning (ML) approach and replace all of the complications of the agent with some kind of nonlinear trainable function, which maps the agent's input (observations from the environment) to some output. The details of the output that this function produces may depend on a particular method or a family of methods, as described in the previous section (such as value-based versus policy-based methods). As our cross-entropy method is policy-based, our nonlinear function (neural network (NN)) produces the *policy*, which basically says for every observation which action the agent should take.
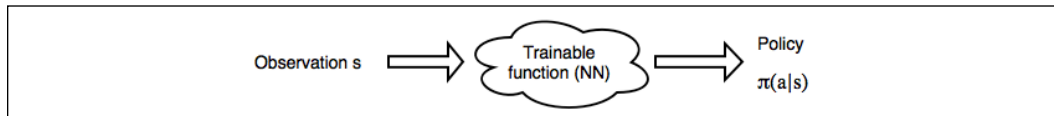
Figure 4.1: A high-level approach to RL

In practice, the policy is usually represented as a probability distribution over actions, which makes it very similar to a classification problem, with the amount of classes being equal to the amount of actions we can carry out.

This abstraction makes our agent very simple: it needs to pass an observation from the environment to the NN, get a probability distribution over actions, and perform random sampling using the probability distribution to get an action to carry out. This random sampling adds randomness to our agent, which is a good thing, as at the beginning of the training, when our weights are random, the agent behaves randomly. After the agent gets an action to issue, it fires the action to the environment and obtains the next observation and reward for the last action. Then the loop continues.

During the agent's lifetime, its experience is presented as episodes. Every episode is a sequence of observations that the agent has got from the environment, actions it has issued, and rewards for these actions. Imagine that our agent has played several such episodes. For every episode, we can calculate the total reward that the agent has claimed. It can be discounted or not discounted; for simplicity, let's assume a discount factor of $\gamma = 1$, which means just a sum of all local rewards for every episode. This total reward shows how good this episode was for the agent.

Let's illustrate this with a diagram, which contains four episodes (note that different episodes have different values for o_i, a_i, and r_i):
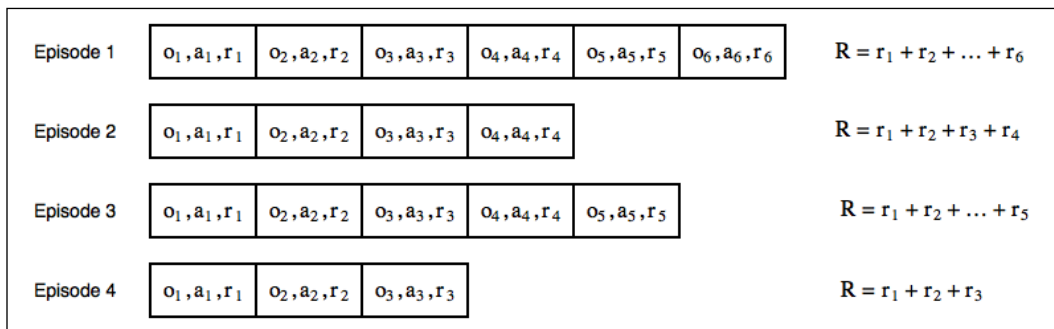
Figure 4.2: Sample episodes with their observations, actions, and rewards

Every cell represents the agent's step in the episode. Due to randomness in the environment and the way that the agent selects actions to take, some episodes will be better than others. The core of the cross-entropy method is to throw away bad episodes and train on better ones. So, the steps of the method are as follows:

1. Play *N* number of episodes using our current model and environment.

2. Calculate the total reward for every episode and decide on a reward boundary. Usually, we use some percentile of all rewards, such as 50th or 70th.

3. Throw away all episodes with a reward below the boundary.

4. Train on the remaining "elite" episodes using observations as the input and issued actions as the desired output.

5. Repeat from step 1 until we become satisfied with the result.

So, that's the cross-entropy method's description. With the preceding procedure, our NN learns how to repeat actions, which leads to a larger reward, constantly moving the boundary higher and higher. Despite the simplicity of this method, it works well in basic environments, it's easy to implement, and it's quite robust to hyperparameters changing, which makes it an ideal baseline method to try. Let's now apply it to our CartPole environment.

The cross-entropy method on CartPole

The whole code for this example is in `Chapter04/01_cartpole.py`, but the following are the most important parts. Our model's core is a one-hidden-layer NN, with rectified linear unit (ReLU) and 128 hidden neurons (which is absolutely arbitrary). Other hyperparameters are also set almost randomly and aren't tuned, as the method is robust and converges very quickly.

```
HIDDEN_SIZE = 128
BATCH_SIZE = 16
PERCENTILE = 70
```

We define constants at the top of the file and they include the count of neurons in the hidden layer, the count of episodes we play on every iteration (`16`), and the percentile of episodes' total rewards that we use for "elite" episode filtering. We will take the 70th percentile, which means that we will leave the top 30% of episodes sorted by reward.

```
class Net(nn.Module):
    def __init__(self, obs_size, hidden_size, n_actions):
        super(Net, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(obs_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, n_actions)
        )

    def forward(self, x):
        return self.net(x)
```

There is nothing special about our NN; it takes a single observation from the environment as an input vector and outputs a number for every action we can perform. The output from the NN is a probability distribution over actions, so a straightforward way to proceed would be to include softmax nonlinearity after the last layer. However, in the preceding NN, we don't apply softmax to increase the numerical stability of the training process. Rather than calculating softmax (which uses exponentiation) and then calculating cross-entropy loss (which uses a logarithm of probabilities), we can use the PyTorch class `nn.CrossEntropyLoss`, which combines both softmax and cross-entropy in a single, more numerically stable expression. `CrossEntropyLoss` requires raw, unnormalized values from the NN (also called logits). The downside of this is that we need to remember to apply softmax every time we need to get probabilities from our NN's output.

```
Episode = namedtuple('Episode', field_names=['reward', 'steps'])
EpisodeStep = namedtuple(
    'EpisodeStep',field_names=['observation', 'action'])
```

Here we will define two helper classes that are named tuples from the `collections` package in the standard library:

- `EpisodeStep`: This will be used to represent one single step that our agent made in the episode, and it stores the observation from the environment and what action the agent completed. We will use episode steps from "elite" episodes as training data.

- `Episode`: This is a single episode stored as total undiscounted reward and a collection of `EpisodeStep`.

Let's look at a function that generates batches with episodes:

```
def iterate_batches(env, net, batch_size):
    batch = []
    episode_reward = 0.0
    episode_steps = []
    obs = env.reset()
    sm = nn.Softmax(dim=1)
```

The preceding function accepts the environment (the `Env` class instance from the Gym library), our NN, and the count of episodes it should generate on every iteration. The `batch` variable will be used to accumulate our batch (which is a list of `Episode` instances). We also declare a reward counter for the current episode and its list of steps (the `EpisodeStep` objects). Then we reset our environment to obtain the first observation and create a softmax layer, which will be used to convert the NN's output to a probability distribution of actions. That's our preparations complete, so we are ready to start the environment loop.

```
while True:
    obs_v = torch.FloatTensor([obs])
    act_probs_v = sm(net(obs_v))
    act_probs = act_probs_v.data.numpy()[0]
```

At every iteration, we convert our current observation to a PyTorch tensor and pass it to the NN to obtain action probabilities. There are several things to note here:

- All `nn.Module` instances in PyTorch expect a batch of data items and the same is true for our NN, so we convert our observation (which is a vector of four numbers in CartPole) into a tensor of size 1×4 (to achieve this, we pass an observation in a single-element list).

- As we haven't used nonlinearity at the output of our NN, it outputs raw action scores, which we need to feed through the softmax function.

- Both our NN and the softmax layer return tensors that track gradients, so we need to unpack this by accessing the `tensor.data` field and then converting the tensor into a NumPy array. This array will have the same two-dimensional structure as the input, with the batch dimension on axis 0, so we need to get the first batch element to obtain a one-dimensional vector of action probabilities.

```
action = np.random.choice(len(act_probs), p=act_probs)
next_obs, reward, is_done, _ = env.step(action)
```

Now that we have the probability distribution of actions, we can use it to obtain the actual action for the current step by sampling this distribution using NumPy's function `random.choice()`. After this, we will pass this action to the environment to get our next observation, our reward, and the indication of the episode ending.

```
episode_reward += reward
step = EpisodeStep(observation=obs,action=action)
episode_steps.append(step)
```

The reward is added to the current episode's total reward, and our list of episode steps is also extended with an `(observation, action)` pair. Note that we save the observation that was used to choose the action, but not the observation returned by the environment as a result of the action. These are the tiny, but important, details that you need to keep in mind.

```
if is_done:
    e = Episode(reward=episode_reward,steps=episode_steps)
    batch.append(e)
    episode_reward = 0.0
    episode_steps = []
    next_obs = env.reset()
```

```
        if len(batch) == batch_size:
            yield batch
            batch = []
```

This is how we handle the situation when the current episode is over (in the case of CartPole, the episode ends when the stick has fallen down despite our efforts). We append the finalized episode to the batch, saving the total reward (as the episode has been completed and we have accumulated all the reward) and steps we have taken. Then we reset our total reward accumulator and clean the list of steps. After that, we reset our environment to start over.

In case our batch has reached the desired count of episodes, we return it to the caller for processing using `yield`. Our function is a generator, so every time the `yield` operator is executed, the control is transferred to the outer iteration loop and then continues after the `yield` line. If you are not familiar with Python's generator functions, refer to the Python documentation: `https://wiki.python.org/moin/Generators`. After processing, we will clean up the batch.

```
        obs = next_obs
```

The last, but very important, step in our loop is to assign an observation obtained from the environment to our current observation variable. After that, everything repeats infinitely—we pass the observation to the NN, sample the action to perform, ask the environment to process the action, and remember the result of this processing.

One very important fact to understand in this function logic is that the training of our NN and the generation of our episodes are performed *at the same time*. They are not completely in parallel, but every time our loop accumulates enough episodes (16), it passes control to this function caller, which is supposed to train the NN using gradient descent. So, when `yield` is returned, the NN will have different, slightly better (we hope) behavior.

We don't need to explore proper synchronization, as our training and data gathering activities are performed at the same thread of execution, but you need to understand those constant jumps from NN training to its utilization.

Okay, now we need to define yet another function and then we will be ready to switch to the training loop.

```
def filter_batch(batch, percentile):
    rewards = list(map(lambda s: s.reward, batch))
    reward_bound = np.percentile(rewards, percentile)
    reward_mean = float(np.mean(rewards))
```

This function is at the core of the cross-entropy method—from the given batch of episodes and percentile value, it calculates a boundary reward, which is used to filter "elite" episodes to train on. To obtain the boundary reward, we will use NumPy's `percentile` function, which, from the list of values and the desired percentile, calculates the percentile's value. Then, we will calculate the mean reward, which is used only for monitoring.

```python
train_obs = []
train_act = []
for reward, steps in batch:
    if reward < reward_bound:
        continue
    train_obs.extend(map(lambda step: step.observation,steps))
    train_act.extend(map(lambda step: step.action, steps))
```

Next, we will filter off our episodes. For every episode in the batch, we will check that the episode has a higher total reward than our boundary and if it has, we will populate lists of observations and actions that we will train on.

```python
train_obs_v = torch.FloatTensor(train_obs)
train_act_v = torch.LongTensor(train_act)
return train_obs_v, train_act_v, reward_bound, reward_mean
```

As the final step of the function, we will convert our observations and actions from "elite" episodes into tensors, and return a tuple of four: observations, actions, the boundary of reward, and the mean reward. The last two values will be used only to write them into TensorBoard to check the performance of our agent.

Now, the final chunk of code that glues everything together, and mostly consists of the training loop, is as follows:

```python
if __name__ == "__main__":
    env = gym.make("CartPole-v0")
    # env = gym.wrappers.Monitor(env, directory="mon", force=True)
    obs_size = env.observation_space.shape[0]
    n_actions = env.action_space.n

    net = Net(obs_size, HIDDEN_SIZE, n_actions)
    objective = nn.CrossEntropyLoss()
    optimizer = optim.Adam(params=net.parameters(), lr=0.01)
    writer = SummaryWriter(comment="-cartpole")
```

In the beginning, we create all the required objects: the environment, our NN, the objective function, the optimizer, and the summary writer for TensorBoard.

The commented line creates a monitor to write videos of your agent's performance.

```
for iter_no, batch in enumerate(iterate_batches(
        env, net,BATCH_SIZE)):
    obs_v, acts_v, reward_b, reward_m = \
        filter_batch(batch, PERCENTILE)
    optimizer.zero_grad()
    action_scores_v = net(obs_v)
    loss_v = objective(action_scores_v, acts_v)
    loss_v.backward()
    optimizer.step()
```

In the training loop, we iterate our batches (a list of `Episode` objects), then we perform filtering of the "elite" episodes using the `filter_batch` function. The result is variables of observations and taken actions, the reward boundary used for filtering, and the mean reward. After that, we zero gradients of our NN and pass observations to the NN, obtaining its action scores. These scores are passed to the `objective` function, which will calculate cross-entropy between the NN output and the actions that the agent took. The idea of this is to reinforce our NN to carry out those "elite" actions that have led to good rewards. Then, we calculate gradients on the loss and ask the optimizer to adjust our NN.

```
print("%d: loss=%.3f, reward_mean=%.1f, rw_bound=%.1f" % (
    iter_no, loss_v.item(), reward_m, reward_b))
writer.add_scalar("loss", loss_v.item(), iter_no)
writer.add_scalar("reward_bound", reward_b, iter_no)
writer.add_scalar("reward_mean", reward_m, iter_no)
```

The rest of the loop is mostly the monitoring of progress. On the console, we show the iteration number, the loss, the mean reward of the batch, and the reward boundary. We also write the same values to TensorBoard, to get a nice chart of the agent's learning performance.

```
    if reward_m > 199:
        print("Solved!")
        break
writer.close()
```

The last check in the loop is the comparison of the mean rewards of our batch episodes. When this becomes greater than 199, we stop our training. Why 199? In Gym, the CartPole environment is considered to be solved when the mean reward for the last 100 episodes is greater than 195, but our method converges so quickly that 100 episodes are usually what we need. The properly trained agent can balance the stick infinitely long (obtaining any amount of score), but the length of an episode in CartPole is limited to 200 steps (if you look at the environment variable of CartPole, you may notice the TimeLimit wrapper, which stops the episode after 200 steps). With all this in mind, we will stop training after the mean reward in the batch is greater than 199, which is a good indication that our agent knows how to balance the stick like a pro.

That's it. So let's start our first RL training!

```
rl_book_samples/Chapter04$ ./01_cartpole.py
[2017-10-04 12:44:39,319] Making new env: CartPole-v0
0: loss=0.701, reward_mean=18.0, rw_bound=21.0
1: loss=0.682, reward_mean=22.6, rw_bound=23.5
2: loss=0.688, reward_mean=23.6, rw_bound=25.5
3: loss=0.675, reward_mean=22.8, rw_bound=22.0
4: loss=0.658, reward_mean=31.9, rw_bound=34.0
.........
36: loss=0.527, reward_mean=135.9, rw_bound=168.5
37: loss=0.527, reward_mean=147.4, rw_bound=160.5
38: loss=0.528, reward_mean=179.8, rw_bound=200.0
39: loss=0.530, reward_mean=178.7, rw_bound=200.0
40: loss=0.532, reward_mean=192.1, rw_bound=200.0
41: loss=0.523, reward_mean=196.8, rw_bound=200.0
42: loss=0.540, reward_mean=200.0, rw_bound=200.0
Solved!
```

It usually doesn't take the agent more than 50 batches to solve the environment. My experiments show something from 25 to 45 episodes, which is a really good learning performance (remember, we need to play only 16 episodes for every batch). TensorBoard shows our agent consistently making progress, pushing the upper boundary at almost every batch (there are some periods of rolling down, but most of the time it improves).
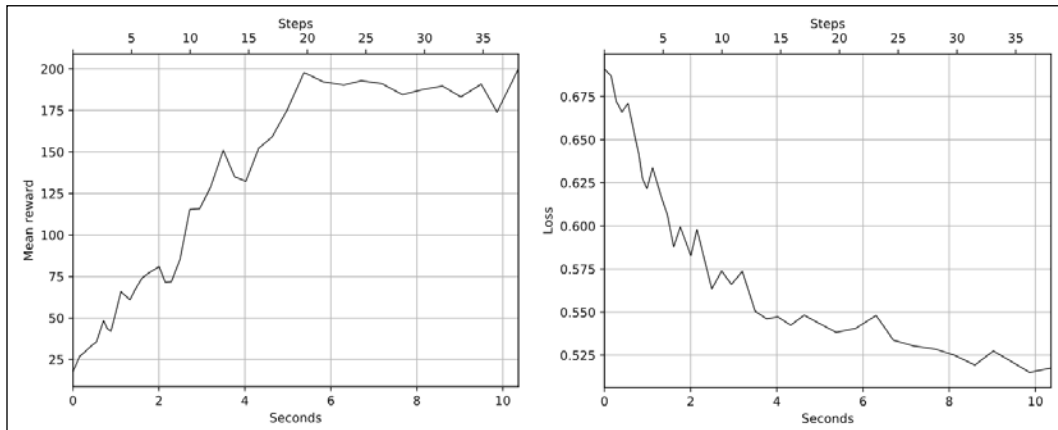
Figure 4.3: Mean reward (left) and loss (right) during the training
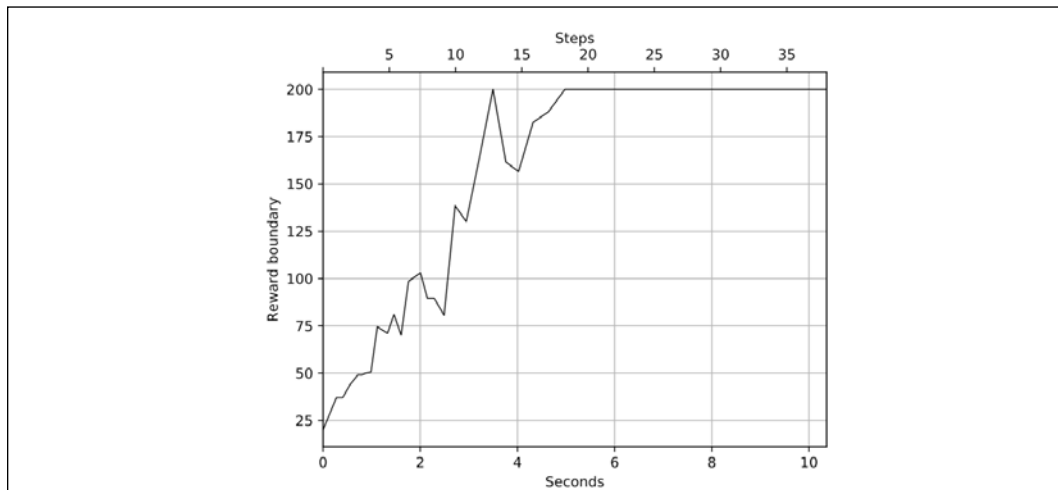
Figure 4.4: The reward boundary during the training

To check our agent in action, you can enable `Monitor` by uncommenting the next line after the environment creation.

After restarting (possibly with `xvfb-run` to provide a virtual X11 display), our program will create a `mon` directory with videos recorded at different training steps.

```
Chapter04$ xvfb-run -s "-screen 0 640x480x24" ./01_cartpole.py
[2017-10-04 13:52:23,806] Making new env: CartPole-v0
[2017-10-04 13:52:23,814] Creating monitor directory mon
[2017-10-04 13:52:23,920] Starting new video recorder writing to mon/
openaigym.video.0.4430.video000000.mp4
[2017-10-04 13:52:25,229] Starting new video recorder writing to mon/
openaigym.video.0.4430.video000001.mp4
[2017-10-04 13:52:25,771] Starting new video recorder writing to mon/
openaigym.video.0.4430.video000008.mp4
0: loss=0.682, reward_mean=18.9, rw_bound=20.5
[2017-10-04 13:52:26,297] Starting new video recorder writing to mon/
openaigym.video.0.4430.video000027.mp4
1: loss=0.687, reward_mean=16.6, rw_bound=19.0
2: loss=0.677, reward_mean=21.1, rw_bound=21.0
[2017-10-04 13:52:26,964] Starting new video recorder writing to mon/
openaigym.video.0.4430.video000064.mp4
3: loss=0.653, reward_mean=33.2, rw_bound=48.5
4: loss=0.642, reward_mean=37.4, rw_bound=42.5
.........
29: loss=0.561, reward_mean=111.6, rw_bound=122.0
30: loss=0.540, reward_mean=135.1, rw_bound=166.0
[2017-10-04 13:52:40,176] Starting new video recorder writing to mon/
openaigym.video.0.4430.video000512.mp4
31: loss=0.546, reward_mean=147.5, rw_bound=179.5
32: loss=0.559, reward_mean=140.0, rw_bound=171.5
33: loss=0.558, reward_mean=160.4, rw_bound=200.0
34: loss=0.547, reward_mean=167.6, rw_bound=195.5
35: loss=0.550, reward_mean=179.5, rw_bound=200.0
36: loss=0.563, reward_mean=173.9, rw_bound=200.0
37: loss=0.542, reward_mean=162.9, rw_bound=200.0
38: loss=0.552, reward_mean=159.1, rw_bound=200.0
39: loss=0.548, reward_mean=189.6, rw_bound=200.0
40: loss=0.546, reward_mean=191.1, rw_bound=200.0
41: loss=0.548, reward_mean=199.1, rw_bound=200.0
Solved!
```

As you can see from the output, it turns a periodical recording of the agent's activity into separate video files, which can give you an idea of what your agent's sessions look like.
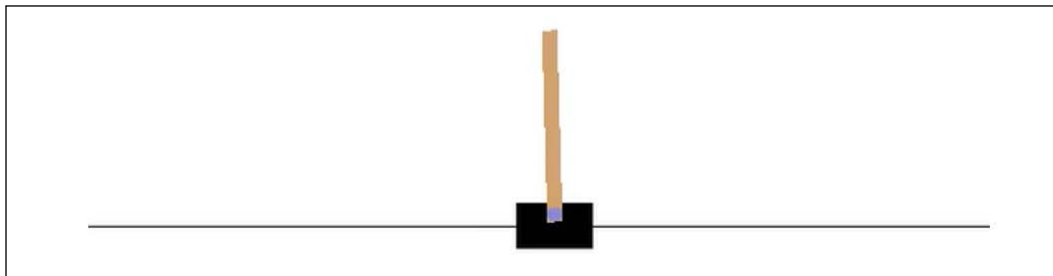
Figure 4.5: Visualization of the CartPole state

Let's now pause a bit and think about what's just happened. Our NN has learned how to play the environment purely from observations and rewards, without any interpretation of observed values. The environment could easily not be a cart with a stick; it could be, say, a warehouse model with product quantities as an observation and money earned as the reward. Our implementation doesn't depend on environment details. This is the beauty of the RL model, and in the next section, we will look at how exactly the same method can be applied to a different environment from the Gym collection.

The cross-entropy method on FrozenLake

The next environment that we will try to solve using the cross-entropy method is FrozenLake. Its world is from the so-called grid world category, when your agent lives in a grid of size 4×4 and can move in four directions: up, down, left, and right. The agent always starts at a top-left position, and its goal is to reach the bottom-right cell of the grid. There are holes in the fixed cells of the grid and if you get into those holes, the episode ends and your reward is zero. If the agent reaches the destination cell, then it obtains a reward of 1.0 and the episode ends.

To make life more complicated, the world is slippery (it's a frozen lake after all), so the agent's actions do not always turn out as expected—there is a 33% chance that it will slip to the right or to the left. If you want the agent to move left, for example, there is a 33% probability that it will, indeed, move left, a 33% chance that it will end up in the cell above, and a 33% chance that it will end up in the cell below. As you will see at the end of the section, this makes progress difficult.
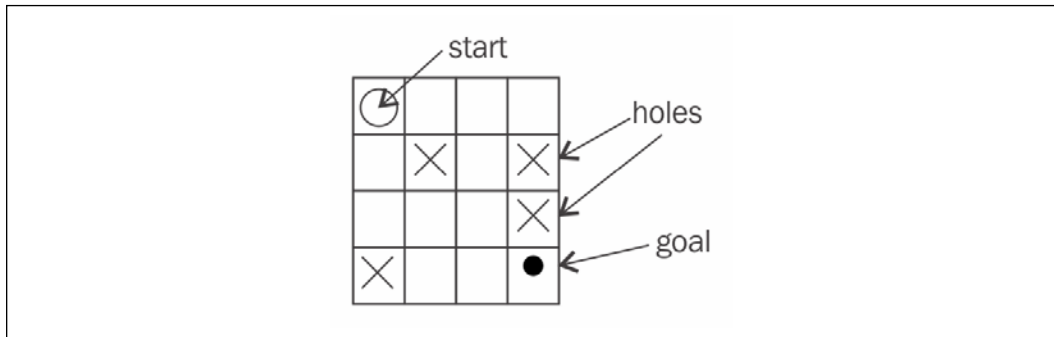
Figure 4.6: The FrozenLake environment

Let's look at how this environment is represented in Gym:

```
>>> e = gym.make("FrozenLake-v0")
[2017-10-05 12:39:35,827] Making new env: FrozenLake-v0
>>> e.observation_space
Discrete(16)
>>> e.action_space
Discrete(4)
>>> e.reset()
0
>>> e.render()
SFFF
FHFH
FFFH
HFFG
```

Our observation space is discrete, which means that it's just a number from zero to 15 inclusive. Obviously, this number is our current position in the grid. The action space is also discrete, but it can be from zero to three. Our NN from the CartPole example expects a vector of numbers. To get this, we can apply the traditional one-hot encoding of discrete inputs, which means that the input to our network will have 16 float numbers and zero everywhere except the index that we will encode. To minimize changes in our code, we can use the `ObservationWrapper` class from Gym and implement our `DiscreteOneHotWrapper` class:

```
class DiscreteOneHotWrapper(gym.ObservationWrapper):
    def __init__(self, env):
        super(DiscreteOneHotWrapper, self).__init__(env)
        assert isinstance(env.observation_space,
                          gym.spaces.Discrete)
        shape = (env.observation_space.n, )
        self.observation_space = gym.spaces.Box(
```

```
                0.0, 1.0, shape, dtype=np.float32)

    def observation(self, observation):
        res = np.copy(self.observation_space.low)
        res[observation] = 1.0
        return res
```

With that wrapper applied to the environment, both the observation space and action space are 100% compatible with our CartPole solution (source code `Chapter04/02_frozenlake_naive.py`). However, by launching it, we can see that this doesn't improve the score over time.
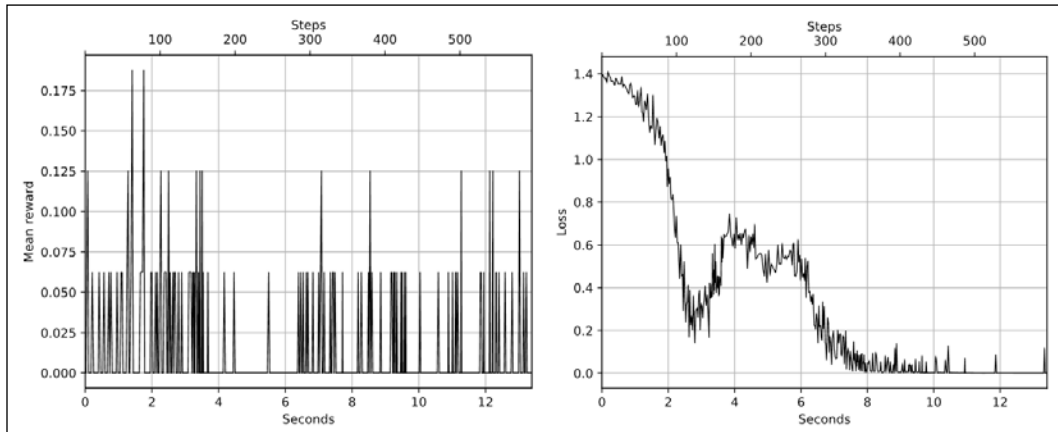
Figure 4.7: Reward (left) and loss (right) of the FrozenLake environment
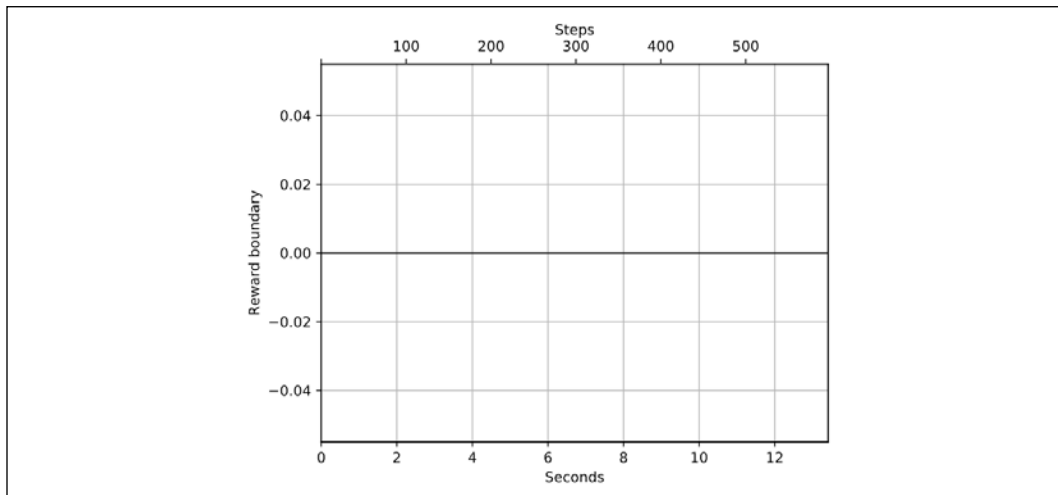
Figure 4.8: The reward boundary during the training (extremely boring)

To understand what's going on, we need to look deeper at the reward structure of both environments. In CartPole, every step of the environment gives us the reward 1.0, until the moment that the pole falls. So, the longer our agent balanced the pole, the more reward it obtained. Due to randomness in our agent's behavior, different episodes were of different lengths, which gave us a pretty normal distribution of the episodes' rewards. After choosing a reward boundary, we rejected less successful episodes and learned how to repeat better ones (by training on successful episodes' data).

This is shown in the following diagram:
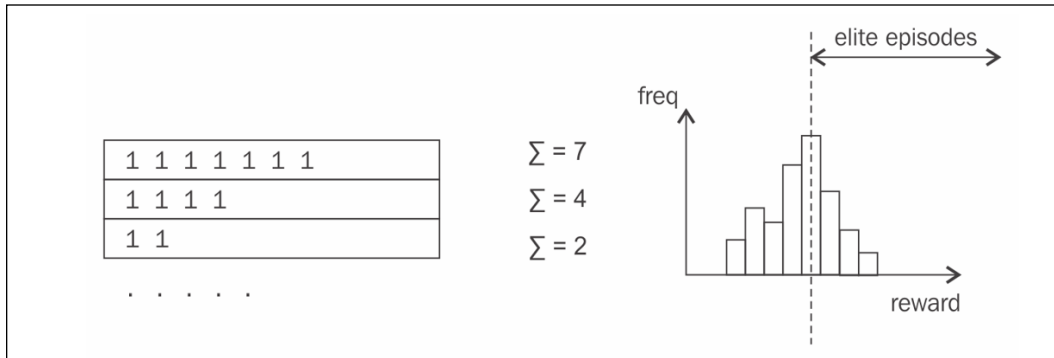
Figure 4.9: Distribution of the reward in the CartPole environment

In the FrozenLake environment, episodes and their rewards look different. We get the reward of 1.0 only when we reach the goal, and this reward says nothing about how good each episode was. Was it quick and efficient or did we make four rounds on the lake before we randomly stepped into the final cell? We don't know; it's just 1.0 reward and that's it. The distribution of rewards for our episodes are also problematic. There are only two kinds of episodes possible, with zero reward (failed) and one reward (successful), and failed episodes will obviously dominate in the beginning of the training. So, our percentile selection of "elite" episodes is totally wrong and gives us bad examples to train on. This is the reason for our training failure.
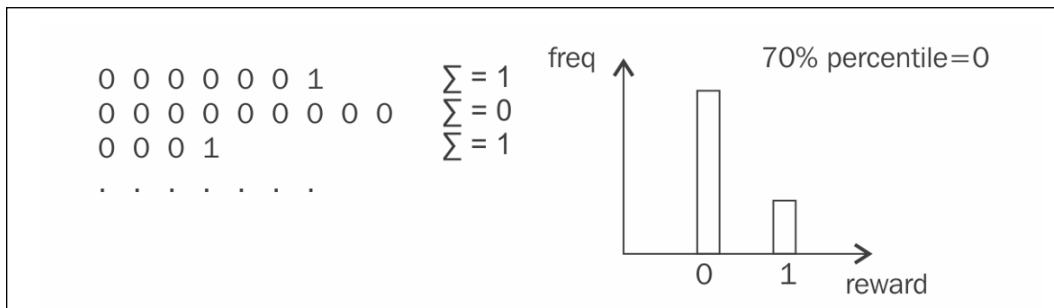
Figure 4.10: Reward distribution of the FrozenLake environment

This example shows us the limitations of the cross-entropy method:

- For training, our episodes have to be finite and, preferably, short
- The total reward for the episodes should have enough variability to separate good episodes from bad ones
- There is no intermediate indication about whether the agent has succeeded or failed

Later in the book, you will become familiar with other methods that address these limitations. For now, if you are curious about how FrozenLake can be solved using the cross-entropy method, here is a list of tweaks of the code that you need to make (the full example is in `Chapter04/03_frozenlake_tweaked.py`):

- **Larger batches of played episodes**: In CartPole, it was enough to have 16 episodes on every iteration, but FrozenLake requires at least 100 just to get some successful episodes.
- **Discount factor applied to the reward**: To make the total reward for an episode depend on its length, and add variety in episodes, we can use a discounted total reward with the discount factor 0.9 or 0.95. In this case, the reward for shorter episodes will be higher than the reward for longer ones. This increases variability in reward distribution, which helps to avoid situations like the one shown in *Figure 4.10*.
- **Keeping "elite" episodes for a longer time**: In the CartPole training, we sampled episodes from the environment, trained on the best ones, and threw them away. In FrozenLake, a successful episode is a much rarer animal, so we need to keep them for several iterations to train on them.
- **Decreasing learning rate**: This will give our NN time to average more training samples.
- **Much longer training time**: Due to the sparsity of successful episodes, and the random outcome of our actions, it's much harder for our NN to get an idea of the best behavior to perform in any particular situation. To reach 50% successful episodes, about 5k training iterations are required.

To incorporate all these into our code, we need to change the `filter_batch` function to calculate discounted reward and return "elite" episodes for us to keep:

```python
def filter_batch(batch, percentile):
    filter_fun = lambda s: s.reward * (GAMMA ** len(s.steps))
    disc_rewards = list(map(filter_fun, batch))
    reward_bound = np.percentile(disc_rewards, percentile)

    train_obs = []
    train_act = []
    elite_batch = []
    for example, discounted_reward in zip(batch, disc_rewards):
        if discounted_reward > reward_bound:
            train_obs.extend(map(lambda step: step.observation,
                                 example.steps))
            train_act.extend(map(lambda step: step.action,
                                 example.steps))
            elite_batch.append(example)
    return elite_batch, train_obs, train_act, reward_bound
```

Then, in the training loop, we will store previous "elite" episodes to pass them to the preceding function on the next training iteration.

```python
full_batch = []
for iter_no, batch in enumerate(iterate_batches(
        env, net, BATCH_SIZE)):
    reward_mean = float(np.mean(list(map(
        lambda s: s.reward, batch))))
    full_batch, obs, acts, reward_bound = \
        filter_batch(full_batch + batch, PERCENTILE)
    if not full_batch:
        continue
    obs_v = torch.FloatTensor(obs)
    acts_v = torch.LongTensor(acts)
    full_batch = full_batch[-500:]
```

The rest of the code is the same, except that the learning rate decreased 10 times and the `BATCH_SIZE` was set to `100`. After a period of patient waiting (the new version takes about one and a half hours to finish 10k iterations), you can see that the training of the model stopped improving at around 55% of solved episodes. There are ways to address this (by applying entropy loss regularization, for example), but those techniques will be discussed in upcoming chapters.
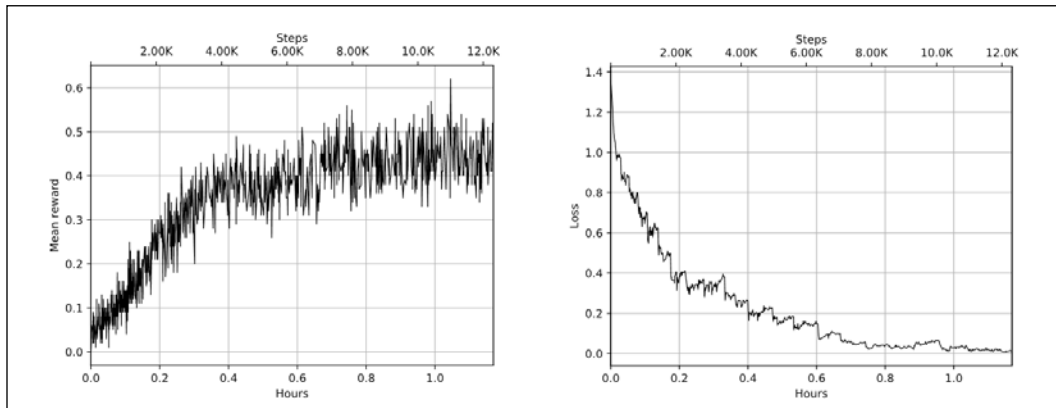
Figure 4.11: Reward (left) and loss (right) of the tweaked training
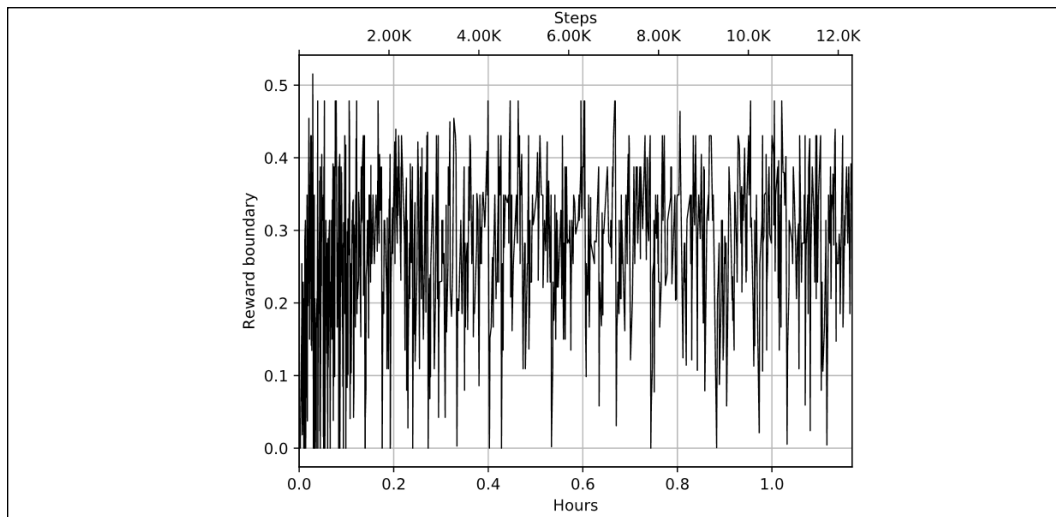
Figure 4.12: The reward boundary of the tweaked version

The final point to note here is the effect of slipperiness in the FrozenLake environment. Each of our actions with 33% probability is replaced with the 90° rotated one (the "up" action, for instance, will succeed with 0.33 probability and there will be a 0.33 chance that it will be replaced with the "left" action and 0.33 with the "right" action).

The nonslippery version is in `Chapter04/04_frozenlake_nonslippery.py`, and the only difference is in the environment creation (we need to peek into the core of Gym to create the instance of the environment with tweaked arguments):

```
env = gym.envs.toy_text.frozen_lake.FrozenLakeEnv(
    is_slippery=False)
env.spec = gym.spec("FrozenLake-v0")
env = gym.wrappers.TimeLimit(env, max_episode_steps=100)
env = DiscreteOneHotWrapper(env)
```

The effect is dramatic! The nonslippery version of the environment can be solved in 120-140 batch iterations, which is 100 times faster than the noisy environment:

```
rl_book_samples/Chapter04$ ./04_frozenlake_nonslippery.py
0: loss=1.379, reward_mean=0.010, reward_bound=0.000, batch=1
1: loss=1.375, reward_mean=0.010, reward_bound=0.000, batch=2
2: loss=1.359, reward_mean=0.010, reward_bound=0.000, batch=3
3: loss=1.361, reward_mean=0.010, reward_bound=0.000, batch=4
4: loss=1.355, reward_mean=0.000, reward_bound=0.000, batch=4
5: loss=1.342, reward_mean=0.010, reward_bound=0.000, batch=5
6: loss=1.353, reward_mean=0.020, reward_bound=0.000, batch=7
7: loss=1.351, reward_mean=0.040, reward_bound=0.000, batch=11
......
124: loss=0.484, reward_mean=0.680, reward_bound=0.000, batch=68
125: loss=0.373, reward_mean=0.710, reward_bound=0.430, batch=114
126: loss=0.305, reward_mean=0.690, reward_bound=0.478, batch=133
128: loss=0.413, reward_mean=0.790, reward_bound=0.478, batch=73
129: loss=0.297, reward_mean=0.810, reward_bound=0.478, batch=108 Solved!
```
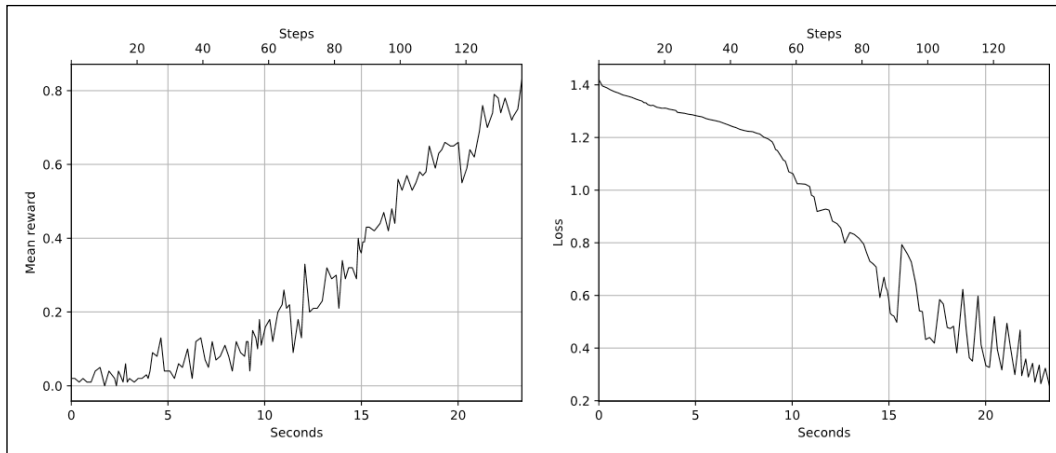
Figure 4.13: Reward (left) and loss (right) of the nonslippery version of FrozenLake
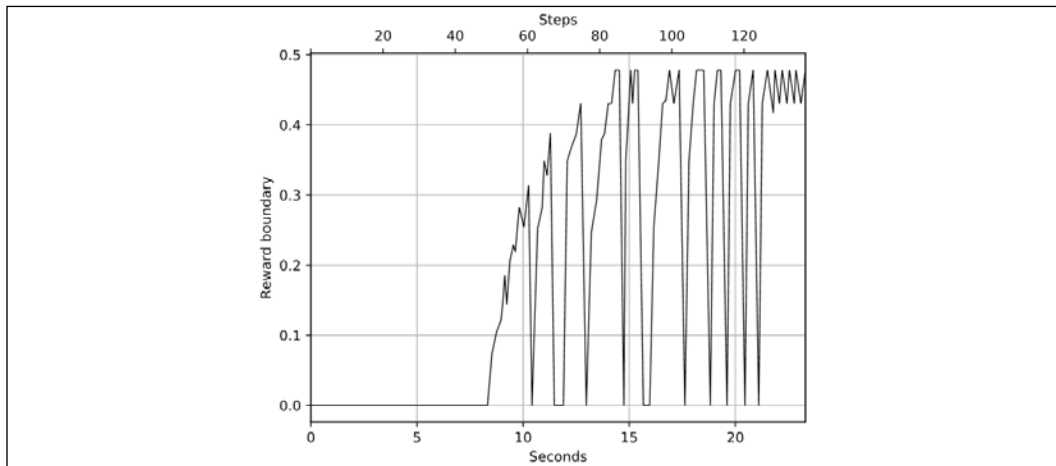
Figure 4.14: The reward boundary for the nonslippery version

The theoretical background of the cross-entropy method

This section is optional and included for readers who are interested in why the method works. If you wish, you can refer to the original paper on the cross-entropy method, which will be given at the end of the section.

The basis of the cross-entropy method lies in the importance sampling theorem, which states this:

$$\mathbb{E}_{x \sim p(x)}[H(x)] = \int_x p(x)H(x)dx = \int_x q(x)\frac{p(x)}{q(x)}H(x)dx = \mathbb{E}_{x \sim q(x)}\left[\frac{p(x)}{q(x)}H(x)\right]$$

In our RL case, $H(x)$ is a reward value obtained by some policy, x, and $p(x)$ is a distribution of all possible policies. We don't want to maximize our reward by searching all possible policies; instead we want to find a way to approximate $p(x)$ $H(x)$ by $q(x)$, iteratively minimizing the distance between them. The distance between two probability distributions is calculated by **Kullback-Leibler** (**KL**) divergence, which is as follows:

$$KL\big(p_1(x) \parallel p_2(x)\big) = \mathbb{E}_{x \sim p_1(x)} \log\frac{p_1(x)}{p_2(x)} = \mathbb{E}_{x \sim p_1(x)}[\log p_1(x)] - \mathbb{E}_{x \sim p_1(x)}[\log p_2(x)]$$

The first term in KL is called entropy and it doesn't depend on $p_2(x)$, so it could be omitted during the minimization. The second term is called **cross-entropy**, which is a very common optimization objective in deep learning.

Combining both formulas, we can get an iterative algorithm, which starts with $q_0(x) = p(x)$ and on every step improves. This is an approximation of $p(x)H(x)$ with an update:

$$q_{i+1}(x) = \underset{q_{i+1}(x)}{\arg\min} - \mathbb{E}_{x \sim q_i(x)}\frac{p(x)}{q_i(x)}H(x)\log q_{i+1}(x)$$

This is a generic cross-entropy method that can be significantly simplified in our RL case. Firstly, we replace our $H(x)$ with an indicator function, which is 1 when the reward for the episode is above the threshold and 0 when the reward is below. Our policy update will look like this:

$$\pi_{i+1}(a|s) = \underset{\pi_{i+1}}{\arg\min} - \mathbb{E}_{z \sim \pi_i(a|s)}[R(z) \geq \psi_i]\log \pi_{i+1}(a|s)$$

Strictly speaking, the preceding formula misses the normalization term, but it still works in practice without it. So, the method is quite clear: we sample episodes using our current policy (starting with some random initial policy) and minimize the negative log likelihood of the most successful samples and our policy.

There is a whole book written by *Dirk P. Kroese* that is dedicated to this method. A shorter description can be found in the *Cross-Entropy Method* paper by him (`https://people.smp.uq.edu.au/DirkKroese/ps/eormsCE.pdf`).

Summary

In this chapter, you became familiar with the cross-entropy method, which is simple but quite powerful, despite its limitations. We applied it to a CartPole environment (with huge success) and to FrozenLake (with much more modest success). In addition, we discussed the taxonomy of RL methods, which will be referenced many times during the rest of the book, as different approaches to RL problems have different properties, which influences their applicability.

This chapter ends the introductory part of the book. In the next part, we will switch to a more systematic study of RL methods and discuss the value-based family of methods. In upcoming chapters, we will explore more complex, but more powerful, tools of deep RL.

5
Tabular Learning and the Bellman Equation

In the previous chapter, you became acquainted with your first reinforcement learning (RL) algorithm, the cross-entropy method, along with its strengths and weaknesses. In this new part of the book, we will look at another group of methods that has much more flexibility and power: Q-learning. This chapter will establish the required background shared by those methods.

We will also revisit the FrozenLake environment and explore how new concepts fit with this environment and help us to address issues of its uncertainty.

In this chapter, we will:

- Review the value of the state and value of the action, and learn how to calculate them in simple cases
- Talk about the **Bellman equation** and how it establishes the optimal policy if we know the values
- Discuss the value iteration method and try it on the FrozenLake environment
- Do the same for the Q-learning method

Despite the simplicity of the environments in this chapter, it establishes the required preparation for deep Q-learning, which is a much more powerful and generic method.

Value, state, and optimality

You may remember our definition of the value of the state from *Chapter 1*, *What Is Reinforcement Learning?*. This is a very important notion and the time has come to explore it further.

This whole part of the book is built around the value and how to approximate it. We defined the value as an expected total reward (optionally discounted) that is obtainable from the state. In a formal way, the value of the state is

$V(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} r_t \gamma^t\right]$, where r_t is the local reward obtained at step t of the episode.

The total reward could be discounted with γ or not (the undiscounted case corresponds to $\gamma = 1$); it's up to us how to define it. The value is always calculated in terms of some policy that our agent follows. To illustrate this, let's consider a very simple environment with three states:

1. The agent's initial state.

2. The final state that the agent is in after executing action "right" from the initial state. The reward obtained from this is 1.

3. The final state that the agent is in after action "down." The reward obtained from this is 2:
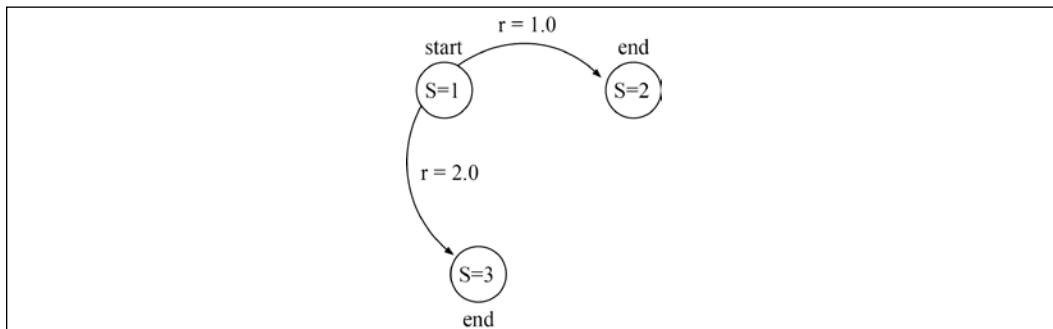
Figure 5.1: An example of an environment's states transition with rewards

The environment is always deterministic — every action succeeds and we always start from state 1. Once we reach either state 2 or state 3, the episode ends. Now, the question is, what's the value of state 1? This question is meaningless without information about our agent's behavior or, in other words, its policy. Even in a simple environment, our agent can have an infinite amount of behaviors, each of which will have its own value for state 1. Consider this example:

*   Agent always goes right

*   Agent always goes down

*   Agent goes right with a probability of 0.5 and down with a probability of 0.5

*   Agent goes right in 10% of cases and in 90% of cases executes the "down" action

To demonstrate how the value is calculated, let's do it for all the preceding policies:

- The value of state 1 in the case of the "always right" agent is **1.0** (every time it goes left, it obtains 1 and the episode ends)
- For the "always down" agent, the value of state 1 is **2.0**
- For the 50% right/50% down agent, the value is 1.0 * 0.5 + 2.0 * 0.5 = **1.5**
- For the 10% right/90% down agent, the value is 1.0 * 0.1 + 2.0 * 0.9 = **1.9**

Now, another question: what's the optimal policy for this agent? The goal of RL is to get as much total reward as possible. For this one-step environment, the total reward is equal to the value of state 1, which, obviously, is at the maximum at policy 2 (always down).

Unfortunately, such simple environments with an obvious optimal policy are not that interesting in practice. For interesting environments, the optimal policies are much harder to formulate and it's even harder to prove their optimality. However, don't worry; we are moving toward the point when we will be able to make computers learn the optimal behavior on their own.

From the preceding example, you may have a false impression that we should always take the action with the highest reward. In general, it's not that simple. To demonstrate this, let's extend our preceding environment with yet another state that is reachable from state 3. State 3 is no longer a terminal state but a transition to state 4, with a bad reward of –20. Once we have chosen the "down" action in state 1, this bad reward is unavoidable, as from state 3 we have only one exit. So, it's a trap for the agent, which has decided that "being greedy" is a good strategy.
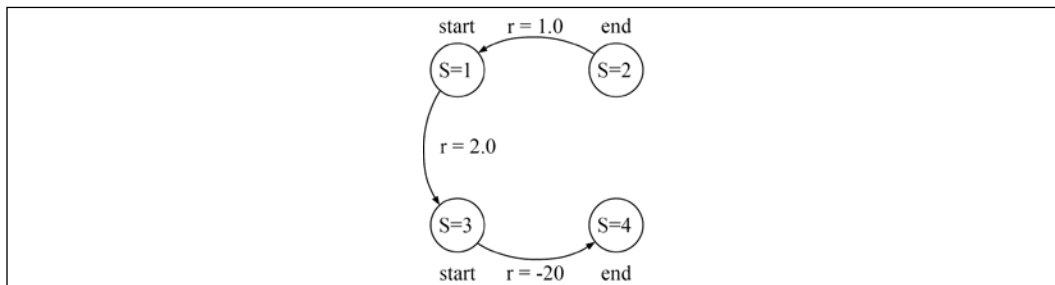
Figure 5.2: The same environment, with an extra state added

With that addition, our values for state 1 will be calculated this way:

- The "always right" agent is the same: **1.0**
- The "always down" agent gets 2.0 + (–20) = **–18**
- The 50%/50% agent gets 0.5 * 1.0 + 0.5 * (2.0 + (–20)) = **–8.5**
- The 10%/90% agent gets 0.1 * 1.0 + 0.9 * (2.0 + (–20)) = **–16.1**

So, the best policy for this new environment is now policy 1: always go right.

We spent some time discussing naïve and trivial environments so that you realize the complexity of this optimality problem and can appreciate the results of Richard Bellman better. Bellman was an American mathematician who formulated and proved his famous Bellman equation. We will talk about it in the next section.

The Bellman equation of optimality

To explain the Bellman equation, it's better to go a bit abstract. Don't be afraid; I'll provide concrete examples later to support your learning! Let's start with a deterministic case, when all our actions have a 100% guaranteed outcome. Imagine that our agent observes state s_0 and has N available actions. Every action leads to another state, $s_1 \dots s_N$, with a respective reward, $r_1 \dots r_N$. Also, assume that we know the values, V_i, of all states connected to state s_0. What will be the best course of action that the agent can take in such a state?
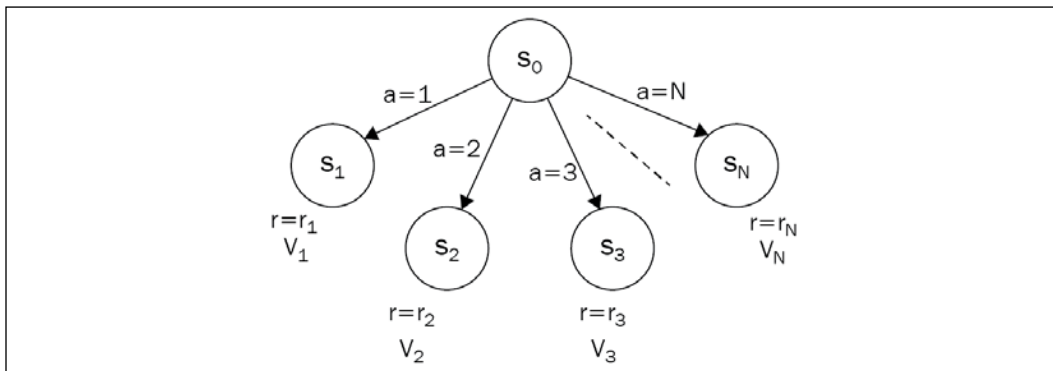
Figure 5.3: An abstract environment with N states reachable from the initial state

If we choose the concrete action, a_i, and calculate the value given to this action, then the value will be $V_0(a = a_i) = r_i + V_i$. So, to choose the best possible action, the agent needs to calculate the resulting values for every action and choose the maximum possible outcome. In other words, $V_0 = \max_{a \in 1 \dots N}(r_a + V_a)$. If we are using the discount factor γ, we need to multiply the value of the next state by gamma:

$$V_0 = \max_{a \in 1 \dots N}(r_a + \gamma V_a)$$

This may look very similar to our greedy example from the previous section, and, in fact, it is. However, there is one difference: when we act greedily, we do not only look at the immediate reward for the action, but at the immediate reward plus the long-term value of the state.

Bellman proved that with that extension, our behavior will get the best possible outcome. In other words, it will be optimal. So, the preceding equation is called the Bellman equation of value (for a deterministic case).

It's not very complicated to extend this idea for a stochastic case, when our actions have the chance of ending up in different states. What we need to do is calculate the expected value for every action, instead of just taking the value of the next state. To illustrate this, let's consider one single action available from state s_0, with three possible outcomes.
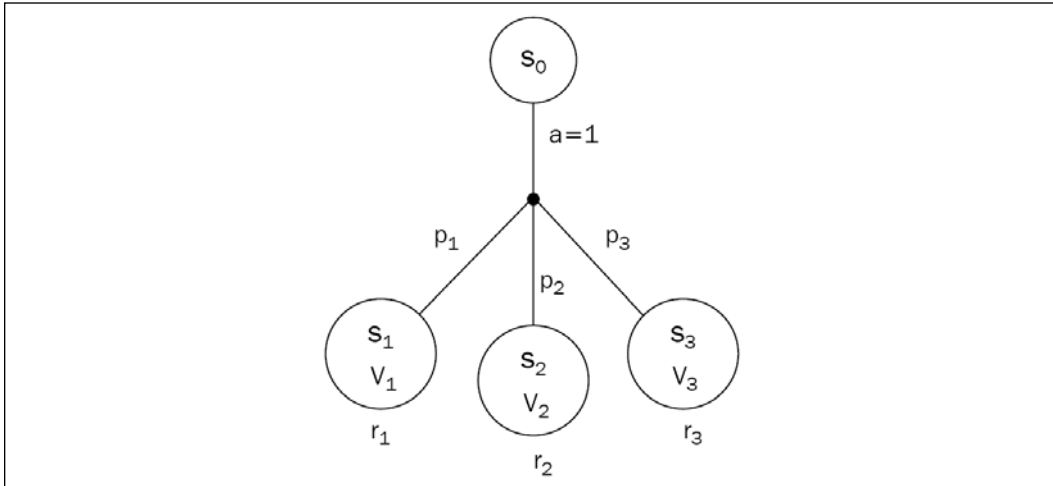
Figure 5.4: An example of the transition from the state in a stochastic case

Here we have one action, which can lead to three different states with different probabilities. With probability p_1, the action can end up in state s_1, with p_2 in state s_2, and with p_3 in state s_3 ($p_1 + p_2 + p_3 = 1$, of course). Every target state has its own reward (r_1, r_2, or r_3). To calculate the expected value after issuing action 1, we need to sum all values, multiplied by their probabilities:

$$V_0(a = 1) = p_1(r_1 + \gamma V_1) + p_2(r_2 + \gamma V_2) + p_3(r_3 + \gamma V_3)$$

Or, more formally:

$$V_0(a) = \mathbb{E}_{s \sim S}[r_{s,a} + \gamma V_s] = \sum_{s \in S} p_{a,0 \to s}(r_{s,a} + \gamma V_s)$$

By combining the Bellman equation, for a deterministic case, with a value for stochastic actions, we get the Bellman optimality equation for a general case:

$$V_0 = \max_{a \in A} \mathbb{E}_{s \sim S}[r_{s,a} + \gamma V_s] = \max_{a \in A} \sum_{s \in S} p_{a,0 \to s}(r_{s,a} + \gamma V_s)$$

> Note that $p_{a,i \to j}$ means the probability of action a, issued in state i, ending up in state j.

The interpretation is still the same: the optimal value of the state is equal to the action, which gives us the maximum possible expected immediate reward, plus the discounted long-term reward for the next state. You may also notice that this definition is recursive: the value of the state is defined via the values of the immediately reachable states. This recursion may look like cheating: we define some value, pretending that we already know it. However, this is a very powerful and common technique in computer science and even in math in general (proof by induction is based on the same trick). This Bellman equation is a foundation not only in RL but also in much more general dynamic programming, which is a widely used method for solving practical optimization problems.

These values not only give us the best reward that we can obtain, but they basically give us the optimal policy to obtain that reward: if our agent knows the value for every state, then it automatically knows how to gather all this reward. Thanks to Bellman's optimality proof, at every state the agent ends up in, it needs to select the action with the maximum expected reward, which is a sum of the immediate reward and the one-step discounted long-term reward. That's it. So, those values are really useful to know. Before you get familiar with a practical way to calculate them, I need to introduce one more mathematical notation. It's not as fundamental as the value of the state, but we need it for our convenience.

The value of the action

To make our life slightly easier, we can define different quantities, in addition to the value of the state, $V(s)$, as the value of the action, $Q(s, a)$. Basically, this equals the total reward we can get by executing action a in state s and can be defined via $V(s)$. Being a much less fundamental entity than $V(s)$, this quantity gave a name to the whole family of methods called Q-learning, because it is more convenient.

In these methods, our primary objective is to get values of Q for every pair of state and action.

$$Q(s,a) = \mathbb{E}_{s' \sim S}[r(s,a) + \gamma V(s')] = \sum_{s' \in S} p_{a,s \to s'}\big(r(s,a) + \gamma V(s')\big)$$

Q for this state, s, and action, a, equals the expected immediate reward and the discounted long-term reward of the destination state. We also can define $V(s)$ via $Q(s, a)$:

$$V(s) = \max_{a \in A} Q(s,a)$$

This just means that the value of some state equals to the value of the maximum action we can execute from this state. Finally, we can express $Q(s, a)$ recursively (which will be used in *Chapter 6, Deep Q-Networks*:

$$Q(s,a) = r(s,a) + \gamma \max_{a' \in A} Q(s',a')$$

In the preceding formula, the index on the immediate reward, s, a, depends on environment details. If immediate reward is given to us after executing a particular action, a, from state s, index (s, a) is used and the formula is exactly as shown above. But if reward is provided for reaching some state, s', via action a', the reward will have the index (s', a') and will need to be moved into the max operator. That difference is not very significant from a mathematical point of view, but it could be important during the implementation of the methods. The first situation is more common, so, we will stick to the preceding formula.

To give you a concrete example, let's consider an environment that is similar to FrozenLake, but has a much simpler structure: we have one initial state (s_0) surrounded by four target states, $s_1, s_2, s_3, s_4,$ with different rewards.
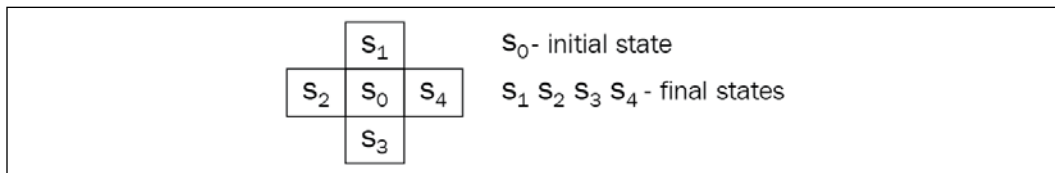
Figure 5.5: A simplified grid-like environment

Every action is probabilistic in the same way as in FrozenLake: with a 33% chance that our action will be executed without modifications, but with a 33% chance that we will slip to the left, relatively, of our target cell and a 33% chance that we will slip to the right. For simplicity, we use discount factor $\gamma = 1$.
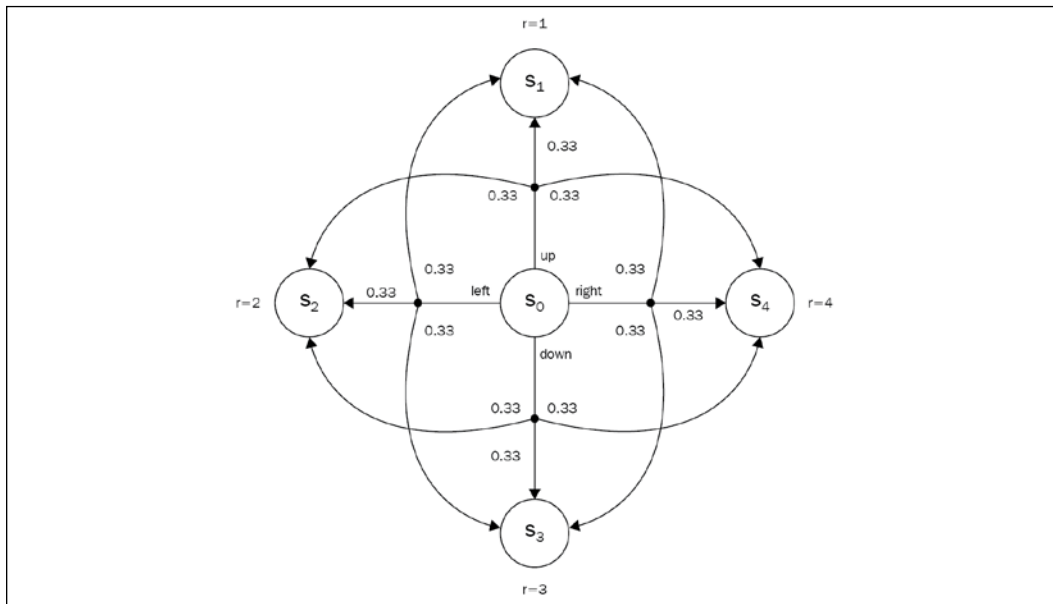
Figure 5.6: A transition diagram of the grid environment

Let's calculate the values of the actions to begin with. Terminal states s_1 ... s_4 have no outbound connections, so Q for those states is zero for all actions. Due to this, the values of the terminal states are equal to their immediate reward (once we get there, our episode ends without any subsequent states): $V_1 = 1$, $V_2 = 2$, $V_3 = 3$, $V_4 = 4$.

The values of the actions for state 0 are a bit more complicated. Let's start with the "up" action. Its value, according to the definition, is equal to the expected sum of the immediate reward plus the long-term value for subsequent steps. We have no subsequent steps for any possible transition for the "up" action:

$$Q(s_0, up) = 0.33 \cdot V_1 + 0.33 \cdot V_2 + 0.33 \cdot V_4 = 0.33 \cdot 1 + 0.33 \cdot 2 + 0.33 \cdot 4 = 2.31$$

Repeating this for the rest of the s_0 actions results in the following:

$$
\begin{aligned}
Q(s_0, left) &= 0.33 \cdot V_1 + 0.33 \cdot V_2 + 0.33 \cdot V_3 &= 1.98 \\
Q(s_0, right) &= 0.33 \cdot V_4 + 0.33 \cdot V_1 + 0.33 \cdot V_3 &= 2.64 \\
Q(s_0, down) &= 0.33 \cdot V_3 + 0.33 \cdot V_2 + 0.33 \cdot V_4 &= 2.97
\end{aligned}
$$

The final value for state s_0 is the maximum of those actions' values, which is 2.97.

Q-values are much more convenient in practice, as for the agent, it's much simpler to make decisions about actions based on Q than on V. In the case of Q, to choose the action based on the state, the agent just needs to calculate Q for all available actions using the current state and choose the action with the largest value of Q. To do the same using values of the states, the agent needs to know not only values, but also probabilities for transitions. In practice, we rarely know them in advance, so the agent needs to estimate transition probabilities for every action and state pair. Later in this chapter, you will see this in practice by solving the FrozenLake environment both ways. However, to be able to do this, we have one important thing still missing: a general way to calculate those Vs and Qs.

The value iteration method

In the simplistic example you just saw, to calculate the values of the states and actions, we exploited the structure of the environment: we had no loops in transitions, so we could start from terminal states, calculate their values, and then proceed to the central state. However, just one loop in the environment builds an obstacle in our approach. Let's consider such an environment with two states:
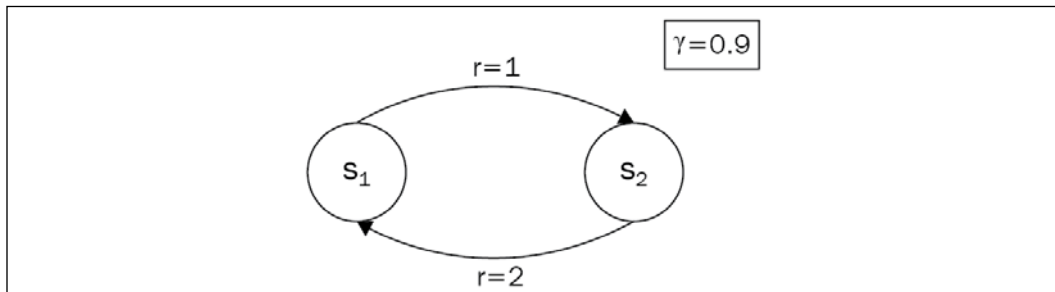
Figure 5.7: A sample environment with a loop in the transition diagram

We start from state s_1, and the only action we can take leads us to state s_2. We get the reward $r = 1$, and the only transition from s_2 is an action, which brings us back to s_1. So, the life of our agent is an infinite sequence of states $[s_1, s_2, s_1, s_2, s_1, s_2, s_1, s_2, ...]$. To deal with this infinity loop, we can use a discount factor: $\gamma = 0.9$. Now, the question is, what are the values for both the states? The answer is not very complicated, in fact. Every transition from s_1 to s_2 gives us a reward of 1 and every back transition gives us 2. So, our sequence of rewards will be $[1, 2, 1, 2, 1, 2, 1, 2, ...]$. As there is only one action available in every state, our agent has no choice, so we can omit the max operation in formulas (there is only one alternative).

The value for every state will be equal to the infinite sum:

$$V(s_1) = 1 + \gamma\left(2 + \gamma\left(1 + \gamma(2 + \cdots)\right)\right) = \sum_{i=0}^{\infty} 1\gamma^{2i} + 2\gamma^{2i+1}$$

$$V(s_2) = 2 + \gamma\left(1 + \gamma\left(2 + \gamma(1 + \cdots)\right)\right) = \sum_{i=0}^{\infty} 2\gamma^{2i} + 1\gamma^{2i+1}$$

Strictly speaking, we can't calculate the exact values for our states, but with $\gamma = 0.9$, the contribution of every transition quickly decreases over time. For example, after 10 steps, $\gamma^{10} = 0.9^{10} = 0.349$, but after 100 steps, it becomes just 0.0000266. Due to this, we can stop after 50 iterations and still get quite a precise estimation.

```
>>> sum([0.9**(2*i) + 2*(0.9**(2*i+1)) for i in range(50)])
14.736450674121663
>>> sum([2*(0.9**(2*i)) + 0.9**(2*i+1) for i in range(50)])
15.262752483911719
```

The preceding example can be used to get the gist of a more general procedure called the **value iteration algorithm**. This allows us to numerically calculate the values of the states and values of the actions of **Markov decision processes** (**MDPs**) with known transition probabilities and rewards. The procedure (for values of the states) includes the following steps:

1. Initialize the values of all states, V_i, to some initial value (usually zero)
2. For every state, s, in the MDP, perform the Bellman update:

$$V_s \leftarrow \max_a \sum_{s'} p_{a,s \rightarrow s'}\left(r_{s,a} + \gamma V_{s'}\right)$$

3. Repeat step 2 for some large number of steps or until changes become too small

In the case of action values (that is, Q), only minor modifications to the preceding procedure are required:

1. Initialize every $Q_{s,a}$ to zero
2. For every state, s, and action, a, in this state, perform this update:

$$Q_{s,a} \leftarrow \sum_{s'} p_{a,s \rightarrow s'}\left(r_{s,a} + \gamma \max_{a'} Q_{s',a'}\right)$$

3. Repeat step 2

Okay, so that's the theory. In practice, this method has several obvious limitations. First of all, our state space should be discrete and small enough to perform multiple iterations over all states. This is not an issue for FrozenLake-4x4 and even for FrozenLake-8x8 (it exists in Gym as a more challenging version), but for CartPole, it's not totally clear what to do. Our observation for CartPole is four float values, which represent some physical characteristics of the system. Even a small difference in those values could have an influence on the state's value. A potential solution for that could be discretization of our observation's values; for example, we can split the observation space of CartPole into bins and treat every bin as an individual discrete state in space. However, this will create lots of practical problems, such as how large bin intervals should be and how much data from the environment we will need to estimate our values. I will address this issue in subsequent chapters, when we get to the usage of neural networks in Q-learning.

The second practical problem arises from the fact that we rarely know the transition probability for the actions and rewards matrix. Remember the interface provided by Gym to the agent's writer: we observe the state, decide on an action, and only then do we get the next observation and reward for the transition. We don't know (without peeking into Gym's environment code) what the probability is of getting into state s_1 from state s_0 by issuing action a_0.

What we do have is just the history from the agent's interaction with the environment. However, in Bellman's update, we need both a reward for every transition and the probability of this transition. So, the obvious answer to this issue is to use our agent's experience as an estimation for both unknowns. Rewards could be used as they are. We just need to remember what reward we got on the transition from s_0 to s_1 using action a, but to estimate probabilities, we need to maintain counters for every tuple (s_0, s_1, a) and normalize them.

Okay, now let's look at how the value iteration method will work for FrozenLake.

Value iteration in practice

The complete example is in `Chapter05/01_frozenlake_v_iteration.py`. The central data structures in this example are as follows:

- **Reward table**: A dictionary with the composite key "source state" + "action" + "target state". The value is obtained from the immediate reward.

- **Transitions table**: A dictionary keeping counters of the experienced transitions. The key is the composite "state" + "action", and the value is another dictionary that maps the target state into a count of times that we have seen it. For example, if in state 0 we execute action 1 ten times, after three times it will lead us to state 4 and after seven times to state 5.

> The entry with the key `(0, 1)` in this table will be a `dict` with contents `{4: 3, 5: 7}`. We can use this table to estimate the probabilities of our transitions.

- **Value table**: A dictionary that maps a state into the calculated value of this state.

The overall logic of our code is simple: in the loop, we play 100 random steps from the environment, populating the reward and transition tables. After those 100 steps, we perform a value iteration loop over all states, updating our value table. Then we play several full episodes to check our improvements using the updated value table. If the average reward for those test episodes is above the 0.8 boundary, then we stop training. During the test episodes, we also update our reward and transition tables to use all data from the environment.

Okay, so let's come to the code. In the beginning, we import the used packages and define constants:

```
import gym
import collections
from tensorboardX import SummaryWriter

ENV_NAME = "FrozenLake-v0"
GAMMA = 0.9
TEST_EPISODES = 20
```

Then we define the `Agent` class, which will keep our tables and contain functions that we will be using in the training loop:

```
class Agent:
    def __init__(self):
        self.env = gym.make(ENV_NAME)
        self.state = self.env.reset()
        self.rewards = collections.defaultdict(float)
        self.transits = collections.defaultdict(
            collections.Counter)
        self.values = collections.defaultdict(float)
```

In the class constructor, we create the environment that we will be using for data samples, obtain our first observation, and define tables for rewards, transitions, and values.

```
    def play_n_random_steps(self, count):
        for _ in range(count):
            action = self.env.action_space.sample()
            new_state, reward, is_done, _ = self.env.step(action)
```

```
self.rewards[(self.state, action, new_state)] = reward
self.transits[(self.state, action)][new_state] += 1
self.state = self.env.reset() \
    if is_done else new_state
```

This function is used to gather random experience from the environment and update the reward and transition tables. Note that we don't need to wait for the end of the episode to start learning; we just perform *N* steps and remember their outcomes. This is one of the differences between value iteration and the cross-entropy method, which can learn only on full episodes.

The next function calculates the value of the action from the state using our transition, reward, and values tables. We will use it for two purposes: to select the best action to perform from the state and to calculate the new value of the state on value iteration. Its logic is illustrated in the following diagram.

We do the following:

1. We extract transition counters for the given state and action from the transition table. Counters in this table have a form of `dict`, with target states as the key and a count of experienced transitions as the value. We sum all counters to obtain the total count of times we have executed the action from the state. We will use this total value later to go from an individual counter to probability.

2. Then we iterate every target state that our action has landed on and calculate its contribution to the total action value using the Bellman equation. This contribution is equal to immediate reward plus discounted value for the target state. We multiply this sum to the probability of this transition and add the result to the final action value.

transit $[(s,a)] = \{s_1:c_1, s_2:c_2\}$

total $= c_1 + c_2$

$$Q(s,a) = \frac{c_1}{total}(r_{s_1} + \gamma V(s_1)) + \frac{c_2}{total}(r_{s_2} + \gamma V(s_2))$$
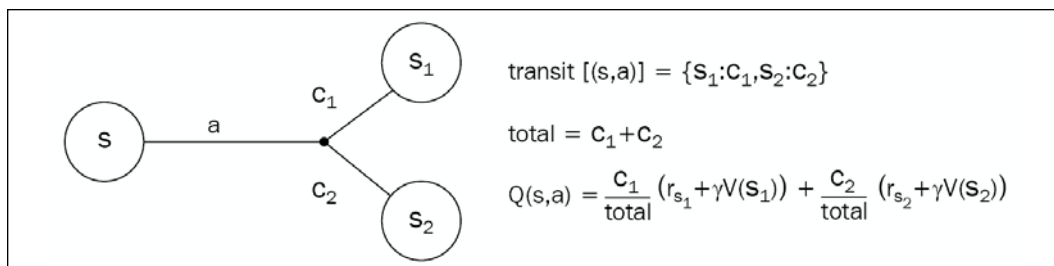
Figure 5.8: The calculation of the state's value

In our diagram, we have an illustration of a calculation of the value for state *s* and action *a*. Imagine that, during our experience, we have executed this action several times $(c_1 + c_2)$ and it ends up in one of two states, s_1 or s_2. How many times we have switched to each of these states is stored in our transition table as `dict` $\{s_1: c_1, s_2: c_2\}$.

Then, the approximate value for the state and action, $Q(s, a)$, will be equal to the probability of every state, multiplied to the value of the state. From the Bellman equation, this equals the sum of the immediate reward and the discounted long-term state value.

```
def calc_action_value(self, state, action):
    target_counts = self.transits[(state, action)]
    total = sum(target_counts.values())
    action_value = 0.0
    for tgt_state, count in target_counts.items():
        reward = self.rewards[(state, action, tgt_state)]
        val = reward + GAMMA * self.values[tgt_state]
        action_value += (count / total) * val
    return action_value
```

The next function uses the function that I just described to make a decision about the best action to take from the given state. It iterates over all possible actions in the environment and calculates the value for every action. The action with the largest value wins and is returned as the action to take. This action selection process is deterministic, as the `play_n_random_steps()` function introduces enough exploration. So, our agent will behave greedily in regard to our value approximation.

```
def select_action(self, state):
    best_action, best_value = None, None
    for action in range(self.env.action_space.n):
        action_value = self.calc_action_value(state, action)
        if best_value is None or best_value < action_value:
            best_value = action_value
            best_action = action
    return best_action
```

The `play_episode()` function uses `select_action()` to find the best action to take and plays one full episode using the provided environment. This function is used to play test episodes, during which we don't want to mess with the current state of the main environment used to gather random data. So, we use the second environment passed as an argument. The logic is very simple and should be already familiar to you: we just loop over states accumulating reward for one episode.

```
def play_episode(self, env):
    total_reward = 0.0
    state = env.reset()
    while True:
        action = self.select_action(state)
        new_state, reward, is_done, _ = env.step(action)
        self.rewards[(state, action, new_state)] = reward
```

```
                    self.transits[(state, action)][new_state] += 1
                    total_reward += reward
                    if is_done:
                        break
                    state = new_state
                return total_reward
```

The final method of the `Agent` class is our value iteration implementation and it is surprisingly simple, thanks to the preceding functions. What we do is just loop over all states in the environment, then for every state, we calculate the values for the states reachable from it, obtaining candidates for the value of the state. Then we update the value of our current state with the maximum value of the action available from the state.

```
        def value_iteration(self):
            for state in range(self.env.observation_space.n):
                state_values = [
                    self.calc_action_value(state, action)
                    for action in range(self.env.action_space.n)
                ]
                self.values[state] = max(state_values)
```

That's all of our agent's methods, and the final piece is a training loop and the monitoring of the code:

```
    if __name__ == "__main__":
        test_env = gym.make(ENV_NAME)
        agent = Agent()
        writer = SummaryWriter(comment="-v-iteration")
```

We create the environment that we will be using for testing, the `Agent` class instance, and the summary writer for TensorBoard.

```
        iter_no = 0
        best_reward = 0.0
        while True:
            iter_no += 1
            agent.play_n_random_steps(100)
            agent.value_iteration()
```

The two lines in the preceding code snippet are the key piece in the training loop. First, we perform 100 random steps to fill our reward and transition tables with fresh data, and then we run value iteration over all states. The rest of the code plays test episodes using the value table as our policy, then writes data into TensorBoard, tracks the best average reward, and checks for the training loop stop condition.

```
        reward = 0.0
        for _ in range(TEST_EPISODES):
            reward += agent.play_episode(test_env)
        reward /= TEST_EPISODES
        writer.add_scalar("reward", reward, iter_no)
        if reward > best_reward:
            print("Best reward updated %.3f -> %.3f" % (
                best_reward, reward))
            best_reward = reward
        if reward > 0.80:
            print("Solved in %d iterations!" % iter_no)
            break
    writer.close()
```

Okay, let's run our program:

```
rl_book_samples/Chapter05$ ./01_frozenlake_v_iteration.py
[2017-10-13 11:39:37,778] Making new env: FrozenLake-v0
[2017-10-13 11:39:37,988] Making new env: FrozenLake-v0
Best reward updated 0.000 -> 0.150
Best reward updated 0.150 -> 0.500
Best reward updated 0.500 -> 0.550
Best reward updated 0.550 -> 0.650
Best reward updated 0.650 -> 0.800
Best reward updated 0.800 -> 0.850
Solved in 36 iterations!
```

Our solution is stochastic, and my experiments usually required from 12 to 100 iterations to reach a solution, but in all cases, it took less than a second to find a good policy that could solve the environment in 80% of runs. If you remember how many hours were required to achieve a 60% success ratio using the cross-entropy method, then you can understand that this is a major improvement. There are several reasons for that.

First of all, the stochastic outcome of our actions, plus the length of the episodes (six to 10 steps on average), makes it hard for the cross-entropy method to understand what was done right in the episode and which step was a mistake. Value iteration works with individual values of the state (or action) and incorporates the probabilistic outcome of actions naturally by estimating probability and calculating the expected value. So, it's much simpler for value iteration and requires much less data from the environment (which is called **sample efficiency** in RL).

The second reason is the fact that value iteration doesn't need full episodes to start learning. In an extreme case, we can start updating our values just from a single example. However, for FrozenLake, due to the reward structure (we get 1 only after successfully reaching the target state), we still need to have at least one successful episode to start learning from a useful value table, which may be challenging to achieve in more complex environments. For example, you can try switching the existing code to a larger version of FrozenLake, which has the name **FrozenLake8x8-v0**. The larger version of FrozenLake can take from 150 to 1,000 iterations to solve, and, according to TensorBoard charts, most of the time it waits for the first successful episode, then it very quickly reaches convergence. Below are two charts: the first one shows reward dynamics during training on FrozenLake-4x4 and the second is for the 8x8 version.
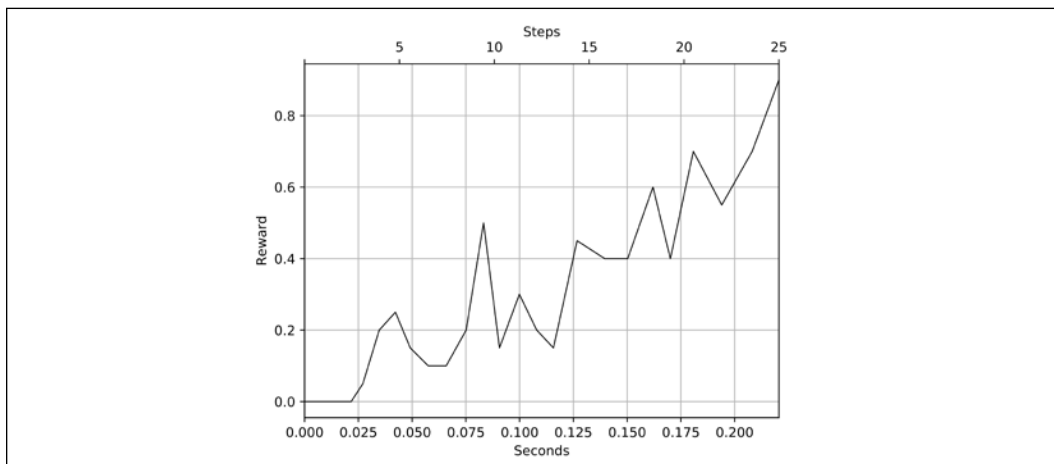
Figure 5.9: The reward dynamics for FrozenLake-4x4
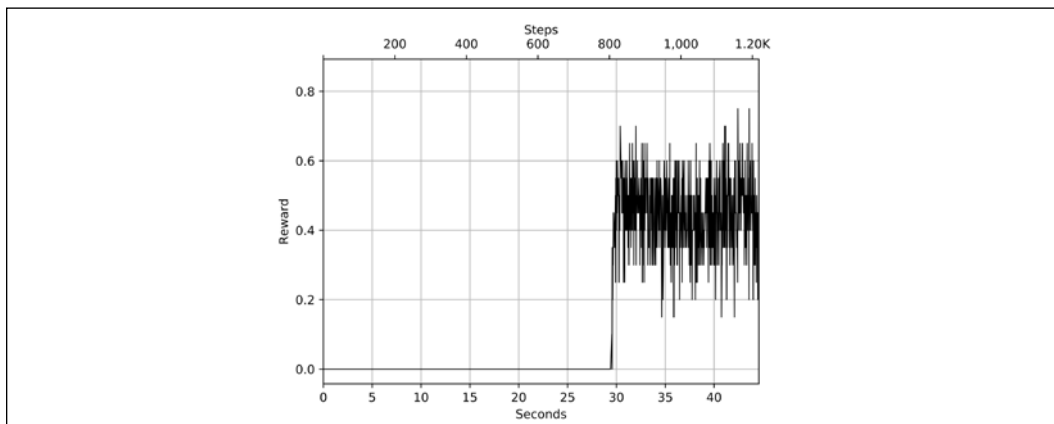
Figure 5.10: The reward dynamics on 8x8

Now it's time to compare the code that learns the values of the states, as we just discussed, with the code that learns the values of the actions.

Q-learning for FrozenLake

The whole example is in the `Chapter05/02_frozenlake_q_iteration.py` file, and the difference is really minor. The most obvious change is to our value table. In the previous example, we kept the value of the state, so the key in the dictionary was just a state. Now we need to store values of the Q-function, which has two parameters: state and action, so the key in the value table is now a composite.

The second difference is in our `calc_action_value()` function. We just don't need it anymore, as our action values are stored in the value table.

Finally, the most important change in the code is in the agent's `value_iteration()` method. Before, it was just a wrapper around the `calc_action_value()` call, which did the job of Bellman approximation. Now, as this function has gone and been replaced by a value table, we need to do this approximation in the `value_iteration()` method.

Let's look at the code. As it's almost the same, I will jump directly to the most interesting `value_iteration()` function:

```python
def value_iteration(self):
    for state in range(self.env.observation_space.n):
        for action in range(self.env.action_space.n):
            action_value = 0.0
            target_counts = self.transits[(state, action)]
            total = sum(target_counts.values())
            for tgt_state, count in target_counts.items():
                key = (state, action, tgt_state)
                reward = self.rewards[key]
                best_action = self.select_action(tgt_state)
                val = reward + GAMMA * \
                    self.values[(tgt_state, best_action)]
                action_value += (count / total) * val
            self.values[(state, action)] = action_value
```

The code is very similar to `calc_action_value()` in the previous example and, in fact, it does almost the same thing. For the given state and action, it needs to calculate the value of this action using statistics about target states that we have reached with the action. To calculate this value, we use the Bellman equation and our counters, which allow us to approximate the probability of the target state. However, in Bellman's equation, we have the value of the state; now, we need to calculate it differently.

Before, we had it stored in the value table (as we approximated the value of the states), so we just took it from this table. We can't do this anymore, so we have to call the `select_action` method, which will choose for us the action with the largest Q-value, and then we take this Q-value as the value of the target state. Of course, we can implement another function that can calculate for us this value of the state, but `select_action` does almost everything we need, so we will reuse it here.

There is another piece of this example that I'd like to emphasize here. Let's look at our `select_action` method:

```
def select_action(self, state):
    best_action, best_value = None, None
    for action in range(self.env.action_space.n):
        action_value = self.values[(state, action)]
        if best_value is None or best_value < action_value:
            best_value = action_value
            best_action = action
    return best_action
```

As I said, we don't have the `calc_action_value` method anymore, so, to select an action, we just iterate over the actions and look up their values in our values table. It could look like a minor improvement, but if you think about the data that we used in `calc_action_value`, it may become obvious why the learning of the Q-function is much more popular in RL than the learning of the V-function.

Our `calc_action_value` function uses both information about the reward and probabilities. It's not a huge problem for the value iteration method, which relies on this information during training. However, in the next chapter, you will learn about the value iteration method extension, which doesn't require probability approximation, but just takes it from the environment samples. For such methods, this dependency on probability adds an extra burden for the agent. In the case of Q-learning, what the agent needs to make the decision is just Q-values.

I don't want to say that V-functions are completely useless, because they are an essential part of the actor-critic method, which we will talk about in part three of this book. However, in the area of value learning, Q-functions is the definite favorite. With regards to convergence speed, both our versions are almost identical (but the Q-learning version requires four times more memory for the value table).

```
rl_book_samples/Chapter05$ ./02_frozenlake_q_iteration.py
[2017-10-13 12:38:56,658] Making new env: FrozenLake-v0
[2017-10-13 12:38:56,863] Making new env: FrozenLake-v0
Best reward updated 0.000 -> 0.050
Best reward updated 0.050 -> 0.200
```

```
Best reward updated 0.200 -> 0.350
Best reward updated 0.350 -> 0.700
Best reward updated 0.700 -> 0.750
Best reward updated 0.750 -> 0.850
Solved in 22 iterations!
```

Summary

My congratulations, you have made another step towards understanding modern, state-of-the-art RL methods! In this chapter, you learned about some very important concepts that are widely used in deep RL: the value of the state, the value of the action, and the Bellman equation in various forms.

We also covered the value iteration method, which is a very important building block in the area of Q-learning. Finally, you got to know how value iteration can improve our FrozenLake solution.

In the next chapter, you will learn about deep Q-networks, which started the deep RL revolution in 2013 by beating humans on lots of Atari 2600 games.

6

Deep Q-Networks

In *Chapter 5*, *Tabular Learning and the Bellman Equation*, you became familiar with the Bellman equation and the practical method of its application called *value iteration*. This approach allowed us to significantly improve our speed and convergence in the FrozenLake environment, which is promising, but can we go further? In this chapter, we will apply the same approach to problems of much greater complexity: arcade games from the Atari 2600 platform, which are the de facto benchmark of the reinforcement learning (RL) research community.

To deal with this new and more challenging goal, in this chapter, we will:

- Talk about problems with the value iteration method and consider its variation, called *Q-learning*.
- Apply Q-learning to so-called grid world environments, which is called **tabular Q-learning**.
- Discuss Q-learning in conjunction with neural networks (NNs). This combination has the name **deep Q-network** (**DQN**).

At the end of the chapter, we will reimplement a DQN algorithm from the famous paper *Playing Atari with Deep Reinforcement Learning* by *V. Mnih* and others, which was published in 2013 and started a new era in RL development.

Real-life value iteration

The improvements that we got in the FrozenLake environment by switching from the cross-entropy method to the value iteration method are quite encouraging, so it's tempting to apply the value iteration method to more challenging problems. However, let's first look at the assumptions and limitations that our value iteration method has.

We will start with a quick recap of the method. On every step, the value iteration method does a loop on all states, and for every state, it performs an update of its value with a Bellman approximation. The variation of the same method for Q-values (values for actions) is almost the same, but we approximate and store values for every state and action. So, what's wrong with this process?

The first obvious problem is the count of environment states and our ability to iterate over them. In value iteration, we assume that we know all states in our environment in advance, can iterate over them, and can store value approximations associated with them. It's definitely true for the simple grid world environment of FrozenLake, but what about other tasks?

First, let's try to understand how scalable the value iteration approach is, or, in other words, how many states we can easily iterate over in every loop. Even a moderate-sized computer can keep several billion float values in memory (8.5 billion in 32 GB of RAM), so the memory required for value tables doesn't look like a huge constraint. Iteration over billions of states and actions will be more central processing unit (CPU) intensive, but not an insurmountable problem.

Nowadays, we have multicore systems that are mostly idle. The real problem is the number of samples required to get good approximations for state transition dynamics. Imagine that you have some environment with, say, a billion states (which corresponds approximately to a FrozenLake of size 31600×31600). To calculate even a rough approximation for every state of this environment, we would need hundreds of billions of transitions evenly distributed over our states, which is not practical.

To give you an example of an environment with an even larger number of potential states, let's consider the Atari 2600 game console again. This was very popular in the 1980s, and many arcade-style games were available for it. The Atari console is archaic by today's gaming standards, but its games provide an excellent set of RL problems that humans can master fairly quickly, yet still are challenging for computers. Not surprisingly, this platform (using an emulator, of course) is a very popular benchmark within RL research, as I mentioned.

Let's calculate the state space for the Atari platform. The resolution of the screen is 210×160 pixels, and every pixel has one of 128 colors. So, every frame of the screen has $210 \times 160 = 33600$ pixels and the total number of different screens possible is 128^{33600}, which is slightly more than 10^{70802}. If we decide to just enumerate all possible states of the Atari once, it will take billions of billions of years even for the fastest supercomputer. Also, 99(.9)% of this job will be a waste of time, as most of the combinations will never be shown during even long gameplay, so we will never have samples of those states. However, the value iteration method wants to iterate over them just in case.

Another problem with the value iteration approach is that it limits us to discrete action spaces. Indeed, both $Q(s, a)$ and $V(s)$ approximations assume that our actions are a mutually exclusive discrete set, which is not true for continuous control problems where actions can represent continuous variables, such as the angle of a steering wheel, the force on an actuator, or the temperature of a heater. This issue is much more challenging than the first, and we will talk about it in the last part of the book, in chapters dedicated to continuous action space problems. For now, let's assume that we have a discrete count of actions and that this count is not very large (orders of 10s). How should we handle the state space size issue?

Tabular Q-learning

First of all, do we really need to iterate over every state in the state space? We have an environment that can be used as a source of real-life samples of states. If some state in the state space is not shown to us by the environment, why should we care about its value? We can use states obtained from the environment to update the values of states, which can save us a lot of work.

This modification of the value iteration method is known as Q-learning, as mentioned earlier, and for cases with explicit state-to-value mappings, it has the following steps:

1. Start with an empty table, mapping states to values of actions.
2. By interacting with the environment, obtain the tuple *s*, *a*, *r*, *s'* (state, action, reward, and the new state). In this step, you need to decide which action to take, and there is no single proper way to make this decision. We discussed this problem as *exploration versus exploitation* in *Chapter 1*, *What Is Reinforcement Learning?* and will talk a lot about it in this chapter.
3. Update the $Q(s, a)$ value using the Bellman approximation:

$$Q(s, a) \leftarrow r + \gamma \max_{a' \in A} Q(s', a')$$

4. Repeat from step 2.

As in value iteration, the end condition could be some threshold of the update, or we could perform test episodes to estimate the expected reward from the policy.

Another thing to note here is how to update the Q-values. As we take samples from the environment, it's generally a bad idea to just assign new values on top of existing values, as training can become unstable.

What is usually done in practice is updating the $Q(s, a)$ with approximations using a "blending" technique, which is just averaging between old and new values of Q using learning rate α with a value from 0 to 1:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \left(r + \gamma \max_{a' \in A} Q(s', a') \right)$$

This allows values of Q to converge smoothly, even if our environment is noisy. The final version of the algorithm is here:

1. Start with an empty table for $Q(s, a)$.

2. Obtain (s, a, r, s') from the environment.

3. Make a Bellman update: $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \left(r + \gamma \max_{a' \in A} Q(s', a') \right)$.

4. Check convergence conditions. If not met, repeat from step 2.

As mentioned earlier, this method is called tabular Q-learning, as we keep a table of states with their Q-values. Let's try it on our FrozenLake environment. The whole example code is in `Chapter06/01_frozenlake_q_learning.py`.

```python
import gym
import collections
from tensorboardX import SummaryWriter

ENV_NAME = "FrozenLake-v0"
GAMMA = 0.9
ALPHA = 0.2
TEST_EPISODES = 20

class Agent:
    def __init__(self):
        self.env = gym.make(ENV_NAME)
        self.state = self.env.reset()
        self.values = collections.defaultdict(float)
```

In the beginning, we import packages and define constants. The new thing here is the value of α, which will be used as the learning rate in the value update. The initialization of our `Agent` class is simpler now, as we don't need to track the history of rewards and transition counters, just our value table. This will make our memory footprint smaller, which is not a big issue for FrozenLake, but can be critical for larger environments.

```python
    def sample_env(self):
        action = self.env.action_space.sample()
        old_state = self.state
```

```
new_state, reward, is_done, _ = self.env.step(action)
self.state = self.env.reset() if is_done else new_state
return old_state, action, reward, new_state
```

The preceding method is used to obtain the next transition from the environment. We sample a random action from the action space and return the tuple of the old state, action taken, reward obtained, and the new state. The tuple will be used in the training loop later.

```
def best_value_and_action(self, state):
    best_value, best_action = None, None
    for action in range(self.env.action_space.n):
        action_value = self.values[(state, action)]
        if best_value is None or best_value < action_value:
            best_value = action_value
            best_action = action
    return best_value, best_action
```

The next method receives the state of the environment and finds the best action to take from this state by taking the action with the largest value that we have in the table. If we don't have the value associated with the state and action pair, then we take it as zero. This method will be used two times: first, in the test method that plays one episode using our current values table (to evaluate our policy's quality), and second, in the method that performs the value update to get the value of the next state.

```
def value_update(self, s, a, r, next_s):
    best_v, _ = self.best_value_and_action(next_s)
    new_v = r + GAMMA * best_v
    old_v = self.values[(s, a)]
    self.values[(s, a)] = old_v * (1-ALPHA) + new_v * ALPHA
```

Here, we update our values table using one step from the environment. To do this, we calculate the Bellman approximation for our state, *s*, and action, *a*, by summing the immediate reward with the discounted value of the next state. Then we obtain the previous value of the state and action pair, and blend these values together using the learning rate. The result is the new approximation for the value of state *s* and action *a*, which is stored in our table.

```
def play_episode(self, env):
    total_reward = 0.0
    state = env.reset()
    while True:
        _, action = self.best_value_and_action(state)
        new_state, reward, is_done, _ = env.step(action)
        total_reward += reward
```

```
            if is_done:
                break
            state = new_state
        return total_reward
```

The last method in our `Agent` class plays one full episode using the provided test environment. The action on every step is taken using our current value table of Q-values. This method is used to evaluate our current policy to check the progress of learning. Note that this method doesn't alter our value table: it only uses it to find the best action to take.

The rest of the example is the training loop, which is very similar to examples from *Chapter 5, Tabular Learning and the Bellman Equation*: we create a test environment, agent, and summary writer, and then, in the loop, we do one step in the environment and perform a value update using the obtained data. Next, we test our current policy by playing several test episodes. If a good reward is obtained, then we stop training.

```
    if __name__ == "__main__":
        test_env = gym.make(ENV_NAME)
        agent = Agent()
        writer = SummaryWriter(comment="-q-learning")

        iter_no = 0
        best_reward = 0.0
        while True:
            iter_no += 1
            s, a, r, next_s = agent.sample_env()
            agent.value_update(s, a, r, next_s)

            reward = 0.0
            for _ in range(TEST_EPISODES):
                reward += agent.play_episode(test_env)
            reward /= TEST_EPISODES
            writer.add_scalar("reward", reward, iter_no)
            if reward > best_reward:
                print("Best reward updated %.3f -> %.3f" % (
                    best_reward, reward))
                best_reward = reward
            if reward > 0.80:
                print("Solved in %d iterations!" % iter_no)
                break
        writer.close()
```

The result of the example is shown here:

```
rl_book_samples/Chapter06$./01_frozenlake_q_learning.py
Best reward updated 0.000 -> 0.150
Best reward updated 0.150 -> 0.250
Best reward updated 0.250 -> 0.300
Best reward updated 0.300 -> 0.350
Best reward updated 0.350 -> 0.400
Best reward updated 0.400 -> 0.450
Best reward updated 0.450 -> 0.550
Best reward updated 0.550 -> 0.600
Best reward updated 0.600 -> 0.650
Best reward updated 0.650 -> 0.700
Best reward updated 0.700 -> 0.750
Best reward updated 0.750 -> 0.800
Best reward updated 0.800 -> 0.850
Solved in 5738 iterations!
```

You may have noticed that this version used more iterations to solve the problem compared to the value iteration method from the previous chapter. The reason for that is that we are no longer using the experience obtained during testing. (In example `Chapter05/02_frozenlake_q_iteration.py`, periodical tests caused an update of Q-table statistics. Here, we don't touch Q-values during the test, which causes more iterations before the environment gets solved.) Overall, the total number of samples required from the environment is almost the same. The reward chart in TensorBoard also shows good training dynamics, which are very similar to the value iteration method.
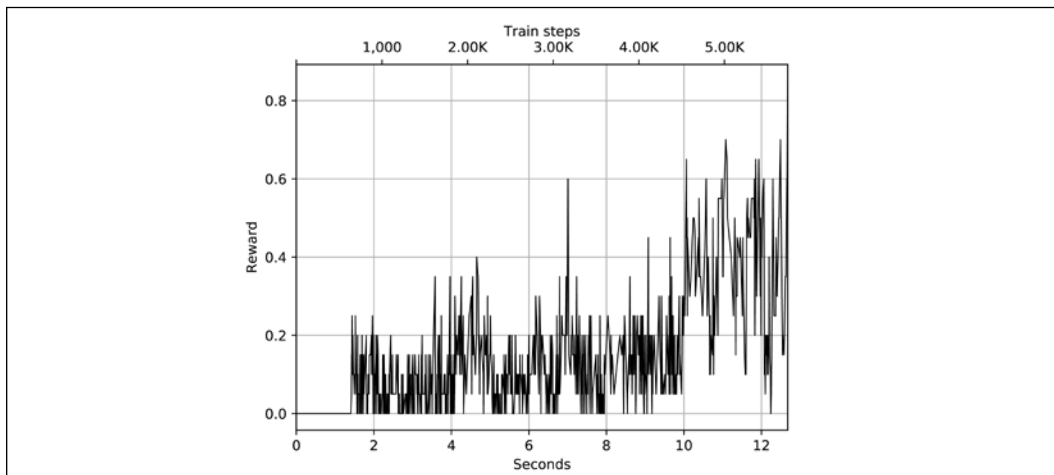
Figure 6.1: Reward dynamics of FrozenLake

Deep Q-learning

The Q-learning method that we have just covered solves the issue of iteration over the full set of states, but still can struggle with situations when the count of the observable set of states is very large. For example, Atari games can have a large variety of different screens, so if we decide to use raw pixels as individual states, we will quickly realize that we have too many states to track and approximate values for.

In some environments, the count of different observable states could be almost infinite. For example, in CartPole, the environment gives us a state that is four floating point numbers. The number of value combinations is finite (they're represented as bits), but this number is extremely large. We could create some bins to discretize those values, but this often creates more problems than it solves: we would need to decide what ranges of parameters are important to distinguish as different states and what ranges could be clustered together.

In the case of Atari, one single pixel change doesn't make much difference, so it's efficient to treat both images as a single state. However, we still need to distinguish some of the states.

The following image shows two different situations in a game of Pong. We're playing against the artificial intelligence (AI) opponent by controlling a paddle (our paddle is on the right, whereas our opponent's is on the left). The objective of the game is to get the bouncing ball past our opponent's paddle, while preventing the ball from getting past our paddle. We can consider the two situations to be completely different: in the right-hand situation, the ball is close to the opponent, so we can relax and watch. However, the situation on the left is more demanding: assuming that the ball is moving from left to right, the ball is moving toward our side, so we need to move our paddle quickly to avoid losing a point. The situations in *Figure 6.2* are just two from the 10^{70802} possible situations, but we want our agent to act on them differently.

Figure 6.2: The ambiguity of observations in Pong. In the left image, the ball is moving to the right toward our paddle, and on the right, its direction is opposite

As a solution to this problem, we can use a nonlinear representation that maps both the state and action onto a value. In machine learning, this is called a "regression problem." The concrete way to represent and train such a representation can vary, but as you may have already guessed from this section's title, using a deep NN is one of the most popular options, especially when dealing with observations represented as screen images. With this in mind, let's make modifications to the Q-learning algorithm:

1. Initialize $Q(s, a)$ with some initial approximation.
2. By interacting with the environment, obtain the tuple (s, a, r, s').
3. Calculate loss: $\mathcal{L} = (Q(s, a) - r)^2$ if the episode has ended, or
   $$\mathcal{L} = \left(Q(s, a) - \left(r + \gamma \max_{a' \in A} Q_{s', a'} \right) \right)^2 \text{ otherwise.}$$
4. Update $Q(s, a)$ using the **stochastic gradient descent** (**SGD**) algorithm, by minimizing the loss with respect to the model parameters.
5. Repeat from step 2 until converged.

The preceding algorithm looks simple, but, unfortunately, it won't work very well. Let's discuss what could go wrong.

Interaction with the environment

First of all, we need to interact with the environment somehow to receive data to train on. In simple environments, such as FrozenLake, we can act randomly, but is this the best strategy to use? Imagine the game of Pong. What's the probability of winning a single point by randomly moving the paddle? It's not zero, but it's extremely small, which just means that we will need to wait for a very long time for such a rare situation. As an alternative, we can use our Q-function approximation as a source of behavior (as we did before in the value iteration method, when we remembered our experience during testing).

If our representation of Q is good, then the experience that we get from the environment will show the agent relevant data to train on. However, we're in trouble when our approximation is not perfect (at the beginning of the training, for example). In such a case, our agent can be stuck with bad actions for some states without ever trying to behave differently. This is the *exploration versus exploitation* dilemma mentioned briefly in *Chapter 1*, *What Is Reinforcement Learning?*, and earlier in this chapter. On the one hand, our agent needs to explore the environment to build a complete picture of transitions and action outcomes. On the other hand, we should use interaction with the environment efficiently: we shouldn't waste time by randomly trying actions that we have already tried and have learned outcomes for.

As you can see, random behavior is better at the beginning of the training when our Q approximation is bad, as it gives us more uniformly distributed information about the environment states. As our training progresses, random behavior becomes inefficient, and we want to fall back to our Q approximation to decide how to act.

A method that performs such a mix of two extreme behaviors is known as an **epsilon-greedy method**, which just means switching between random and Q policy using the probability hyperparameter ε. By varying ε, we can select the ratio of random actions. The usual practice is to start with $\varepsilon = 1.0$ (100% random actions) and slowly decrease it to some small value, such as 5% or 2% random actions. Using an epsilon-greedy method helps us both to explore the environment in the beginning and stick to good policy at the end of the training. There are other solutions to the exploration versus exploitation problem, and we will discuss some of them in the third part of the book. This problem is one of the fundamental open questions in RL and an active area of research that is not even close to being resolved completely.

SGD optimization

The core of our Q-learning procedure is borrowed from supervised learning. Indeed, we are trying to approximate a complex, nonlinear function, $Q(s, a)$, with an NN. To do this, we must calculate targets for this function using the Bellman equation and then pretend that we have a supervised learning problem at hand. That's okay, but one of the fundamental requirements for SGD optimization is that the training data is **independent and identically distributed** (frequently abbreviated as **i.i.d.**).

In our case, data that we are going to use for the SGD update doesn't fulfill these criteria:

1. Our samples are not independent. Even if we accumulate a large batch of data samples, they will all be very close to each other, as they will belong to the same episode.

2. Distribution of our training data won't be identical to samples provided by the optimal policy that we want to learn. Data that we have will be a result of some other policy (our current policy, random, or both in the case of epsilon-greedy), but we don't want to learn how to play randomly: we want an optimal policy with the best reward.

To deal with this nuisance, we usually need to use a large buffer of our past experience and sample training data from it, instead of using our latest experience. This technique is called **replay buffer**. The simplest implementation is a buffer of fixed size, with new data added to the end of the buffer so that it pushes the oldest experience out of it.

Replay buffer allows us to train on more-or-less independent data, but the data will still be fresh enough to train on samples generated by our recent policy. In the next chapter, we will check another kind of replay buffer: prioritized, which provides a more sophisticated sampling approach.

Correlation between steps

Another practical issue with the default training procedure is also related to the lack of i.i.d data, but in a slightly different manner. The Bellman equation provides us with the value of $Q(s, a)$ via $Q(s', a')$ (this process is called *bootstrapping*). However, both the states s and s' have only one step between them. This makes them very similar, and it's very hard for NNs to distinguish between them. When we perform an update of our NN's parameters to make $Q(s, a)$ closer to the desired result, we can indirectly alter the value produced for $Q(s', a')$ and other states nearby. This can make our training very unstable, like chasing our own tail: when we update Q for state s, then on subsequent states we will discover that $Q(s', a')$ becomes worse, but attempts to update it can spoil our $Q(s, a)$ approximation, and so on.

To make training more stable, there is a trick, called *target network*, by which we keep a copy of our network and use it for the $Q(s', a')$ value in the Bellman equation. This network is synchronized with our main network only periodically, for example, once in N steps (where N is usually quite a large hyperparameter, such as 1k or 10k training iterations).

The Markov property

Our RL methods use Markov decision process (MDP) formalism as their basis, which assumes that the environment obeys the Markov property: observations from the environment are all that we need to act optimally. (In other words, our observations allow us to distinguish states from one another.)

As you saw from the preceding Pong screenshot, one single image from the Atari game is not enough to capture all the important information (using only one image, we have no idea about the speed and direction of objects, like the ball and our opponent's paddle). This obviously violates the Markov property and moves our single-frame Pong environment into the area of **partially observable MDPs (POMDPs)**. A POMDP is basically MDP without the Markov property, and it is very important in practice. For example, for most card games in which you don't see your opponents' cards, game observations are POMDPs, because the current observation (your cards and cards on the table) could correspond to different cards in your opponents' hands.

We won't discuss POMDPs in detail in this book, but we will use a small technique to push our environment back into the MDP domain. The solution is maintaining several observations from the past and using them as a state. In the case of Atari games, we usually stack k subsequent frames together and use them as the observation at every state. This allows our agent to deduct the dynamics of the current state, for instance, to get the speed of the ball and its direction. The usual "classical" number of k for Atari is four. Of course, it's just a hack, as there can be longer dependencies in the environment, but for most of the games, it works well.

The final form of DQN training

There are many more tips and tricks that researchers have discovered to make DQN training more stable and efficient, and we will cover the best of them in the next chapter. However, epsilon-greedy, replay buffer, and target network form the basis that has allowed the AI company DeepMind to successfully train a DQN on a set of 49 Atari games and demonstrate the efficiency of this approach when applied to complicated environments.

The original paper (without target network) was published at the end of 2013 (*Playing Atari with Deep Reinforcement Learning*, 1312.5602v1, *Mnih* and others) and used seven games for testing. Later, at the beginning of 2015, a revised version of the article, with 49 different games, was published in *Nature* (*Human-Level Control Through Deep Reinforcement Learning*, doi:10.1038/nature14236, *Mnih* and others).

The algorithm for DQN from the preceding papers has the following steps:

1. Initialize the parameters for $Q(s, a)$ and $\hat{Q}(s, a)$ with random weights, $\varepsilon \leftarrow 1.0$, and empty the replay buffer.

2. With probability ε, select a random action, a; otherwise, $a = \arg\max_a Q(s, a)$.

3. Execute action a in an emulator and observe the reward, r, and the next state, s'.

4. Store transition (s, a, r, s') in the replay buffer.

5. Sample a random mini-batch of transitions from the replay buffer.

6. For every transition in the buffer, calculate target $y = r$ if the episode has ended at this step, or $y = r + \gamma \max_{a' \in A} \hat{Q}(s', a')$ otherwise.

7. Calculate loss: $\mathcal{L} = (Q(s, a) - y)^2$.

8. Update $Q(s, a)$ using the SGD algorithm by minimizing the loss in respect to the model parameters.

9. Every N steps, copy weights from Q to $\hat{Q}$.

10. Repeat from step 2 until converged.

Let's implement it now and try to beat some of the Atari games!

DQN on Pong

Before we jump into the code, some introduction is needed. Our examples are becoming increasingly challenging and complex, which is not surprising, as the complexity of the problems that we are trying to tackle is also growing. The examples are as simple and concise as possible, but some of the code may be difficult to understand at first.

Another thing to note is performance. Our previous examples for FrozenLake, or CartPole, were not demanding from a performance perspective, as observations were small, NN parameters were tiny, and shaving off extra milliseconds in the training loop wasn't important. However, from now on, that's not the case. One single observation from the Atari environment is 100k values, which have to be rescaled, converted to floats, and stored in the replay buffer. One extra copy of this data array can cost you training speed, which will not be seconds and minutes anymore, but could be hours on even the fastest graphics processing unit (GPU) available.

The NN training loop could also be a bottleneck. Of course, RL models are not as huge monsters as state-of-the-art ImageNet models, but even the DQN model from 2015 has more than 1.5M parameters, which is a lot for a GPU to crunch. So, to make a long story short, performance matters, especially when you are experimenting with hyperparameters and need to wait not for a single model to train, but for dozens of them.

PyTorch is quite expressive, so more-or-less efficient processing code could look much less cryptic than optimized TensorFlow graphs, but there is still a significant opportunity for doing things slowly and making mistakes. For example, a naïve version of DQN loss computation, which loops over every batch sample, is about two times slower than a parallel version. However, a single extra copy of the data batch could make the speed of the same code 13 times slower, which is quite significant.

This example has been split into three modules due to its length, logical structure, and reusability. The modules are as follows:

- `Chapter06/lib/wrappers.py`: These are Atari environment wrappers, mostly taken from the OpenAI **Baselines** project.
- `Chapter06/lib/dqn_model.py`: This is the DQN NN layer, with the same architecture as the DeepMind DQN from the *Nature* paper.
- `Chapter06/02_dqn_pong.py`: This is the main module, with the training loop, loss function calculation, and experience replay buffer.

Wrappers

Tackling Atari games with RL is quite demanding from a resource perspective. To make things faster, several transformations are applied to the Atari platform interaction, which are described in DeepMind's paper. Some of these transformations influence only performance, but some address Atari platform features that make learning long and unstable. Transformations are usually implemented as OpenAI Gym wrappers of various kinds. The full list is quite lengthy and there are several implementations of the same wrappers in various sources. My personal favorite is in the OpenAI Baselines repository, which is a set of RL methods and algorithms implemented in TensorFlow and applied to popular benchmarks to establish the common ground for comparing methods. The repository is available from `https://github.com/openai/baselines`, and wrappers are available in this file: `https://github.com/openai/baselines/blob/master/baselines/common/atari_wrappers.py`.

The list of the most popular Atari transformations used by RL researchers includes:

- *Converting individual lives in the game into separate episodes.* In general, an episode contains all the steps from the beginning of the game until the "Game over" screen appears, which can last for thousands of game steps (observations and actions). Usually, in arcade games, the player is given several lives, which provide several attempts in the game. This transformation splits a full episode into individual small episodes for every life that a player has. Not all games support this feature (for example, Pong doesn't), but for the supported environments, it usually helps to speed up convergence, as our episodes become shorter.

- *At the beginning of the game, performing a random amount (up to 30) of no-op actions.* This skips intro screens in some Atari games, which are not relevant for the gameplay.

- *Making an action decision every* K *steps, where* K *is usually 4 or 3.* On intermediate frames, the chosen action is simply repeated. This allows training to speed up significantly, as processing every frame with an NN is quite a demanding operation, but the difference between consequent frames is usually minor.

- *Taking the maximum of every pixel in the last two frames and using it as an observation.* Some Atari games have a flickering effect, which is due to the platform's limitation. (Atari has a limited number of sprites that can be shown on a single frame.) For the human eye, such quick changes are not visible, but they can confuse NNs.

- *Pressing* **FIRE** *at the beginning of the game.* Some games (including Pong and Breakout) require a user to press the **FIRE** button to start the game. Without this, the environment becomes a POMDP, as from observation, an agent cannot tell whether **FIRE** was already pressed.

- *Scaling every frame down from 210×160, with three color frames, to a single-color 84×84 image.* Different approaches are possible. For example, the DeepMind paper describes this transformation as taking the Y-color channel from the YCbCr color space and then rescaling the full image to an 84×84 resolution. Some other researchers do grayscale transformation, cropping non-relevant parts of the image and then scaling down. In the Baselines repository (and in the following example code), the latter approach is used.

- *Stacking several (usually four) subsequent frames together to give the network information about the dynamics of the game's objects.* This approach was already discussed as a quick solution to the lack of game dynamics in a single game frame.

- *Clipping the reward to –1, 0, and 1 values.* The obtained score can vary wildly among the games. For example, in Pong you get a score of 1 for every ball that your opponent passes behind you. However, in some games, like KungFuMaster, you get a reward of 100 for every enemy killed. This spread in reward values makes our loss have completely different scales between the games, which makes it harder to find common hyperparameters for a set of games. To fix this, the reward just gets clipped to the range [–1...1].

- *Converting observations from* `unsigned` *bytes to* `float32` *values.* The screen obtained from the emulator is encoded as a tensor of bytes with values from 0 to 255, which is not the best representation for an NN. So, we need to convert the image into floats and rescale the values to the range [0.0…1.0].

In our Pong example, we don't need some of these wrappers, such as converting lives into separate episodes and reward clipping, so those wrappers aren't included in the example code. However, you should be aware of them, just in case you decide to experiment with other games. Sometimes, when the DQN is not converging, the problem is not in the code but in the wrongly wrapped environment. I've spent several days debugging convergence issues caused by missing the **FIRE** button press at the beginning of a game!

Let's take a look at the implementation of individual wrappers from `Chapter06/lib/wrappers.py`:

```
import cv2
import gym
import gym.spaces
import numpy as np
import collections
```

```
class FireResetEnv(gym.Wrapper):
    def __init__(self, env=None):
        super(FireResetEnv, self).__init__(env)
        assert env.unwrapped.get_action_meanings()[1] == 'FIRE'
        assert len(env.unwrapped.get_action_meanings()) >= 3

    def step(self, action):
        return self.env.step(action)

    def reset(self):
        self.env.reset()
        obs, _, done, _ = self.env.step(1)
        if done:
            self.env.reset()
        obs, _, done, _ = self.env.step(2)
        if done:
            self.env.reset()
        return obs
```

The preceding wrapper presses the **FIRE** button in environments that require that for the game to start. In addition to pressing **FIRE**, this wrapper checks for several corner cases that are present in some games.

```
class MaxAndSkipEnv(gym.Wrapper):
    def __init__(self, env=None, skip=4):
        super(MaxAndSkipEnv, self).__init__(env)
        self._obs_buffer = collections.deque(maxlen=2)
        self._skip = skip

    def step(self, action):
        total_reward = 0.0
        done = None
        for _ in range(self._skip):
            obs, reward, done, info = self.env.step(action)
            self._obs_buffer.append(obs)
            total_reward += reward
            if done:
                break
        max_frame = np.max(np.stack(self._obs_buffer), axis=0)
        return max_frame, total_reward, done, info

    def reset(self):
        self._obs_buffer.clear()
```

```
        obs = self.env.reset()
        self._obs_buffer.append(obs)
        return obs
```

This wrapper combines the repetition of actions during *K* frames and pixels from two consecutive frames.

```
class ProcessFrame84(gym.ObservationWrapper):
    def __init__(self, env=None):
        super(ProcessFrame84, self).__init__(env)
        self.observation_space = gym.spaces.Box(
            low=0, high=255, shape=(84, 84, 1), dtype=np.uint8)

    def observation(self, obs):
        return ProcessFrame84.process(obs)

    @staticmethod
    def process(frame):
        if frame.size == 210 * 160 * 3:
            img = np.reshape(frame, [210, 160, 3]).astype(
                np.float32)
        elif frame.size == 250 * 160 * 3:
            img = np.reshape(frame, [250, 160, 3]).astype(
                np.float32)
        else:
            assert False, "Unknown resolution."
        img = img[:, :, 0] * 0.299 + img[:, :, 1] * 0.587 + \
            img[:, :, 2] * 0.114
        resized_screen = cv2.resize(
            img, (84, 110), interpolation=cv2.INTER_AREA)
        x_t = resized_screen[18:102, :]
        x_t = np.reshape(x_t, [84, 84, 1])
        return x_t.astype(np.uint8)
```

The goal of this wrapper is to convert input observations from the emulator, which normally has a resolution of 210×160 pixels with RGB color channels, to a grayscale 84×84 image. It does this using a colorimetric grayscale conversion (which is closer to human color perception than a simple averaging of color channels), resizing the image, and cropping the top and bottom parts of the result.

```
class BufferWrapper(gym.ObservationWrapper):
    def __init__(self, env, n_steps, dtype=np.float32):
        super(BufferWrapper, self).__init__(env)
        self.dtype = dtype
        old_space = env.observation_space
```

```
        self.observation_space = gym.spaces.Box(
            old_space.low.repeat(n_steps, axis=0),
            old_space.high.repeat(n_steps, axis=0), dtype=dtype)

    def reset(self):
        self.buffer = np.zeros_like(
            self.observation_space.low, dtype=self.dtype)
        return self.observation(self.env.reset())

    def observation(self, observation):
        self.buffer[:-1] = self.buffer[1:]
        self.buffer[-1] = observation
        return self.buffer
```

This class creates a stack of subsequent frames along the first dimension and returns them as an observation. The purpose is to give the network an idea about the dynamics of the objects, such as the speed and direction of the ball in Pong or how enemies are moving. This is very important information, which it is not possible to obtain from a single image.

```
    class ImageToPyTorch(gym.ObservationWrapper):
        def __init__(self, env):
            super(ImageToPyTorch, self).__init__(env)
            old_shape = self.observation_space.shape
            new_shape = (old_shape[-1], old_shape[0], old_shape[1])
            self.observation_space = gym.spaces.Box(
                low=0.0, high=1.0, shape=new_shape, dtype=np.float32)

        def observation(self, observation):
            return np.moveaxis(observation, 2, 0)
```

This simple wrapper changes the shape of the observation from HWC (height, width, channel) to the CHW (channel, height, width) format required by PyTorch. The input shape of the tensor has a color channel as the last dimension, but PyTorch's convolution layers assume the color channel to be the first dimension.

```
    class ScaledFloatFrame(gym.ObservationWrapper):
        def observation(self, obs):
            return np.array(obs).astype(np.float32) / 255.0
```

The final wrapper we have in the library converts observation data from bytes to floats, and scales every pixel's value to the range [0.0...1.0].

```
    def make_env(env_name):
        env = gym.make(env_name)
        env = MaxAndSkipEnv(env)
```

```
env = FireResetEnv(env)
env = ProcessFrame84(env)
env = ImageToPyTorch(env)
env = BufferWrapper(env, 4)
return ScaledFloatFrame(env)
```

At the end of the file is a simple function that creates an environment by its name and applies all the required wrappers to it. That's it for wrappers, so let's look at our model.

The DQN model

The model published in *Nature* has three convolution layers followed by two fully connected layers. All layers are separated by rectified linear unit (ReLU) nonlinearities. The output of the model is Q-values for every action available in the environment, without nonlinearity applied (as Q-values can have any value). The approach of having all Q-values calculated with one pass through the network helps us to increase speed significantly in comparison to treating $Q(s, a)$ literally and feeding observations and actions to the network to obtain the value of the action.

The code of the model is in Chapter06/lib/dqn_model.py:

```python
import torch
import torch.nn as nn
import numpy as np


class DQN(nn.Module):
    def __init__(self, input_shape, n_actions):
        super(DQN, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )

        conv_out_size = self._get_conv_out(input_shape)
        self.fc = nn.Sequential(
            nn.Linear(conv_out_size, 512),
            nn.ReLU(),
            nn.Linear(512, n_actions)
        )
```

To be able to write our network in the generic way, it was implemented in two parts: *convolution* and *sequential*. PyTorch doesn't have a "flatter" layer that could transform a 3D tensor into a 1D vector of numbers, with the requirement of feeding convolution output to the fully connected layer. This problem is solved in the `forward()` function, where we can reshape our batch of 3D tensors into a batch of 1D vectors.

Another small problem is that we don't know the exact number of values in the output from the convolution layer produced with the input of the given shape. However, we need to pass this number to the first fully connected layer constructor. One possible solution would be to hard-code this number, which is a function of input shape (for 84×84 input, the output from the convolution layer will have 3136 values); however, it's not the best way, as our code will become less robust to input shape change. The better solution would be to have a simple function, `_get_conv_out()`, that accepts the input shape and applies the convolution layer to a fake tensor of such a shape. The result of the function would be equal to the number of parameters returned by this application. It would be fast, as this call would be done once on model creation, but also, it would allow us to have generic code.

```
def _get_conv_out(self, shape):
    o = self.conv(torch.zeros(1, *shape))
    return int(np.prod(o.size()))

def forward(self, x):
    conv_out = self.conv(x).view(x.size()[0], -1)
    return self.fc(conv_out)
```

The final piece of the model is the `forward()` function, which accepts the 4D input tensor. (The first dimension is batch size and the second is the color channel, which is our stack of subsequent frames; the third and fourth are image dimensions.)

The application of transformations is done in two steps: first we apply the convolution layer to the input, and then we obtain a 4D tensor on output. This result is flattened to have two dimensions: a batch size and all the parameters returned by the convolution for this batch entry as one long vector of numbers. This is done by the `view()` function of the tensors, which lets one single dimension be a `-1` argument as a *wildcard* for the rest of the parameters. For example, if we have a tensor, `T`, of shape `(2, 3, 4)`, which is a 3D tensor of 24 elements, we can reshape it into a 2D tensor with six rows and four columns using `T.view(6, 4)`. This operation doesn't create a new memory object or move the data in memory; it just changes the higher-level shape of the tensor. The same result could be obtained by `T.view(-1, 4)` or `T.view(6, -1)`, which is very convenient when your tensor has a batch size in the first dimension. Finally, we pass this flattened 2D tensor to our fully connected layers to obtain Q-values for every batch input.

Training

The third module contains the experience replay buffer, the agent, the loss function calculation, and the training loop itself. Before going into the code, something needs to be said about the training hyperparameters. DeepMind's *Nature* paper contained a table with all the details about hyperparameters used to train its model on *all* 49 Atari games used for evaluation. DeepMind kept all those parameters the same for all games (but trained individual models for every game), and it was the team's intention to show that the method is robust enough to solve lots of games with varying complexity, action space, reward structure, and other details using one single model architecture and hyperparameters. However, our goal here is much more modest: we want to solve just the Pong game.

Pong is quite simple and straightforward in comparison to other games in the Atari test set, so the hyperparameters in the paper are overkill for our task. For example, to get the best result on all 49 games, DeepMind used a million-observations replay buffer, which requires approximately 20 GB of RAM to keep and lots of samples from the environment to populate.

The epsilon decay schedule that was used is also not the best for a single Pong game. In the training, DeepMind linearly decayed epsilon from 1.0 to 0.1 during the first million frames obtained from the environment. However, my own experiments have shown that for Pong, it's enough to decay epsilon over the first 150k frames and then keep it stable. The replay buffer also can be much smaller: 10k transitions will be enough.

In the following example, I've used my parameters. These differ from the parameters in the paper but will allow us to solve Pong about 10 times faster. On a GeForce GTX 1080 Ti, the following version converges to a mean score of 19.0 in one to two hours, but with DeepMind's hyperparameters, it will require at least a day.

This speedup, of course, is fine-tuning for one particular environment and can break convergence on other games. You are free to play with the options and other games from the Atari set.

```
from lib import wrappers
from lib import dqn_model

import argparse
import time
import numpy as np
import collections

import torch
import torch.nn as nn
```

```
import torch.optim as optim

from tensorboardX import SummaryWriter
```

First, we import required modules and define the hyperparameters.

```
DEFAULT_ENV_NAME = "PongNoFrameskip-v4"
MEAN_REWARD_BOUND = 19.0
```

These two values set the default environment to train on and the reward boundary for the last 100 episodes to stop training. If you want, you can redefine the environment name using the command line.

```
GAMMA = 0.99
BATCH_SIZE = 32
REPLAY_SIZE = 10000
REPLAY_START_SIZE = 10000
LEARNING_RATE = 1e-4
SYNC_TARGET_FRAMES = 1000
```

These parameters define the following:

- Our gamma value used for Bellman approximation (GAMMA)

- The batch size sampled from the replay buffer (BATCH_SIZE)

- The maximum capacity of the buffer (REPLAY_SIZE)

- The count of frames we wait for before starting training to populate the replay buffer (REPLAY_START_SIZE)

- The learning rate used in the Adam optimizer, which is used in this example (LEARNING_RATE)

- How frequently we sync model weights from the training model to the target model, which is used to get the value of the next state in the Bellman approximation (SYNC_TARGET_FRAMES)

```
EPSILON_DECAY_LAST_FRAME = 150000
EPSILON_START = 1.0
EPSILON_FINAL = 0.01
```

The last batch of hyperparameters is related to the epsilon decay schedule. To achieve proper exploration, we start with epsilon = 1.0 at the early stages of training, which causes all actions to be selected randomly. Then, during the first 150,000 frames, epsilon is linearly decayed to 0.01, which corresponds to the random action taken in 1% of steps. A similar scheme was used in the original DeepMind paper, but the duration of decay was almost 10 times longer (so, epsilon = 0.01 was reached after a million frames).

The next chunk of code defines our experience replay buffer, the purpose of which is to keep the transitions obtained from the environment (tuples of the observation, action, reward, `done` flag, and the next state). Each time we do a step in the environment, we push the transition into the buffer, keeping only a fixed number of steps (in our case, 10k transitions). For training, we randomly sample the batch of transitions from the replay buffer, which allows us to break the correlation between subsequent steps in the environment.

```python
Experience = collections.namedtuple(
    'Experience', field_names=['state', 'action', 'reward',
                               'done', 'new_state'])


class ExperienceBuffer:
    def __init__(self, capacity):
        self.buffer = collections.deque(maxlen=capacity)

    def __len__(self):
        return len(self.buffer)

    def append(self, experience):
        self.buffer.append(experience)

    def sample(self, batch_size):
        indices = np.random.choice(len(self.buffer), batch_size,
                                   replace=False)
        states, actions, rewards, dones, next_states = \
            zip(*[self.buffer[idx] for idx in indices])
        return np.array(states), np.array(actions), \
            np.array(rewards, dtype=np.float32), \
            np.array(dones, dtype=np.uint8), \
            np.array(next_states)
```

Most of the experience replay buffer code is quite straightforward: it basically exploits the capability of the `deque` class to maintain the given number of entries in the buffer. In the `sample()` method, we create a list of random indices and then repack the sampled entries into NumPy arrays for more convenient loss calculation.

The next class we need to have is an `Agent`, which interacts with the environment and saves the result of the interaction into the experience replay buffer that you have just seen:

```python
class Agent:
    def __init__(self, env, exp_buffer):
        self.env = env
        self.exp_buffer = exp_buffer
```

```
        self._reset()

    def _reset(self):
        self.state = env.reset()
        self.total_reward = 0.0
```

During the agent's initialization, we need to store references to the environment and experience replay buffer, tracking the current observation and the total reward accumulated so far.

```
    @torch.no_grad()
    def play_step(self, net, epsilon=0.0, device="cpu"):
        done_reward = None

        if np.random.random() < epsilon:
            action = env.action_space.sample()
        else:
            state_a = np.array([self.state], copy=False)
            state_v = torch.tensor(state_a).to(device)
            q_vals_v = net(state_v)
            _, act_v = torch.max(q_vals_v, dim=1)
            action = int(act_v.item())
```

The main method of the agent is to perform a step in the environment and store its result in the buffer. To do this, we need to select the action first. With the probability epsilon (passed as an argument), we take the random action; otherwise, we use the past model to obtain the Q-values for all possible actions and choose the best.

```
        new_state, reward, is_done, _ = self.env.step(action)
        self.total_reward += reward

        exp = Experience(self.state, action, reward,
                         is_done, new_state)
        self.exp_buffer.append(exp)
        self.state = new_state
        if is_done:
            done_reward = self.total_reward
            self._reset()
        return done_reward
```

As the action has been chosen, we pass it to the environment to get the next observation and reward, store the data in the experience buffer, and then handle the end-of-episode situation. The result of the function is the total accumulated reward if we have reached the end of the episode with this step, or None if not.

Now it is time for the last function in the training module, which calculates the loss for the sampled batch. This function is written in a form to maximally exploit GPU parallelism by processing all batch samples with vector operations, which makes it harder to understand when compared with a naïve loop over the batch. Yet this optimization pays off: the parallel version is more than two times faster than an explicit loop over the batch.

As a reminder, here is the loss expression we need to calculate:

$$\mathcal{L} = \left(Q(s, a) - \left(r + \gamma \max_{a' \in A} \hat{Q}(s', a') \right) \right)^2 \text{ for steps that aren't at the end of the episode,}$$

or $\mathcal{L} = (Q(s, a) - r)^2$ for final steps.

```
def calc_loss(batch, net, tgt_net, device="cpu"):
    states, actions, rewards, dones, next_states = batch
```

In arguments, we pass our batch as a tuple of arrays (repacked by the `sample()` method in the experience buffer), our network that we are training, and the target network, which is periodically synced with the trained one.

The first model (passed as the argument network) is used to calculate gradients; the second model in the `tgt_net` argument is used to calculate values for the next states, and this calculation shouldn't affect gradients. To achieve this, we use the `detach()` function of the PyTorch tensor to prevent gradients from flowing into the target network's graph. This function was described in *Chapter 3, Deep Learning with PyTorch*.

```
states_v = torch.tensor(np.array(
    states, copy=False)).to(device)
next_states_v = torch.tensor(np.array(
    next_states, copy=False)).to(device)
actions_v = torch.tensor(actions).to(device)
rewards_v = torch.tensor(rewards).to(device)
done_mask = torch.BoolTensor(dones).to(device)
```

The preceding code is simple and straightforward: we wrap individual NumPy arrays with batch data in PyTorch tensors and copy them to GPU if the CUDA device was specified in arguments.

```
state_action_values = net(states_v).gather(
    1, actions_v.unsqueeze(-1)).squeeze(-1)
```

In the preceding line, we pass observations to the first model and extract the specific Q-values for the taken actions using the `gather()` tensor operation. The first argument to the `gather()` call is a dimension index that we want to perform gathering on (in our case, it is equal to 1, which corresponds to actions).

The second argument is a tensor of indices of elements to be chosen. Extra `unsqueeze()` and `squeeze()` calls are required to compute the index argument for the gather functions, and to get rid of the extra dimensions that we created, respectively. (The index should have the same number of dimensions as the data we are processing.) In *Figure 6.3*, you can see an illustration of what `gather()` does on the example case, with a batch of six entries and four actions:
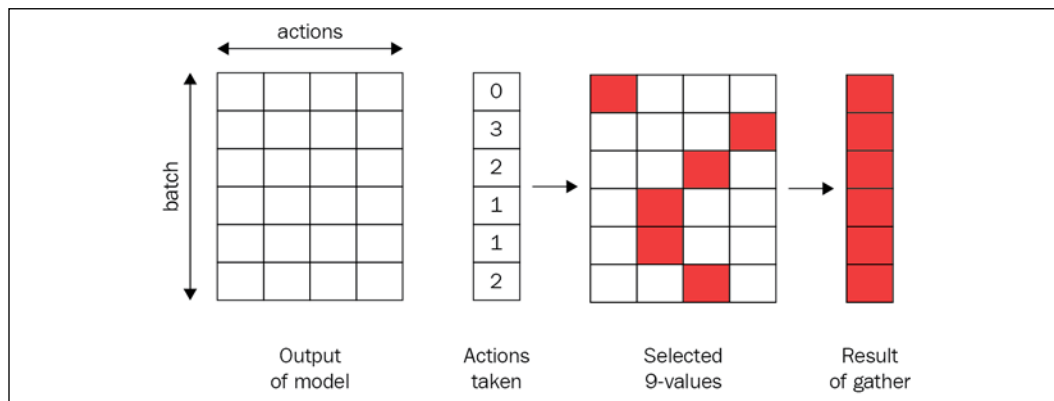
Figure 6.3: Transformation of tensors during a DQN loss calculation

Keep in mind that the result of `gather()` applied to tensors is a differentiable operation that will keep all gradients with respect to the final loss value.

```
next_state_values = tgt_net(next_states_v).max(1)[0]
```

In the preceding line, we apply the target network to our next state observations and calculate the maximum Q-value along the same *action* dimension, 1. Function `max()` returns both maximum values and indices of those values (so it calculates both max and argmax), which is very convenient. However, in this case, we are interested only in values, so we take the first entry of the result.

```
next_state_values[done_mask] = 0.0
```

Here we make one simple, but very important, point: if transition in the batch is from the last step in the episode, then our value of the action doesn't have a discounted reward of the next state, as there is no next state from which to gather the reward. This may look minor, but it is very important in practice: without this, training will not converge.

```
next_state_values = next_state_values.detach()
```

In this line, we detach the value from its computation graph to prevent gradients from flowing into the NN used to calculate Q approximation for the next states.

This is important, as without this, our backpropagation of the loss will start to affect both predictions for the current state and the next state. However, we don't want to touch predictions for the next state, as they are used in the Bellman equation to calculate reference Q-values. To block gradients from flowing into this branch of the graph, we are using the `detach()` method of the tensor, which returns the tensor without connection to its calculation history.

```
expected_state_action_values = next_state_values * GAMMA + \
                                    rewards_v
return nn.MSELoss()(state_action_values,
                        expected_state_action_values)
```

Finally, we calculate the Bellman approximation value and the mean squared error loss. This ends our loss function calculation, and the rest of the code is our training loop.

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--cuda", default=False,
                        action="store_true", help="Enable cuda")
    parser.add_argument("--env", default=DEFAULT_ENV_NAME,
                        help="Name of the environment, default=" +
                            DEFAULT_ENV_NAME)
    args = parser.parse_args()
    device = torch.device("cuda" if args.cuda else "cpu")
```

To begin with, we create a parser of command-line arguments. Our script allows us to enable CUDA and train on environments that are different from the default.

```
    env = wrappers.make_env(args.env)
    net = dqn_model.DQN(env.observation_space.shape,
                        env.action_space.n).to(device)
    tgt_net = dqn_model.DQN(env.observation_space.shape,
                            env.action_space.n).to(device)
```

Here, we create our environment with all required wrappers applied, the NN that we are going to train, and our target network with the same architecture. In the beginning, they will be initialized with different random weights, but it doesn't matter much, as we will sync them every 1k frames, which roughly corresponds to one episode of Pong.

```
    writer = SummaryWriter(comment="-" + args.env)
    print(net)

    buffer = ExperienceBuffer(REPLAY_SIZE)
    agent = Agent(env, buffer)
    epsilon = EPSILON_START
```

Then we create our experience replay buffer of the required size and pass it to the agent. Epsilon is initially initialized to 1.0, but will be decreased every iteration.

```
optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE)
total_rewards = []
frame_idx = 0
ts_frame = 0
ts = time.time()
best_m_reward = None
```

The last things we do before the training loop are to create an optimizer, a buffer for full episode rewards, a counter of frames and several variables to track our speed, and the best mean reward reached. Every time our mean reward beats the record, we will save the model in the file.

```
while True:
    frame_idx += 1
    epsilon = max(EPSILON_FINAL, EPSILON_START -
                        frame_idx / EPSILON_DECAY_LAST_FRAME)
```

At the beginning of the training loop, we count the number of iterations completed and decrease epsilon according to our schedule. Epsilon will drop linearly during the given number of frames (`EPSILON_DECAY_LAST_FRAME=150k`) and then be kept on the same level of `EPSILON_FINAL=0.01`.

```
reward = agent.play_step(net, epsilon, device=device)
if reward is not None:
    total_rewards.append(reward)
    speed = (frame_idx - ts_frame) / (time.time() - ts)
    ts_frame = frame_idx
    ts = time.time()
    m_reward = np.mean(total_rewards[-100:])
    print("%d: done %d games, reward %.3f, "
            "eps %.2f, speed %.2f f/s" % (
        frame_idx, len(total_rewards), m_reward, epsilon,
        speed
    ))
    writer.add_scalar("epsilon", epsilon, frame_idx)
    writer.add_scalar("speed", speed, frame_idx)
    writer.add_scalar("reward_100", m_reward, frame_idx)
    writer.add_scalar("reward", reward, frame_idx)
```

In this block of code, we ask our agent to make a single step in the environment (using our current network and value for epsilon). This function returns a non-`None` result only if this step is the final step in the episode.

In that case, we report our progress. Specifically, we calculate and show, both in the console and in TensorBoard, these values:

- Speed as a count of frames processed per second
- Count of episodes played
- Mean reward for the last 100 episodes
- Current value for epsilon

```
if best_m_reward is None or best_m_reward < m_reward:
    torch.save(net.state_dict(), args.env +
            "-best_%.0f.dat" % m_reward)
    if best_m_reward is not None:
        print("Best reward updated %.3f -> %.3f" % (
            best_m_reward, m_reward))
    best_m_reward = m_reward
if m_reward > MEAN_REWARD_BOUND:
    print("Solved in %d frames!" % frame_idx)
    break
```

Every time our mean reward for the last 100 episodes reaches a maximum, we report this and save the model parameters. If our mean reward exceeds the specified boundary, then we stop training. For Pong, the boundary is 19.0, which means winning more than 19 from 21 possible games.

```
if len(buffer) < REPLAY_START_SIZE:
    continue

if frame_idx % SYNC_TARGET_FRAMES == 0:
    tgt_net.load_state_dict(net.state_dict())
```

Here we check whether our buffer is large enough for training. In the beginning, we should wait for enough data to start, which in our case is 10k transitions. The next condition syncs parameters from our main network to the target network every `SYNC_TARGET_FRAMES`, which is 1k by default.

```
optimizer.zero_grad()
batch = buffer.sample(BATCH_SIZE)
loss_t = calc_loss(batch, net, tgt_net, device=device)
loss_t.backward()
optimizer.step()
```

The last piece of the training loop is very simple but requires the most time to execute: we zero gradients, sample data batches from the experience replay buffer, calculate loss, and perform the optimization step to minimize the loss.

Running and performance

This example is demanding on resources. On Pong, it requires about 400k frames to reach a mean reward of 17 (which means winning more than 80% of games). A similar number of frames will be required to get from 17 to 19, as our learning progress will saturate and it will be hard for the model to improve the score. So, on average, a million frames are needed to train it fully. On the GTX 1080 Ti, I have a speed of about 120 frames per second, which is about two hours of training. On a CPU, the speed is much slower: about nine frames per second, which will take about a day and a half. Remember that this is for Pong, which is relatively easy to solve. Other games require hundreds of millions of frames and a 100 times larger experience replay buffer.

In *Chapter 8*, *DQN Extensions*, we will look at various approaches found by researchers since 2015 that can help to increase both training speed and data efficiency. *Chapter 9*, *Ways to Speed up RL*, will be devoted to engineering tricks to speed up RL methods' performance. Nevertheless, for Atari, you will need resources and patience. The following figure shows charts with training dynamics:
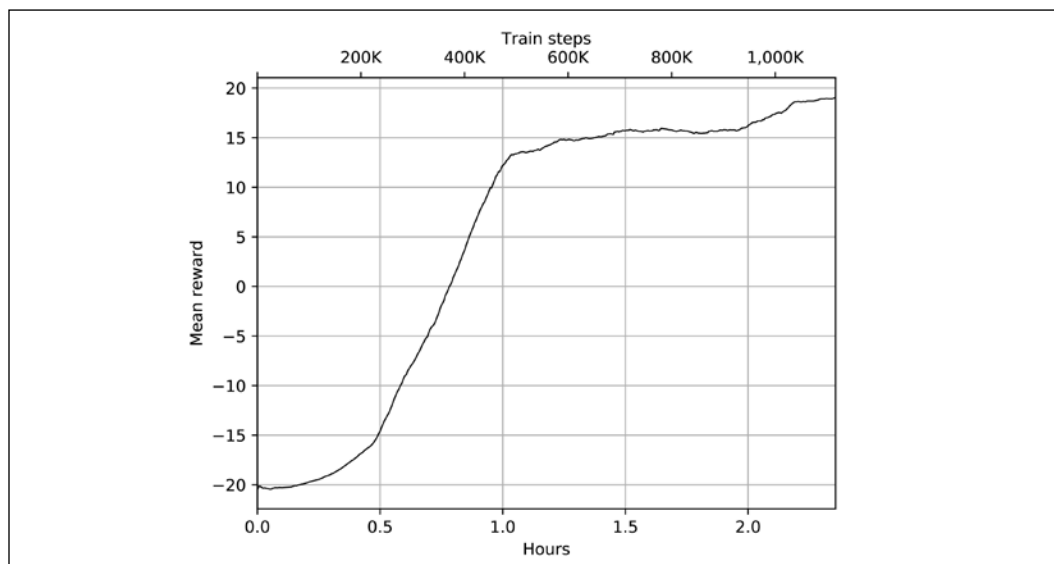

Figure 6.4: Dynamics of average reward calculated over the last 100 episodes

In the beginning of the training:

```
rl_book_samples/Chapter06$ ./02_dqn_pong.py --cuda
  (conv): Sequential(
    (0): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU()
```

```
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (5): ReLU()
  )
  (fc): Sequential(
    (0): Linear(in_features=3136, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=6, bias=True)
  )
)
971: done 1 games, reward -21.000, eps 0.99, speed 890.81 f/s
1733: done 2 games, reward -21.000, eps 0.98, speed 984.59 f/s
2649: done 3 games, reward -20.667, eps 0.97, speed 987.53 f/s
Best reward updated -21.000 -> -20.667, saved
3662: done 4 games, reward -20.500, eps 0.96, speed 921.47 f/s
Best reward updated -20.667 -> -20.500, saved
4619: done 5 games, reward -20.600, eps 0.95, speed 965.86 f/s
5696: done 6 games, reward -20.500, eps 0.94, speed 963.99 f/s
6671: done 7 games, reward -20.429, eps 0.93, speed 967.28 f/s
Best reward updated -20.500 -> -20.429, saved
7648: done 8 games, reward -20.375, eps 0.92, speed 948.15 f/s
Best reward updated -20.429 -> -20.375, saved
8528: done 9 games, reward -20.444, eps 0.91, speed 954.72 f/s
9485: done 10 games, reward -20.400, eps 0.91, speed 936.82 f/s
10394: done 11 games, reward -20.455, eps 0.90, speed 256.84 f/s
11292: done 12 games, reward -20.417, eps 0.89, speed 127.90 f/s
12132: done 13 games, reward -20.385, eps 0.88, speed 130.18 f/s
```

During the first 10k steps, our speed is very high, as we don't do any training, which is the most expensive operation in our code. After 10k, we start sampling the training batches and the performance drops significantly.

Hundreds of games later, our DQN should start to figure out how to win one or two games out of 21. The speed has decreased due to epsilon drop; we need to use our model not only for training, but also for the environment step:

```
94101: done 86 games, reward -19.512, eps 0.06, speed 120.67 f/s
Best reward updated -19.541 -> -19.512, saved
96279: done 87 games, reward -19.460, eps 0.04, speed 120.21 f/s
Best reward updated -19.512 -> -19.460, saved
98140: done 88 games, reward -19.455, eps 0.02, speed 119.10 f/s
Best reward updated -19.460 -> -19.455, saved
99884: done 89 games, reward -19.416, eps 0.02, speed 123.34 f/s
Best reward updated -19.455 -> -19.416, saved
101451: done 90 games, reward -19.411, eps 0.02, speed 120.53 f/s
Best reward updated -19.416 -> -19.411, saved
103812: done 91 games, reward -19.330, eps 0.02, speed 122.41 f/s
Best reward updated -19.411 -> -19.330, saved
105908: done 92 games, reward -19.283, eps 0.02, speed 119.85 f/s
Best reward updated -19.330 -> -19.283, saved
108259: done 93 games, reward -19.172, eps 0.02, speed 122.09 f/s
Best reward updated -19.283 -> -19.172, saved
```

Finally, after many more games, our DQN can finally dominate and beat the (not very sophisticated) built-in Pong AI opponent:

```
1097050: done 522 games, reward 18.800, eps 0.01, speed 132.71 f/s
Best reward updated 18.770 -> 18.800, saved
1098741: done 523 games, reward 18.820, eps 0.01, speed 134.58 f/s
Best reward updated 18.800 -> 18.820, saved
1100507: done 524 games, reward 18.890, eps 0.01, speed 132.11 f/s
Best reward updated 18.820 -> 18.890, saved
1102198: done 525 games, reward 18.920, eps 0.01, speed 133.68 f/s
Best reward updated 18.890 -> 18.920, saved
1103947: done 526 games, reward 18.920, eps 0.01, speed 130.07 f/s
1105745: done 527 games, reward 18.920, eps 0.01, speed 130.27 f/s
1107423: done 528 games, reward 18.960, eps 0.01, speed 130.08 f/s
Best reward updated 18.920 -> 18.960, saved
1109286: done 529 games, reward 18.940, eps 0.01, speed 129.04 f/s
1111058: done 530 games, reward 18.940, eps 0.01, speed 128.59 f/s
1112836: done 531 games, reward 18.930, eps 0.01, speed 130.84 f/s
1114622: done 532 games, reward 18.980, eps 0.01, speed 130.34 f/s
Best reward updated 18.960 -> 18.980, saved
1116437: done 533 games, reward 19.080, eps 0.01, speed 130.09 f/s
Best reward updated 18.980 -> 19.080, saved
Solved in 1116437 frames!
```

Due to randomness in the training process, your actual dynamics might differ from what is displayed here. In some rare cases (one run from 10 according to my experiments), the training does not converge at all, which looks like a constant stream of rewards –21 for a long time. If your training doesn't show any positive dynamics for the first 100k-200k iterations, you should restart it.

Your model in action

The training process is just one half of the whole. Our final goal is not just to train the model; we also want our model to play the game with a good outcome. During the training, every time we update the maximum of the mean reward for the last 100 games, we save the model into the file `PongNoFrameskip-v4-best.dat`. In the `Chapter06/03_dqn_play.py` file, we have a program that can load this model file and play one episode, displaying the model's dynamics.

The code is very simple, but it can be like magic seeing how several matrices, with a million parameters, can play Pong with superhuman accuracy by observing only the pixels.

```
import gym
import time
import argparse
import numpy as np
import torch

from lib import wrappers
from lib import dqn_model

import collections

DEFAULT_ENV_NAME = "PongNoFrameskip-v4"
FPS = 25
```

In the beginning, we import the familiar PyTorch and Gym modules. The preceding `FPS` (frames per second) parameter specifies the approximate speed of the shown frames.

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-m", "--model", required=True,
                        help="Model file to load")
    parser.add_argument("-e", "--env", default=DEFAULT_ENV_NAME,
                        help="Environment name to use, default=" +
                            DEFAULT_ENV_NAME)
    parser.add_argument("-r", "--record", help="Directory for video")
    parser.add_argument("--no-vis", default=True, dest='vis',
                        help="Disable visualization",
                        action='store_false')
    args = parser.parse_args()
```

The script accepts the filename of the saved model and allows the specification of the Gym environment. (Of course, the model and environment have to match.) Additionally, you can pass option `-r` with the name of a nonexistent directory, which will be used to save a video of your game (using the `Monitor` wrapper). By default, the script just shows frames, but if you want to upload your model's gameplay to YouTube, for example, `-r` could be handy.

```
env = wrappers.make_env(args.env)
if args.record:
    env = gym.wrappers.Monitor(env, args.record)
net = dqn_model.DQN(env.observation_space.shape,
                    env.action_space.n)
state = torch.load(args.model, map_location=lambda stg,_: stg)
net.load_state_dict(state)

state = env.reset()
total_reward = 0.0
c = collections.Counter()
```

The preceding code should be clear without comments: we create the environment and our model, and then we load weights from the file passed in the arguments. The argument `map_location`, passed to the `torch.load()` function, is needed to map the loaded tensor location from GPU to CPU. By default, `torch` tries to load tensors on the same device where they were saved, but if you copy the model from the machine you used for training (with a GPU) to a laptop without a GPU, the locations need to be remapped. Our example doesn't use GPU at all, as inference is fast enough without acceleration.

```
while True:
    start_ts = time.time()
    if args.vis:
        env.render()
    state_v = torch.tensor(np.array([state], copy=False))
    q_vals = net(state_v).data.numpy()[0]
    action = np.argmax(q_vals)
    c[action] += 1
```

This is almost an exact copy of the `Agent` class' method `play_step()` from the training code, without the epsilon-greedy action selection. We just pass our observation to the agent and select the action with maximum value. The only new thing here is the `render()` method in the environment, which is a standard way in Gym to display the current observation. (You need to have a graphical user interface (GUI) for this.)

```
        state, reward, done, _ = env.step(action)
        total_reward += reward
        if done:
            break
        if args.vis:
            delta = 1/FPS - (time.time() - start_ts)
            if delta > 0:
                time.sleep(delta)
    print("Total reward: %.2f" % total_reward)
    print("Action counts:", c)
    if args.record:
        env.env.close()
```

The rest of the code is also simple. We pass the action to the environment, count the total reward, and stop our loop when the episode ends. After the episode, we show the total reward and the count of times that the agent executed the action.

In this YouTube playlist, you can find recordings of the gameplay at different stages of the training: `https://www.youtube.com/playlist?list=PLMVwuZENsfJklt4vC ltrWq0KV9aEZ3ylu`.

Things to try

If you are curious and want to experiment with this chapter's material on your own, then here is a short list of directions to explore. Be warned though: they can take lots of time and may cause you some moments of frustration during your experiments. However, these experiments are a very efficient way to really master the material from a practical point of view:

- Try to take some other games from the Atari set, such as Breakout, Atlantis, or River Raid (my childhood favorite). This could require the tuning of hyperparameters.

- As an alternative to FrozenLake, there is another tabular environment, Taxi, which emulates a taxi driver who needs to pick up passengers and take them to a destination.

- Play with Pong hyperparameters. Is it possible to train faster? OpenAI claims that it can solve Pong in 30 minutes using the asynchronous advantage actor-critic method (which is a subject of part three of this book). Maybe it's possible with a DQN.

- Can you make the DQN training code faster? The OpenAI Baselines project has shown 350 FPS using TensorFlow on GTX 1080 Ti. So, it looks like it's possible to optimize the PyTorch code. We will discuss this topic in *Chapter 8*, *DQN Extensions*, but meanwhile, you can do your own experiments.

- In the video recording, you might notice that models with a mean score around zero play quite well. In fact, I had the impression that those models play better than models with mean scores of 10-19. This might be the case due to overfitting to the particular game situations. Could you try to fix this? Maybe it will be possible to use a generative adversarial network-style approach to make one model play with another?

- Can you get *The Ultimate Pong Dominator* model with a mean score of 21? It shouldn't be very hard—learning rate decay is the obvious method to try.

Summary

In this chapter, we covered a lot of new and complex material. You became familiar with the limitations of value iteration in complex environments with large observation spaces, and we discussed how to overcome them with Q-learning. We checked the Q-learning algorithm on the FrozenLake environment and discussed the approximation of Q-values with NNs, and the extra complications that arise from this approximation.

We covered several tricks for DQNs to improve their training stability and convergence, such as an experience replay buffer, target networks, and frame stacking. Finally, we combined those extensions into one single implementation of DQN that solves the Pong environment from the Atari games suite.

In the next chapter, we will look at a set of tricks that researchers have found since 2015 to improve DQN convergence and quality, which (combined) can produce state-of-the-art results on most of the 54 (new games have been added) Atari games. This set was published in 2017, and we will analyze and reimplement all of the tricks.

7
Higher-Level RL Libraries

In *Chapter 6*, *Deep Q-Networks*, we implemented the deep Q-network (DQN) model published by DeepMind in 2015 (`https://deepmind.com/research/publications/playing-atari-deep-reinforcement-learning`). This paper had a significant effect on the RL field by demonstrating that, despite common belief, it's possible to use nonlinear approximators in RL. This proof of concept stimulated great interest in the deep Q-learning field and in deep RL in general.

In this chapter, we will take another step towards practical RL by discussing higher-level RL libraries, which will allow you to build your code from higher-level blocks and focus on the details of the method that you are implementing. Most of the chapter will describe the **PyTorch Agent Net** (**PTAN**) library, which will be used in the rest of the book to avoid code repetition, so will be covered in detail.

We will cover:

- The motivation for using high-level libraries, rather than reimplementing everything from scratch
- The PTAN library, including coverage of the most important parts, which will be illustrated with code examples
- DQN on CartPole, implemented using the PTAN library
- Other RL libraries that you might consider

Why RL libraries?

Our implementation of basic DQN in *Chapter 6*, *Deep Q-Networks* wasn't very, long and complicated—about 200 lines of training code plus 120 lines in environment wrappers. When you are becoming familiar with RL methods, it is very useful to implement everything yourself to understand how things actually work. However, the more involved you become in the field, the more often you will realize that you are writing the same code over and over again.

This repetition comes from the generality of RL methods. As we already discussed in *Chapter 1*, *What Is Reinforcement Learning?*, RL is quite flexible and many real-life problems fall into the environment-agent interaction scheme. RL methods don't make many assumptions about the specifics of observations and actions, so code implemented for the CartPole environment will be applicable to Atari games (maybe with some minor tweaks).

Writing the same code over and over again is not very efficient, as bugs might be introduced every time, which will cost you time for debugging and understanding. In addition, carefully designed code that has been used in several projects usually has a higher quality in terms of performance, unit tests, readability, and documentation.

The practical applications of RL are quite young by computer science standards, so in comparison to other more mature domains, you might not have that rich a choice of approaches. For example, in web development, even if you limit yourself with just Python, you have hundreds of very good libraries of all sorts: Django for heavyweight, fully functional websites; Flask for light **Web Server Gateway Interface** (**WSGI**) apps; and much more, large and small.

In RL, this process has just started, but still, you can choose from several projects that are trying to simplify RL practitioners' lives. In addition, you can always write your own set of tools, as I did several years ago. This library is called PTAN, and, as mentioned, it will be used in the rest of the book to illustrate examples.

The PTAN library

The library is available in GitHub: `https://github.com/Shmuma/ptan`. All the subsequent examples were implemented using version 0.6 of PTAN, which can be installed in your virtual environment by running the following:

```
pip install ptan==0.6
```

The original goal of PTAN was to simplify my RL experiments, and it tries to keep the balance between two extremes:

- Import the library and then write one line with tons of parameters to train one of the provided methods, like DQN (a very vivid example is the OpenAI Baselines project)
- Implement everything from scratch

The first approach is very inflexible. It works well when you are using the library the way it is supposed to be used. But if you want to do something fancy, you will quickly find yourself hacking the library and fighting with the constraints that I imposed, rather than solving the problem you want to solve.

The second extreme gives too much freedom and requires implementing replay buffers and trajectory handling over and over again, which is error-prone, boring, and inefficient.

PTAN tries to balance those extremes, providing high-quality building blocks to simplify your RL code, but at the same time being flexible and not limiting your creativity.

At a high level, PTAN provides the following entities:

- `Agent`: a class that knows how to convert a batch of observations to a batch of actions to be executed. It can contain an optional state, in case you need to track some information between consequent actions in one episode. (We will use this approach in *Chapter 17*, *Continuous Action Space*, in the **deep deterministic policy gradient** (**DDPG**) method, which includes the Ornstein–Uhlenbeck random process for exploration.) The library provides several agents for the most common RL cases, but you always can write your own subclass of `BaseAgent`.
- `ActionSelector`: a small piece of logic that knows how to choose the action from some output of the network. It works in tandem with `Agent`.
- `ExperienceSource` and variations: the `Agent` instance and a Gym environment object can provide information about the trajectory from episodes. In its simplest form, it is one single (a, r, s') transition at a time, but its functionality goes beyond this.
- `ExperienceSourceBuffer` and friends: replay buffers with various characteristics. They include a simple replay buffer and two versions of prioritized replay buffers.
- Various utility classes, like `TargetNet` and wrappers for time series preprocessing (used for tracking training progress in TensorBoard).
- PyTorch Ignite helpers to integrate PTAN into the Ignite framework.
- Wrappers for Gym environments, for example, wrappers for Atari games (copied and pasted from OpenAI Baselines with some tweaks).

And that's basically it. In the following sections, we will take a look at those pieces in detail.

Action selectors

In PTAN terminology, an *action selector* is an object that helps with going from network output to concrete action values. The most common cases include:

- **Argmax**: commonly used by Q-value methods when the network predicts Q-values for a set of actions and the desired action is the action with the largest $Q(s, a)$.

- **Policy-based**: the network outputs the probability distribution (in the form of logits or normalized distribution), and an action needs to be sampled from this distribution. You have already seen this case in *Chapter 4*, *The Cross-Entropy Method*, where we discussed the cross-entropy method.

An action selector is used by the `Agent` and rarely needs to be customized (but you have this option). The concrete classes provided by the library are:

- `ArgmaxActionSelector`: applies argmax on the second axis of a passed tensor. (It assumes a matrix with batch dimension along the first axis.)

- `ProbabilityActionSelector`: samples from the probability distribution of a discrete set of actions.

- `EpsilonGreedyActionSelector`: has the parameter `epsilon`, which specifies the probability of a random action to be taken.

All the classes assume that NumPy arrays will be passed to them. The complete example from this section can be found in `Chapter07/01_actions.py`.

```
>>> import numpy as np
>>> import ptan
>>> q_vals = np.array([[1, 2, 3], [1, -1, 0]])
>>> q_vals
array([[ 1,  2,  3],
       [ 1, -1,  0]])
>>> selector = ptan.actions.ArgmaxActionSelector()
>>> selector(q_vals)
array([2, 0])
```

As you can see, the selector returns indices of actions with the largest values.

```
>>> selector = ptan.actions.EpsilonGreedyActionSelector(epsilon=0.0)
>>> selector(q_vals)
array([2, 0])
```

The result of the `EpsilonGreedyActionSelector` application is the same, as `epsilon` is `0.0`, which means no random actions are taken. If we change `epsilon` to `1`, actions will be random:

```
>>> selector = ptan.actions.EpsilonGreedyActionSelector(epsilon=1.0)
>>> selector(q_vals)
array([1, 1])
```

Working with `ProbabilityActionSelector` is the same, but the input needs to be a normalized probability distribution:

```
>>> selector = ptan.actions.ProbabilityActionSelector()
>>> for _ in range(10):
...     acts = selector(np.array([
...         [0.1, 0.8, 0.1],
...         [0.0, 0.0, 1.0],
...         [0.5, 0.5, 0.0]
...     ]))
...     print(acts)
...
[1 2 1]
[0 2 1]
[1 2 0]
[1 2 0]
[1 2 0]
[2 2 0]
[1 2 0]
[1 2 0]
[1 2 0]
[1 2 0]
```

In the preceding example, we sample from three distributions: in the first, the action with index `1` is chosen with probability 80%; in the second distribution, we always select action number 2; and in the third, actions 0 and 1 are equally likely.

The agent

The agent entity provides a unified way of bridging observations from the environment and the actions that we want to execute. So far, you have seen only a simple, stateless DQN agent that uses a neural network (NN) to obtain actions' values from the current observation and behaves greedily on those values. We have used epsilon-greedy behavior to explore the environment, but this doesn't change the picture much.

In the RL field, this could be more complicated. For example, instead of predicting the values of the actions, our agent could predict a probability distribution over the actions. Such agents are called *policy agents*, and we will talk about those methods in part three of the book.

The other requirement could be the necessity for the agent to keep a state between observations. For example, very often one observation (or even the k last observation) is not enough to make a decision about the action, and we want to keep some memory in the agent to capture the necessary information. There is a whole subdomain of RL that tries to address this complication with partially observable Markov decision process (POMDP) formalism, which is not covered in the book.

The third variant of the agent is very common in *continuous control problems*, which will be discussed in part four of the book. For now, it will be enough to say that in such cases, actions are not discrete anymore but some continuous value, and the agent needs to predict them from the observations.

To capture all those variants and make the code flexible, the agent in PTAN is implemented as an extensible hierarchy of classes with the `ptan.agent.BaseAgent` abstract class at the top. From a high level, the agent needs to accept the batch of observations (in the form of a NumPy array) and return the batch of actions that it wants to take. The batch is used to make the processing more efficient, as processing several observations in one pass in a graphics processing unit (GPU) is frequently much faster than processing them individually.

The abstract base class doesn't define the types of input and output, which makes it very flexible and easy to extend. For example, in the continuous domain, our actions will no longer be indices of discrete actions, but float values.

In any case, the agent can be seen as something that knows how to convert observations into actions, and it's up to the agent how to do this. In general, there are no assumptions made on observation and action types, but the concrete implementation of agents is more limiting. PTAN provides two of the most common ways to convert observations into actions: `DQNAgent` and `PolicyAgent`.

In real problems, a custom agent is often needed. These are some of the reasons:

- The architecture of the NN is fancy—its action space is a mixture of continuous and discrete, it has multimodal observations (text and pixels, for example), or something like that.
- You want to use non-standard exploration strategies, for example, the Ornstein–Uhlenbeck process (a very popular exploration strategy in the continuous control domain).

- You have a POMDP environment, and the agent's decision is not fully defined by observations, but by some internal agent state (which is also the case for Ornstein–Uhlenbeck exploration).

All those cases are easily supported by subclassing the `BaseAgent` class, and in the rest of the book, several examples of such redefinition will be given.

Let's now check the standard agents provided by the library: `DQNAgent` and `PolicyAgent`. The complete example is in `Chapter07/02_agents.py`.

DQNAgent

This class is applicable in Q-learning when the action space is not very large, which covers Atari games and lots of classical problems. This representation is not universal, and later in the book, you will see ways of dealing with that. `DQNAgent` takes a batch of observations on input (as a NumPy array), applies the network on them to get Q-values, and then uses the provided `ActionSelector` to convert Q-values to indices of actions.

Let's consider a small example. For simplicity, our network always produces the same output for the input batch.

```
class DQNNet(nn.Module):
    def __init__(self, actions: int):
        super(DQNNet, self).__init__()
        self.actions = actions

    def forward(self, x):
        return torch.eye(x.size()[0], self.actions)
```

Once we have defined the above class, we can use it as a DQN model:

```
>>> net = DQNNet(actions=3)
>>> net(torch.zeros(2, 10))
tensor([[1., 0., 0.],
        [0., 1., 0.]])
```

We start with the simple argmax policy, so the agent will always return actions corresponding to 1s in the network output.

```
>>> selector = ptan.actions.ArgmaxActionSelector()
>>> agent = ptan.agent.DQNAgent(dqn_model=net, action_
selector=selector)
>>> agent(torch.zeros(2, 5))
(array([0, 1]), [None, None])
```

On the input, a batch of two observations, each having five values, was given, and on the output, the agent returned a tuple of two objects:

- An array with actions to be executed for every batch. In our case, this is action 0 for the first batch sample and action 1 for the second.

- A list with the agent's internal state. This is used for stateful agents and is a list of `None` in our case. As our agent is stateless, you can ignore it.

Now let's make the agent with an epsilon-greedy exploration strategy. For this, we just need to pass a different action selector:

```
>>> selector = ptan.actions.EpsilonGreedyActionSelector(epsilon=1.0)
>>> agent = ptan.agent.DQNAgent(dqn_model=net, action_
selector=selector)
>>> agent(torch.zeros(10, 5))[0]
array([2, 0, 0, 0, 1, 2, 1, 2, 2, 1])
```

As `epsilon` is `1.0`, all the actions will be random, regardless of the network's output. But we can change the epsilon value on the fly, which is very handy during the training when we are supposed to anneal `epsilon` over time:

```
>>> selector.epsilon = 0.5
>>> agent(torch.zeros(10, 5))[0]
array([0, 1, 0, 1, 0, 0, 2, 1, 1, 2])
>>> selector.epsilon = 0.1
>>> agent(torch.zeros(10, 5))[0]
array([0, 1, 2, 0, 0, 0, 0, 0, 2, 0])
```

PolicyAgent

`PolicyAgent` expects the network to produce policy distribution over a discrete set of actions. Policy distribution could be either logits (unnormalized) or a normalized distribution. In practice, you should always use logits to improve the numeric stability of the training process.

Let's reimplement our previous example, but now the network will produce probability:

```
class PolicyNet(nn.Module):
    def __init__(self, actions: int):
        super(PolicyNet, self).__init__()
        self.actions = actions

    def forward(self, x):
        # Now we produce the tensor with first two actions
        # having the same logit scores
```

```
        shape = (x.size()[0], self.actions)
        res = torch.zeros(shape, dtype=torch.float32)
        res[:, 0] = 1
        res[:, 1] = 1
        return res
```

The class above could be used to get the action logits for a batch of observations (which is ignored in our example):

```
>>> net = PolicyNet(actions=5)
>>> net(torch.zeros(6, 10))
tensor([[1., 1., 0., 0., 0.],
        [1., 1., 0., 0., 0.],
        [1., 1., 0., 0., 0.],
        [1., 1., 0., 0., 0.],
        [1., 1., 0., 0., 0.],
        [1., 1., 0., 0., 0.]])
```

Now we can use `PolicyAgent` in combination with `ProbabilityActionSelector`. As the latter expects normalized probabilities, we need to ask `PolicyAgent` to apply softmax to the network's output.

```
>>> selector = ptan.actions.ProbabilityActionSelector()
>>> agent = ptan.agent.PolicyAgent(model=net, action_
selector=selector, apply_softmax=True)
>>> agent(torch.zeros(6, 5))[0]
array([0, 4, 0, 0, 1, 2])
```

Please note that the softmax operation produces non-zero probabilities for zero logits, so our agent can still select actions >1.

```
>>> torch.nn.functional.softmax(net(torch.zeros(1, 10)), dim=1)
tensor([[0.3222, 0.3222, 0.1185, 0.1185, 0.1185]])
```

Experience source

The agent abstraction described in the previous section allows us to implement environment communications in a generic way. These communications happen in the form of trajectories, produced by applying the agent's actions to the Gym environment.

At a high level, experience source classes take the agent instance and environment and provide you with step-by step data from the trajectories. The functionality of those classes includes:

- Support of multiple environments being communicated at the same time. This allows efficient GPU utilization as a batch of observations is being processed by the agent at once.

- A trajectory can be preprocessed and presented in a convenient form for further training. For example, there is an implementation of subtrajectory rollouts with accumulation of the reward. That preprocessing is convenient for DQN and n-step DQN, when we are not interested in individual intermediate steps in subtrajectories, so they can be dropped. This saves memory and reduces the amount of code we need to write.

- Support of vectorized environments from OpenAI Universe. We will cover this in *Chapter 17, Continuous Action Space*, for web automation and MiniWoB environments.

So, the experience source classes act as a "magic black box" to hide the environment interaction and trajectory handling complexities from the library user. But the overall PTAN philosophy is to be flexible and extensible, so if you want, you can subclass one of the existing classes or implement your own version as needed.

There are three classes provided by the system:

- `ExperienceSource`: using the agent and the set of environments, it produces n-step subtrajectories with all intermediate steps

- `ExperienceSourceFirstLast`: this is the same as `ExperienceSource`, but instead of a full subtrajectory (with all steps), it keeps only the first and last steps, with proper reward accumulation in between. This can save a lot of memory in the case of n-step DQN or advantage actor-critic (A2C) rollouts

- `ExperienceSourceRollouts`: this follows the asynchronous advantage actor-critic (A3C) rollouts scheme described in Mnih's paper about Atari games (referenced in *Chapter 12, The Actor-Critic Method*)

All the classes are written to be efficient both in terms of central processing unit (CPU) and memory, which is not very important for toy problems, but might become an issue when you want to solve Atari games and need to keep 10M samples in the replay buffer using commodity hardware.

Toy environment

For demonstration, we will implement a very simple Gym environment with a small predictable observation state to show how experience source classes work. This environment has integer observation, which increases from 0 to 4, integer action, and a reward equal to the action given.

```
class ToyEnv(gym.Env):
    def __init__(self):
        super(ToyEnv, self).__init__()
        self.observation_space = gym.spaces.Discrete(n=5)
        self.action_space = gym.spaces.Discrete(n=3)
```

```
        self.step_index = 0

    def reset(self):
        self.step_index = 0
        return self.step_index

    def step(self, action):
        is_done = self.step_index == 10
        if is_done:
            return self.step_index % self.observation_space.n, \
                    0.0, is_done, {}
        self.step_index += 1
        return self.step_index % self.observation_space.n, \
                float(action), self.step_index == 10, {}
```

In addition to this environment, we will use an agent that always generates fixed actions regardless of observations:

```
class DullAgent(ptan.agent.BaseAgent):
    """
    Agent always returns the fixed action
    """
    def __init__(self, action: int):
        self.action = action

    def __call__(self, observations: List[Any],
                    state: Optional[List] = None) \
            -> Tuple[List[int], Optional[List]]:
        return [self.action for _ in observations], state
```

The ExperienceSource class

The first class is `ptan.experience.ExperienceSource`, which generates chunks of agent trajectories of the given length. The implementation automatically handles the end of episode situation (when the `step()` method in the environment returns `is_done=True`) and resets the environment.

The constructor accepts several arguments:

- The Gym environment to be used. Alternatively, it could be the list of environments.

- The agent instance.

- `steps_count=2`: the length of subtrajectories to be generated.

- `vectorized=False`: if set to `True`, the environment needs to be an OpenAI Universe vectorized environment. We will discuss such environments in detail in *Chapter 16, Web Navigation*.

The class instance provides the standard Python iterator interface, so you can just iterate over this to get subtrajectories:

```
>>> env = ToyEnv()
>>> agent = DullAgent(action=1)
>>> exp_source = ptan.experience.ExperienceSource(env=env,
agent=agent, steps_count=2)
>>> for idx, exp in enumerate(exp_source):
...     if idx > 2:
...         break
...     print(exp)
...
(Experience(state=0, action=1, reward=1.0, done=False),
Experience(state=1, action=1, reward=1.0, done=False))
(Experience(state=1, action=1, reward=1.0, done=False),
Experience(state=2, action=1, reward=1.0, done=False))
(Experience(state=2, action=1, reward=1.0, done=False),
Experience(state=3, action=1, reward=1.0, done=False))
```

On every iteration, `ExperienceSource` returns a piece of the agent's trajectory in environment communication. It might look simple, but there are several things happening under the hood of our example:

1. `reset()` was called in the environment to get the initial state
2. The agent was asked to select the action to execute from the state returned
3. The `step()` method was executed to get the reward and the next state
4. This next state was passed to the agent for the next action
5. Information about the transition from one state to the next state was returned
6. The process iterated (from step 3) until it iterated over the experience source

If the agent changes the way it generates actions (we can get this by updating the network weights, decreasing epsilon, or by some other means), it will immediately affect the experience trajectories that we get.

The `ExperienceSource` instance returns tuples of length equal to or less than the argument `step_count` passed on construction. In our case, we asked for two-step subtrajectories, so tuples will be of length 2 or 1 (at the end of episodes). Every object in a tuple is an instance of the `ptan.experience.Experience` class, which is a `namedtuple` with the following fields:

- `state`: the state we observed before taking the action
- `action`: the action we completed
- `reward`: the immediate reward we got from `env`
- `done`: whether the episode was done

If the episode reaches the end, the subtrajectory will be shorter and the underlying environment will be reset automatically, so we don't need to bother with this and can just keep iterating.

```
>>> for idx, exp in enumerate(exp_source):
>>>     if idx > 15:
>>>         break
>>>     print(exp)
(Experience(state=0, action=1, reward=1.0, done=False),
Experience(state=1, action=1, reward=1.0, done=False))
......
(Experience(state=3, action=1, reward=1.0, done=False),
Experience(state=4, action=1, reward=1.0, done=True))
(Experience(state=4, action=1, reward=1.0, done=True),)
(Experience(state=0, action=1, reward=1.0, done=False),
Experience(state=1, action=1, reward=1.0, done=False))
......
(Experience(state=0, action=1, reward=1.0, done=False),
Experience(state=1, action=1, reward=1.0, done=False))
```

We can ask `ExperienceSource` for subtrajectories of any length.

```
>>> exp_source = ptan.experience.ExperienceSource(env=env,
...                 agent=agent, steps_count=4)
>>> next(iter(exp_source))
(Experience(state=0, action=1, reward=1.0, done=False),
 Experience(state=1, action=1, reward=1.0, done=False),
 Experience(state=2, action=1, reward=1.0, done=False),
 Experience(state=3, action=1, reward=1.0, done=False))
```

We can pass it several instances of `gym.Env`. In that case, they will be used in round-robin fashion.

```
>>> exp_source = ptan.experience.ExperienceSource(env=[env, env],
agent=agent, steps_count=4)
>>> for idx, exp in enumerate(exp_source):
...     if idx > 4:
...         break
...     print(exp)
(Experience(state=0, action=1, reward=1.0, done=False),
Experience(state=1, action=1, reward=1.0, done=False))
(Experience(state=0, action=1, reward=1.0, done=False),
Experience(state=1, action=1, reward=1.0, done=False))
(Experience(state=1, action=1, reward=1.0, done=False),
Experience(state=2, action=1, reward=1.0, done=False))
(Experience(state=1, action=1, reward=1.0, done=False),
Experience(state=2, action=1, reward=1.0, done=False))
```

```
(Experience(state=2, action=1, reward=1.0, done=False),
Experience(state=3, action=1, reward=1.0, done=False))
```

ExperienceSourceFirstLast

The class `ExperienceSource` provides us with full subtrajectories of the given length as the list of (*s*, *a*, *r*) objects. The next state, *s'*, is returned in the next tuple, which is not always convenient. For example, in DQN training, we want to have tuples (*s*, *a*, *r*, *s'*) at once to do one-step Bellman approximation during the training. In addition, some extension of DQN, like n-step DQN, might want to collapse longer sequences of observations into (*first-state*, *action*, *total-reward-for-n-steps*, *state-after-step-n*).

To support this in a generic way, a simple subclass of `ExperienceSource` is implemented: `ExperienceSourceFirstLast`. It accepts almost the same arguments in the constructor, but returns different data.

```
>>> exp_source = ptan.experience.ExperienceSourceFirstLast(env, agent,
gamma=1.0, steps_count=1)

>>> for idx, exp in enumerate(exp_source):
...     print(exp)
...     if idx > 10:
...         break
ExperienceFirstLast(state=0, action=1, reward=1.0, last_state=1)
ExperienceFirstLast(state=1, action=1, reward=1.0, last_state=2)
ExperienceFirstLast(state=2, action=1, reward=1.0, last_state=3)
ExperienceFirstLast(state=3, action=1, reward=1.0, last_state=4)
ExperienceFirstLast(state=4, action=1, reward=1.0, last_state=0)
ExperienceFirstLast(state=0, action=1, reward=1.0, last_state=1)
ExperienceFirstLast(state=1, action=1, reward=1.0, last_state=2)
ExperienceFirstLast(state=2, action=1, reward=1.0, last_state=3)
ExperienceFirstLast(state=3, action=1, reward=1.0, last_state=4)
ExperienceFirstLast(state=4, action=1, reward=1.0, last_state=None)
ExperienceFirstLast(state=0, action=1, reward=1.0, last_state=1)
ExperienceFirstLast(state=1, action=1, reward=1.0, last_state=2)
```

Now it returns a single object on every iteration, which is again a `namedtuple` with the following fields:

- `state`: the state we used to decide on the action to take
- `action`: the action we took at this step
- `reward`: the partial accumulated reward for `steps_count` (in our case, `steps_count=1`, so it is equal to the immediate reward)
- `last_state`: the state we got after executing the action. If our episode ends, we have `None` here

This data is much more convenient for DQN training, as we can apply Bellman approximation directly to it.

Let's check the result with a larger number of steps:

```
>>> exp_source = ptan.experience.ExperienceSourceFirstLast(env,
...                    agent, gamma=1.0, steps_count=2)
>>> for idx, exp in enumerate(exp_source):
...     print(exp)
...     if idx > 10:
...         break
ExperienceFirstLast(state=0, action=1, reward=2.0, last_state=2)
ExperienceFirstLast(state=1, action=1, reward=2.0, last_state=3)
ExperienceFirstLast(state=2, action=1, reward=2.0, last_state=4)
ExperienceFirstLast(state=3, action=1, reward=2.0, last_state=0)
ExperienceFirstLast(state=4, action=1, reward=2.0, last_state=1)
ExperienceFirstLast(state=0, action=1, reward=2.0, last_state=2)
ExperienceFirstLast(state=1, action=1, reward=2.0, last_state=3)
ExperienceFirstLast(state=2, action=1, reward=2.0, last_state=4)
ExperienceFirstLast(state=3, action=1, reward=2.0, last_state=None)
ExperienceFirstLast(state=4, action=1, reward=1.0, last_state=None)
ExperienceFirstLast(state=0, action=1, reward=2.0, last_state=2)
ExperienceFirstLast(state=1, action=1, reward=2.0, last_state=3)
```

So, now we are collapsing two steps on every iteration and calculating the immediate reward (that's why `reward=2.0` for most of the samples). More interesting samples are at the end of the episode:

```
ExperienceFirstLast(state=3, action=1, reward=2.0, last_state=None)
ExperienceFirstLast(state=4, action=1, reward=1.0, last_state=None)
```

As the episode ends, we have `last_state=None` in those samples, but additionally, we calculate the reward for the tail of the episode. Those tiny details are very easy to implement wrongly if you are doing all the trajectory handling yourself.

Experience replay buffers

In DQN, we rarely deal with immediate experience samples, as they are heavily correlated, which leads to instability in the training. Normally, we have large replay buffers, which are populated with experience pieces. Then the buffer is sampled (randomly or with priority weights) to get the training batch. The replay buffer normally has a maximum capacity, so old samples are pushed out when the replay buffer reaches the limit.

There are several implementation tricks here, which become extremely important when you need to deal with large problems:

- How to efficiently sample from a large buffer
- How to push old samples from the buffer
- In the case of a prioritized buffer, how priorities need to be maintained and handled in the most efficient way

All this becomes a quite non-trivial task if you want to solve Atari, keeping 10-100M samples, where every sample is an image from the game. A small mistake can lead to a 10-100x memory increase and major slowdowns of the training process.

PTAN provides several variants of replay buffers, which integrate simply with `ExperienceSource` and `Agent` machinery. Normally, what you need to do is ask the buffer to pull a new sample from the source and sample the training batch. The provided classes are:

- `ExperienceReplayBuffer`: a simple replay buffer of predefined size with uniform sampling.
- `PrioReplayBufferNaive`: a simple, but not very efficient, prioritized replay buffer implementation. The complexity of sampling is $O(n)$, which might become an issue with large buffers. This version has the advantage over the optimized class, having much easier code.
- `PrioritizedReplayBuffer`: uses segment trees for sampling, which makes the code cryptic, but with $O(\log(n))$ sampling complexity.

The following shows how the replay buffer could be used:

```
>>> env = ToyEnv()
>>> agent = DullAgent(action=1)
>>> exp_source = ptan.experience.ExperienceSourceFirstLast(env, agent,
gamma=1.0, steps_count=1)
>>> buffer = ptan.experience.ExperienceReplayBuffer(exp_source,
buffer_size=100)
>>> len(buffer)
0
```

All replay buffers provide the following interface:

- A Python iterator interface to walk over all the samples in the buffer
- The method `populate(N)` to get *N* samples from the experience source and put them into the buffer
- The method `sample(N)` to get the batch of *N* experience objects

So, the normal training loop for DQN looks like an infinite repetition of the following steps:

1. Call `buffer.populate(1)` to get a fresh sample from the environment
2. `batch = buffer.sample(BATCH_SIZE)` to get the batch from the buffer
3. Calculate the loss on the sampled batch
4. Backpropagate
5. Repeat until convergence (hopefully)

All the rest happens automatically: resetting the environment, handling subtrajectories, buffer size maintenance, and so on.

```
>>> for step in range(6):
...     buffer.populate(1)
...     if len(buffer) < 5:
...         continue
...     batch = buffer.sample(4)
...     print("Train time, %d batch samples:" % len(batch))
...     for s in batch:
...         print(s)
Train time, 4 batch samples:
ExperienceFirstLast(state=0, action=1, reward=1.0, last_state=1)
ExperienceFirstLast(state=3, action=1, reward=1.0, last_state=4)
ExperienceFirstLast(state=2, action=1, reward=1.0, last_state=3)
ExperienceFirstLast(state=4, action=1, reward=1.0, last_state=0)
Train time, 4 batch samples:
ExperienceFirstLast(state=3, action=1, reward=1.0, last_state=4)
ExperienceFirstLast(state=2, action=1, reward=1.0, last_state=3)
ExperienceFirstLast(state=0, action=1, reward=1.0, last_state=1)
ExperienceFirstLast(state=3, action=1, reward=1.0, last_state=4)
```

The TargetNet class

`TargetNet` is a small but useful class that allows us to synchronize two NNs of the same architecture. The purpose of this was described in the previous chapter: improving training stability. `TargetNet` supports two modes of such synchronization:

- `sync()`: weights from the source network are copied into the target network.
- `alpha_sync()`: the source network's weights are blended into the target network with some alpha weight (between 0 and 1).

The first mode is the standard way to perform a target network sync in discrete action space problems, like Atari and CartPole. We did this in *Chapter 6, Deep Q-Networks*. The latter mode is used in continuous control problems, which will be described in several chapters in part four of the book. In such problems, the transition between two networks' parameters should be smooth, so alpha blending is used, given by the formula $w_i = w_i \alpha + s_i(1 - \alpha)$, where w_i is the target network's ith parameter and s_i is the source network's weight. The following is a small example of how `TargetNet` should be used in code.

Assume we have the following network:

```
class DQNNet(nn.Module):
    def __init__(self):
        super(DQNNet, self).__init__()
        self.ff = nn.Linear(5, 3)

    def forward(self, x):
        return self.ff(x)
```

The target network could be created as follows:

```
>>> net = DQNNet()
>>> net
DQNNet(
  (ff): Linear(in_features=5, out_features=3, bias=True)
)
>>> tgt_net = ptan.agent.TargetNet(net)
```

The target network contains two fields: `model`, which is the reference to the original network, and `target_model`, which is a deep copy of it. If we examine both networks' weights, they will be the same:

```
>>> net.ff.weight
Parameter containing:
tensor([[-0.3287,  0.1219,  0.1804,  0.0993, -0.0996],
        [ 0.1890, -0.1656, -0.0889,  0.3874,  0.1428],
        [-0.3872, -0.2714, -0.0618, -0.2972,  0.4414]], requires_
grad=True)

>>> tgt_net.target_model.ff.weight
Parameter containing:
tensor([[-0.3287,  0.1219,  0.1804,  0.0993, -0.0996],
        [ 0.1890, -0.1656, -0.0889,  0.3874,  0.1428],
        [-0.3872, -0.2714, -0.0618, -0.2972,  0.4414]], requires_
grad=True)
```

They are independent of each other, however, just having the same architecture:

```
>>> net.ff.weight.data += 1.0
>>> net.ff.weight
Parameter containing:
tensor([[0.6713, 1.1219, 1.1804, 1.0993, 0.9004],
        [1.1890, 0.8344, 0.9111, 1.3874, 1.1428],
        [0.6128, 0.7286, 0.9382, 0.7028, 1.4414]], requires_grad=True)
>>> tgt_net.target_model.ff.weight
Parameter containing:
tensor([[-0.3287,  0.1219,  0.1804,  0.0993, -0.0996],
        [ 0.1890, -0.1656, -0.0889,  0.3874,  0.1428],
        [-0.3872, -0.2714, -0.0618, -0.2972,  0.4414]], requires_
grad=True)
```

To synchronize them again, the `sync()` method can be used:

```
>>> tgt_net.sync()
>>> net.ff.weight
Parameter containing:
tensor([[0.6713, 1.1219, 1.1804, 1.0993, 0.9004],
        [1.1890, 0.8344, 0.9111, 1.3874, 1.1428],
        [0.6128, 0.7286, 0.9382, 0.7028, 1.4414]], requires_grad=True)

>>> tgt_net.target_model.ff.weight
Parameter containing:
tensor([[0.6713, 1.1219, 1.1804, 1.0993, 0.9004],
        [1.1890, 0.8344, 0.9111, 1.3874, 1.1428],
        [0.6128, 0.7286, 0.9382, 0.7028, 1.4414]], requires_grad=True)
```

Ignite helpers

PyTorch Ignite was briefly discussed in *Chapter 3, Deep Learning with PyTorch* and it will be used in the rest of the book to reduce the amount of training loop code. PTAN provides several small helpers to simplify integration with Ignite, which reside in the `ptan.ignite` package:

- `EndOfEpisodeHandler`: attached to the `ignite.Engine`, it emits an `EPISODE_COMPLETED` event, and tracks the reward and number of steps in the event in the engine's metrics. It also can emit an event when the average reward for the last episodes reaches the predefined boundary, which is supposed to be used to stop the training on some goal reward.

- `EpisodeFPSHandler`: tracks the number of interactions between the agent and environment that are performed and calculates performance metrics as frames per second. It also keeps the number of seconds passed since the start of the training.

- `PeriodicEvents`: emits corresponding events every 10, 100, or 1,000 training iterations. It is useful for reducing the amount of data being written into TensorBoard.

A detailed illustration of how the preceding classes can be used will be given in the next chapter, when we will use them to reimplement DQN training from *Chapter 6, Deep Q-Networks*, and then check several DQN extensions and tweaks to improve basic DQN convergence.

The PTAN CartPole solver

Let's now take the PTAN classes (without Ignite so far) and try to combine everything together to solve our first environment: CartPole. The complete code is in `Chapter07/06_cartpole.py`. I will show only the important parts of the code related to the material that we have just covered.

```
net = Net(obs_size, HIDDEN_SIZE, n_actions)
tgt_net = ptan.agent.TargetNet(net)
selector = ptan.actions.ArgmaxActionSelector()
selector = ptan.actions.EpsilonGreedyActionSelector(
    epsilon=1, selector=selector)
agent = ptan.agent.DQNAgent(net, selector)
exp_source = ptan.experience.ExperienceSourceFirstLast(
    env, agent, gamma=GAMMA)
buffer = ptan.experience.ExperienceReplayBuffer(
    exp_source, buffer_size=REPLAY_SIZE)
```

In the beginning, we create the NN (the simple two-layer feed-forward NN that we used for CartPole before) and target the NN epsilon-greedy action selector and `DQNAgent`. Then the experience source and replay buffer are created. With those few lines, we have finished with our data pipeline. Now we just need to call `populate()` on the buffer and sample training batches from it:

```
while True:
    step += 1
    buffer.populate(1)

    for reward, steps in exp_source.pop_rewards_steps():
        episode += 1
        print("%d: episode %d done, reward=%.3f, epsilon=%.2f" % (
            step, episode, reward, selector.epsilon))
        solved = reward > 150
    if solved:
        print("Congrats!")
```

```
        break

    if len(buffer) < 2*BATCH_SIZE:
        continue
    batch = buffer.sample(BATCH_SIZE)
```

In the beginning of every training loop iteration, we ask the buffer to fetch one sample from the experience source and then check for the finished episode. The method `pop_rewards_steps()` in the `ExperienceSource` class returns the list of tuples with information about episodes completed since the last call to the method.

```
        states_v, actions_v, tgt_q_v = unpack_batch(
            batch, tgt_net.target_model, GAMMA)
        optimizer.zero_grad()
        q_v = net(states_v)
        q_v = q_v.gather(1, actions_v.unsqueeze(-1)).squeeze(-1)
        loss_v = F.mse_loss(q_v, tgt_q_v)
        loss_v.backward()
        optimizer.step()
        selector.epsilon *= EPS_DECAY

        if step % TGT_NET_SYNC == 0:
            tgt_net.sync()
```

Later in the training loop, we convert a batch of `ExperienceFirstLast` objects into tensors suitable for DQN training, calculate the loss, and do a backpropagation step. Finally, we decay epsilon in our action selector (with the hyperparameters used, epsilon decays to zero at training step 500) and ask the target network to sync every 10 training iterations.

When started, the code should converge in 1,000-2,000 training iterations:

```
$ python 06_cartpole.py
18: episode 1 done, reward=17.000, steps=17, epsilon=1.00
33: episode 2 done, reward=15.000, steps=15, epsilon=0.99
58: episode 3 done, reward=25.000, steps=25, epsilon=0.77
100: episode 4 done, reward=42.000, steps=42, epsilon=0.50
116: episode 5 done, reward=16.000, steps=16, epsilon=0.43
129: episode 6 done, reward=13.000, steps=13, epsilon=0.38
140: episode 7 done, reward=11.000, steps=11, epsilon=0.34
152: episode 8 done, reward=12.000, steps=12, epsilon=0.30
......
348: episode 27 done, reward=9.000, steps=9, epsilon=0.04
```

```
358: episode 28 done, reward=10.000, steps=10, epsilon=0.04
435: episode 29 done, reward=77.000, steps=77, epsilon=0.02
537: episode 30 done, reward=102.000, steps=102, epsilon=0.01
737: episode 31 done, reward=200.000, steps=200, epsilon=0.00
Congrats!
```

Other RL libraries

As we discussed earlier, there are several RL-specific libraries available. Overall, TensorFlow is more popular than PyTorch, as it is more widespread in the deep learning community. The following is my (very biased) list of libraries:

- **Keras-RL**: started by Matthias Plappert in 2016, this includes basic deep RL methods. As suggested by the name, this library was implemented using Keras, which is a higher-level wrapper around TensorFlow (`https://github.com/keras-rl/keras-rl`).

- **Dopamine**: a library from Google published in 2018. It is TensorFlow-specific, which is not surprising for a library from Google (`https://github.com/google/dopamine`).

- **Ray**: a library for distributed execution of machine learning code. It includes RL utilities as part of the library (`https://github.com/ray-project/ray`).

- **TF-Agents**: another library from Google published in 2018 (`https://github.com/tensorflow/agents`).

- **ReAgent**: a library from Facebook Research. It uses PyTorch internally and uses a declarative style of configuration (when you are creating a JSON file to describe your problem), which limits extensibility. But, of course, as it is open source, you can always extend the functionality (`https://github.com/facebookresearch/ReAgent`).

- **Catalyst.RL**: a project started by Sergey Kolesnikov (one of this book's technical reviewers). It uses PyTorch as the backend (`https://github.com/catalyst-team/catalyst`).

- **SLM Lab**: another PyTorch RL library (`https://github.com/kengz/SLM-Lab`).

Summary

In this chapter, we talked about higher-level RL libraries, their motivation, and their requirements. Then we took a deep look into the PTAN library, which will be used in the rest of the book to simplify example code.

In the next chapter, we will return to DQN methods by exploring extensions that researchers and practitioners have discovered since the classic DQN introduction to improve the stability and performance of the method.

8
DQN Extensions

Since DeepMind published its paper on the **deep Q-network** (**DQN**) model (`https://deepmind.com/research/publications/playing-atari-deep-reinforcement-learning`) in 2015, many improvements have been proposed, along with tweaks to the basic architecture, which, significantly, have improved the convergence, stability, and sample efficiency of DeepMind's basic DQN. In this chapter, we will take a deeper look at some of those ideas.

Very conveniently, in October 2017, DeepMind published a paper called *Rainbow: Combining Improvements in Deep Reinforcement Learning* ([1] *Hessel* and others, 2017), which presented the seven most important improvements to DQN; some were invented in 2015, but others were very recent. In this paper, state-of-the-art results on the Atari games suite were reached, just by combining those seven methods. This chapter will go through all those methods. We will analyze the ideas behind them, alongside how they can be implemented and compared to the classic DQN performance. Finally, we will check the combined system with all the methods.

The DQN extensions that we will become familiar with are the following:

- **N-step DQN**: how to improve convergence speed and stability with a simple unrolling of the Bellman equation, and why it's not an ultimate solution
- **Double DQN**: how to deal with DQN overestimation of the values of the actions
- **Noisy networks**: how to make exploration more efficient by adding noise to the network weights
- **Prioritized replay buffer**: why uniform sampling of our experience is not the best way to train
- **Dueling DQN**: how to improve convergence speed by making our network's architecture represent more closely the problem that we are solving

- **Categorical DQN**: how to go beyond the single expected value of the action and work with full distributions

Basic DQN

To get started, we will implement the same DQN method as in *Chapter 6*, *Deep Q-Networks*, but leveraging the high-level libraries described in *Chapter 7*, *Higher-Level RL Libraries*. This will make our code much more compact, which is good, as non-relevant details won't distract us from the method's logic.

At the same time, the purpose of this book is not to teach you how to use the existing libraries, but rather how to develop intuition about RL methods and, if necessary, implement everything from scratch. From my perspective, this is a much more valuable skill, as libraries come and go, but true understanding of the domain will allow you to quickly make sense of other people's code and apply it consciously.

In the basic DQN implementation, we have three modules:

- `Chapter08/lib/dqn_model.py`: the DQN neural network (NN), which is the same as *Chapter 6*, so I won't repeat it

- `Chapter08/lib/common.py`: common functions and declarations shared by the code in this chapter

- `Chapter08/01_dqn_basic.py`: 60 lines of code leveraging the PTAN and Ignite libraries, implementing the basic DQN method

Common library

Let's start with the contents of `lib/common.py`. First of all, we have hyperparameters for our Pong environment from the previous chapter. The hyperparameters are stored in the `SimpleNamespace` object, which is a class from the Python standard library that provides simple access to a variable set of keys and values. This makes it easy to add another configuration set for different, more complicated Atari games and allows us to experiment with hyperparameters:

```
HYPERPARAMS = {
    'pong': SimpleNamespace(**{
        'env_name':         "PongNoFrameskip-v4",
        'stop_reward':      18.0,
        'run_name':         'pong',
        'replay_size':      100000,
        'replay_initial':   10000,
        'target_net_sync':  1000,
        'epsilon_frames':   10**5,
```

```
        'epsilon_start':    1.0,
        'epsilon_final':    0.02,
        'learning_rate':    0.0001,
        'gamma':            0.99,
        'batch_size':       32
    }),
```

The instance of the `SimpleNamespace` class provides a generic container for values; for example, with the preceding hyperparameters, you can write:

```
params = common.HYPERPARAMS['pong']
print("Env %s, gamma %.2f" % (params.env_name, params.gamma))
```

The next function from `lib/common.py` has the name `unpack_batch` and it takes the batch of transitions and converts it into the set of NumPy arrays suitable for training. Every transition from `ExperienceSourceFirstLast` has a type of `ExperienceFirstLast`, which is a `namedtuple` with the following fields:

- `state`: observation from the environment.

- `action`: integer action taken by the agent.

- `reward`: if we have created `ExperienceSourceFirstLast` with the attribute `steps_count=1`, it's just the immediate reward. For larger step counts, it contains the discounted sum of rewards for this number of steps.

- `last_state`: if the transition corresponds to the final step in the environment, then this field is `None`; otherwise, it contains the last observation in the experience chain.

The code of `unpack_batch` is as follows:

```
def unpack_batch(batch:List[ptan.experience.ExperienceFirstLast]):
    states, actions, rewards, dones, last_states = [],[],[],[],[]
    for exp in batch:
        state = np.array(exp.state, copy=False)
        states.append(state)
        actions.append(exp.action)
        rewards.append(exp.reward)
        dones.append(exp.last_state is None)
        if exp.last_state is None:
            lstate = state # the result will be masked anyway
        else:
            lstate = np.array(exp.last_state, copy=False)
        last_states.append(lstate)
    return np.array(states, copy=False), np.array(actions), \
           np.array(rewards, dtype=np.float32), \
           np.array(dones, dtype=np.uint8), \
           np.array(last_states, copy=False)
```

Note how we handle the final transitions in the batch. To avoid the special handling of such cases, for terminal transitions, we store the initial state in the `last_states` array. To make our calculations of the Bellman update correct, we can mask such batch entries during the loss calculation using the `dones` array. Another solution would be to calculate the value of the last states only for non-terminal transitions, but it would make our loss function logic a bit more complicated.

Calculation of the DQN loss function is provided by the function `calc_loss_dqn`, and the code is almost the same as in *Chapter 6, Deep Q-Networks*. One small addition is `torch.no_grad()`, which stops the PyTorch calculation graph from being recorded.

```
def calc_loss_dqn(batch, net, tgt_net, gamma, device="cpu"):
    states, actions, rewards, dones, next_states = \
        unpack_batch(batch)

    states_v = torch.tensor(states).to(device)
    next_states_v = torch.tensor(next_states).to(device)
    actions_v = torch.tensor(actions).to(device)
    rewards_v = torch.tensor(rewards).to(device)
    done_mask = torch.BoolTensor(dones).to(device)

    actions_v = actions_v.unsqueeze(-1)
    state_action_vals = net(states_v).gather(1, actions_v)
    state_action_vals = state_action_vals.squeeze(-1)
    with torch.no_grad():
        next_state_vals = tgt_net(next_states_v).max(1)[0]
        next_state_vals[done_mask] = 0.0

    bellman_vals = next_state_vals.detach() * gamma + rewards_v
    return nn.MSELoss()(state_action_vals, bellman_vals)
```

Besides those core DQN functions, `common.py` provides several utilities related to our training loop, data generation, and TensorBoard tracking. The first such utility is a small class that implements epsilon decay during the training. Epsilon defines the probability of taking the random action by the agent. It should be decayed from 1.0 in the beginning (fully random agent) to some small number, like 0.02 or 0.01. The code is trivial but needed in almost any DQN, so it is provided by the following little class:

```
class EpsilonTracker:
    def __init__(self,
                 selector: ptan.actions.EpsilonGreedyActionSelector,
                 params: SimpleNamespace):
        self.selector = selector
        self.params = params
        self.frame(0)
```

```
def frame(self, frame_idx: int):
    eps = self.params.epsilon_start - \
            frame_idx / self.params.epsilon_frames
    self.selector.epsilon = max(self.params.epsilon_final,eps)
```

Another small function is `batch_generator`, which takes `ExperienceReplayBuffer` (the PTAN class described in *Chapter 7, Higher-Level RL Libraries*) and infinitely generates training batches sampled from the buffer. In the beginning, the function ensures that the buffer contains the required amount of samples.

```
def batch_generator(buffer:ptan.experience.ExperienceReplayBuffer,
                    initial: int, batch_size: int):
    buffer.populate(initial)
    while True:
        buffer.populate(1)
        yield buffer.sample(batch_size)
```

Finally, a lengthy, but nevertheless very useful, function called `setup_ignite` attaches the needed Ignite handlers, showing the training progress and writing metrics to TensorBoard. Let's look at this function piece by piece.

```
def setup_ignite(engine: Engine, params: SimpleNamespace,
                 exp_source, run_name: str,
                 extra_metrics: Iterable[str] = ()):
    warnings.simplefilter("ignore", category=UserWarning)
    handler = ptan_ignite.EndOfEpisodeHandler(
        exp_source, bound_avg_reward=params.stop_reward)
    handler.attach(engine)
    ptan_ignite.EpisodeFPSHandler().attach(engine)
```

Initially, `setup_ignite` attaches two Ignite handlers provided by PTAN:

- `EndOfEpisodeHandler`, which emits the Ignite event every time a game episode ends. It can also fire an event when the averaged reward for episodes crosses some boundary. We use this to detect when the game is finally solved.

- `EpisodeFPSHandler`, a small class that tracks the time the episode has taken and the amount of interactions that we have had with the environment. From this, we calculate frames per second (FPS), which is an important performance metric to track.

```
@engine.on(ptan_ignite.EpisodeEvents.EPISODE_COMPLETED)
def episode_completed(trainer: Engine):
    passed = trainer.state.metrics.get('time_passed', 0)
    print("Episode %d: reward=%.0f, steps=%s, "
            "speed=%.1f f/s, elapsed=%s" % (
```

```
                trainer.state.episode, trainer.state.episode_reward,
                trainer.state.episode_steps,
                trainer.state.metrics.get('avg_fps', 0),
                timedelta(seconds=int(passed))))

        @engine.on(ptan_ignite.EpisodeEvents.BOUND_REWARD_REACHED)
        def game_solved(trainer: Engine):
            passed = trainer.state.metrics['time_passed']
            print("Game solved in %s, after %d episodes "
                  "and %d iterations!" % (
                timedelta(seconds=int(passed)),
                trainer.state.episode, trainer.state.iteration))
            trainer.should_terminate = True
```

Then we install two event handlers, with one being called at the end of an episode. It will show information about the completed episode on the console. Another function will be called when the average reward grows above the boundary defined in the hyperparameters (18.0 in the case of Pong). This function shows a message about the solved game and stops the training.

The rest of the function is related to the TensorBoard data that we want to track:

```
        now = datetime.now().isoformat(timespec='minutes')
        logdir = f"runs/{now}-{params.run_name}-{run_name}"
        tb = tb_logger.TensorboardLogger(log_dir=logdir)
        run_avg = RunningAverage(output_transform=lambda v: v['loss'])
        run_avg.attach(engine, "avg_loss")
```

First, we create a `TensorboardLogger`, a special class provided by Ignite to write into TensorBoard. Our processing function will return the loss value, so we attach the `RunningAverage` transformation (also provided by Ignite) to get a smoothed version of the loss over time.

```
        metrics = ['reward', 'steps', 'avg_reward']
        handler = tb_logger.OutputHandler(
            tag="episodes", metric_names=metrics)
        event = ptan_ignite.EpisodeEvents.EPISODE_COMPLETED
        tb.attach(engine, log_handler=handler, event_name=event)
```

`TensorboardLogger` can track two groups of values from Ignite: outputs (values returned by the transformation function) and metrics (calculated during the training and kept in the `engine` state). `EndOfEpisodeHandler` and `EpisodeFPSHandler` provide metrics, which are updated at the end of every game episode. So, we attach `OutputHandler`, which will write into TensorBoard information about the episode every time it is completed.

```
# write to tensorboard every 100 iterations
ptan_ignite.PeriodicEvents().attach(engine)
metrics = ['avg_loss', 'avg_fps']
metrics.extend(extra_metrics)
handler = tb_logger.OutputHandler(
    tag="train", metric_names=metrics,
    output_transform=lambda a: a)
event = ptan_ignite.PeriodEvents.ITERS_100_COMPLETED
tb.attach(engine, log_handler=handler, event_name=event)
```

Another group of values that we want to track are metrics from the training process: loss, FPS, and, possibly, some custom metrics. Those values are updated every training iteration, but we are going to do millions of iterations, so we will store values in TensorBoard every 100 training iterations; otherwise, the data files will be huge. All this functionality might look too complicated, but it provides us with the unified set of metrics gathered from the training process. In fact, Ignite is not very complicated and provides a very flexible framework. That's it for `common.py`.

Implementation

Now, let's take a look at `01_dqn_basic.py`, which creates the needed classes and starts the training. I'm going to omit non-relevant code and focus only on important pieces. The full version is available in the GitHub repo.

```
env = gym.make(params.env_name)
env = ptan.common.wrappers.wrap_dqn(env)
env.seed(common.SEED)

net = dqn_model.DQN(env.observation_space.shape,
                    env.action_space.n).to(device)
tgt_net = ptan.agent.TargetNet(net)
```

First, we create the environment and apply a set of standard wrappers. We have already discussed them in *Chapter 6, Deep Q-Networks* and will also touch upon them in the next chapter, when we optimize the performance of the Pong solver. Then, we create the DQN model and the target network.

```
selector = ptan.actions.EpsilonGreedyActionSelector(
    epsilon=params.epsilon_start)
epsilon_tracker = common.EpsilonTracker(selector, params)
agent = ptan.agent.DQNAgent(net, selector, device=device)
```

Next, we create the agent, passing it an epsilon-greedy action selector. During the training, epsilon will be decreased by the `EpsilonTracker` class that we have already discussed.

This will decrease the amount of randomly selected actions and give more control to our NN.

```
exp_source = ptan.experience.ExperienceSourceFirstLast(
    env, agent, gamma=params.gamma)
buffer = ptan.experience.ExperienceReplayBuffer(
    exp_source, buffer_size=params.replay_size)
```

The next two very important objects are `ExperienceSource` and `ExperienceReplayBuffer`. The first one takes the agent and environment and provides transitions over game episodes. Those transitions will be kept in the experience replay buffer.

```
optimizer = optim.Adam(net.parameters(),
                       lr=params.learning_rate)


def process_batch(engine, batch):
    optimizer.zero_grad()
    loss_v = common.calc_loss_dqn(
        batch, net, tgt_net.target_model,
        gamma=params.gamma, device=device)
    loss_v.backward()
    optimizer.step()
    epsilon_tracker.frame(engine.state.iteration)
    if engine.state.iteration % params.target_net_sync == 0:
        tgt_net.sync()
    return {
        "loss": loss_v.item(),
        "epsilon": selector.epsilon,
    }
```

Then we create an optimizer and define the processing function, which will be called for every batch of transitions to train the model. To do this, we call function `common.calc_loss_dqn` and then backpropagate on the result.

This function also asks `EpsilonTracker` to decrease the epsilon and does periodical target network synchronization.

```
engine = Engine(process_batch)
common.setup_ignite(engine, params, exp_source, NAME)
engine.run(common.batch_generator(buffer, params.replay_initial,
                                  params.batch_size))
```

And, finally, we create the Ignite `Engine` object, configure it using a function from `common.py`, and run our training process.

Results

Okay, let's start the training.

```
rl_book_samples/Chapter08$ ./01_dqn_basic.py --cuda
Episode 2: reward=-21, steps=809, speed=0.0 f/s, elapsed=0:00:15
Episode 3: reward=-21, steps=936, speed=0.0 f/s, elapsed=0:00:15
Episode 4: reward=-21, steps=817, speed=0.0 f/s, elapsed=0:00:15
Episode 5: reward=-20, steps=927, speed=0.0 f/s, elapsed=0:00:15
Episode 6: reward=-19, steps=1164, speed=0.0 f/s, elapsed=0:00:15
Episode 7: reward=-20, steps=955, speed=0.0 f/s, elapsed=0:00:15
Episode 8: reward=-21, steps=783, speed=0.0 f/s, elapsed=0:00:15
Episode 9: reward=-21, steps=785, speed=0.0 f/s, elapsed=0:00:15
Episode 10: reward=-19, steps=1030, speed=0.0 f/s, elapsed=0:00:15
Episode 11: reward=-21, steps=761, speed=0.0 f/s, elapsed=0:00:15
Episode 12: reward=-21, steps=968, speed=162.7 f/s, elapsed=0:00:19
Episode 13: reward=-19, steps=1081, speed=162.7 f/s, elapsed=0:00:26
Episode 14: reward=-19, steps=1184, speed=162.7 f/s, elapsed=0:00:33
Episode 15: reward=-20, steps=894, speed=162.7 f/s, elapsed=0:00:39
Episode 16: reward=-21, steps=880, speed=162.6 f/s, elapsed=0:00:44
...
```

Every line in the output is written at the end of the next episode, showing the episode reward, a count of steps, speed, and the total training time. For the basic DQN version, it usually takes about 1M frames to reach the mean reward of 18, so be patient. During the training, we can check the dynamics of the training process in TensorBoard, which shows charts for epsilon, raw reward values, average reward, and speed. The following charts show the reward and the number of steps for episodes. (The bottom x axis shows wall clock time, and the top axis is the episode number.)
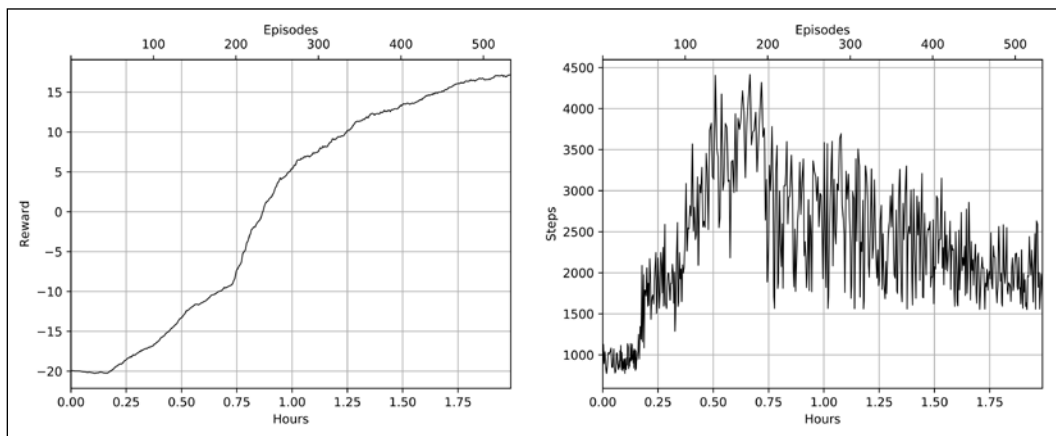
Figure 8.1: Information about episodes played during the training
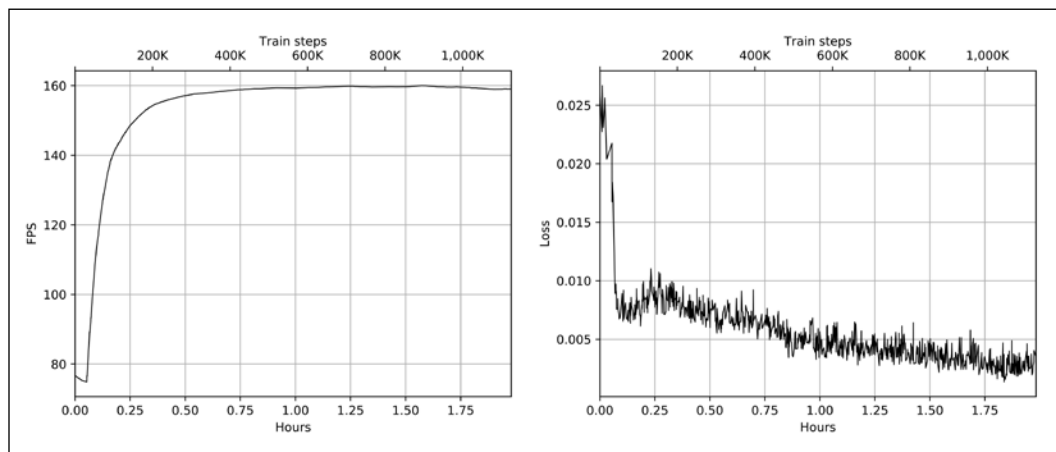
Figure 8.2: Training metrics: speed and loss

N-step DQN

The first improvement that we will implement and evaluate is quite an old one. It was first introduced in the paper *Learning to Predict by the Methods of Temporal Differences*, by *Richard Sutton* ([2] *Sutton*, 1988). To get the idea, let's look at the Bellman update used in Q-learning once again:

$$Q(s_t, a_t) = r_t + \gamma \max_a Q(s_{t+1}, a_{t+1})$$

This equation is recursive, which means that we can express $Q(s_{t+1}, a_{t+1})$ in terms of itself, which gives us this result:

$$Q(s_t, a_t) = r_t + \gamma \max_a \left[r_{a,t+1} + \gamma \max_{a'} Q(s_{t+2}, a') \right]$$

Value $r_{a,t+1}$ means local reward at time *t+1*, after issuing action *a*. However, if we assume that action *a* at the step *t+1* was chosen optimally, or close to optimally, we can omit the $\max_a$ operation and obtain this:

$$Q(s_t, a_t) = r_t + \gamma r_{t+1} + \gamma^2 \max_{a'} Q(s_{t+2}, a')$$

This value can be unrolled again and again any number of times. As you may guess, this unrolling can be easily applied to our DQN update by replacing one-step transition sampling with longer transition sequences of *n*-steps. To understand why this unrolling will help us to speed up training, let's consider the example illustrated in *Figure 8.3*. Here, we have a simple environment of four states (s_1, s_2, s_3, s_4) and the only action available at every state, except s_4, which is a terminal state.
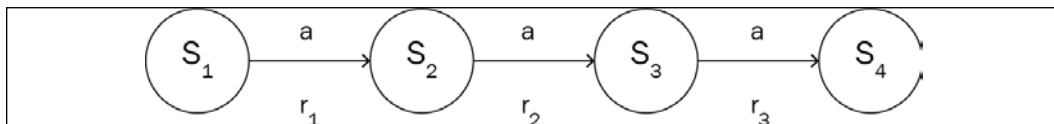
Figure 8.3: A transition diagram for a simple environment

So, what happens in a one-step case? We have three total updates possible (we don't use max, as there is only one action available):

1. $Q(s_1, a) \leftarrow r_1 + \gamma\, Q(s_2, a)$
2. $Q(s_2, a) \leftarrow r_2 + \gamma\, Q(s_3, a)$
3. $Q(s_3, a) \leftarrow r_3$

Let's imagine that, at the beginning of the training, we complete the preceding updates in this order. The first two updates will be useless, as our current $Q(s_2, a)$ and $Q(s_3, a)$ are incorrect and contain initial random data. The only useful update will be update 3, which will correctly assign reward r_3 to the state s_3 prior to the terminal state.

Now let's complete the updates over and over again. On the second iteration, the correct value will be assigned to $Q(s_2, a)$, but the update of $Q(s_1, a)$ will still be noisy. Only on the third iteration will we get the valid values for every Q. So, even in a one-step case, it takes three steps to propagate the correct values to all the states.

Now let's consider a two-step case. This situation again has three updates:

1. $Q(s_1, a) \leftarrow r_1 + \gamma r_2 + \gamma^2 Q(s_3, a)$
2. $Q(s_2, a) \leftarrow r_2 + \gamma r_3$
3. $Q(s_3, a) \leftarrow r_3$

In this case, on the first loop over the updates, the correct values will be assigned to both $Q(s_2, a)$ and $Q(s_3, a)$. On the second iteration, the value of $Q(s_1, a)$ also will be properly updated. So, multiple steps improve the propagation speed of values, which improves convergence. You may be thinking, "If it's so helpful, let's unroll the Bellman equation, say, 100 steps ahead. Will it speed up our convergence 100 times?" Unfortunately, the answer is no.

Despite our expectations, our DQN will fail to converge at all. To understand why, let's again return to our unrolling process, especially where we dropped the $\max_a$. Was it correct? Strictly speaking, no. We omitted the max operation at the intermediate step, assuming that our action selection during experience gathering (or our *policy*) was optimal. What if it wasn't, for example, at the beginning of the training, when our agent acted randomly? In that case, our calculated value for $Q(s_t, a_t)$ may be smaller than the optimal value of the state (as some steps have been taken randomly, but not following the most promising paths by maximizing the Q-value). The more steps on which we unroll the Bellman equation, the more incorrect our update could be.

Our large experience replay buffer will make the situation even worse, as it will increase the chance of getting transitions obtained from the old bad policy (dictated by old bad approximations of Q). This will lead to a wrong update of the current Q approximation, so it can easily break our training progress. This problem is a fundamental characteristic of RL methods, as was briefly mentioned in *Chapter 4, The Cross-Entropy Method*, when we talked about RL methods' taxonomy.

There are two large classes: the **off-policy** and **on-policy** methods. The first class of off-policy methods doesn't depend on "freshness of data." For example, a simple DQN is off-policy, which means that we can use very old data sampled from the environment several million steps ago, and this data will still be useful for learning. That's because we are just updating the value of the action, $Q(s_t, a_t)$, with the immediate reward, plus the discounted current approximation of the best action's value. Even if action a_t was sampled randomly, it doesn't matter because for this particular action a_t, in the state s_t, our update will be correct. That's why in off-policy methods, we can use a very large experience buffer to make our data closer to being independent and identically distributed (i.i.d.).

On the other hand, on-policy methods heavily depend on the training data to be sampled according to the current policy that we are updating. That happens because on-policy methods are trying to improve the current policy indirectly (as in the previous n-step DQN) or directly (all of part three of the book is devoted to such methods).

So, which class of methods is better? Well, it depends. Off-policy methods allow you to train on the previous large history of data or even on human demonstrations, but they usually are slower to converge. On-policy methods are typically faster, but require much more fresh data from the environment, which can be costly. Just imagine a self-driving car trained with the on-policy method. It will cost you a lot of crashed cars before the system learns that walls and trees are things that it should avoid!

You may have a question: why are we talking about an n-step DQN if this "n-stepness" turns it into an on-policy method, which will make our large experience buffer useless? In practice, this is usually not black and white. You may still use an n-step DQN if it will help to speed up DQNs, but you need to be modest with the selection of n. Small values of two or three usually work well, because our trajectories in the experience buffer are not that different from one-step transitions. In such cases, convergence speed usually improves proportionally, but large values of n can break the training process. So, the number of steps should be tuned, but convergence speeding up usually makes it worth doing.

Implementation

As the `ExperienceSourceFirstLast` class already supports the multi-step Bellman unroll, our n-step version of a DQN is extremely simple. There are only two modifications that we need to make to the basic DQN to turn it into an n-step version:

- Pass the count of steps that we want to unroll on `ExperienceSourceFirstLast` creation in the `steps_count` parameter.

- Pass the correct gamma to the `calc_loss_dqn` function. This modification is really easy to overlook, which could be harmful to convergence. As our Bellman is now n-steps, the discount coefficient for the last state in the experience chain will no longer be just γ, but γ^n.

You can find the whole example in `Chapter08/02_dqn_n_steps.py`, but here are the modified lines:

```
exp_source = ptan.experience.ExperienceSourceFirstLast(
    env, agent, gamma=params.gamma, steps_count=args.n)
```

The `args.n` value is a count of steps passed in command-line arguments; the default is to use four steps. Another modification is in `gamma` passed to the `calc_loss_dqn` function:

```
loss_v = common.calc_loss_dqn(
    batch, net, tgt_net.target_model,
    gamma=params.gamma**args.n, device=device)
```

Results

The training module `Chapter08/02_dqn_n_steps.py` can be started as before, with the additional command-line option `-n`, which gives a count of steps to unroll the Bellman equation.

These are charts for our baseline and n-step DQN, with *n* being equal to 2 and 3. As you can see, the Bellman unroll has given a significant boost in convergence.
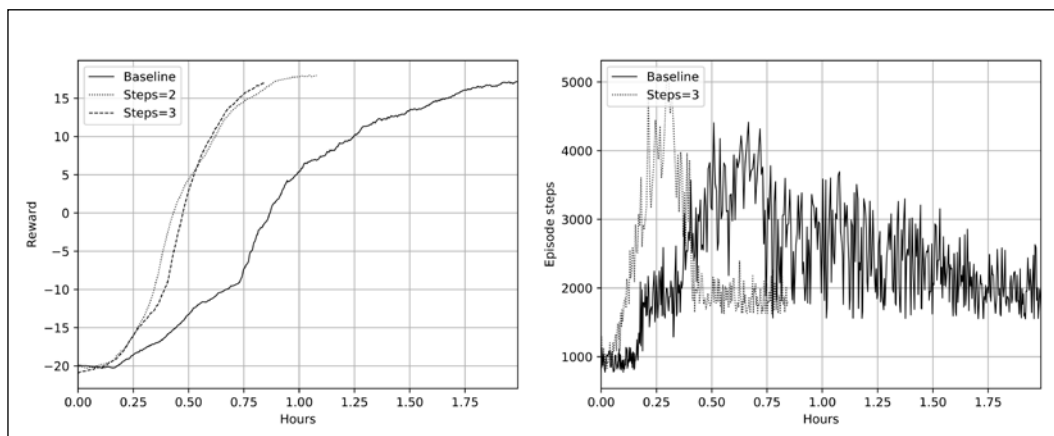
Figure 8.4: The reward and number of steps for basic (one-step) DQN and n-step versions

As you can see in the diagram, the three-step DQN converges more than twice as fast as the simple DQN, which is a nice improvement. So, what about a larger n? The following chart shows the reward dynamics for $n = 3\ldots6$:
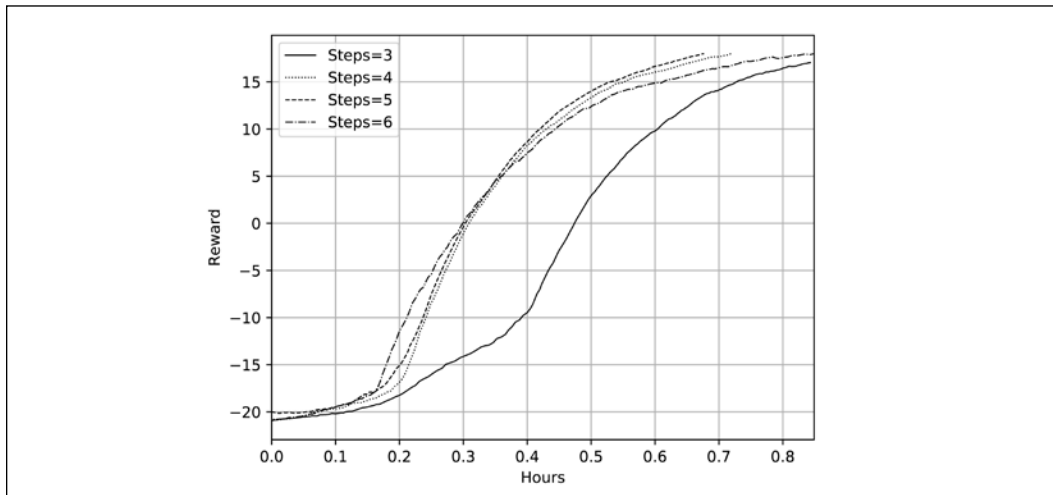
Figure 8.5: A comparison of a two-step DQN and three-step DQN

As you can see, going from three steps to four has given some improvement, but it is much less than before. The variant with $n = 5$ is almost the same as $n = 4$, but $n = 6$ behaves worse. So, in our case, $n = 4$ looks optimal.

Double DQN

The next fruitful idea on how to improve a basic DQN came from DeepMind researchers in the paper titled *Deep Reinforcement Learning with Double Q-Learning* ([3] *van Hasselt, Guez*, and *Silver, 2015*). In the paper, the authors demonstrated that the basic DQN tends to overestimate values for Q, which may be harmful to training performance and sometimes can lead to suboptimal policies. The root cause of this is the max operation in the Bellman equation, but the strict proof is too complicated to write down here. As a solution to this problem, the authors proposed modifying the Bellman update a bit.

In the basic DQN, our target value for Q looked like this:

$$Q(s_t, a_t) = r_t + \gamma \max_a Q'(s_{t+1}, a)$$

$Q'(s_{t+1}, a)$ was Q-values calculated using our target network, so we update with the trained network every *n* steps. The authors of the paper proposed choosing actions for the next state using the trained network, but taking values of Q from the target network. So, the new expression for target Q-values will look like this:

$$Q(s_t, a_t) = r_t + \gamma \max_a Q'\left(s_{t+1}, \arg\max_a Q(s_{t+1}, a)\right)$$

The authors proved that this simple tweak fixes overestimation completely, and they called this new architecture **double DQN**.

Implementation

The core implementation is very simple. What we need to do is slightly modify our loss function. Let's go a step further and compare action values produced by the basic DQN and double DQN. To do this, we store a random held-out set of states and periodically calculate the mean value of the best action for every state in the evaluation set.

The complete example is in `Chapter08/03_dqn_double.py`. Let's first take a look at the loss function:

```
def calc_loss_double_dqn(batch, net, tgt_net, gamma,
                         device="cpu", double=True):
    states, actions, rewards, dones, next_states = \
        common.unpack_batch(batch)
```

The `double` extra argument turns on and off the double DQN way of calculating actions to take.

```
        states_v = torch.tensor(states).to(device)
        actions_v = torch.tensor(actions).to(device)
        rewards_v = torch.tensor(rewards).to(device)
        done_mask = torch.BoolTensor(dones).to(device)
```

The preceding section is the same as before.

```
        actions_v = actions_v.unsqueeze(-1)
        state_action_vals = net(states_v).gather(1, actions_v)
        state_action_vals = state_action_vals.squeeze(-1)
        with torch.no_grad():
            next_states_v = torch.tensor(next_states).to(device)
            if double:
```

```
            next_state_acts = net(next_states_v).max(1)[1]
            next_state_acts = next_state_acts.unsqueeze(-1)
            next_state_vals = tgt_net(next_states_v).gather(
                1, next_state_acts).squeeze(-1)
        else:
            next_state_vals = tgt_net(next_states_v).max(1)[0]
        next_state_vals[done_mask] = 0.0
        exp_sa_vals = next_state_vals.detach()*gamma+rewards_v
    return nn.MSELoss()(state_action_vals, exp_sa_vals)
```

Here is the difference compared to the basic DQN loss function. If double DQN is enabled, we calculate the best action to take in the next state using our main trained network, but values corresponding to this action come from the target network. Of course, this part could be implemented in a faster way, by combining `next_states_v` with `states_v` and calling our main network only once, but it will make the code less clear.

The rest of the function is the same: we mask completed episodes and compute the mean squared error (MSE) loss between Q-values predicted by the network and approximated Q-values. The last function that we consider calculates the values of our held-out state:

```
def calc_values_of_states(states, net, device="cpu"):
    mean_vals = []
    for batch in np.array_split(states, 64):
        states_v = torch.tensor(batch).to(device)
        action_values_v = net(states_v)
        best_action_values_v = action_values_v.max(1)[0]
        mean_vals.append(best_action_values_v.mean().item())
    return np.mean(mean_vals)
```

There is nothing too complicated here: we just split our held-out states array into equal chunks and pass every chunk to the network to obtain action values. From those values, we choose the action with the largest value (for every state) and calculate the mean of such values. As our array with states is fixed for the whole training process, and this array is large enough (in the code we store 1,000 states), we can compare the dynamics of this mean value in both DQN variants.

The rest of the `03_dqn_double.py` file is almost the same; the two differences are usage of our tweaked loss function and keep randomly sampled 1,000 states for periodical evaluation.

Results

To train a double DQN, with the extension enabled, pass the `--double` command-line argument:

```
Chapter08$ ./03_dqn_double.py --cuda --double
```

To compare the Q-values with those for a basic DQN, train it again without the `--double` option. Training will take some time, depending on your computing power. On GTX 1080 Ti, 1M frames take about two hours. In addition, I've noticed that double extension convergence is less frequent than with the basic version. In basic DQN, about one in 10 runs fails to converge, but with double extension, it is about one in three. Very likely, hyperparameters need to be tuned, but our comparison is supposed to check the effect of extensions without touching hyperparameters.

Anyway, as you can see on the following chart, double DQN shows better reward dynamics (the average reward for episodes starts to grow earlier), but the final time to solve the game is almost the same as in our baseline.
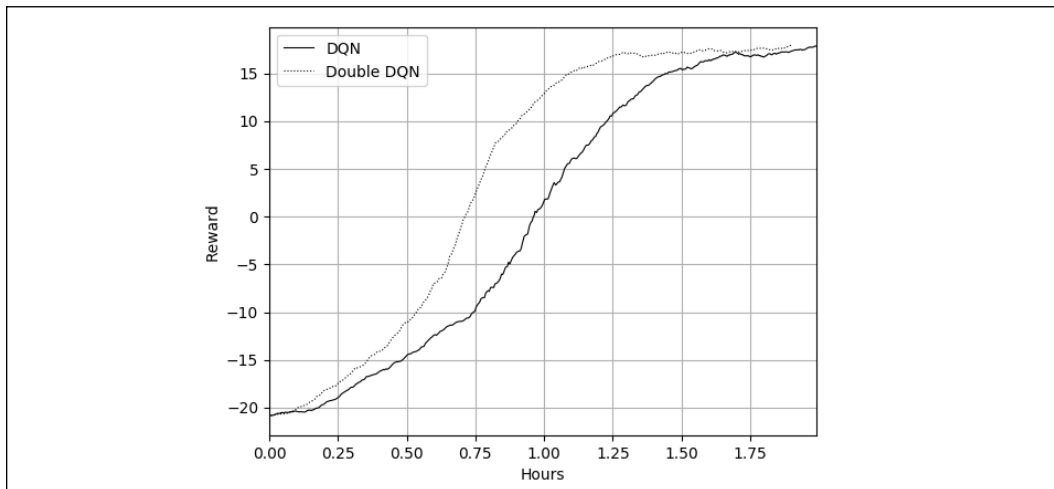
Figure 8.6: Reward dynamics for double and baseline DQN

Besides the standard metrics, the example also outputs the mean value for the held-out set of states. They are shown on the next chart; indeed, basic DQN does overestimation of values, so values are decreasing after some level. In contrast, double DQN grows more consistently. In my experiments, double DQN has only a small effect on the training time, but this doesn't necessary mean that double DQN is useless, as Pong is a simple environment. In more complicated games, double DQN could give better results.
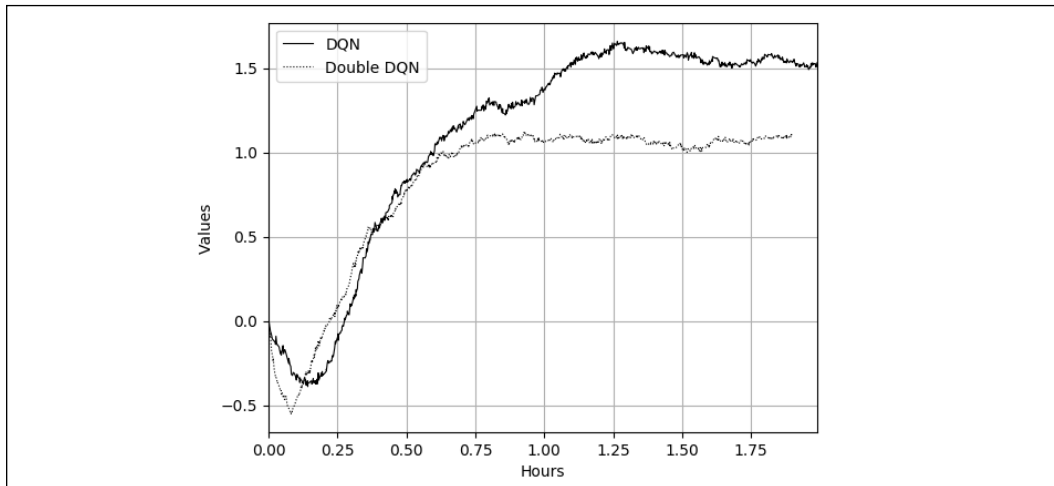
Figure 8.7: Values predicted by the network for held-out states

Noisy networks

The next improvement that we are going to look at addresses another RL problem: exploration of the environment. The paper that we will draw from is called *Noisy Networks for Exploration* ([4] *Fortunato* and others, *2017*) and it has a very simple idea for learning exploration characteristics during training instead of having a separate schedule related to exploration.

Classical DQN achieves exploration by choosing random actions with a specially defined hyperparameter epsilon, which is slowly decreased over time from 1.0 (fully random actions) to some small ratio of 0.1 or 0.02. This process works well for simple environments with short episodes, without much non-stationarity during the game; but even in such simple cases, it requires tuning to make the training processes efficient.

In the *Noisy Networks* paper, the authors proposed a quite simple solution that, nevertheless, works well. They add noise to the weights of fully connected layers of the network and adjust the parameters of this noise during training using backpropagation. Of course, this method shouldn't be confused with "the network decides where to explore more," which is a much more complex approach that also has widespread support (for example, see articles about intrinsic motivation and count-based exploration methods, [5] or [6]). We will discuss advanced exploration techniques in *Chapter 21, Advanced Exploration*.

The authors proposed two ways of adding the noise, both of which work according to their experiments, but they have different computational overheads:

1. **Independent Gaussian noise**: for every weight in a fully connected layer, we have a random value that we draw from the normal distribution. Parameters of the noise, μ and σ, are stored inside the layer and get trained using backpropagation in the same way that we train weights of the standard linear layer. The output of such a "noisy layer" is calculated in the same way as in a linear layer.

2. **Factorized Gaussian noise**: to minimize the number of random values to be sampled, the authors proposed keeping only two random vectors: one with the size of the input and another with the size of the output of the layer. Then, a random matrix for the layer is created by calculating the outer product of the vectors.

Implementation

In PyTorch, both methods can be easily implemented in a very straightforward way. What we need to do is create our own `nn.Linear` layer equivalent with the additional random values sampled every time `forward()` gets called. I've implemented both noisy layers, and their implementations are in `Chapter08/lib/dqn_extra.py` in classes `NoisyLinear` (for independent Gaussian noise) and `NoisyFactorizedLinear` (for the factorized noise variant).

```
class NoisyLinear(nn.Linear):
    def __init__(self, in_features, out_features,
                 sigma_init=0.017, bias=True):
        super(NoisyLinear, self).__init__(
            in_features, out_features, bias=bias)
        w = torch.full((out_features, in_features), sigma_init)
        self.sigma_weight = nn.Parameter(w)
        z = torch.zeros(out_features, in_features)
        self.register_buffer("epsilon_weight", z)
        if bias:
            w = torch.full((out_features,), sigma_init)
            self.sigma_bias = nn.Parameter(w)
            z = torch.zeros(out_features)
            self.register_buffer("epsilon_bias", z)
        self.reset_parameters()
```

In the constructor, we create a matrix for σ. (Values of μ will be stored in a matrix inherited from `nn.Linear`.) To make sigmas trainable, we need to wrap the tensor in an `nn.Parameter`.

The `register_buffer` method creates a tensor in the network that won't be updated during backpropagation, but will be handled by the `nn.Module` machinery (for example, it will be copied to GPU with the `cuda()` call). An extra parameter and buffer are created for the bias of the layer. The initial value for sigmas (0.017) was taken from the *Noisy Networks* article cited in the beginning of this section. At the end, we call the `reset_parameters()` method, which was overridden from `nn.Linear` and is supposed to perform the initialization of the layer.

```
def reset_parameters(self):
    std = math.sqrt(3 / self.in_features)
    self.weight.data.uniform_(-std, std)
    self.bias.data.uniform_(-std, std)
```

In the `reset_parameters` method, we perform initialization of the `nn.Linear` weight and bias according to the recommendations in the article.

```
def forward(self, input):
    self.epsilon_weight.normal_()
    bias = self.bias
    if bias is not None:
        self.epsilon_bias.normal_()
        bias = bias + self.sigma_bias * \
               self.epsilon_bias.data
    v = self.sigma_weight * self.epsilon_weight.data + \
        self.weight
    return F.linear(input, v, bias)
```

In the `forward()` method, we sample random noise in both weight and bias buffers, and perform linear transformation of the input data in the same way that `nn.Linear` does.

Factorized Gaussian noise works in a similar way and I haven't found much difference in the results, so I'll just include its code here for completeness. If you are curious, you can find the details and equations in the article [4].

```
class NoisyFactorizedLinear(nn.Linear):
    def __init__(self, in_features, out_features,
                 sigma_zero=0.4, bias=True):
        super(NoisyFactorizedLinear, self).__init__(
            in_features, out_features, bias=bias)
        sigma_init = sigma_zero / math.sqrt(in_features)
        w = torch.full((out_features, in_features), sigma_init)
        self.sigma_weight = nn.Parameter(w)
        z1 = torch.zeros(1, in_features)
        self.register_buffer("epsilon_input", z1)
        z2 = torch.zeros(out_features, 1)
```

```
        self.register_buffer("epsilon_output", z2)
        if bias:
            w = torch.full((out_features,), sigma_init)
            self.sigma_bias = nn.Parameter(w)

    def forward(self, input):
        self.epsilon_input.normal_()
        self.epsilon_output.normal_()

        func = lambda x: torch.sign(x) * \
                         torch.sqrt(torch.abs(x))
        eps_in = func(self.epsilon_input.data)
        eps_out = func(self.epsilon_output.data)

        bias = self.bias
        if bias is not None:
            bias = bias + self.sigma_bias * eps_out.t()
        noise_v = torch.mul(eps_in, eps_out)
        v = self.weight + self.sigma_weight * noise_v
        return F.linear(input, v, bias)
```

From the implementation point of view, that's it. What we now need to do to turn the classic DQN into a noisy network variant is just replace `nn.Linear` (which are the two last layers in our DQN network) with the `NoisyLinear` layer (or `NoisyFactorizedLinear` if you wish). Of course, you have to remove all the code related to the epsilon-greedy strategy.

To check the internal noise level during training, we can monitor the signal-to-noise ratio (SNR) of our noisy layers, which is a ratio of RMS(μ) / RMS(σ), where RMS is the root mean square of the corresponding weights. In our case, SNR shows how many times the stationary component of the noisy layer is larger than the injected noise.

Results

After the training, the TensorBoard charts show much better training dynamics. The model was able to reach the mean score of 18 in less than 600k frames seen.
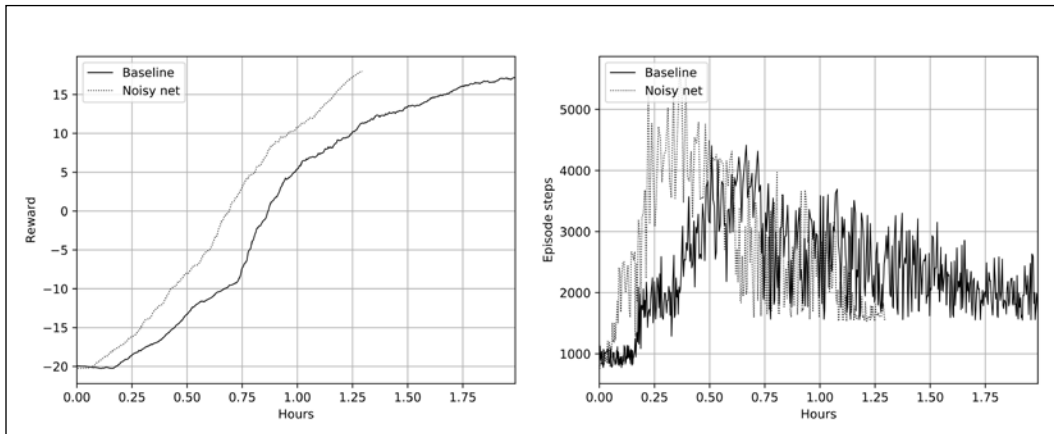
Figure 8.8: Noisy networks compared to the baseline DQN

After checking the SNR chart, you may notice that both layers' noise levels have decreased very quickly. The first layer has gone from 1 to almost 1/2.5 ratio of noise. The second layer is even more interesting, as its noise level decreased from 1/3 in the beginning to 1/15, but after 250k frames (roughly the same time as when raw rewards climbed close to the 20 score), the level of noise in the last layer started to increase back, pushing the agent to explore the environment more. This makes a lot of sense, as after reaching high score levels, the agent basically knows how to play at a good level, but still needs to "polish" its actions to improve the results even more.
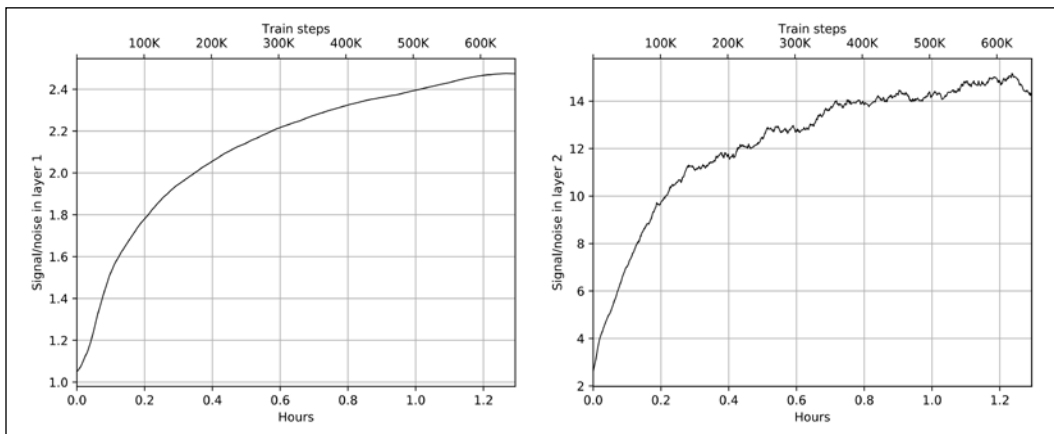
Figure 8.9: Noise level changes during the training

Prioritized replay buffer

The next very useful idea on how to improve DQN training was proposed in 2015 in the paper *Prioritized Experience Replay* ([7] *Schaul* and others, *2015*). This method tries to improve the efficiency of samples in the replay buffer by prioritizing those samples according to the training loss.

The basic DQN used the replay buffer to break the correlation between immediate transitions in our episodes. As we discussed in *Chapter 6*, *Deep Q-Networks*, the examples we experience during the episode will be highly correlated, as most of the time, the environment is "smooth" and doesn't change much according to our actions. However, the stochastic gradient descent (SGD) method assumes that the data we use for training has an i.i.d. property. To solve this problem, the classic DQN method uses a large buffer of transitions, randomly sampled to get the next training batch.

The authors of the paper questioned this uniform random sample policy and proved that by assigning priorities to buffer samples, according to training loss and sampling the buffer proportional to those priorities, we can significantly improve convergence and the policy quality of the DQN. This method's basic idea could be explained as "train more on data that surprises you." The tricky point here is to keep the balance of training on an "unusual" sample and training on the rest of the buffer. If we focus only on a small subset of the buffer, we can lose our i.i.d. property and simply overfit on this subset.

From the mathematical point of view, the priority of every sample in the buffer is calculated as $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$, where p_i is the priority of the ith sample in the buffer and α is the number that shows how much emphasis we give to the priority. If $\alpha = 0$, our sampling will become uniform as in the classic DQN method. Larger values for α put more stress on samples with higher priority. So, it's another hyperparameter to tune, and the starting value of α proposed by the paper is 0.6.

There were several options proposed in the paper for how to define the priority, and the most popular is to make it proportional to the loss for this particular example in the Bellman update. New samples added to the buffer need to be assigned a maximum value of priority to be sure that they will be sampled soon.

By adjusting the priorities for the samples, we are introducing bias into our data distribution (we sample some transitions much more frequently than others), which we need to compensate for if SGD is to work. To get this result, the authors of the study used sample weights, which needed to be multiplied to the individual sample loss. The value of the weight for each sample is defined as $w_i = (N \cdot P(i))^{-\beta}$, where β is another hyperparameter that should be between 0 and 1.

With $\beta = 1$, the bias introduced by the sampling is fully compensated, but the authors showed that it's good for convergence to start with some β between 0 and 1 and slowly increase it to 1 during the training.

Implementation

To implement this method, we have to introduce several changes in our code. First of all, we need a new replay buffer that will track priorities, sample a batch according to them, calculate weights, and let us update priorities after the loss has become known. The second change will be the loss function itself. Now we not only need to incorporate weights for every sample, but we need to pass loss values back to the replay buffer to adjust the priorities of the sampled transitions.

In the example file `Chapter08/05_dqn_prio_replay.py`, we have all those changes implemented. For the sake of simplicity, the new priority replay buffer class uses a very similar storage scheme to our previous replay buffer. Unfortunately, new requirements for prioritization make it impossible to implement sampling in O(1) time to buffer size. If we are using simple lists, every time that we sample a new batch, we need to process all the priorities, which makes our sampling have *O(N)* time complexity in proportion to the buffer size. It's not a big deal if our buffer is small, such as 100k samples, but may become an issue for real-life large buffers of millions of transitions. There are other storage schemes that support efficient sampling in O(log *N*) time, for example, using the segment tree data structure. You can find such implementation in the OpenAI Baselines project: `https://github.com/openai/baselines`. The PTAN library also provides an efficient prioritized replay buffer in the class `ptan.experience.PrioritizedReplayBuffer`. You can update the example to use the more efficient version and check the effect on training performance.

But, for now, let's take a look at the naïve version, whose source code is in `lib/dqn_extra.py`.

```
BETA_START = 0.4
BETA_FRAMES = 100000
```

In the beginning, we define parameters for the β increase rate. Our beta will be changed from 0.4 to 1.0 during the first 100k frames. Now the prioritized replay buffer class:

```
class PrioReplayBuffer:
    def __init__(self, exp_source, buf_size, prob_alpha=0.6):
        self.exp_source_iter = iter(exp_source)
        self.prob_alpha = prob_alpha
        self.capacity = buf_size
        self.pos = 0
        self.buffer = []
```

```
            self.priorities = np.zeros(
                (buf_size, ), dtype=np.float32)
            self.beta = BETA_START
```

The class for the priority replay buffer stores samples in a circular buffer (it allows us to keep a fixed amount of entries without reallocating the list) and a NumPy array to keep priorities. We also store the iterator to the experience source object to pull the samples from the environment.

```
        def update_beta(self, idx):
            v = BETA_START + idx * (1.0 - BETA_START) / \
                BETA_FRAMES
            self.beta = min(1.0, v)
            return self.beta

        def __len__(self):
            return len(self.buffer)

        def populate(self, count):
            max_prio = self.priorities.max() if \
                self.buffer else 1.0
            for _ in range(count):
                sample = next(self.exp_source_iter)
                if len(self.buffer) < self.capacity:
                    self.buffer.append(sample)
                else:
                    self.buffer[self.pos] = sample
                self.priorities[self.pos] = max_prio
                self.pos = (self.pos + 1) % self.capacity
```

The `populate()` method needs to pull the given number of transitions from the `ExperienceSource` object and store them in the buffer. As our storage for the transitions is implemented as a circular buffer, we have two different situations with this buffer:

- When our buffer hasn't reached the maximum capacity, we just need to append a new transition to the buffer.
- If the buffer is already full, we need to overwrite the oldest transition, which is tracked by the `pos` class field, and adjust this position modulo buffer's size.

The method `update_beta` needs to be called periodically to increase beta according to schedule.

```
        def sample(self, batch_size):
            if len(self.buffer) == self.capacity:
```

```
            prios = self.priorities
        else:
            prios = self.priorities[:self.pos]
        probs = prios ** self.prob_alpha
        probs /= probs.sum()
```

In the sample method, we need to convert priorities to probabilities using our α hyperparameter.

```
        indices = np.random.choice(len(self.buffer),
                                batch_size, p=probs)
        samples = [self.buffer[idx] for idx in indices]
```

Then, using those probabilities, we sample our buffer to obtain a batch of samples.

```
        total = len(self.buffer)
        weights = (total * probs[indices]) ** (-self.beta)
        weights /= weights.max()
        return samples, indices, \
                np.array(weights, dtype=np.float32)
```

As the last step, we calculate weights for samples in the batch and return three objects: the batch, indices, and weights. Indices for batch samples are required to update priorities for sampled items.

```
    def update_priorities(self, batch_indices,
                          batch_priorities):
        for idx, prio in zip(batch_indices,
                             batch_priorities):
            self.priorities[idx] = prio
```

The last function of the priority replay buffer allows us to update new priorities for the processed batch. It's the responsibility of the caller to use this function with the calculated losses for the batch.

The next custom function that we have in our example is the loss calculation. As the `MSELoss` class in PyTorch doesn't support weights (which is understandable, as MSE is loss used in regression problems, but weighting of the samples is commonly utilized in classification losses), we need to calculate the MSE and explicitly multiply the result on the weights:

```
def calc_loss(batch, batch_weights, net, tgt_net,
              gamma, device="cpu"):
    states, actions, rewards, dones, next_states = \
        common.unpack_batch(batch)

    states_v = torch.tensor(states).to(device)
```

```
    actions_v = torch.tensor(actions).to(device)
    rewards_v = torch.tensor(rewards).to(device)
    done_mask = torch.BoolTensor(dones).to(device)
    batch_weights_v = torch.tensor(batch_weights).to(device)

    actions_v = actions_v.unsqueeze(-1)
    state_action_vals = net(states_v).gather(1, actions_v)
    state_action_vals = state_action_vals.squeeze(-1)
    with torch.no_grad():
        next_states_v = torch.tensor(next_states).to(device)
        next_s_vals = tgt_net(next_states_v).max(1)[0]
        next_s_vals[done_mask] = 0.0
        exp_sa_vals = next_s_vals.detach() * gamma + rewards_v
    l = (state_action_vals - exp_sa_vals) ** 2
    losses_v = batch_weights_v * l
    return losses_v.mean(), \
           (losses_v + 1e-5).data.cpu().numpy()
```

In the last part of the loss calculation, we implement the same MSE loss but write our expression explicitly, rather than using the library. This allows us to take into account the weights of samples and keep individual loss values for every sample. Those values will be passed to the priority replay buffer to update priorities. A small value is added to every loss to handle the situation of zero loss value, which will lead to zero priority for an entry in the replay buffer.

In the main section of the utility, we have only two updates: the creation of the replay buffer and our processing function. Buffer creation is straightforward, so we will take a look at only a new processing function:

```
def process_batch(engine, batch_data):
    batch, batch_indices, batch_weights = batch_data
    optimizer.zero_grad()
    loss_v, sample_prios = calc_loss(
        batch, batch_weights, net, tgt_net.target_model,
        gamma=params.gamma, device=device)
    loss_v.backward()
    optimizer.step()
    buffer.update_priorities(batch_indices, sample_prios)
    epsilon_tracker.frame(engine.state.iteration)
    if engine.state.iteration % params.target_net_sync == 0:
        tgt_net.sync()
    return {
        "loss": loss_v.item(),
```

```
       "epsilon": selector.epsilon,
       "beta": buffer.update_beta(engine.state.iteration),
    }
```

There are several changes here:

- Our batch now contains three entities: the batch of data, indices of sampled items, and samples' weights.

- We call our new loss function, which accepts weights and returns the additional items' priorities. They are passed to the `buffer.update_priorities` function to reprioritize items that we have sampled.

- We call the `update_beta` method of the buffer to change the beta parameter according to schedule.

Results

This example can be trained as usual. According to my experiments, the prioritized replay buffer took almost the same absolute time to solve the environment: almost two hours. But it took fewer training iterations and fewer episodes. So, wall clock time is the same mostly due to the less efficient replay buffer, which, of course, could be resolved by proper O(log *N*) implementation of the buffer.

Here are the charts of reward dynamics of the baseline (left) and prioritized replay buffer (right). The top axes show the number of game episodes.
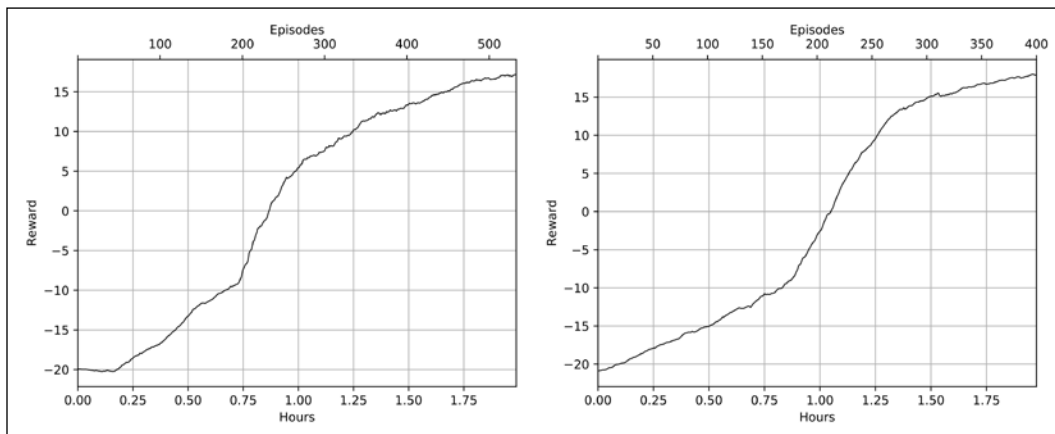
Figure 8.10: Prioritized replay buffer in comparison to basic DQN

Another difference to note on the TensorBoard charts is a much lower loss for the prioritized replay buffer. The following chart shows the comparison.
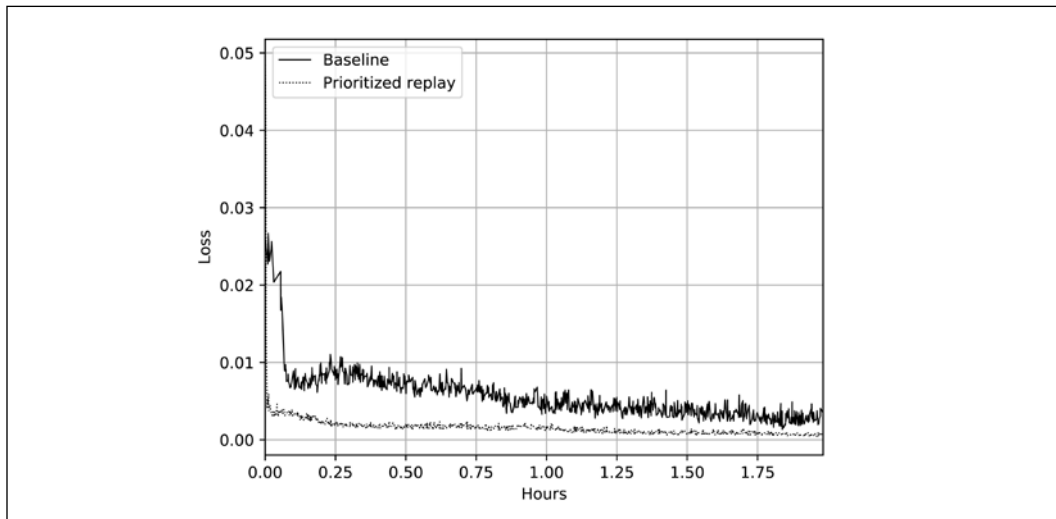
Figure 8.11: The comparison of loss during the training

Lower loss value is also expected and is a good sign that our implementation works. The idea of prioritization is to train more on samples with high loss value, so training becomes more efficient. But there is a danger here: loss value during the training is not the primary objective to optimize; we can have very low loss, but due to a lack of exploration, the final policy learned could be far from being optimal.

Dueling DQN

This improvement to DQN was proposed in 2015, in the paper called *Dueling Network Architectures for Deep Reinforcement Learning* ([8] *Wang* et al., *2015*). The core observation of this paper is that the Q-values, $Q(s, a)$, that our network is trying to approximate can be divided into quantities: the value of the state, $V(s)$, and the advantage of actions in this state, $A(s, a)$.

You have seen the quantity $V(s)$ before, as it was the core of the value iteration method from *Chapter 5*, *Tabular Learning and the Bellman Equation*. It is just equal to the discounted expected reward achievable from this state. The advantage $A(s, a)$ is supposed to bridge the gap from $A(s)$ to $Q(s, a)$, as, by definition, $Q(s, a) = V(s) + A(s, a)$. In other words, the advantage $A(s, a)$ is just the delta, saying how much extra reward some particular action from the state brings us. The advantage could be positive or negative and, in general, can have any magnitude. For example, at some *tipping point*, the choice of one action over another can cost us a lot of the total reward.

The *Dueling* paper's contribution was an explicit separation of the value and the advantage in the network's architecture, which brought better training stability, faster convergence, and better results on the Atari benchmark. The architecture difference from the classic DQN network is shown in the following illustration. The classic DQN network (top) takes features from the convolution layer and, using fully connected layers, transforms them into a vector of Q-values, one for each action. On the other hand, dueling DQN (bottom) takes convolution features and processes them using two independent paths: one path is responsible for $V(s)$ prediction, which is just a single number, and another path predicts individual advantage values, having the same dimension as Q-values in the classic case. After that, we add $V(s)$ to every value of $A(s, a)$ to obtain $Q(s, a)$, which is used and trained as normal.
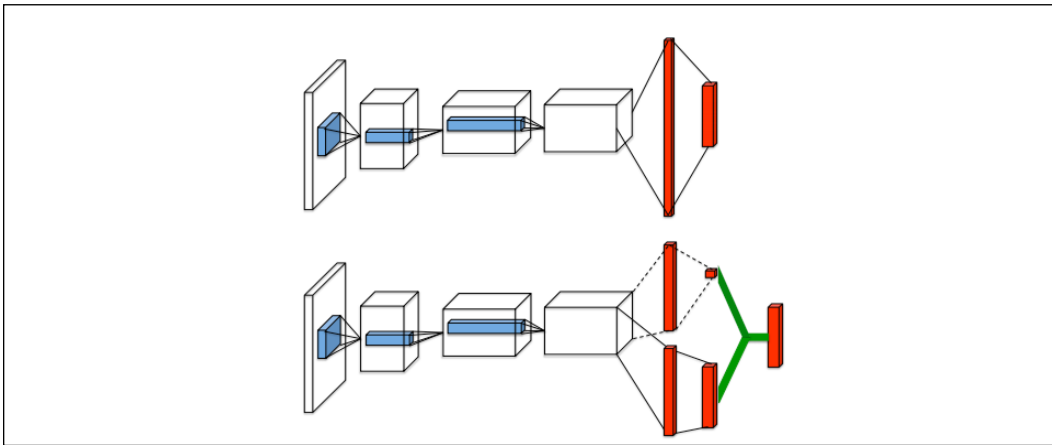
Figure 8.12: A basic DQN (top) and dueling architecture (bottom)

These changes in the architecture are not enough to make sure that the network will learn $V(s)$ and $A(s, a)$ as we want it to. Nothing prevents the network, for example, from predicting some state, $V(s) = 0$, and $A(s) = [1, 2, 3, 4]$, which is completely wrong, as the predicted $V(s)$ is not the expected value of the state. We have yet another constraint to set: we want the mean value of the advantage of any state to be zero. In that case, the correct prediction for the preceding example will be $V(s) = 2.5$ and $A(s) = [-1.5, -0.5, 0.5, 1.5]$.

This constraint could be enforced in various ways, for example, via the loss function; but in the *Dueling* paper, the authors proposed a very elegant solution of subtracting the mean value of the advantage from the Q expression in the network, which effectively pulls the mean for the advantage to zero: $Q(s, a) = V(s) + A(s, a) - \frac{1}{N} \sum_k A(s, k)$. This keeps the changes that need to be made in the classic DQN very simple: to convert it to the double DQN, you need to change only the network architecture, without affecting other pieces of the implementation.

Implementation

A complete example is available in `Chapter08/06_dqn_dueling.py`. All the changes sit in the network architecture, so here I'll show only the network class (which is in the `lib/dqn_extra.py` module).

```python
class DuelingDQN(nn.Module):
    def __init__(self, input_shape, n_actions):
        super(DuelingDQN, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32,
                      kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )
```

The convolution layers are completely the same as before.

```python
        conv_out_size = self._get_conv_out(input_shape)
        self.fc_adv = nn.Sequential(
            nn.Linear(conv_out_size, 256),
            nn.ReLU(),
            nn.Linear(256, n_actions)
        )
        self.fc_val = nn.Sequential(
            nn.Linear(conv_out_size, 256),
            nn.ReLU(),
            nn.Linear(256, 1)
        )
```

Instead of defining a single path of fully connected layers, we create two different transformations: one for advantages and one for value prediction. Also, to keep the number of parameters in the model comparable to the original network, the inner dimension in both paths is decreased from 512 to 256.

```python
    def _get_conv_out(self, shape):
        o = self.conv(torch.zeros(1, *shape))
        return int(np.prod(o.size()))

    def forward(self, x):
        adv, val = self.adv_val(x)
        return val + (adv - adv.mean(dim=1, keepdim=True))
```

```
def adv_val(self, x):
    fx = x.float() / 256
    conv_out = self.conv(fx).view(fx.size()[0], -1)
    return self.fc_adv(conv_out), self.fc_val(conv_out)
```

The changes in the `forward()` function are also very simple, thanks to PyTorch's expressiveness: we calculate the value and advantage for our batch of samples and add them together, subtracting the mean of the advantage to obtain the final Q-values. A subtle, but important, difference lies in calculating the mean along the second dimension of the tensor, which produces a vector of the mean advantage for every sample in our batch.

Results

After training a dueling DQN, we can compare it to the classic DQN convergence on our Pong benchmark, as shown here.
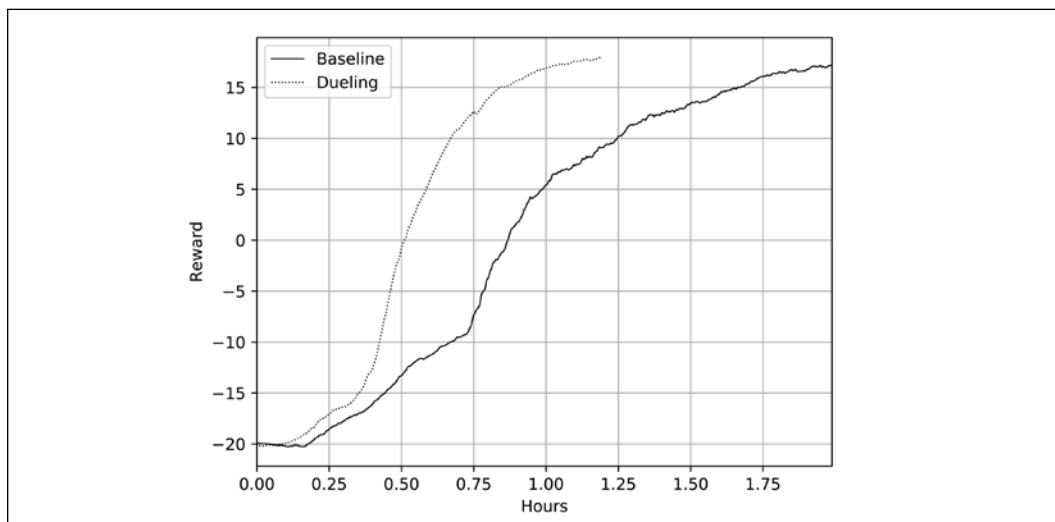
Figure 8.13: The reward dynamic of dueling DQN compared to the baseline version

Our example also outputs the advantage and value for a fixed set of states, shown in the following charts.

They meet our expectations: the advantage is not very different from zero, but the value improves over time (and resembles the value from the *Double DQN* section).
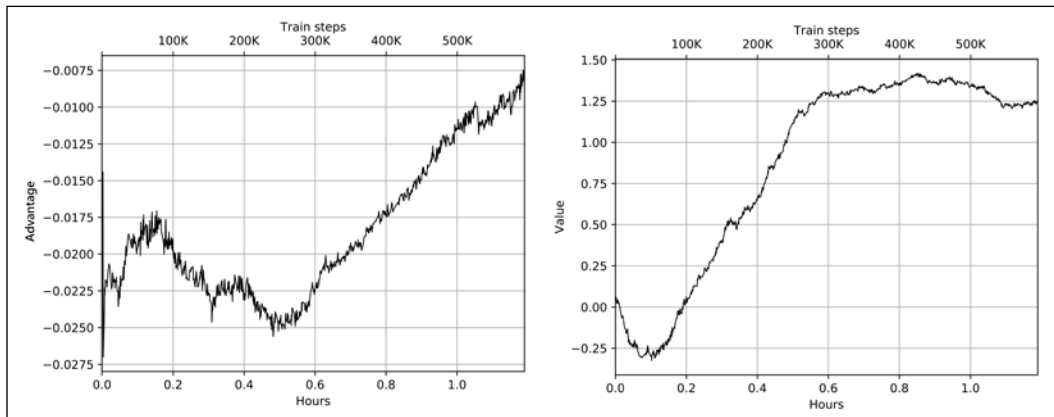
Figure 8.14: The advantage (left) and value (right) on a fixed set of states

Categorical DQN

The last, and the most complicated, method in our *DQN improvements toolbox* is from a very recent paper, published by DeepMind in June 2017, called *A Distributional Perspective on Reinforcement Learning* ([9] *Bellemare*, *Dabney*, and *Munos*, 2017).

In the paper, the authors questioned the fundamental piece of Q-learning—Q-values—and tried to replace them with a more generic Q-value probability distribution. Let's try to understand the idea. Both the Q-learning and value iteration methods work with the values of the actions or states represented as simple numbers and showing how much total reward we can achieve from a state, or an action and a state. However, is it practical to squeeze all future possible rewards into one number? In complicated environments, the future could be stochastic, giving us different values with different probabilities.

For example, imagine the commuter scenario when you regularly drive from home to work. Most of the time, the traffic isn't that heavy, and it takes you around 30 minutes to reach your destination. It's not exactly 30 minutes, but on average it's 30. From time to time, something happens, like road repairs or an accident, and due to traffic jams, it takes you three times longer to get to work. The probability of your commute time can be represented as a distribution of the "commute time" random variable and it is shown in the following chart.
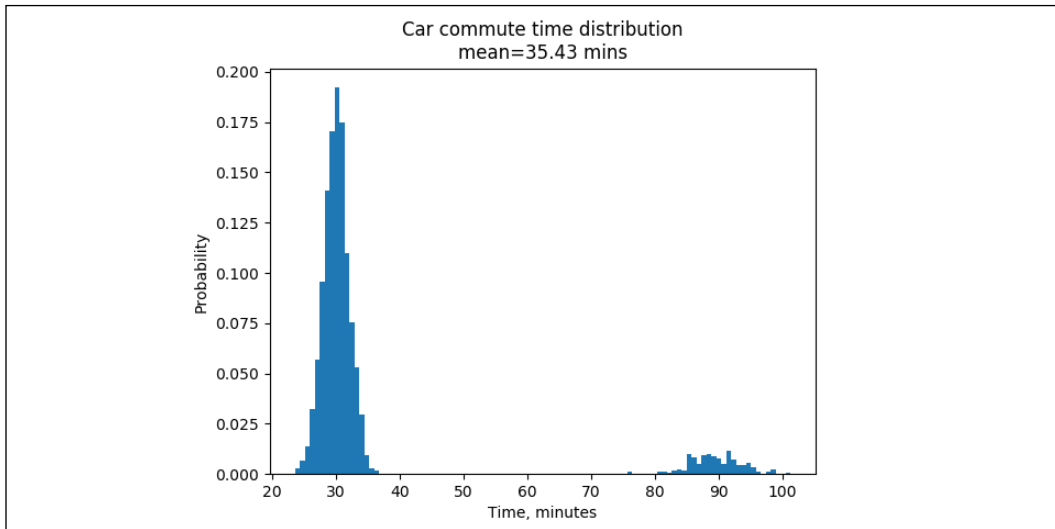
Figure 8.15: The probability distribution of commute time

Now imagine that you have an alternative way to get to work: the train. It takes a bit longer, as you need to get from home to the train station and from the station to the office, but they are much more reliable. Say, for example, that the train commute time is 40 minutes on average, with a small chance of train disruption, which adds 20 minutes of extra time to the journey. The distribution of the train commute is shown in the following graph.
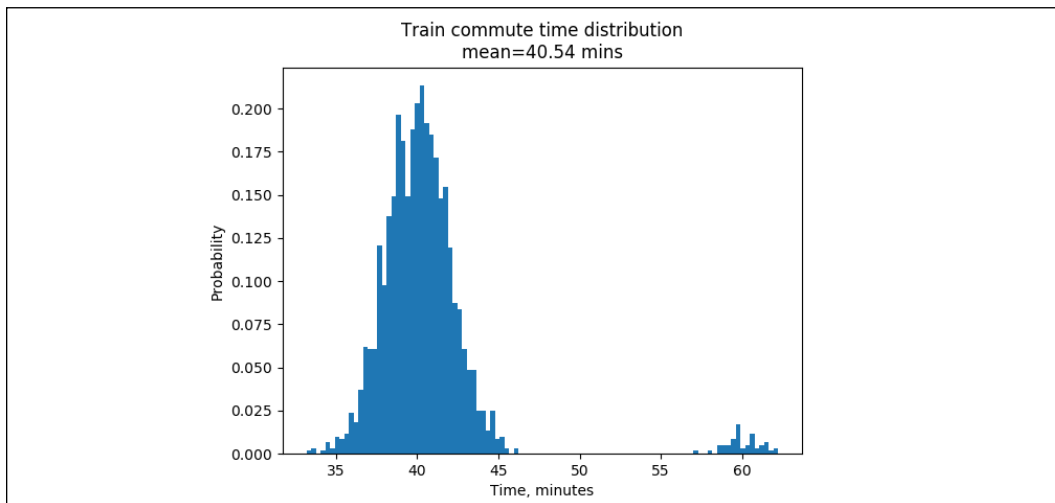
Figure 8.16: The probability distribution of train commute time

Imagine that now we want to make the decision on how to commute. If we know only the mean time for both car and train, a car looks more attractive, as on average it takes 35.43 minutes to travel, which is better than 40.54 minutes for the train.

However, if we look at full distributions, we may decide to go by train, as even in the worst-case scenario, it will be one hour of commuting versus one hour and 30 minutes. Switching to statistical language, the car distribution has much higher **variance**, so in situations when you really have to be at the office in 60 minutes max, the train is better.

The situation becomes even more complicated in the Markov decision process (MDP) scenario, when the sequence of decisions needs to be made and every decision might influence the future situation. In the commute example, it might be the time of an important meeting that you need to arrange given the way that you are going to commute. In that case, working with mean reward values might mean losing lots of information about the underlying dynamics.

Exactly the same idea was proposed by the authors of *Distributional Perspective on Reinforcement Learning* [9]. Why do we limit ourselves by trying to predict an average value for an action, when the underlying value may have a complicated underlying distribution? Maybe it will help us to work with distributions directly.

The results presented in the paper show that, in fact, this idea could be helpful, but at the cost of introducing a more complicated method. I'm not going to put a strict mathematical definition here, but the overall idea is to predict the distribution of value for every action, similar to the distributions for our car/train example. As the next step, the authors showed that the Bellman equation can be generalized for a distribution case, and it will have the form $Z(x,a) \overset{D}{=} R(x,a) + \gamma \, Z(x',a')$, which is very similar to the familiar Bellman equation, but now $Z(x, a)$ and $R(x, a)$ are the probability distributions and not numbers.

The resulting distribution can be used to train our network to give better predictions of value distribution for every action of the given state, exactly in the same way as with Q-learning. The only difference will be in the loss function, which now has to be replaced with something suitable for distribution comparison. There are several alternatives available, for example, Kullback-Leibler (KL) divergence (or cross-entropy loss), which is used in classification problems, or the Wasserstein metric. In the paper, the authors gave theoretical justification for the Wasserstein metric, but when they tried to apply it in practice, they faced limitations. So, in the end, the paper used KL divergence. The paper is very recent, so it's quite probable that improvements to the methods will follow.

Implementation

As mentioned, the method is quite complex, so it took me a while to implement it and make sure it was working. The complete code is in `Chapter08/07_dqn_distrib.py`, which uses a function in `lib/dqn_extra.py` that we haven't discussed before to perform distribution projection. Before we start it, I need to say several words about the implementation logic.

The central part of the method is the probability distribution, which we are approximating. There are lots of ways to represent the distribution, but the authors of the paper chose a quite generic *parametric distribution*, which is basically a fixed number of values placed regularly on a values range. The range of values should cover the range of possible accumulated discounted reward. In the paper, the authors did experiments with various numbers of atoms, but the best results were obtained with the range split on `N_ATOMS=51` intervals in the range of values from `Vmin=-10` to `Vmax=10`.

For every atom (we have 51 of them), our network predicts the probability that the future discounted value will fall into this atom's range. The central part of the method is the code, which performs the contraction of distribution of the next state's best action using gamma, adds local reward to the distribution, and projects the results back into our original atoms. The following is the function that does exactly this:

```
def distr_projection(next_distr, rewards, dones, gamma):
    batch_size = len(rewards)
    proj_distr = np.zeros((batch_size, N_ATOMS),
                          dtype=np.float32)
    delta_z = (Vmax - Vmin) / (N_ATOMS - 1)
```

In the beginning, we allocate the array that will keep the result of the projection. This function expects the batch of distributions with a shape `(batch_size, N_ATOMS)`, the array of rewards, flags for completed episodes, and our hyperparameters: `Vmin`, `Vmax`, `N_ATOMS`, and `gamma`. The `delta_z` variable is the width of every atom in our value range.

```
        for atom in range(N_ATOMS):
            v = rewards + (Vmin + atom * delta_z) * gamma
            tz_j = np.minimum(Vmax, np.maximum(Vmin, v))
```

In the preceding code, we iterate over every atom in the original distribution that we have and calculate the place that this atom will be projected to by the Bellman operator, taking into account our value bounds.

For example, the very first atom, with index `0`, corresponds with the value `Vmin=-10`, but for the sample with reward +1 will be projected into the value –10 * 0.99 + 1 = –8.9. In other words, it will be shifted to the right (assume `gamma=0.99`). If the value falls beyond our value range given by `Vmin` and `Vmax`, we clip it to the bounds.

```
b_j = (tz_j - Vmin) / delta_z
```

In the next line, we calculate the atom numbers that our samples have projected. Of course, samples can be projected between atoms. In such situations, we spread the value in the original distribution at the source atom between the two atoms that it falls between. This spreading should be carefully handled, as our target atom can land exactly at some atom's position. In that case, we just need to add the source distribution value to the target atom.

```
l = np.floor(b_j).astype(np.int64)
u = np.ceil(b_j).astype(np.int64)
eq_mask = u == l
proj_distr[eq_mask, l[eq_mask]] += \
    next_distr[eq_mask, atom]
```

The preceding code handles the situation when the projected atom lands exactly on the target atom. Otherwise, `b_j` won't be the integer value and variables `l` and `u` (which correspond to the indices of atoms below and above the projected point).

```
ne_mask = u != l
proj_distr[ne_mask, l[ne_mask]] += \
    next_distr[ne_mask, atom] * (u - b_j)[ne_mask]
proj_distr[ne_mask, u[ne_mask]] += \
    next_distr[ne_mask, atom] * (b_j - l)[ne_mask]
```

When the projected point lands between atoms, we need to spread the probability of the source atom between the atoms below and above. This is carried out by two lines in the preceding code, and, of course, we need to properly handle the final transitions of episodes. In that case, our projection shouldn't take into account the next distribution and should just have a 1 probability corresponding to the reward obtained. However, we again need to take into account our atoms and properly distribute this probability if the reward value falls between atoms. This case is handled by the following code branch, which zeroes the resulting distribution for samples with the `done` flag set and then calculates the resulting projection.

```
if dones.any():
    proj_distr[dones] = 0.0
    tz_j = np.minimum(
        Vmax, np.maximum(Vmin, rewards[dones]))
    b_j = (tz_j - Vmin) / delta_z
    l = np.floor(b_j).astype(np.int64)
```

```
        u = np.ceil(b_j).astype(np.int64)
        eq_mask = u == l
        eq_dones = dones.copy()
        eq_dones[dones] = eq_mask
        if eq_dones.any():
            proj_distr[eq_dones, l[eq_mask]] = 1.0
        ne_mask = u != l
        ne_dones = dones.copy()
        ne_dones[dones] = ne_mask
        if ne_dones.any():
            proj_distr[ne_dones, l[ne_mask]] = \
                (u - b_j)[ne_mask]
            proj_distr[ne_dones, u[ne_mask]] = \
                (b_j - l)[ne_mask]
    return proj_distr
```

To give you an illustration of what this function does, let's look at artificially made distributions processed by this function (*Figure 8.17*). I used them to debug the function and make sure that it worked as intended. The code for these checks is in `Chapter08/adhoc/distr_test.py`.
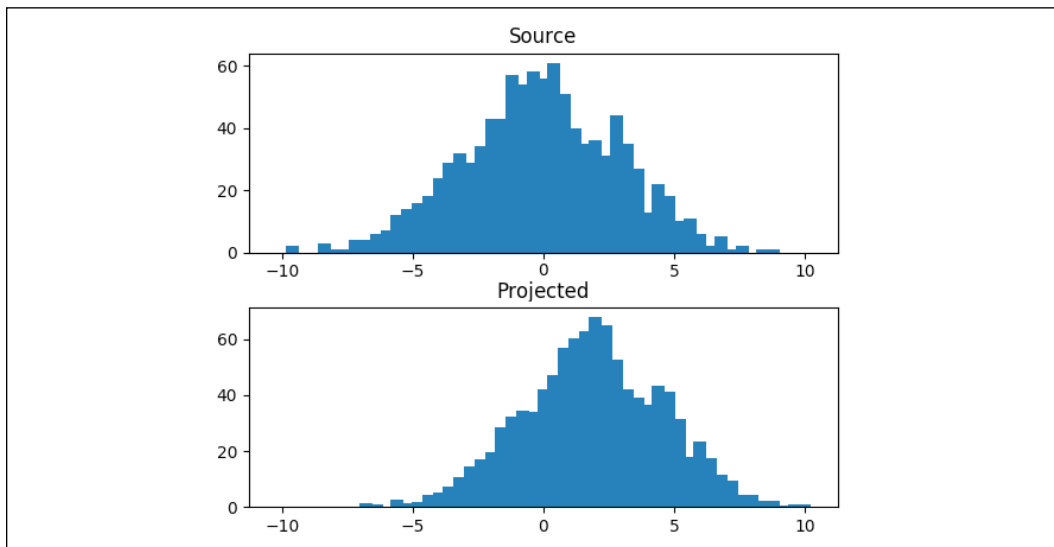
Figure 8.17: The sample of the probability distribution transformation applied to a normal distribution

The top chart of *Figure 8.17* (named "Source") is a normal distribution with zero mean and `scale=3`. The chart following that (named "Projected") is obtained from distribution projection with `gamma=0.9` and is shifted to the right with `reward=2`.

In the situation when we pass `done=True` with the same data, the result will be different and is shown on *Figure 8.18*. In such cases, the source distribution will be ignored completely, and the result will have only the reward projected.
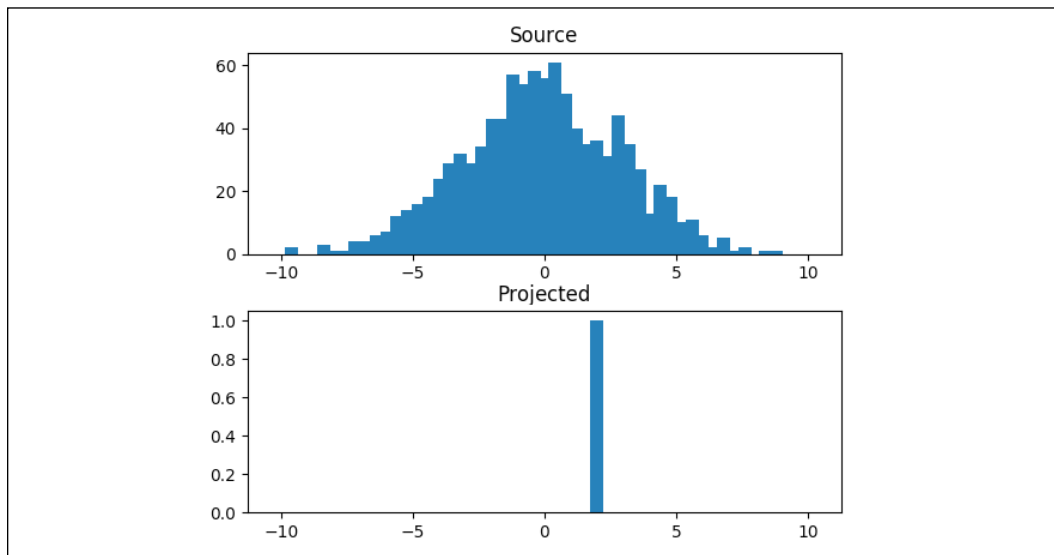
Figure 8.18: The projection of distribution for the final step in the episode

There are two versions of this method: in `Chapter08/07_dqn_distrib.py` there is only the essential code; I put the same code in source `Chapter08/07_dqn_distrib_plots.py`, but it additionally saved images of underlying probability distributions for a fixed set of states. Those images were essential during the development of the code, and I believe they could be useful if you want to understand and visualize the convergence dynamics. Sample images are shown in *Figure 8.21* and *Figure 8.22*.

I'm going to show only essential pieces of the implementation here. The core of the method, the function `distr_projection`, was already covered, and it is the most complicated piece. What is still missing is the network architecture and modified loss function.

Let's start with the network, which is in `lib/dqn_extra.py`, in the class `DistributionalDQN`.

```
Vmax = 10
Vmin = -10
N_ATOMS = 51
DELTA_Z = (Vmax - Vmin) / (N_ATOMS - 1)

class DistributionalDQN(nn.Module):
    def __init__(self, input_shape, n_actions):
```

```
        super(DistributionalDQN, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32,
                      kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )

        conv_out_size = self._get_conv_out(input_shape)
        self.fc = nn.Sequential(
            nn.Linear(conv_out_size, 512),
            nn.ReLU(),
            nn.Linear(512, n_actions * N_ATOMS)
        )

        sups = torch.arange(Vmin, Vmax + DELTA_Z, DELTA_Z)
        self.register_buffer("supports", sups)
        self.softmax = nn.Softmax(dim=1)
```

The main difference is the output of the fully connected layer. Now it outputs the vector of n_actions * N_ATOMS values, which is 6 × 51 = 306 for Pong. For every action, it needs to predict the probability distribution on 51 atoms. Every atom (called "support") has a value, which corresponds to a particular reward. Those atoms' rewards are evenly distributed from –10 to 10, which gives a grid with step 0.4. Those supports are stored in the network's buffer.

```
    def forward(self, x):
        batch_size = x.size()[0]
        fx = x.float() / 256
        conv_out = self.conv(fx).view(batch_size, -1)
        fc_out = self.fc(conv_out)
        return fc_out.view(batch_size, -1, N_ATOMS)

    def both(self, x):
        cat_out = self(x)
        probs = self.apply_softmax(cat_out)
        weights = probs * self.supports
        res = weights.sum(dim=2)
        return cat_out, res

    def qvals(self, x):
```

```
        return self.both(x)[1]

    def apply_softmax(self, t):
        return self.softmax(t.view(-1, N_ATOMS)).view(t.size())
```

The method `forward()` returns the predicted probability distribution as a 3D tensor (`batch, actions, and supports`). The network also defines several helper functions to simplify the calculation of Q-values and apply softmax on the probability distribution.

The final change is the new loss function that has to apply distribution projection instead of the Bellman equation, and calculate KL divergence between predicted and projected distributions.

```
def calc_loss(batch, net, tgt_net, gamma, device="cpu"):
    states, actions, rewards, dones, next_states = \
        common.unpack_batch(batch)
    batch_size = len(batch)

    states_v = torch.tensor(states).to(device)
    actions_v = torch.tensor(actions).to(device)
    next_states_v = torch.tensor(next_states).to(device)

    next_distr_v, next_qvals_v = tgt_net.both(next_states_v)
    next_acts = next_qvals_v.max(1)[1].data.cpu().numpy()
    next_distr = tgt_net.apply_softmax(next_distr_v)
    next_distr = next_distr.data.cpu().numpy()

    next_best_distr = next_distr[range(batch_size), next_acts]
    dones = dones.astype(np.bool)

    proj_distr = dqn_extra.distr_projection(
        next_best_distr, rewards, dones, gamma)

    distr_v = net(states_v)
    sa_vals = distr_v[range(batch_size), actions_v.data]
    state_log_sm_v = F.log_softmax(sa_vals, dim=1)
    proj_distr_v = torch.tensor(proj_distr).to(device)

    loss_v = -state_log_sm_v * proj_distr_v
    return loss_v.sum(dim=1).mean()
```

The preceding code is not very complicated; it is just preparing to call `distr_projection` and KL divergence, which is defined as $D_{KL}(P \parallel Q) = -\sum_i p_i \log q_i$.

To calculate the logarithm of probability, we use the PyTorch function `log_softmax`, which performs both `log` and `softmax` in a numerical and stable way.

Results

From my experiments, the distributional version of DQN converged a bit slower and less stably than the original DQN, which is not surprising, as the network output is now 51 times larger and the loss function has changed. So, hyperparameter tuning is required. I've tried to avoid this, as it might take another chapter to describe the process and not allow us to compare the effect of the methods in a meaningful way. The following charts show the reward dynamics and loss value over time. Obviously, the learning rate could be increased.
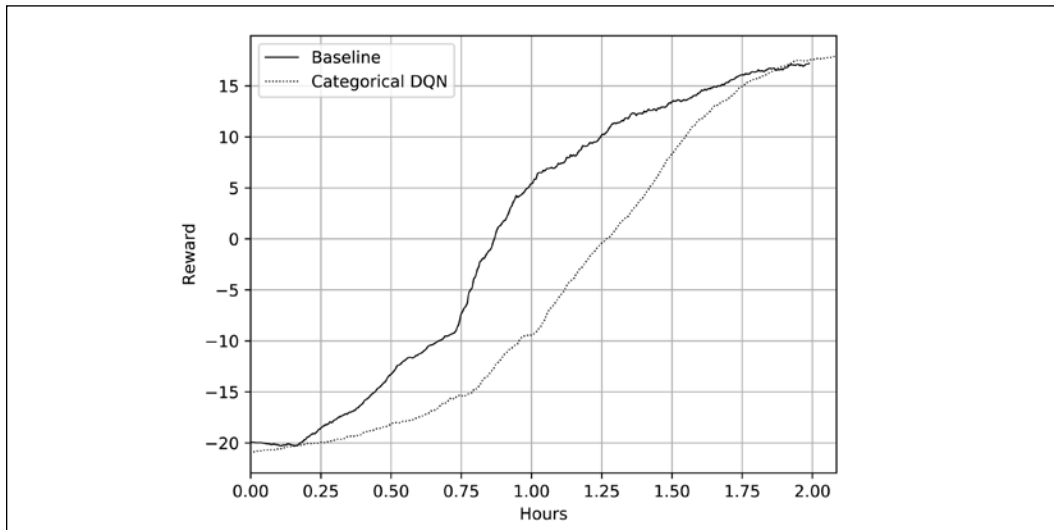
Figure 8.19: Reward dynamics
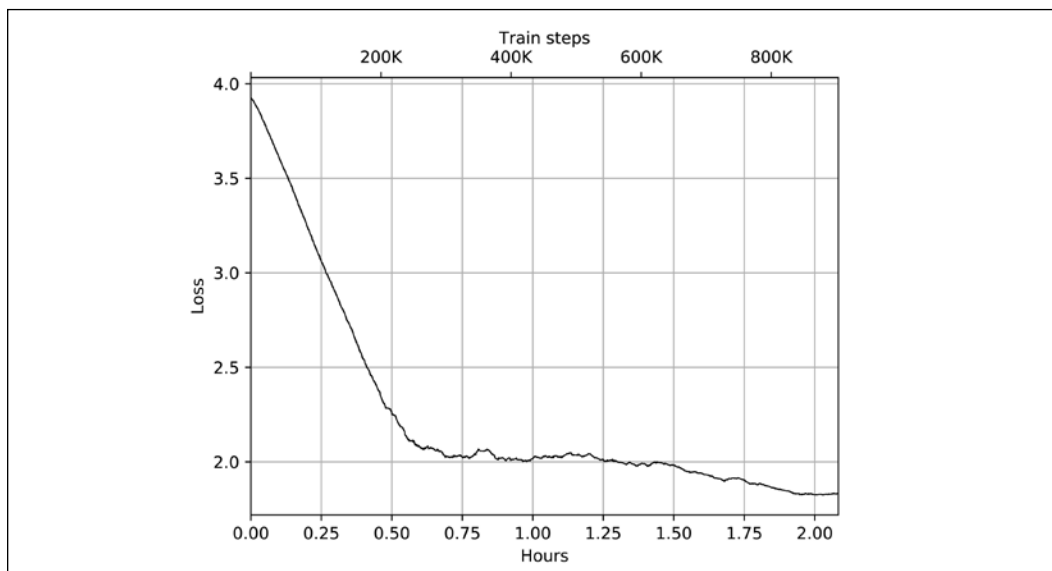
Figure 8.20: Loss decrease

As you can see, the categorical DQN is the only method that we have compared, and it behaves *worse* than the classic DQN. However, there is one factor that protects this new method: Pong is too simple a game to draw conclusions. In the 2017 *A Distributional Perspective* paper, the authors reported state-of-the-art scores for more than half of the games from the Atari benchmark (Pong was not among them).

It might be interesting to look into the dynamics of the probability distribution during the training. The extended version of the code in `Chapter08/07_dqn_distrib_plots.py` has two flags — `SAVE_STATES_IMG` and `SAVE_TRANSITIONS_IMG` — that enable the saving of probability distribution images during the training. For example, the following image shows the probability distribution for all six actions for one state at the beginning of the training (after 30k frames).
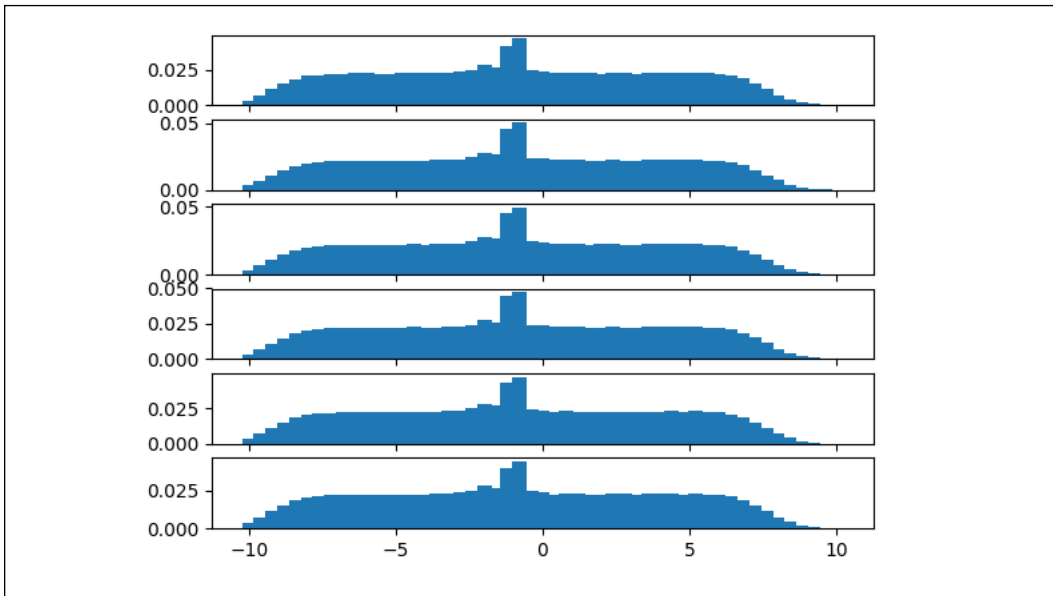
Figure 8.21: Probability distribution at the beginning of training

All the distributions are very wide (as the network hasn't converged yet), and the peak in the middle corresponds to the negative reward that the network expects to get from its actions. The same state after 500k frames of training is shown in the following figure:
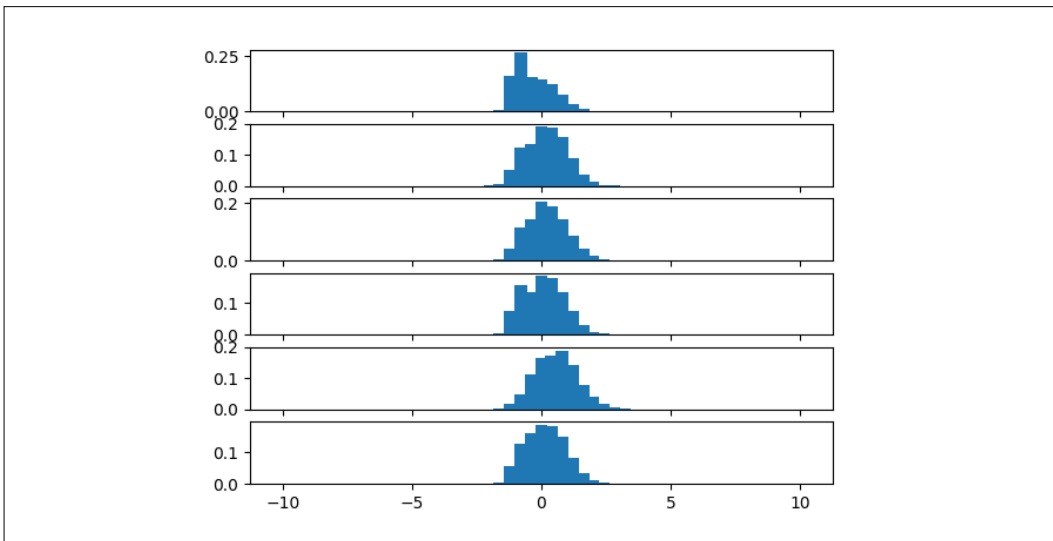
Figure 8.22: Probability distribution produced by the trained network

Now we can see that different actions have different distributions. The first action (which corresponds to the NOOP, the *do nothing* action) has its distribution shifted to the left, so doing nothing in this state usually leads to losing. The fifth action, which is RIGHTFIRE, has the mean value shifted to the right, so this action leads to a better score.

Combining everything

You have now seen all the DQN improvements mentioned in the paper *Rainbow: Combining Improvements in Deep Reinforcement Learning*, but it was done in an incremental way, which helped you to understand the idea and implementation of every improvement. The main point of the paper was to combine those improvements and check the results. In the final example, I've decided to exclude categorical DQN and double DQN from the final system, as they haven't shown too much improvement on our guinea pig environment. If you want, you can add them and try using a different game. The complete example is available in `Chapter08/08_dqn_rainbow.py`.

First of all, we need to define our network architecture and the methods that have contributed to it:

- **Dueling DQN**: our network will have two separate paths for the value of the state distribution and advantage distribution. On the output, both paths will be summed together, providing the final value probability distributions for actions. To force the advantage distribution to have a zero mean, we will subtract the distribution with the mean advantage in every atom.
- **Noisy networks**: our linear layers in the value and advantage paths will be noisy variants of `nn.Linear`.

In addition to network architecture changes, we will use the prioritized replay buffer to keep environment transitions and sample them proportionally to the MSE loss. Finally, we will unroll the Bellman equation to n-steps.

I'm not going to repeat all the code, as individual methods have already been given in the preceding sections, and it should be obvious what the final result of combining the methods will look like. If you have any trouble, you can find the code on GitHub.

Results

The following are reward charts. On the left is the combined system alone, and on the right, it is compared to our baseline code.
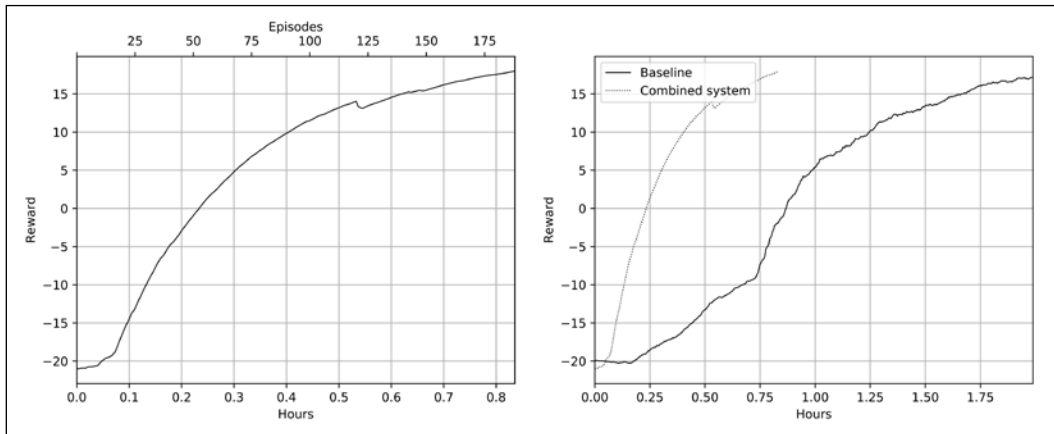
Figure 8.23: Reward dynamics of the combined system

As you can see, we got an improvement both in terms of wall clock time (50 minutes versus two hours) and the number of game episodes required (180 versus 520). The wall clock time speed-up is much more modest than sample efficiency, due to lower system performance in terms of FPS. The following chart compares the performance:
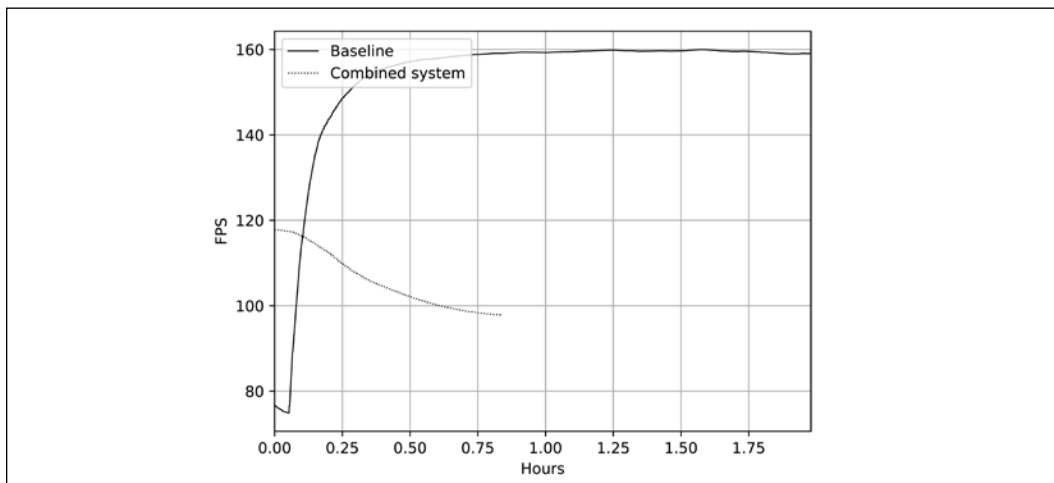

Figure 8.24: The performance comparison of the combined and baseline systems

From the chart, you can see that at the beginning of the training, the combined system showed 110 FPS, but the performance dropped to 95 due to inefficiencies in the replay buffer. The drop from 160 to 110 FPS was caused by the more complicated network architecture (noisy networks, dueling). Probably the code could be optimized in some way. Anyway, the final result is still positive.

The final chart I'd like to discuss is the number of steps in episodes (*Figure 8.25*). As you can see, the combined system discovered very quickly how to win very efficiently. The first peak to 3k steps corresponds to the initial phase of training when the system found ways to stand against the opponent for longer. The later drop in the number of steps corresponds to the "policy polishing" phase, when our agent optimized the actions to win faster (as discounted factor gamma pushed it toward shorter episodes).
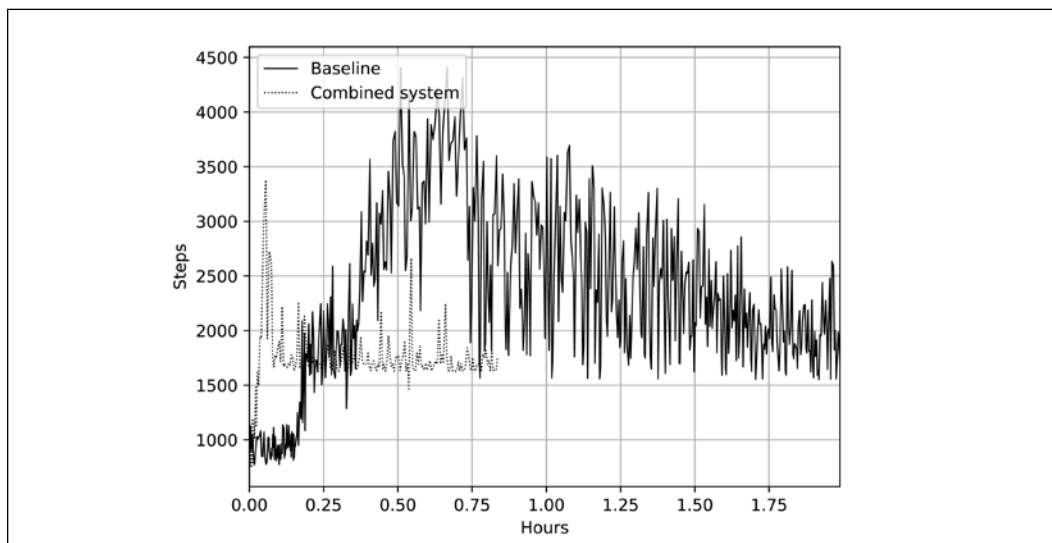
Figure 8.25: The number of steps in episodes

Of course, you are free to experiment with other combinations of improvements, or you can even combine them with the material of the next chapter, which discusses ways to speed up RL methods from the technical angle.

Summary

In this chapter, we have walked through and implemented a lot of DQN improvements that have been discovered by researchers since the first DQN paper was published in 2015. This list is far from complete. First of all, for the list of methods, I used the paper *Rainbow: Combining Improvements in Deep Reinforcement Learning*, which was published by DeepMind, so the list of methods is definitely biased to DeepMind papers. Secondly, RL is so active nowadays that new papers come out almost every day, which makes it very hard to keep up, even if we limit ourselves to one kind of RL model, such as a DQN. The goal of this chapter was to give you a practical view of different ideas that the field has developed.

In the next chapter, we will continue discussing practical DQN applications from an engineering perspective by talking about ways to improve DQN performance without touching the underlying method.

References

1. *Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, David Silver, 2017, Rainbow: Combining Improvements in Deep Reinforcement Learning. arXiv:1710.02298*

2. *Sutton, R.S., 1988, Learning to Predict by the Methods of Temporal Differences, Machine Learning 3(1):9-44*

3. *Hado Van Hasselt, Arthur Guez, David Silver, 2015, Deep Reinforcement Learning with Double Q-Learning. arXiv:1509.06461v3*

4. *Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Pilot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, Shane Legg, 2017, Noisy Networks for Exploration. arXiv:1706.10295v1*

5. *Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaus, David Saxton, Remi Munos, 2016, Unifying Count-Based Exploration and Intrinsic Motivation. arXiv:1606.01868v2*

6. *Jarryd Martin, Suraj Narayanan Sasikumar, Tom Everitt, Marcus Hutter, 2017, Count-Based Exploration in Feature Space for Reinforcement Learning. arXiv:1706.08090*

7. *Tom Schaul, John Quan, Ioannis Antonoglou, David Silver, 2015, Prioritized Experience Replay. arXiv:1511.05952*

8. *Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas, 2015, Dueling Network Architectures for Deep Reinforcement Learning. arXiv:1511.06581*

9. *Marc G. Bellemare, Will Dabney, Rémi Munos, 2017, A Distributional Perspective on Reinforcement Learning. arXiv:1707.06887*

9

Ways to Speed up RL

In *Chapter 8*, *DQN Extensions*, you saw several practical tricks to make the deep
Q-network (DQN) method more stable and converge faster. They involved the basic
DQN method modifications (like injecting noise into the network or unrolling the
Bellman equation) to get a better policy, with less time spent on training. But there is
another way: tweaking the implementation details of the method to improve the speed
of the training. This is a pure engineering approach, but it's also important in practice.

In this chapter, we will:

- Take the Pong environment from *Chapter 8* and try to get it solved as fast as
  possible
- In a step-by-step manner, get Pong solved 3.5 times faster using exactly the
  same commodity hardware
- Discuss fancier ways to speed up reinforcement learning (RL) training that
  could become common in the future

Why speed matters

First, let's talk a bit about why speed is important and why we optimize it at all. It
might not be obvious, but enormous hardware performance improvements have
happened in the last decade or two. 14 years ago, I was involved with a project
that focused on building a supercomputer for computational fluid dynamics (CFD)
simulations performed by an aircraft engine design company. The system consisted
of 64 servers, occupied three 42-inch racks, and required dedicated cooling and
power subsystems. The hardware alone (without cooling) cost almost $1M.

In 2005, this supercomputer occupied fourth place for Russian supercomputers and
was the fastest system installed in the industry. Its theoretical performance was 922
GFLOPS (billion floating-point operations per second), but in comparison to the
GTX 1080 Ti released 12 years later, all the capabilities of this pile of iron look tiny.

One single GTX 1080 Ti is able to perform 11,340 GFLOPS, which is 12.3 times more. And the price was $700 per card! If we count computation power per $1, we get a price drop of more than 17,500 times for every GFLOP.

It has been said many times that artificial intelligence (AI) progress (and machine learning (ML) in general) is being driven by data availability and computing power increases, and I believe that this is absolutely true. Imagine some computations that require a month to complete on one machine (a very common situation in CFD and other physics simulations). If we are able to increase speed by five times, this month of patient waiting will turn into six days. Speeding up by 100 times will mean that this heavy one-month computation will end up taking eight hours, so you could have three of them done in just one day! It's very cool to be able to get 20,000 times more power for the same money nowadays. (By the way, speeding up by 20k times will mean that our one-month problem will be done in two to three minutes!)

This has happened not only in the "big iron" (also known as *high-performance computing*) world; basically, it is everywhere. Modern microcontrollers have the performance characteristics of desktops that we worked with 15 years ago. (For example, you can build a pocket computer for $50, with a 32-bit microcontroller running at 120 MHz, that is able to run the Atari 2600 emulator [1]: `https://hackaday.io/project/80627-badge-for-hackaday-conference-2018-in-belgrade`.) I'm not even talking about modern smartphones, which normally have four to eight cores, a graphics processing unit (GPU), and several GBs of RAM.

> The Atari 2600 emulator for this hardware is a project that I am currently working on: `https://hackaday.io/project/166288-atari-emulator-for-hackaday-badge`. I hope it will be done when the book is published.

Of course, there are a lot of complications there. It's not just taking the same code that you used a decade ago and now, magically, finding that it works several thousand times faster. It might be the opposite: you might not be able to run it at all, due to a change in libraries, operating system interfaces, and other factors. (Have you ever tried to read old CD-RW disks written just seven years ago?) Nowadays, to get the full capabilities of modern hardware, you need to parallelize your code, which automatically means tons of details about distributed systems, data locality, communications, and the internal characteristics of the hardware and libraries. High-level libraries are trying to hide all those complications from you, but you can't ignore all of them to use them efficiently. However, it is definitely worth it—one month of patient waiting could be turned into three minutes, remember.

It might not be fully obvious why we need to speed things up in the first place. One month is not that long, after all; just lock the computer in a server room and go on vacation!

But think about the process involved in preparing and making this computation work. You might already have noticed that even simple ML problems can be almost impossible to implement properly from the first attempt.

They require many trial runs before you find good hyperparameters, and fix all the bugs and the code ready for a clean launch. There is exactly the same process in physics simulations, RL research, big data processing, and programming in general. So, if we are able to make something run faster, it's not only beneficial for the single run, but also enables us to iterate quickly and do more experiments with code, which might significantly speed up the whole process and improve the quality of the final result.

I remember one situation from my career when we deployed a Hadoop cluster in our department, where we were developing a web search engine (similar to Google, but for Russian websites). Before the deployment, it took several weeks to conduct even simple experiments with data. Several terabytes of data were lying on different servers; you needed to run your code several times on every machine, gather and combine intermediate results, deal with occasional hardware failures, and do a lot of manual tasks not related to the problem that you were supposed to solve. After integrating the Hadoop platform into the data processing, the time needed for experiments dropped to several hours, which was completely game-changing. Since then, developers have been able to conduct many experiments much more easily and faster without bothering with unnecessary details. The number of experiments (and willingness to run them) has increased significantly, which has also increased the quality of the final product.

Another reason in favor of optimization is the size of problems that we can deal with. Making some method run faster might mean two different things: we can get the results sooner, or we can increase the size (or some other measure of the problem's complexity). A complexity increase might have different meanings in different cases, like getting more accurate results, making fewer simplifications of the real world, or taking into account more data, but, almost always, this is a good thing.

Returning to the main topic of the book, let's outline how RL methods might benefit from speed-ups. First of all, even state-of-the-art RL methods are not very sample efficient, which means that training needs to communicate with the environment many times (in the case of Atari, millions of times) before learning a good policy, and that might mean weeks of training. If we can speed up this process a bit, we can get the results faster, do more experiments, and find better hyperparameters. Besides this, if we have faster code, we might increase the complexity of the problems that they are applied to.

In modern RL, Atari games are considered solved; even so-called "hard-exploration games," like Montezuma's Revenge, can be trained to superhuman accuracy.

Therefore, new frontiers in research require more complex problems, with richer observation and action spaces, which inevitably require more training time and more hardware. Such research has already been started (and has increased the complexity of problems a bit too much, from my point of view) by DeepMind and OpenAI, which have switched from Atari to much more challenging games like StarCraft II and Dota 2. Those games require hundreds of GPUs for training, but allow us to explore ways of improving existing methods and developing new methods.

I want to end this discussion with a small warning: all performance optimizations make sense only when the core method is working properly (which is not always obvious in cases of RL and ML in general). As an instructor in an online course about performance optimizations said, "It's much better to have a slow and correct program than a fast but incorrect one."

The baseline

In the rest of the chapter, we will take the Atari Pong environment that you are already familiar with and try to speed up its convergence. As a baseline, we will take the same simple DQN that we used in *Chapter 8*, *DQN Extensions*, and the hyperparameters will also be the same. To compare the effect of our changes, we will use two characteristics:

- **The number of frames** that we consume from the environment every second (FPS). It indicates how fast we can communicate with the environment during the training. It is very common in RL papers to indicate the number of frames that the agent observed during the training; normal numbers are 25M-50M frames. So, if our FPS=200, it will take 50M/200/60/60/24 ≈ 2.8 days. In such calculations, you need to take into account that RL papers commonly report *raw* environment frames. But if frame skip is used (and it almost always is), this number needs to be divided by the frame skip factor, which is commonly equal to 4. In our measurements, we calculate FPS in terms of agent communications with the environment, so the environment FPS will be four times larger.

- **The wall clock time** before the game is solved. We stop training when the smoothed reward for the last 100 episodes reaches 17. (The maximum score in Pong is 21.) This boundary could be increased, but normally 17 is a good indication that the agent has almost mastered the game and polishing the policy to perfection is just a matter of the training time. We check the wall clock time because FPS alone is not the best indicator of training speed-up.

Due to our manipulations performed with the code, we can get a very high FPS, but convergence might suffer. This value alone also can't be used as a reliable characteristic of our improvements, as the training process is stochastic. Even by specifying random seeds (we need to set seeds explicitly for PyTorch, Gym, and NumPy), parallelization (which will be used in subsequent steps) adds randomness to the process, which is almost impossible to avoid. So, the best we can do is run the benchmark several times and average the results. But one single run's outcome can't be used to make any decisions.

All the benchmarks in this chapter use the same machine with an i5-6600K central processing unit (CPU), a GTX 1080 Ti GPU with CUDA 10.0, and NVIDIA drivers version 410.79. Our first benchmark will be our baseline version, which is in `Chapter09/01_baseline.py`. I will not provide the source code here, as it has already been given in *Chapter 8* and is the same here. During the training, the code writes into TensorBoard several metrics:

- `reward`: the raw undiscounted reward from the episode; the x axis is the episode number.

- `avg_reward`: the same as reward but smoothed by running the average with alpha=0.98.

- `steps`: the number of steps that the episode lasted. Normally, in the beginning, the agent loses very quickly, so every episode is around 1,000 steps. Then, it learns how to act better, so the number of steps increases with the reward increase; but, in the end, when the agent masters the game, the number of steps drops back to 2,000 steps, as the policy is polished to win as quickly as possible (due to the discount factor gamma). In fact, this drop in episode length might be an indication of overfitting to the environment, which is a huge problem in RL. However, this experiment is beyond the scope of the book.

- `loss`: the loss during the training, sampled every 100 iterations. It should be around 2e-3…7e-3, with occasional increases when the agent discovers new behavior, leading to a different reward than that learned by the Q-value.

- `avg_loss`: a smoothed version of the loss.

- `epsilon`: the current value of epsilon.

- `avg_fps`: the speed of agent communication with environment (observations per second), smoothed with a running average.

In *Figure 9.1* and *Figure 9.2*, the charts are averaged from several baseline runs. As usual, each chart is drawn with two *x* axes: the bottom one is the wall clock time in hours, and the top is the step number (episode in *Figure 9.1* and training iteration in *Figure 9.2*).
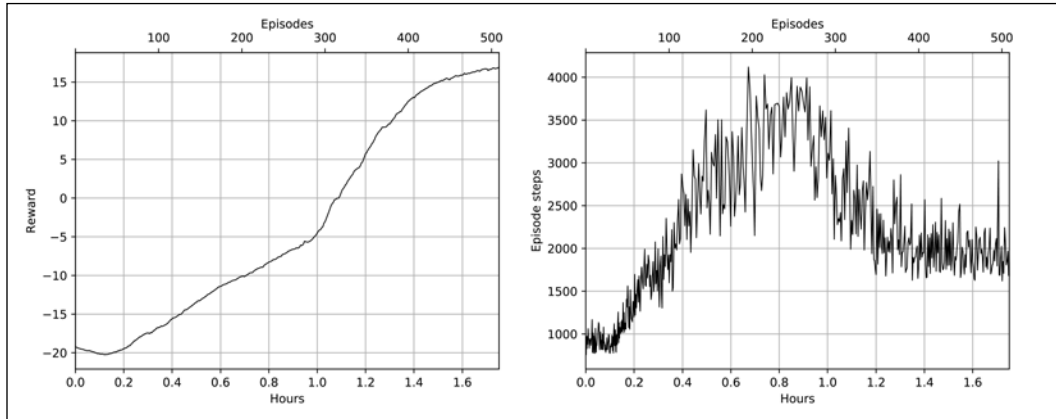
Figure 9.1: Reward and episode length
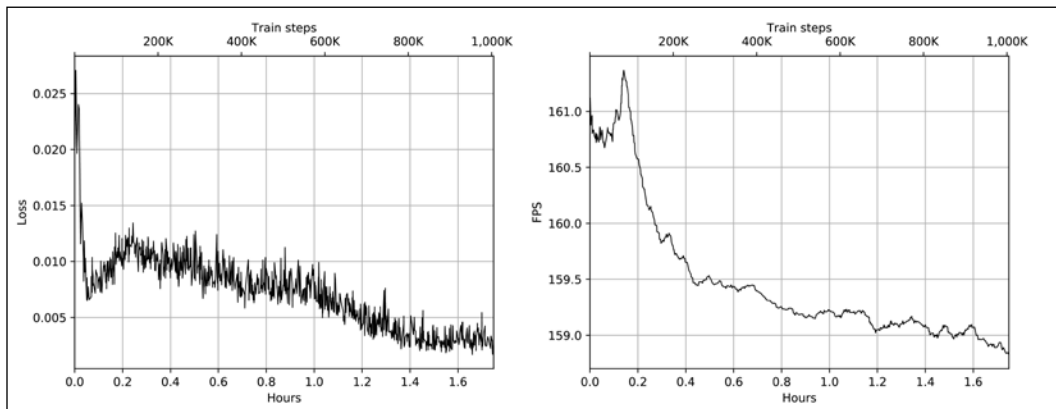
Figure 9.2: Loss and FPS dynamics during the training

The computation graph in PyTorch

Our first examples won't be around speeding up the baseline, but will show one common, and not always obvious, situation that can cost you performance. In *Chapter 3, Deep Learning with PyTorch*, we discussed the way PyTorch calculates gradients: it builds the graph of all operations that you perform on tensors, and when you call the `backward()` method of the final loss, all gradients in the model parameters are automatically calculated.

This works well, but RL code is normally much more complex than traditional supervised learning models, so the RL model that we are currently training is also being applied to get the actions that the agent needs to perform in the environment. The target network discussed in *Chapter 6* makes it even more tricky. So, in DQN, a neural network (NN) is normally used in three different situations:

1. When we want to calculate Q-values predicted by the network to get the loss in respect to reference Q-values approximated by the Bellman equation

2. When we apply the target network to get Q-values for the next state to calculate a Bellman approximation

3. When the agent wants to make a decision about the action to perform

It is very important to make sure that gradients are calculated only for the first situation. In *Chapter 6*, we avoided gradients by explicitly calling `detach()` on the tensor returned by the target network. This `detach` is very important, as it prevents gradients from flowing into our model "from the unexpected direction." (Without this, the DQN might not converge at all.) In the third situation, gradients were stopped by converting the network result into a NumPy array.

Our code in *Chapter 6, Deep Q-Networks*, worked, but we missed one subtle detail: the computation graph that is created for all three situations. This is not a major problem, but creating the graph still uses some resources (in terms of both speed and memory), which are wasted because PyTorch creates this computation graph even if we don't call `backward()` on some graph. To prevent this, one very nice thing exists: the decorator `torch.no_grad()`.

Decorators in Python is a very wide topic. They give the developer a lot of power (when properly used), but are well beyond the scope of this book. Here, I'll just give an example.

```
>>> import torch
>>> @torch.no_grad()
... def fun_a(t):
...     return t*2
>>> def fun_b(t):
...     return t*2
```

We define two functions, both doing the same thing: doubling its argument; but the first function is declared with `torch.no_grad()`, and the second is just a normal function. This decorator temporarily disables gradient computation for all tensors passed to the function.

```
>>> t = torch.ones(3, requires_grad=True)
>>> t
tensor([1., 1., 1.], requires_grad=True)
>>> a = fun_a(t)
```

```
>>> b = fun_b(t)
>>> b
tensor([2., 2., 2.], grad_fn=<MulBackward0>)
>>> a
tensor([2., 2., 2.])
```

As you can see, although the tensor, `t`, requires grad, the result from `fun_a` (the decorated function) doesn't have gradients. But this effect is bound inside the decorated function:

```
>>> a*t
tensor([2., 2., 2.], grad_fn=<MulBackward0>)
```

The function `torch.no_grad()` also could be used as a *context manager* (another powerful Python concept that I recommend you learn about), to stop gradients in some chunk of code:

```
>>> with torch.no_grad():
...        c = t*2
...
>>> c
tensor([2., 2., 2.])
```

This functionality provides you with a very convenient way to indicate parts of your code that should be excluded from the gradient machinery completely. This has already been done in `ptan.agent.DQNAgent` (and other agents provided by PTAN) and in the `common.calc_loss_dqn` function. But if you are writing a custom agent or implementing your own code, it might be very easy to forget about this.

To benchmark the effect of unnecessary graph calculation, I've provided the modified baseline code in `Chapter09/00_slow_grads.py`, which is exactly the same, but the agent and loss calculations are copied without `torch.no_grad()`. The following charts show the effect of this.
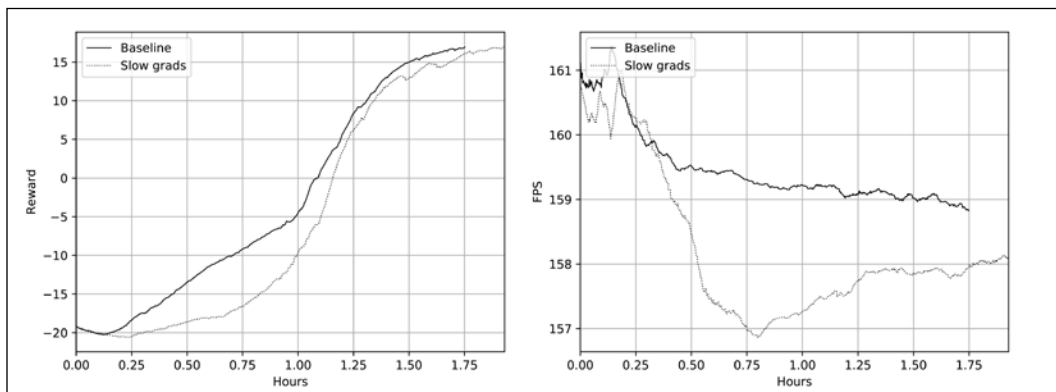
Figure 9.3: A comparison of reward and FPS between the baseline and version without torch.no_grad()

As you can see, the speed penalty is not that large (only a couple of FPS), but that might become different in the case of a larger network with a more complicated structure. I've seen a 50% performance boost in more complex recurrent NNs obtained after adding `torch.no_grad()`.

Several environments

The first idea that we usually apply to speed up deep learning training is *larger batch size*. It's applicable to the domain of deep RL, but you need to be careful here. In the normal supervised learning case, the simple rule "a large batch is better" is usually true: you just increase your batch as your GPU memory allows, and a larger batch normally means more samples will be processed in a unit of time thanks to enormous GPU parallelism.

The RL case is slightly different. During the training, two things happen simultaneously:

- Your network is trained to get better predictions on the current data
- Your agent explores the environment

As the agent explores the environment and learns about the outcome of its actions, the training data changes. In a shooter example, your agent can run randomly for a time while being shot by monsters and have only a miserable "death is everywhere" experience in the training buffer. But after a while, the agent will discover that it has a weapon it can use. This new experience can dramatically change the data that we are using for training.

RL convergence usually lies on a fragile balance between training and exploration. If we just increase a batch size without tweaking other options, we can easily overfit to the current data. (For our shooter example, your agent can start thinking that "dying young" is the only option to minimize suffering and may never discover the gun it has.)

So, in example `Chapter09/02_n_envs.py`, our agent uses several copies of the same environment to gather the training data. On every training iteration, we populate our replay buffer with samples from all those environments and then sample a proportionally larger batch size. This also allows us to speed up *inference time* a bit, as we can make a decision about the actions to execute for all *N* environments in one forward pass of the NN.

In terms of implementation, the preceding logic requires just a few changes in the code:

- As PTAN supports several environments out of the box, what we need to do is just pass *N* Gym environments to the `ExperienceSource` instance

- The agent code (in our case `DQNAgent`) is already optimized for the batched application of the NN

Here are several pieces of code that were changed:

```
def batch_generator(
        buffer: ptan.experience.ExperienceReplayBuffer,
        initial: int, batch_size: int, steps: int):
    buffer.populate(initial)
    while True:
        buffer.populate(steps)
        yield buffer.sample(batch_size)
```

The function that generates batches now performs several steps in the environment for every training iteration.

```
    envs = []
    for _ in range(args.envs):
        env = gym.make(params.env_name)
        env = ptan.common.wrappers.wrap_dqn(env)
        env.seed(common.SEED)
        envs.append(env)

    params.batch_size *= args.envs
    exp_source = ptan.experience.ExperienceSourceFirstLast(
        envs, agent, gamma=params.gamma)
```

The experience source accepts the array of environments instead of a single environment. Other changes are just minor tweaks of constants to adjust the FPS tracker and compensated speed of epsilon decay (ratio of random steps).

As the number of environments is the new hyperparameter that needs to be tuned, I ran several experiments with *N* from 2…6. The following charts show the averaged dynamics.
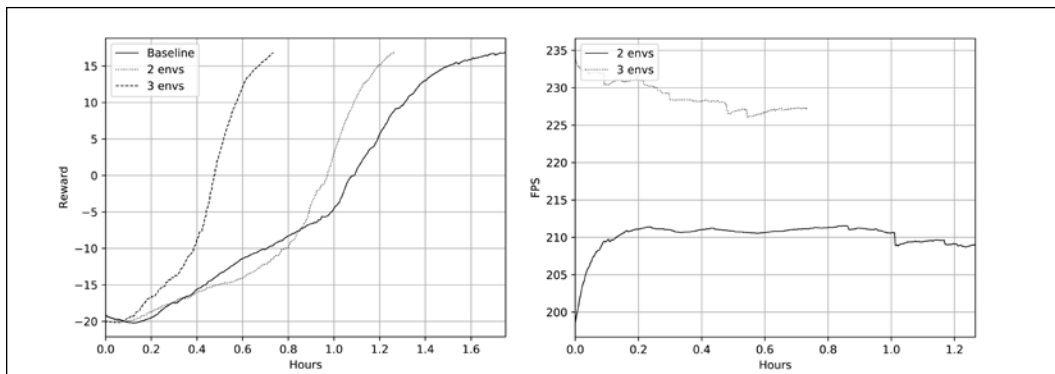
Figure 9.4: Reward dynamics and FPS in the baseline, two, and three environments
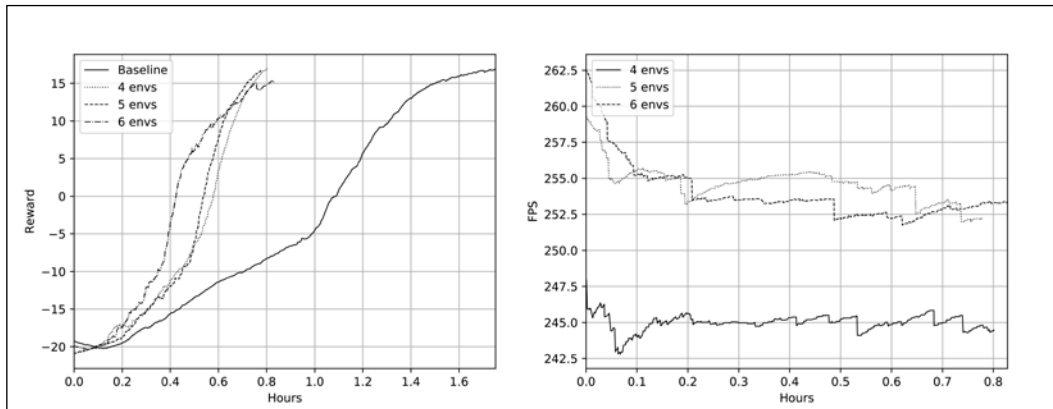
Figure 9.5: Reward and FPS in the baseline, four, five, and six environments

As you can see from the charts, adding an extra environment added 30% to FPS and sped up the convergence. The same effect came from adding the third environment (+43% to FPS), but adding more environments had a negative effect on convergence speed, despite a further increase in FPS. So, it looks like $N = 3$ is more or less the optimal value for our hyperparameter, but, of course, you are free to tweak and experiment.

Play and train in separate processes

At a high level, our training contains a repetition of the following steps:

1. Ask the current network to choose actions and execute them in our array of environments
2. Put observations into the replay buffer
3. Randomly sample the training batch from the replay buffer
4. Train on this batch

The purpose of the first two steps is to populate the replay buffer with samples from the environment (which are observation, action, reward, and next observation). The last two steps are for training our network.

The following is an illustration of the preceding steps that will make potential parallelism slightly more obvious. On the left, the training flow is shown. The training steps use environments, the replay buffer, and our NN. The solid lines show data and code flow.

Dotted lines represent usage of the NN for training and inference.
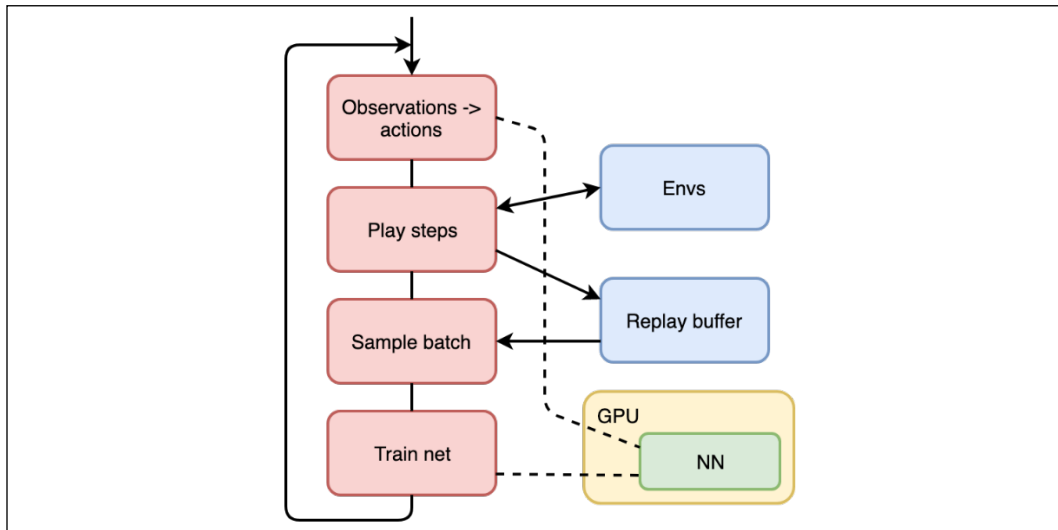
Figure 9.6: A sequential diagram of the training process

As you can see, the top two steps communicate with the bottom only via the replay buffer and NN. This makes it possible to separate those two parts as distinct processes. The following figure is a diagram of the scheme.
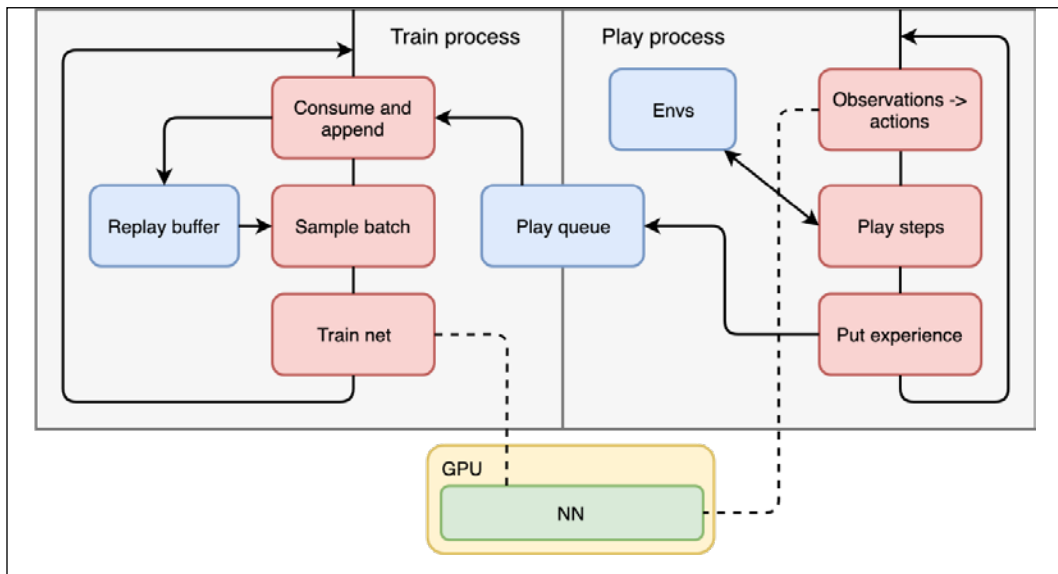
Figure 9.7: The parallel version of the training and play steps

In the case of our Pong environment, it might look like an unnecessary complication of the code, but this separation might be extremely useful in some cases. Imagine that you have a very slow and heavy environment, so every step takes seconds of computations. That's not a contrived example; for instance, recent NeurIPS competitions, such as *Learning to Run*, *AI for Prosthetics Challenge*, and *Learn to Move* [1], have very slow neuromuscular simulators, so you have to separate experience gathering from the training process. In such cases, you can have many concurrent environments that deliver the experience to the central training process.

To turn our serial code into a parallel one, some modifications are needed. In the file `Chapter09/03_parallel.py`, you can find the full source of the example. In the following, I'll focus only on major differences.

```python
import torch.multiprocessing as mp

BATCH_MUL = 4

EpisodeEnded = collections.namedtuple(
    'EpisodeEnded', field_names=('reward', 'steps', 'epsilon'))

def play_func(params, net, cuda, exp_queue):
    env = gym.make(params.env_name)
    env = ptan.common.wrappers.wrap_dqn(env)
    env.seed(common.SEED)
    device = torch.device("cuda" if cuda else "cpu")

    selector = ptan.actions.EpsilonGreedyActionSelector(
        epsilon=params.epsilon_start)
    epsilon_tracker = common.EpsilonTracker(selector, params)
    agent = ptan.agent.DQNAgent(net, selector, device=device)
    exp_source = ptan.experience.ExperienceSourceFirstLast(
        env, agent, gamma=params.gamma)

    for frame_idx, exp in enumerate(exp_source):
        epsilon_tracker.frame(frame_idx/BATCH_MUL)
        exp_queue.put(exp)
        for reward, steps in exp_source.pop_rewards_steps():
            exp_queue.put(EpisodeEnded(reward, steps,
                                       selector.epsilon))
```

First of all, we use the `torch.multiprocessing` module as a drop-in replacement for the standard Python `multiprocessing` module.

The version from the standard library provides several primitives to work with code executed in separated processes, such as `mp.Queue` (distributed queue), `mp.Process` (child process), and others. PyTorch provides a wrapper around the standard multiprocessing library, which allows torch tensors to be shared between processes without copying them. This is implemented using shared memory in the case of CPU tensors, or CUDA references for tensors on a GPU. This sharing mechanism removes the major bottleneck when communication is performed within the single computer. Of course, in the case of truly distributed communications, you need to serialize data yourself.

The function `play_func` implements our "play process" and will be running in a separate child process started by the "training process." Its responsibility is to get experience from the environment and push it into the shared queue. In addition, it wraps information about episode ends into a `namedtuple` and pushes it into the same queue to keep the training process informed about the episode reward and the number of steps.

```python
class BatchGenerator:
    def __init__(self,
                 buffer: ptan.experience.ExperienceReplayBuffer,
                 exp_queue: mp.Queue,
                 fps_handler: ptan_ignite.EpisodeFPSHandler,
                 initial: int, batch_size: int):
        self.buffer = buffer
        self.exp_queue = exp_queue
        self.fps_handler = fps_handler
        self.initial = initial
        self.batch_size = batch_size
        self._rewards_steps = []
        self.epsilon = None

    def pop_rewards_steps(self) -> List[Tuple[float, int]]:
        res = list(self._rewards_steps)
        self._rewards_steps.clear()
        return res

    def __iter__(self):
        while True:
            while exp_queue.qsize() > 0:
                exp = exp_queue.get()
                if isinstance(exp, EpisodeEnded):
                    self._rewards_steps.append((exp.reward,
                                                exp.steps))
                    self.epsilon = exp.epsilon
                else:
                    self.buffer._add(exp)
                    self.fps_handler.step()
            if len(self.buffer) < self.initial:
                continue
            yield self.buffer.sample(self.batch_size * BATCH_MUL)
```

The function `batch_generator` is replaced by the class `BatchGenerator`, which provides an iterator over batches and additionally mimics the `ExperienceSource` interface with the method `pop_reward_steps()`. The logic of this class is simple: it consumes the queue (populated by the "play process"), and if it is information about episode reward and steps, it remembers it; otherwise, the object is a piece of experience that needs to be added into the replay buffer. From the queue, we consume all objects available at the moment, and then the training batch is sampled from the buffer and yielded.

```
if __name__ == "__main__":
    warnings.simplefilter("ignore", category=UserWarning)

    mp.set_start_method('spawn')
```

In the beginning of the training process, we need to tell `torch.multiprocess` which start method to use. There are several of them, but `spawn` is the most flexible.

```
    exp_queue = mp.Queue(maxsize=BATCH_MUL*2)
    play_proc = mp.Process(target=play_func, args=(params, net,
                                                   args.cuda,
                                                   exp_queue))
    play_proc.start()
```

Then the queue for communication is created, and we start our `play_func` as a separate process. As arguments, we pass the NN, hyperparameters, and queue to be used for experience.

The rest of the code is almost the same, with the exception that we use a `BatchGenerator` instance as the data source for Ignite and for `EndOfEpisodeHandler` (which requires the method `pop_rewards_steps()`).

The following charts were obtained from my benchmarks.
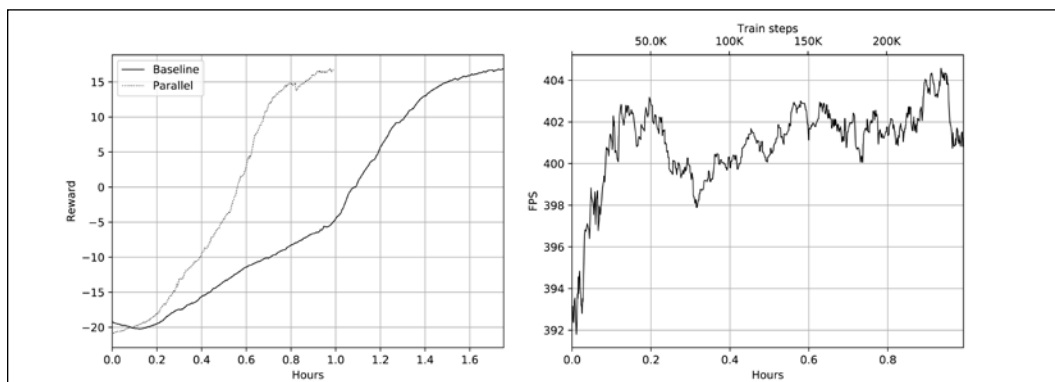
Figure 9.8: The parallel version versus the baseline (left), and the parallel version's FPS (right)

As you can see, in terms of FPS, we got a large boost of 152%: 402 FPS in the parallel version versus 159 in the baseline. But the total training time looks worse than the results obtained by the version with `N_envs=3` in the previous section: one hour of training of the parallel version versus 45 minutes. That happened due to more data being fed into the replay buffer, which caused a longer training time. Probably, some hyperparameters could be tweaked in the parallel version to improve convergence, but that's left as an exercise for the curious reader.

Tweaking wrappers

The final step in our sequence of experiments will be tweaking wrappers applied to the environment. This is very easy to overlook, as wrappers are normally written once, or just borrowed from other code, applied to the environment, and left to sit there. But you should be aware of their importance in terms of the speed and convergence of your method. For example, the normal DeepMind-style stack of wrappers applied to an Atari game looks like this:

1. `NoopResetEnv`: applies a random amount of NOOP operations to the game reset. In some Atari games, this is needed to remove weird initial observations.

2. `MaxAndSkipEnv`: applies max to *N* observations (four by default) and returns this as an observation for the step. This solves the "flickering" problem in some Atari games, when the game draws different portions of the screen on even and odd frames (a normal practice among 2600 developers to increase the complexity of the game's sprites).

3. `EpisodicLifeEnv`: in some games, this detects a lost life and turns this situation into the end of the episode. This significantly increases convergence, as our episodes become shorter (one single life versus several given by the game logic). This is relevant only for some games supported by the Atari 2600 Learning Environment.

4. `FireResetEnv`: executes a FIRE action on game reset. Some games require this to start the gameplay. Without this, our environment becomes a partially observable Markov decision process (POMDP), which makes it impossible to converge.

5. `WarpFrame` also known as `ProcessFrame84`, it converts an image to grayscale and resizes it to 84×84.

6. `ClipRewardEnv`: clips the reward to a –1…1 range, which is not the best, but a working solution to a wide variety of scores in different Atari games. For example, Pong might have a –21…21 score range, but the score in the River Raid game (my favorite) could be 0 … ∞.

7. `FrameStack`: stacks *N* sequential observations into the stack (the default is four). As we already discussed in *Chapter 6*, *Deep Q-Networks*, in some games this is required to fulfill the Markov property. For example, in Pong, from one single frame it is impossible to get the direction the ball is moving in.

The code of those wrappers was heavily optimized by many people and several versions exist. My personal favorite is the OpenAI Baselines set of wrappers, which can be found here: `https://github.com/openai/baselines/blob/master/baselines/common/atari_wrappers.py`. But you shouldn't take this code as the final source of truth, as your concrete environment might have different requirements and specifics. For example, if you are interested in speeding up one specific game from the Atari suite, `NoopResetEnv` and `MaxAndSkipEnv` (more precisely, max pooling from `MaxAndSkipEnv`) might be not needed. Another thing that could be tweaked is the number of frames in the `FrameStack` wrapper. The normal practice is to use four, but you need to understand that this number was used by DeepMind and other researchers to train on the *full* Atari 2600 game suite, which currently includes more than 50 games. For your specific case, a history of two frames might be enough to give you a performance boost, as less data will need to be processed by the NN.

Finally, the image resize could be the bottleneck of wrappers, so you might want to optimize libraries used by wrappers, for example, rebuilding them or replacing them with faster versions. In our particular case, my experiments have shown that the following tweaks could be applied to our Pong example to improve performance:

- Replace the `cv2` library with the optimized Pillow fork called `pillow-simd`: `https://github.com/uploadcare/pillow-simd`
- Disable `NoopResetEnv`
- Replace `MaxAndSkipEnv` with my wrapper, which just skips four frames without max pooling
- Keep only two frames in `FrameStack`

Let's discuss the code changes in detail. For brevity, I'll show only the relevant changes. The full code can be found in the files `Chapter09/04_new_wrappers_n_env.py`, `Chapter09/04_new_wrappers_parallel.py`, and `Chapter09/lib/atari_wrappers.py`.

First of all, the `pillow-simd` library was installed with `pip install pillow-simd`, which is a heavily optimized drop-in replacement of the Pillow library. Standard wrappers use `cv2`, but my experiments have shown that `pillow-simd` is faster.

Then, the latest version of `atari_wrappers.py` was taken from the OpenAI Baselines repository and modified for PyTorch tensor shapes. This was needed, as Baselines is implemented on TensorFlow, which has a different organization of image tensors. Specifically, TensorFlow tensors are (width, height, channel), but PyTorch uses (channel, width, height). To take this into account, minor modifications were required in the `FrameStack` and `LazyFrames` classes. In addition, a small wrapper was implemented to change the axes:

```python
class ImageToPyTorch(gym.ObservationWrapper):
    """
    Change image shape to CWH
    """
    def __init__(self, env):
        super(ImageToPyTorch, self).__init__(env)
        old_shape = self.observation_space.shape
        new_shape = (old_shape[-1], old_shape[0], old_shape[1])
        self.observation_space = gym.spaces.Box(
            low=0.0, high=1.0, shape=new_shape, dtype=np.uint8)

    def observation(self, observation):
        return np.swapaxes(observation, 2, 0)
```

Then, `WarpFrame` was modified to use the `pillow-simd` library instead of `cv2`, but it was kept simple to give the option to switch back:

```python
USE_PIL = True
if USE_PIL:
    from PIL import Image
else:
    import cv2
    cv2.ocl.setUseOpenCL(False)


# ... WarpFrame class (beginning of class is omitted)
    def observation(self, obs):
        if self._key is None:
            frame = obs
        else:
            frame = obs[self._key]
        if USE_PIL:
            frame = Image.fromarray(frame)
            if self._grayscale:
                frame = frame.convert("L")
            frame = frame.resize((self._width, self._height))
            frame = np.array(frame)
        else:
            if self._grayscale:
                frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
            frame = cv2.resize(
                frame, (self._width, self._height),
```

```
                interpolation=cv2.INTER_AREA
        )
```

A trimmed version of `MaxAndSkipEnv` was implemented, which just does a skip of *N* steps, without calculating a maxpool of observations:

```python
class SkipEnv(gym.Wrapper):
    def __init__(self, env, skip=4):
        """Return only every 'skip'-th frame"""
        gym.Wrapper.__init__(self, env)
        self._skip = skip

    def step(self, action):
        """Repeat action, sum reward, and max over last
observations."""
        total_reward = 0.0
        done = None
        for i in range(self._skip):
            obs, reward, done, info = self.env.step(action)
            total_reward += reward
            if done:
                break
        return obs, total_reward, done, info

    def reset(self, **kwargs):
        return self.env.reset(**kwargs)
```

Then, extra flags were added in the `make_atari` and `wrap_deepmind` functions to give more control over the set of wrappers applied:

```python
def make_atari(env_id, max_episode_steps=None,
               skip_noop=False, skip_maxskip=False):
    env = gym.make(env_id)
    assert 'NoFrameskip' in env.spec.id
    if not skip_noop:
        env = NoopResetEnv(env, noop_max=30)
    if not skip_maxskip:
        env = MaxAndSkipEnv(env, skip=4)
    else:
        env = SkipEnv(env, skip=4)
    if max_episode_steps is not None:
        env = TimeLimit(env, max_episode_steps=max_episode_steps)
    return env

def wrap_deepmind(env, episode_life=True, clip_rewards=True,
                  frame_stack=False, scale=False,
                  pytorch_img=False,
                  frame_stack_count=4, skip_firereset=False):
    """Configure environment for DeepMind-style Atari.
    """
```

```
    if episode_life:
        env = EpisodicLifeEnv(env)
    if 'FIRE' in env.unwrapped.get_action_meanings():
        if not skip_firereset:
            env = FireResetEnv(env)
    env = WarpFrame(env)
    if pytorch_img:
        env = ImageToPyTorch(env)
    if scale:
        env = ScaledFloatFrame(env)
    if clip_rewards:
        env = ClipRewardEnv(env)
    if frame_stack:
        env = FrameStack(env, frame_stack_count)
    return env
```

So, as you can see, nothing really fancy was done. Then, the environment creation code was changed to use the new version:

```
env = atari_wrappers.make_atari(params.env_name,
                                skip_noop=True,
                                skip_maxskip=True)
env = atari_wrappers.wrap_deepmind(env, pytorch_img=True,
                                   frame_stack=True,
                                   frame_stack_count=2)
```

That's it! Benchmarks were used for two versions: sequential with three environments and parallel from the previous section. The following charts came from benchmarking both versions.
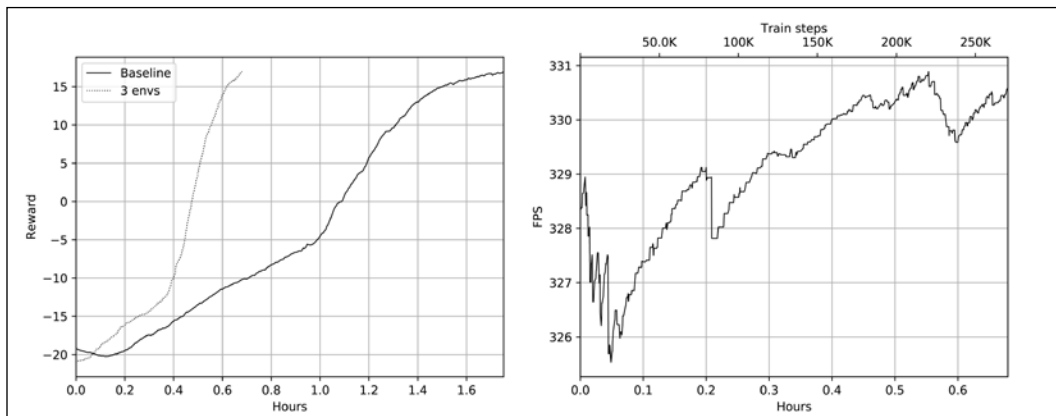
Figure 9.9: The effect of new wrappers applied to the version with three environments
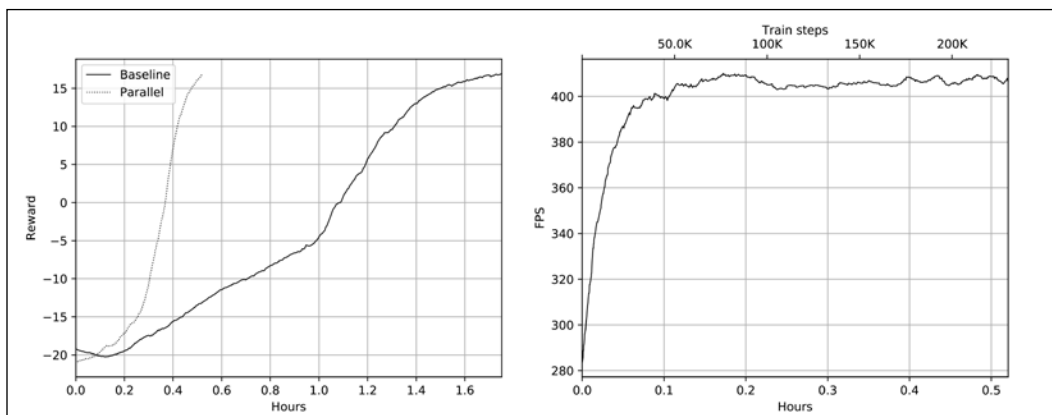
Figure 9.10: The effect of new wrappers applied to the parallel version

Benchmark summary

I summarized our experiments in the following table. The percentages show the changes versus the baseline version.

Step	FPS	Solution time in minutes
Baseline	159	105
Version without `torch.no_grad()`	157 (-1.5%)	115 (+10%)
Three environments	228 (+43%)	44 (-58%)
Parallel version	402 (+152%)	59 (-43%)
Tweaked wrappers, three environments	330 (+108%)	40 (-62%)
Tweaked wrappers, parallel version	409 (+157%)	31 (-70%)

So, by using only engineering tweaks, we were able to increase the FPS 2.5 times and get Pong solved 3.5 times faster. Not bad!

Going hardcore: CuLE

During the writing of this chapter, NVIDIA researchers published the paper and code for their latest experiments with porting the Atari emulator on GPU: *Steven Dalton, Iuri Frosio, GPU-Accelerated Atari Emulation for Reinforcement Learning*, 2019, arXiv:1907.08467. The code of their Atari port is called CuLE (CUDA Learning Environment) and is available on GitHub: `https://github.com/NVlabs/cule`.

According to their paper, by keeping both the Atari emulator and NN on the GPU, they were able to get Pong solved within one to two minutes and reach FPS of 50k (on the advantage actor-critic (A2C) method, which will be the subject of the next part of the book).

Unfortunately, at the time of writing, the code wasn't stable enough. I failed to make it work on my hardware, but I hope that when you read this, the situation will have already changed. In any case, this project shows a somewhat extreme, but very efficient, way to increase RL methods' performance. By eliminating CPU-GPU interaction, speed could be increased by a factor of hundreds.

Another example of such "hardcore" ways to speed up an RL environment and methods uses a field-programmable gate array (FPGA) to implement an environment. One such project was a Game Boy emulator implemented in Verilog (`https://github.com/krocki/gb`), which was able to reach millions of FPS. Of course, the major challenge is what to do with those frames, as FPGA is not very suitable for ML, but who knows where the progress will take us!

Summary

In this chapter, you saw several ways to improve the performance of the RL method using a pure engineering approach, which was in contrast to the "algorithmic" or "theoretical" approach covered in *Chapter 8*, *DQN Extensions*. From my perspective, both approaches complement each other, and a good RL practitioner needs to both know the latest tricks that researchers have found and be aware of the implementation details.

In the next chapter, we will apply our DQN knowledge to stocks trading.

References

1. `https://www.aicrowd.com/challenges/neurips-2019-learn-to-move-walk-around`

10
Stocks Trading Using RL

Rather than learning new methods to solve toy reinforcement learning (RL) problems in this chapter, we will try to utilize our deep Q-network (DQN) knowledge to deal with the much more practical problem of financial trading. I can't promise that the code will make you super rich on the stock market or Forex, because my goal is much less ambitious: to demonstrate how to go beyond the Atari games and apply RL to a different practical domain.

In this chapter, we will:

- Implement our own OpenAI Gym environment to simulate the stock market
- Apply the DQN method that you learned in *Chapter 6*, *Deep Q-Networks*, and *Chapter 8*, *DQN Extensions*, to train an agent to trade stocks to maximize profit

Trading

There are a lot of financial instruments traded on markets every day: goods, stocks, and currencies. Even weather forecasts can be bought or sold using so-called "weather derivatives," which is just a consequence of the complexity of the modern world and financial markets. If your income depends on future weather conditions, like a business growing crops, then you might want to hedge the risks by buying weather derivatives. All these different items have a price that changes over time. Trading is the activity of buying and selling financial instruments with different goals, like making a profit (investment), gaining protection from future price movement (hedging), or just getting what you need (like buying steel or exchanging USD for JPY to pay a contract).

Since the first financial market was established, people have been trying to predict future price movements, as this promises many benefits, like "profit from nowhere" or protecting capital from sudden market movements.

This problem is known to be complex, and there are a lot of financial consultants, investment funds, banks, and individual traders trying to predict the market and find the best moments to buy and sell to maximize profit.

The question is: can we look at the problem from the RL angle? Let's say that we have some observation of the market, and we want to make a decision: buy, sell, or wait. If we buy before the price goes up, our profit will be positive; otherwise, we will get a negative reward. What we're trying to do is get as much profit as possible. The connections between market trading and RL are quite obvious.

Data

In our example, we will use the Russian stock market prices from the period of 2015-2016, which are placed in `Chapter08/data/ch08-small-quotes.tgz` and have to be unpacked before model training.

Inside the archive, we have CSV files with M1 bars, which means that every row in each CSV file corresponds to a single minute in time, and price movement during that minute is captured with four prices: open, high, low, and close. Here, an **open** price is the price at the beginning of the minute, **high** is the maximum price during the interval, **low** is the minimum price, and the **close** price is the last price of the minute time interval. Every minute interval is called a **bar** and allows us to have an idea of price movement within the interval. For example, in the `YNDX_160101_161231.csv` file (which has Yandex company stocks for 2016), we have 130k lines in this form:

```
<DATE>,<TIME>,<OPEN>,<HIGH>,<LOW>,<CLOSE>,<VOL>
20160104,100100,1148.9,1148.9,1148.9,1148.9,0
20160104,100200,1148.9,1148.9,1148.9,1148.9,50
20160104,100300,1149.0,1149.0,1149.0,1149.0,33
20160104,100400,1149.0,1149.0,1149.0,1149.0,4
20160104,100500,1153.0,1153.0,1153.0,1153.0,0
20160104,100600,1156.9,1157.9,1153.0,1153.0,43
20160104,100700,1150.6,1150.6,1150.4,1150.4,5
20160104,100800,1150.2,1150.2,1150.2,1150.2,4
...
```

The first two columns are the date and time for the minute; the next four columns are open, high, low, and close prices; and the last value represents the number of buy and sell orders performed during the bar. The exact interpretation of this number is stock- and market-dependent, but usually, volumes give you an idea about how active the market was.

The typical way to represent those prices is called a **candlestick chart**, where every bar is shown as a candle. Part of Yandex's quotes for one day in February 2016 is shown in the following chart. The archive contains two files with M1 data for 2016 and 2015. We will use data from 2016 for model training and data from 2015 for validation.

Figure 10.1: Price data for Yandex in February 2016

Problem statements and key decisions

The finance domain is large and complex, so you can easily spend several years learning something new every day. In our example, we will just scratch the surface a bit with our RL tools, and our problem will be formulated as simply as possible, using price as an observation. We will investigate whether it will be possible for our agent to learn when the best time is to buy one single share and then close the position to maximize the profit. The purpose of this example is to show how flexible the RL model can be and what the first steps are that you usually need to take to apply RL to a real-life use case.

As you already know, to formulate RL problems, three things are needed: observation of the environment, possible actions, and a reward system. In previous chapters, all three were already given to us, and the internal machinery of the environment was hidden. Now we're in a different situation, so we need to decide ourselves what our agent will see and what set of actions it can take. The reward system is also not given as a strict set of rules; rather, it will be guided by our feelings and knowledge of the domain, which gives us lots of flexibility.

Flexibility, in this case, is good and bad at the same time. It's good that we have the freedom to pass some information to the agent that we feel will be important to learn efficiently. For example, you can pass to the trading agent not only prices, but also news or important statistics (which are known to influence financial markets a lot). The bad part is that this flexibility usually means that to find a good agent, you need to try a lot of variants of data representation, and it's not always obvious which will work better. In our case, we will implement the basic trading agent in its simplest form. The observation will include the following information:

- N past bars, where each has open, high, low, and close prices
- An indication that the share was bought some time ago (only one share at a time will be possible)
- Profit or loss that we currently have from our current position (the share bought)

At every step, after every minute's bar, the agent can take one of the following actions:

- **Do nothing**: skip the bar without taking an action
- **Buy a share**: if the agent has already got the share, nothing will be bought; otherwise, we will pay the commission, which is usually some small percentage of the current price
- **Close the position**: if we do not have a previously purchased share, nothing will happen; otherwise, we will pay the commission for the trade

The reward that the agent receives can be expressed in various ways. On the one hand, we can split the reward into multiple steps during our ownership of the share. In that case, the reward on every step will be equal to the last bar's movement. On the other hand, the agent will receive the reward only after the *close* action and receive the full reward at once. At first sight, both variants should have the same final result, but maybe with different convergence speeds. However, in practice, the difference could be dramatic. We will implement both variants to compare them.

One last decision to make is how to represent the prices in our environment observation. Ideally, we would like our agent to be independent of actual price values and take into account relative movement, such as "the stock has grown 1% during the last bar" or "the stock has lost 5%." This makes sense, as different stocks' prices can vary, but they can have similar movement patterns. In finance, there is a branch of analytics called *technical analysis* that studies such patterns to help to make predictions from them. We would like our system to be able to discover the patterns (if they exist). To achieve this, we will convert every bar's open, high, low, and close prices to three numbers showing high, low, and close prices represented as a percentage of the open price.

This representation has its own drawbacks, as we're potentially losing the information about key price levels. For example, it's known that markets have a tendency to bounce from round price numbers (like $8,000 per bitcoin) and levels that were turning points in the past. However, as already stated, we're just playing with the data here and checking the concept. Representation in the form of relative price movement will help the system to find repeating patterns in the price level (if they exist, of course), regardless of the absolute price position. Potentially, the NN could learn this on its own (it's just the mean price that needs to be subtracted from the absolute price values), but relative representation simplifies the NN's task.

The trading environment

As we have a lot of code (methods, utility classes in PTAN, and so on) that is supposed to work with OpenAI Gym, we will implement the trading functionality following Gym's `Env` class API, which should be familiar to you. Our environment is implemented in the `StocksEnv` class in the `Chapter10/lib/environ.py` module. It uses several internal classes to keep its state and encode observations. Let's first look at the public API class:

```
import gym
import gym.spaces
from gym.utils import seeding
from gym.envs.registration import EnvSpec
import enum
import numpy as np

from . import data

class Actions(enum.Enum):
    Skip = 0
    Buy = 1
    Close = 2
```

We encode all available actions as an enumerator's fields. We support a very simple set of actions with only three options: do nothing, buy a single share, and close the existing position.

```
class StocksEnv(gym.Env):
    metadata = {'render.modes': ['human']}
    spec = EnvSpec("StocksEnv-v0")
```

The fields `metadata` and `spec` are required for `gym.Env` compatibility. We don't provide render functionality, so you can ignore this.

```
@classmethod
def from_dir(cls, data_dir, **kwargs):
    prices = {
        file: data.load_relative(file)
        for file in data.price_files(data_dir)
    }
    return StocksEnv(prices, **kwargs)
```

Our environment class provides two ways to create its instance. The first way is to call the class method `from_dir` with the data directory as the argument. In that case, it will load all quotes from CSV files in the directory and construct the environment. To deal with price data in our form, we have several helper functions in `Chapter10/lib/data.py`. Another way is to construct the class instance directly. In that case, you should pass the `prices` dictionary, which has to map the quote tag to the `Prices` tuple declared in `data.py`. This object has five fields containing `open`, `high`, `low`, `close`, and `volume` time series in the NumPy array format. You can construct such objects using `data.py` library functions, like `data.load_relative()`.

```
def __init__(self, prices, bars_count=DEFAULT_BARS_COUNT,
             commission=DEFAULT_COMMISSION_PERC,
             reset_on_close=True, conv_1d=False,
             random_ofs_on_reset=True, reward_on_close=False,
             volumes=False):
```

The constructor of the environment accepts a lot of arguments to tweak the environment's behavior and observation representation:

- `prices`: Contains one or more stock prices for one or more instruments as a `dict`, where keys are the instrument's name and the value is a container object `data.Prices`, which holds price data arrays.

- `bars_count`: The count of bars that we pass in the observation. By default, this is 10 bars.

- `commission`: The percentage of the stock price that we have to pay to the broker on buying and selling the stock. By default, it's 0.1%.

- `reset_on_close`: If this parameter is set to `True`, which it is by default, every time the agent asks us to close the existing position (in other words, sell a share), we stop the episode. Otherwise, the episode will continue until the end of our time series, which is one year of data.

- `conv_1d`: This Boolean argument switches between different representations of price data in the observation passed to the agent. If it is set to `True`, observations have a 2D shape, with different price components for subsequent bars organized in rows. For example, high prices (max price for the bar) are placed on the first row, low prices on the second, and close prices on the third. This representation is suitable for doing 1D convolution on time series, where every row in the data has the same meaning as different color planes (red, green, or blue) in Atari 2D images. If we set this option to `False`, we have one single array of data with every bar's components placed together. This organization is convenient for a fully connected network architecture. Both representations are illustrated in *Figure 10.2*.

- `random_ofs_on_reset`: If the parameter is `True` (by default), on every reset of the environment, the random offset in the time series will be chosen. Otherwise, we will start from the beginning of the data.

- `reward_on_close`: This Boolean parameter switches between the two reward schemes discussed previously. If it is set to `True`, the agent will receive a reward only on the "close" action issue. Otherwise, we will give a small reward every bar, corresponding to price movement during that bar.

- `volumes`: This argument switches on volumes in observations and is disabled by default.
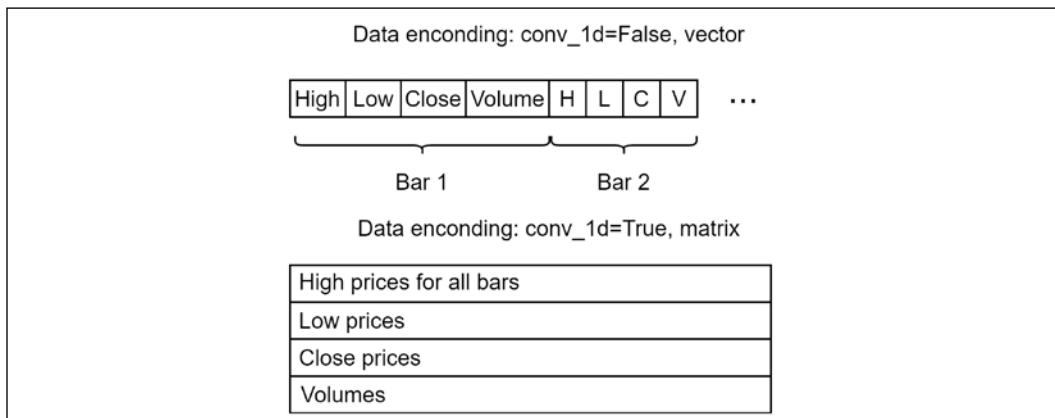
Figure 10.2: Different data representations for the NN

Now we will continue looking at the environment constructor:

```
assert isinstance(prices, dict)
self._prices = prices
if conv_1d:
    self._state = State1D(
```

```
            bars_count, commission, reset_on_close,
            reward_on_close=reward_on_close, volumes=volumes)
    else:
        self._state = State(
            bars_count, commission, reset_on_close,
            reward_on_close=reward_on_close, volumes=volumes)
    self.action_space = gym.spaces.Discrete(n=len(Actions))
    self.observation_space = gym.spaces.Box(
        low=-np.inf, high=np.inf,
        shape=self._state.shape, dtype=np.float32)
    self.random_ofs_on_reset = random_ofs_on_reset
    self.seed()
```

Most of the functionality of the `StocksEnv` class is implemented in two internal classes: `State` and `State1D`. They are responsible for observation preparation and our bought share state and reward. They implement a different representation of our data in the observations, and we will take a look at their code later. In the constructor, we create the state object, action space, and observation space fields that are required by Gym.

```
def reset(self):
    self._instrument = self.np_random.choice(
        list(self._prices.keys()))
    prices = self._prices[self._instrument]
    bars = self._state.bars_count
    if self.random_ofs_on_reset:
        offset = self.np_random.choice(
            prices.high.shape[0]-bars*10) + bars
    else:
        offset = bars
    self._state.reset(prices, offset)
    return self._state.encode()
```

This method defines the `reset()` functionality for our environment. According to the `gym.Env` semantics, we randomly switch the time series that we will work on and select the starting offset in this time series. The selected price and offset are passed to our internal state instance, which then asks for an initial observation using its `encode()` function.

```
def step(self, action_idx):
    action = Actions(action_idx)
    reward, done = self._state.step(action)
    obs = self._state.encode()
    info = {
        "instrument": self._instrument,
```

```
            "offset": self._state._offset
        }
        return obs, reward, done, info
```

This method has to handle the action chosen by the agent and return the next observation, `reward`, and `done` flag. All real functionality is implemented in our state classes, so this method is a very simple wrapper around the call to state methods.

```
def render(self, mode='human', close=False):
    pass

def close(self):
    pass
```

The API for `gym.Env` allows you to define the `render()` method handler, which is supposed to render the current state in human or machine-readable format. Generally, this method is used to peek inside the environment state and is useful for debugging or tracing the agent's behavior. For example, the market environment could render current prices as a chart to visualize what the agent sees at that moment. Our environment doesn't support rendering, so this method does nothing. Another method is `close()`, which gets called on the environment's destruction to free the allocated resources.

```
def seed(self, seed=None):
    self.np_random, seed1 = seeding.np_random(seed)
    seed2 = seeding.hash_seed(seed1 + 1) % 2 ** 31
    return [seed1, seed2]
```

This method is part of Gym's magic related to Python random number generator problems. For example, when you create several environments at the same time, their random number generators could be initialized with the same seed (which is the current timestamp, by default). It's not very relevant for our code (as we're using only one environment instance for DQN), but it will become useful in the next part of the book, when we look at the asynchronous advantage actor-critic (A3C) method, which is supposed to use several environments concurrently.

Let's now look at the internal `environ.State` class, which implements most of the environment's functionality:

```
class State:
    def __init__(self, bars_count, commission_perc,
                 reset_on_close, reward_on_close=True,
                 volumes=True):
        assert isinstance(bars_count, int)
        assert bars_count > 0
```

```
                assert isinstance(commission_perc, float)
                assert commission_perc >= 0.0
                assert isinstance(reset_on_close, bool)
                assert isinstance(reward_on_close, bool)
                self.bars_count = bars_count
                self.commission_perc = commission_perc
                self.reset_on_close = reset_on_close
                self.reward_on_close = reward_on_close
                self.volumes = volumes
```

The constructor does nothing more than just check and remember the arguments in the object's fields.

```
        def reset(self, prices, offset):
                assert isinstance(prices, data.Prices)
                assert offset >= self.bars_count-1
                self.have_position = False
                self.open_price = 0.0
                self._prices = prices
                self._offset = offset
```

The `reset()` method is called every time that the environment is asked to reset and has to save the passed prices data and starting offset. In the beginning, we don't have any shares bought, so our state has have_position=False and open_price=0.0.

```
        @property
        def shape(self):
                # [h, l, c] * bars + position_flag + rel_profit
                if self.volumes:
                        return 4 * self.bars_count + 1 + 1,
                else:
                        return 3 * self.bars_count + 1 + 1,
```

This property returns the shape of the state representation in a NumPy array.

The `State` class is encoded into a single vector, which includes prices with optional volumes and two numbers indicating the presence of a bought share and position profit.

```
        def encode(self):
                res = np.ndarray(shape=self.shape, dtype=np.float32)
                shift = 0
                for bar_idx in range(-self.bars_count+1, 1):
                        ofs = self._offset + bar_idx
                        res[shift] = self._prices.high[ofs]
```

```
            shift += 1
            res[shift] = self._prices.low[ofs]
            shift += 1
            res[shift] = self._prices.close[ofs]
            shift += 1
            if self.volumes:
                res[shift] = self._prices.volume[ofs]
                shift += 1
        res[shift] = float(self.have_position)
        shift += 1
        if not self.have_position:
            res[shift] = 0.0
        else:
            res[shift] = self._cur_close() / self.open_price - 1.0
        return res
```

The preceding method encodes prices at the current offset into a NumPy array, which will be the observation of the agent.

```
    def _cur_close(self):
        open = self._prices.open[self._offset]
        rel_close = self._prices.close[self._offset]
        return open * (1.0 + rel_close)
```

This helper method calculates the current bar's close price. Prices passed to the `State` class have the relative form in respect of the open price: the high, low, and close components are relative ratios to the open price. This representation was already discussed when we talked about the training data, and it will (probably) help our agent to learn price patterns that are independent of actual price value.

```
    def step(self, action):
        assert isinstance(action, Actions)
        reward = 0.0
        done = False
        close = self._cur_close()
```

This method is the most complicated piece of code in the `State` class, which is responsible for performing one step in our environment. On exit, it has to return the reward in a percentage and an indication of the episode ending.

```
        if action == Actions.Buy and not self.have_position:
            self.have_position = True
            self.open_price = close
            reward -= self.commission_perc
```

If the agent has decided to buy a share, we change our state and pay the commission. In our state, we assume the instant order execution at the current bar's close price, which is a simplification on our side; normally, an order can be executed on a different price, which is called *price slippage*.

```
elif action == Actions.Close and self.have_position:
    reward -= self.commission_perc
    done |= self.reset_on_close
    if self.reward_on_close:
        reward += 100.0 * (close / self.open_price - 1.0)
    self.have_position = False
    self.open_price = 0.0
```

If we have a position and the agent asks us to close it, we pay commission again, change the `done` flag if we're in `reset_on_close` mode, give a final reward for the whole position, and change our state.

```
self._offset += 1
prev_close = close
close = self._cur_close()
done |= self._offset >= self._prices.close.shape[0]-1
if self.have_position and not self.reward_on_close:
    reward += 100.0 * (close / prev_close - 1.0)
return reward, done
```

In the rest of the function, we modify the current offset and give the reward for the last bar movement. That's it for the `State` class, so let's look at `State1D`, which has the same behavior and just overrides the representation of the state passed to the agent:

```
class State1D(State):
    @property
    def shape(self):
        if self.volumes:
            return (6, self.bars_count)
        else:
            return (5, self.bars_count)
```

The shape of this representation is different, as our prices are encoded as a 2D matrix suitable for a 1D convolution operator.

```
def encode(self):
    res = np.zeros(shape=self.shape, dtype=np.float32)
    start = self._offset-(self.bars_count-1)
    stop = self._offset+1
    res[0] = self._prices.high[start:stop]
```

```
        res[1] = self._prices.low[start:stop]
        res[2] = self._prices.close[start:stop]
        if self.volumes:
            res[3] = self._prices.volume[start:stop]
            dst = 4
        else:
            dst = 3
        if self.have_position:
            res[dst] = 1.0
            res[dst+1] = self._cur_close() / self.open_price - 1.0
        return res
```

This method encodes the prices in our matrix, depending on the current offset, whether we need volumes, and whether we have stock. That's it for our trading environment. Compatibility with the Gym API allows us to plug it into the familiar classes that we used to handle the Atari games. Let's do that now.

Models

In this example, two architectures of DQN are used: a simple feed-forward network with three layers and a network with 1D convolution as a feature extractor, followed by two fully connected layers to output Q-values. Both of them use the dueling architecture described in *Chapter 8, DQN Extensions*. Double DQN and two-step Bellman unrolling have also been used. The rest of the process is the same as in a classical DQN (from *Chapter 6, Deep Q-Networks*).

Both models are in `Chapter10/lib/models.py` and are very simple.

```
class SimpleFFDQN(nn.Module):
    def __init__(self, obs_len, actions_n):
        super(SimpleFFDQN, self).__init__()

        self.fc_val = nn.Sequential(
            nn.Linear(obs_len, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 1)
        )

        self.fc_adv = nn.Sequential(
            nn.Linear(obs_len, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
```

```
            nn.ReLU(),
            nn.Linear(512, actions_n)
        )

    def forward(self, x):
        val = self.fc_val(x)
        adv = self.fc_adv(x)
        return val + (adv - adv.mean(dim=1, keepdim=True))
```

The convolutional model has a common feature extraction layer with the 1D convolution operations and two fully connected heads to output the value of the state and advantages for actions.

```
class DQNConv1D(nn.Module):
    def __init__(self, shape, actions_n):
        super(DQNConv1D, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv1d(shape[0], 128, 5),
            nn.ReLU(),
            nn.Conv1d(128, 128, 5),
            nn.ReLU(),
        )
        out_size = self._get_conv_out(shape)

        self.fc_val = nn.Sequential(
            nn.Linear(out_size, 512),
            nn.ReLU(),
            nn.Linear(512, 1)
        )

        self.fc_adv = nn.Sequential(
            nn.Linear(out_size, 512),
            nn.ReLU(),
            nn.Linear(512, actions_n)
        )

    def _get_conv_out(self, shape):
        o = self.conv(torch.zeros(1, *shape))
        return int(np.prod(o.size()))

    def forward(self, x):
        conv_out = self.conv(x).view(x.size()[0], -1)
        val = self.fc_val(conv_out)
        adv = self.fc_adv(conv_out)
        return val + (adv - adv.mean(dim=1, keepdim=True))
```

Training code

We have two very similar training modules in this example: one for the feed-forward model and one for 1D convolutions. For both of them, there is nothing new added to our examples from *Chapter 8, DQN Extensions*:

- They're using epsilon-greedy action selection to perform exploration. The epsilon linearly decays over the first 1M steps from 1.0 to 0.1.

- A simple experience replay buffer of size 100k is being used, which is initially populated with 10k transitions.

- For every 1,000 steps, we calculate the mean value for the fixed set of states to check the dynamics of the Q-values during the training.

- For every 100k steps, we perform validation: 100 episodes are played on the training data and on previously unseen quotes. Characteristics of orders are recorded in TensorBoard, such as the mean profit, the mean count of bars, and the share held. This step allows us to check for overfitting conditions.

The training modules are in `Chapter10/train_model.py` (feed-forward model) and `Chapter10/train_model_conv.py` (with a 1D convolutional layer). Both versions accept the same command-line options.

To start the training, you need to pass training data with the `--data` option, which could be an individual CSV file or the whole directory with files. By default, the training module uses Yandex quotes for 2016 (file `data/YNDX_160101_161231.csv`). For the validation data, there is an option, `--val`, that takes Yandex 2015 quotes by default. Another required option will be `-r`, which is used to pass the name of the run. This name will be used in the TensorBoard run name and to create directories with saved models.

Results

Let's now take a look at the results.

The feed-forward model

The convergence on Yandex data for one year requires about 10M training steps, which can take a while. (GTX 1080 Ti trains at a speed of 230-250 steps per second.)

During the training, we have several charts in TensorBoard showing us what's going on.
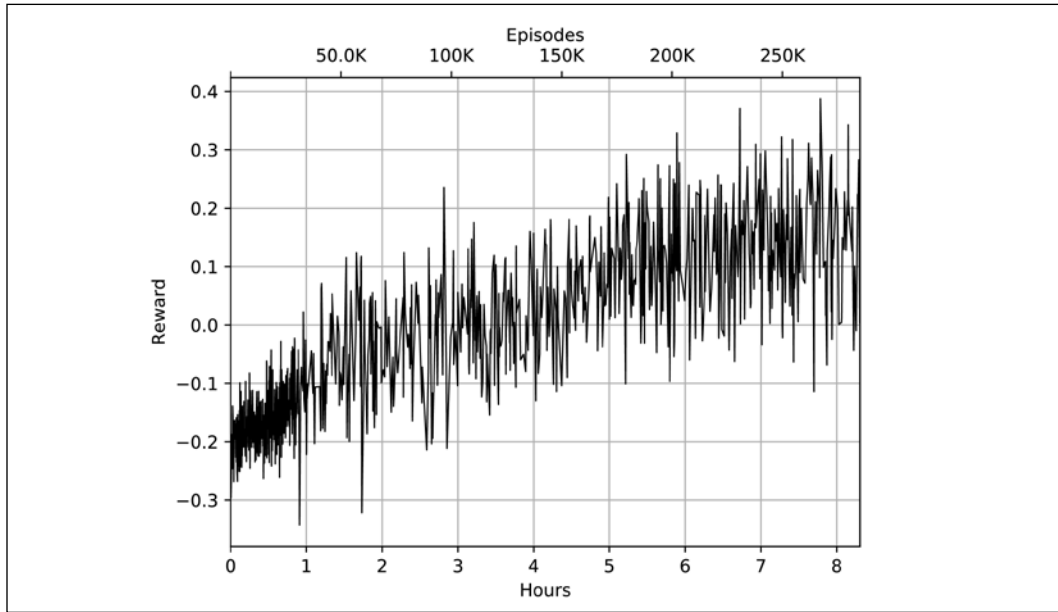
Figure 10.3: The reward for episodes during the training
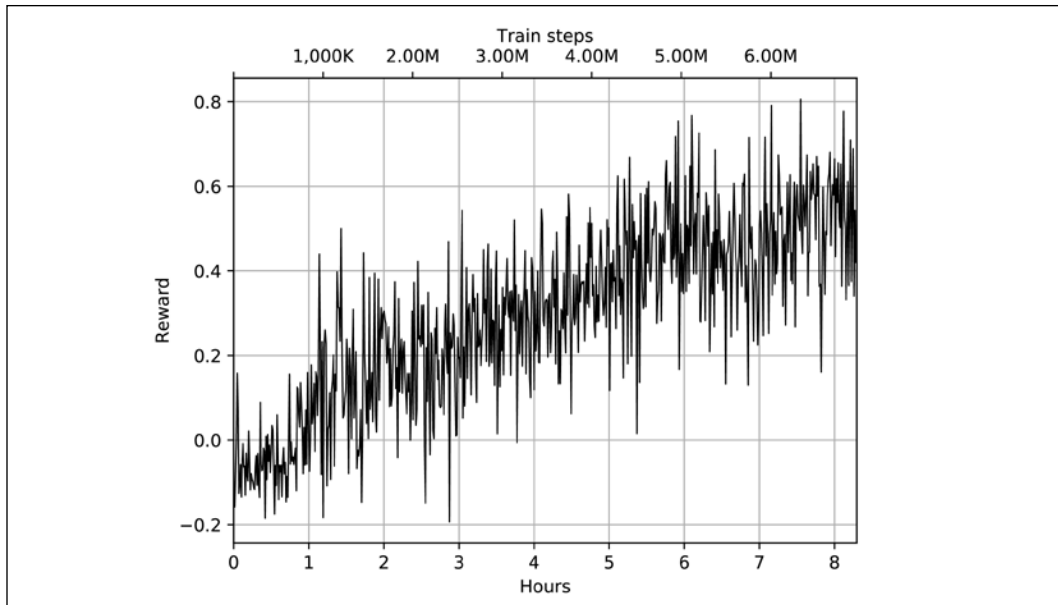
Figure 10.4: The reward for test episodes

The two preceding charts show the reward for episodes played during the training and the reward obtained from testing (which is done on the same quotes, but with `epsilon=0`). From them, we see that our agent is learning how to increase the profit from its actions over time.
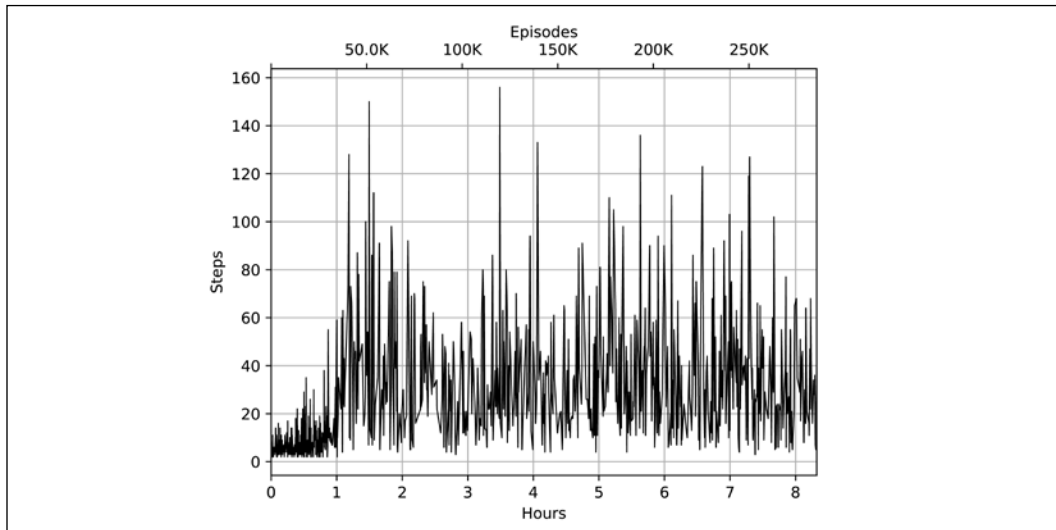
Figure 10.5: The lengths of played episodes
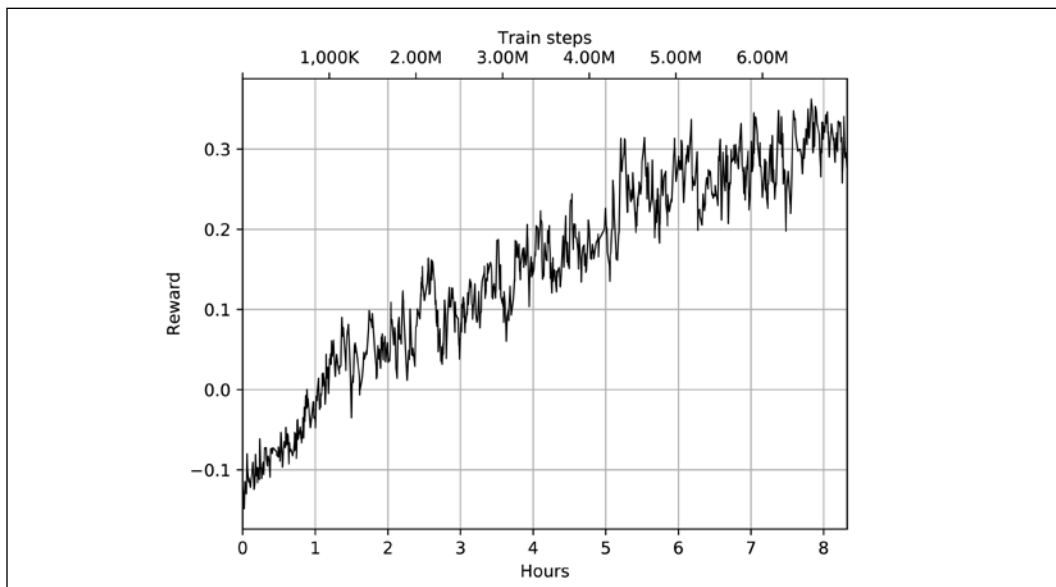
Figure 10.6: The values predicted by the network on a subset of states

The lengths of episodes also increased after 1M training iterations. The number of values predicted by the network is growing.
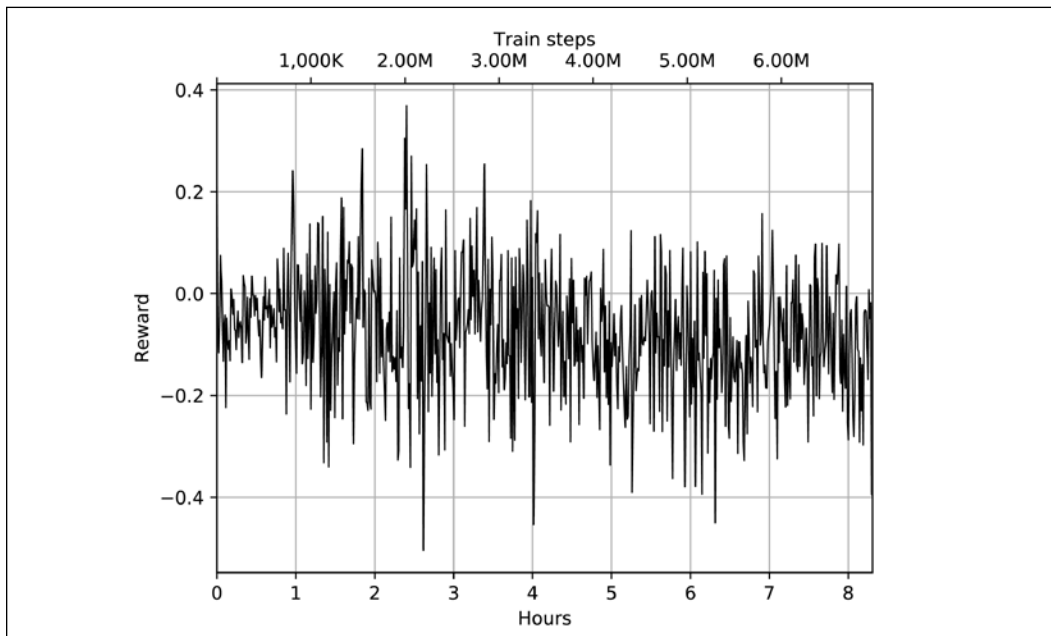
Figure 10.7: The reward obtained on the validation dataset

Figure 10.7 shows quite important information: the amount of reward obtained during the training on the validation set (which is quotes from 2015 by default). This reward doesn't have as obvious a trend as the reward on the training data. This might be an indication of overfitting of the agent, which starts after 3M training iterations. But still, the reward is above line –0.2% (which is a broker commission in our environment) and means that our agent is better than a random "buying and selling monkey."

During the training, our code saves models for later experiments. It does this every time the mean Q-values on our held-out states set update the maximum or when the reward on the validation sets beats the previous record. There is a tool that loads the model, trades on prices you've provided to it with the command-line option, and draws the plots with the profit change over time. The tool is called `Chapter10/run_model.py` and it can be used as shown here:

```
$ ./run_model.py -d data/YNDX_160101_161231.csv -m saves/ff- YNDX16/mean_
val-0.332.data -b 10 -n test
```

The options that the tool accepts are as follows:

- `-d`: This is the path to the quotes to use. In the preceding example, we apply the model to the data that it was trained on.

- `-m`: This is the path to the model file. By default, the training code saves it in the `saves` dir.

- `-b`: This shows how many bars to pass to the model in the context. It has to match the count of bars used on training, which is 10 by default and can be changed in the training code.

- `-n`: This is the suffix to be prepended to the images produced.

- `--commission`: This allows you to redefine the broker's commission, which has a default of 0.1%.

At the end, the tool creates a chart of the total profit dynamics (in percentages). The following is the reward chart on Yandex 2016 quotes (used for training).
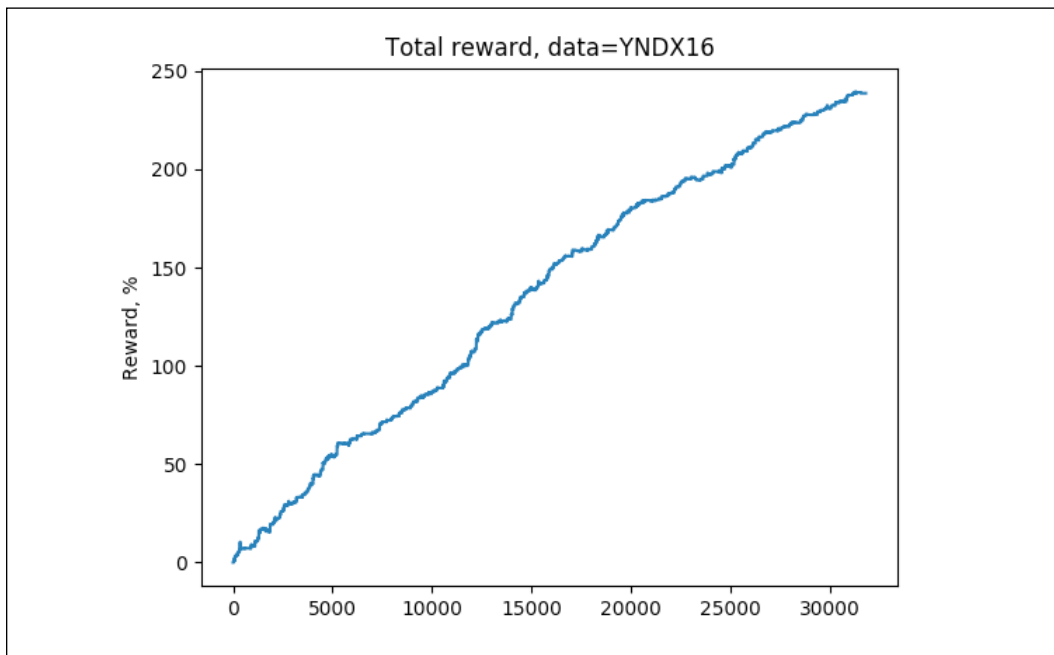
Figure 10.8: Trading profit on the 2016 Yandex training data

The result looks amazing: more than 200% profit in a year. However, let's look at what will happen with the 2015 data:
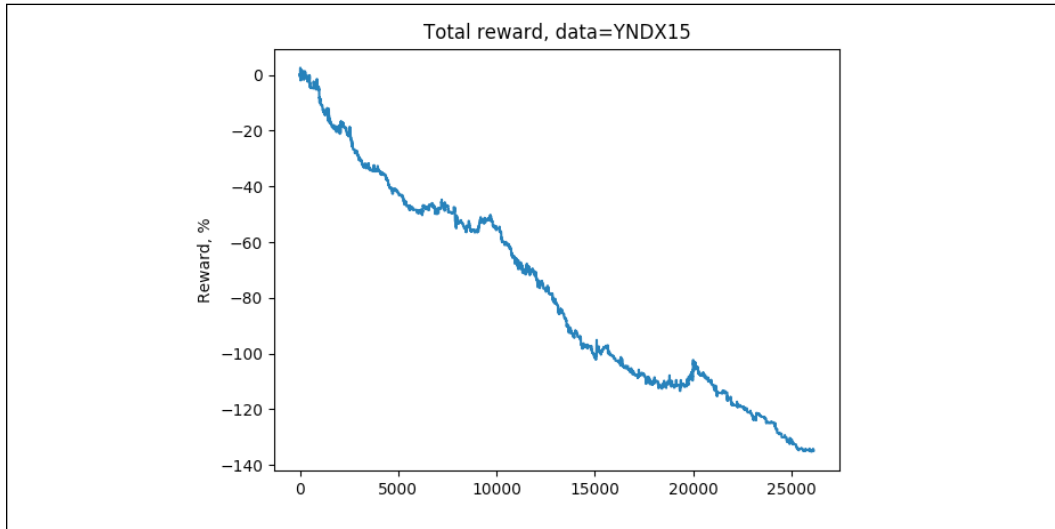
Figure 10.9: Trading profit on the 2015 Yandex validation data

This result is much worse, as we've seen from the validation plots in TensorBoard. To check that our system is profitable with zero commission, we must rerun on the same data with the `--commission 0.0` option.
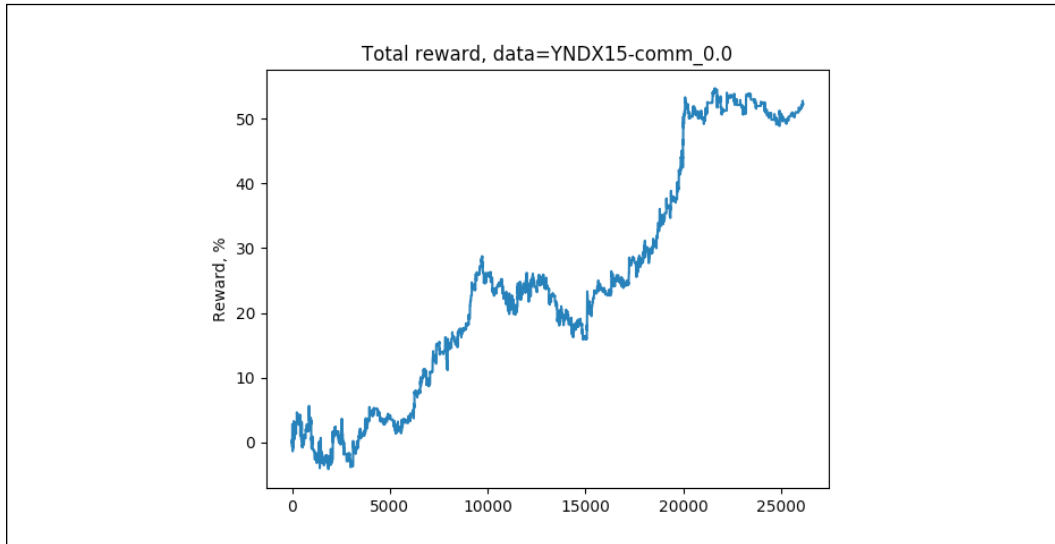
Figure 10.10: Trading profit on validation with zero broker commission

We have some bad days with drawdown, but the overall results are good: without commission, our agent can be profitable. Of course, the commission is not the only issue. Our order simulation is very primitive and doesn't take into account real-life situations, such as price spread and a slip in order execution.

If we take the model with the best reward on the validation set, the reward dynamics are a bit better. Profitability is lower, but the drawdown on unseen quotes is much less.
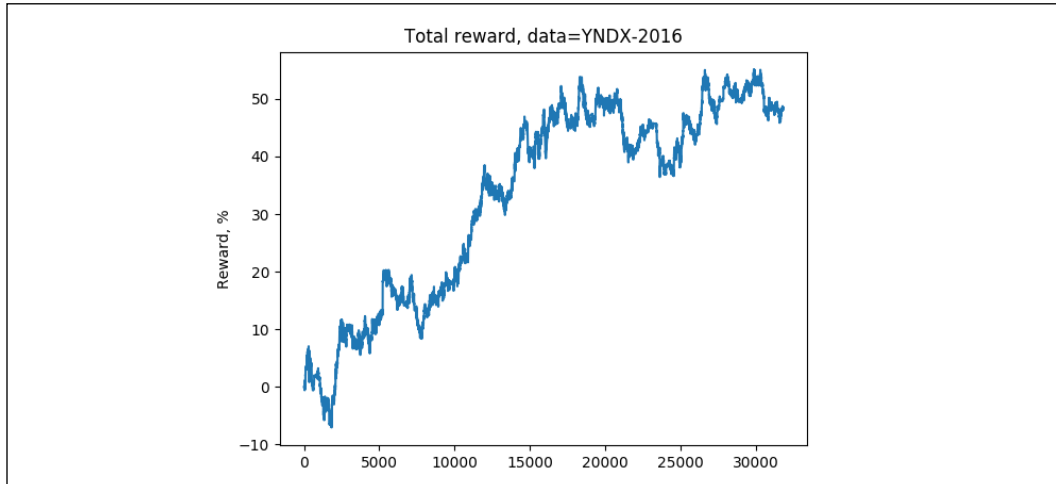
Figure 10.11: The reward from the model with the best validation reward (on 2016 data)
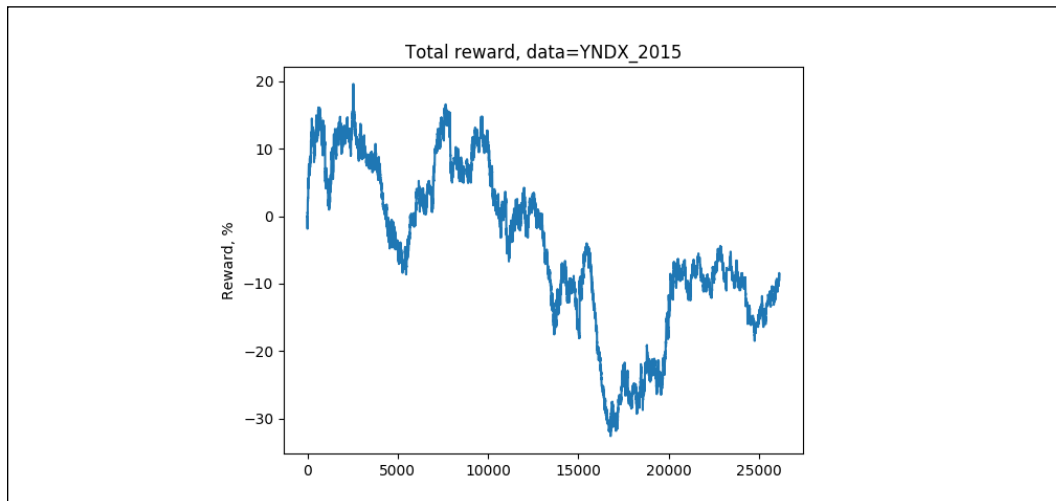
Figure 10.12: The reward from the model with the best validation reward (on 2015 data)

The convolution model

The second model implemented in this example uses 1D convolution filters to extract features from the price data. This allows us to increase the number of bars in the context window that our agent sees on every step without a significant increase in the network size. By default, the convolution model example uses 50 bars of context. The training code is in `Chapter10/train_model_conv.py`, and it accepts the same set of command-line parameters as the feed-forward version.

Training dynamics are almost identical, but the reward obtained on the validation set is slightly higher and starts to overfit later.
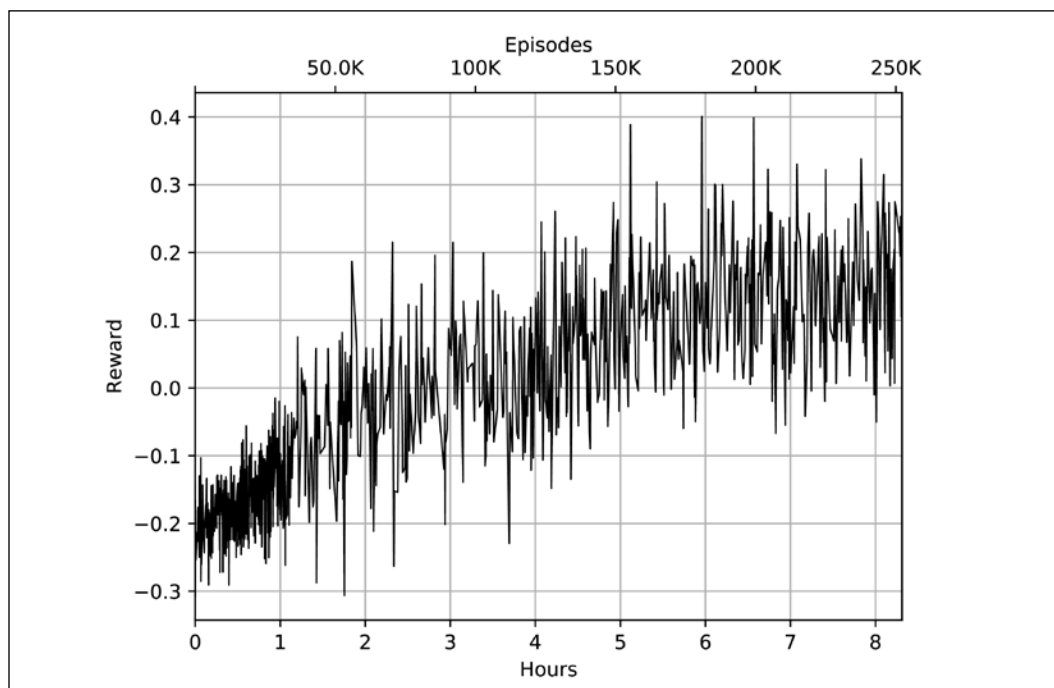
Figure 10.13: The reward during the training of the convolution version
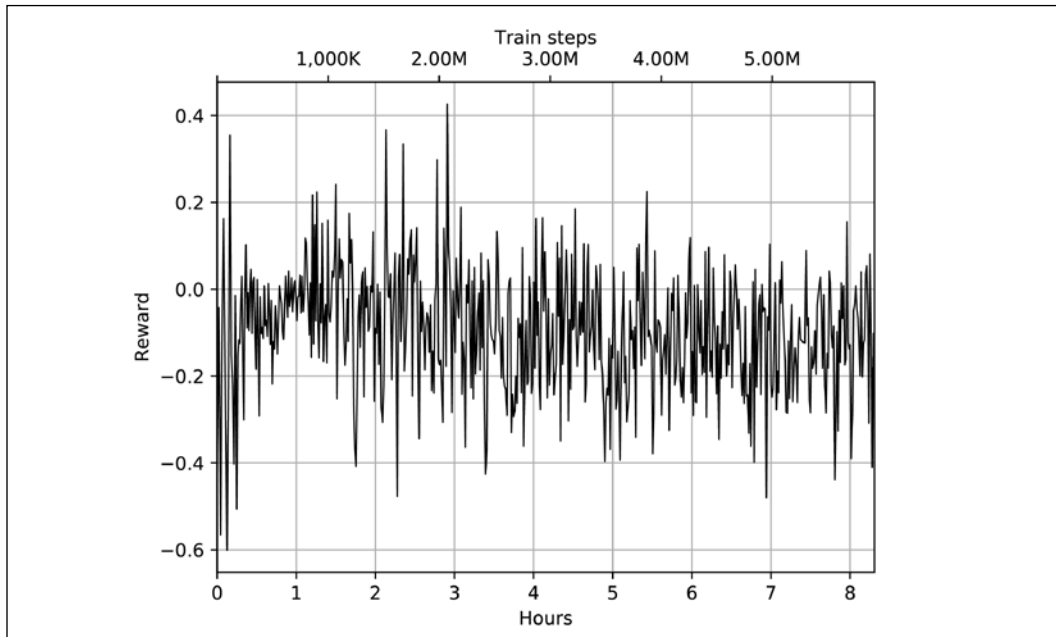
Figure 10.14: The validation reward of the convolution version

Things to try

As already mentioned, financial markets are large and complicated. The methods that we've tried are just the very beginning. Using RL to create a complete and profitable trading strategy is a large project, which can take several months of dedicated labor. However, there are things that we can try to get a better understanding of the topic:

- Our data representation is definitely not perfect. We don't take into account significant price levels (support and resistance), round price values, and others. Incorporating them into the observation could be a challenging problem.

- Market prices are usually analyzed at different timeframes. Low-level data like one-minute bars are noisy (as they include lots of small price movements caused by individual trades), and it is like looking at the market using a microscope. At larger scales, such as one-hour or one-day bars, you can see large, long trends in data movement, which could be extremely important for price prediction.

- More training data is needed. One year of data for one stock is just 130k bars, which might be not enough to capture all market situations. Ideally, a real-life agent should be trained on a much larger dataset, such as the prices for hundreds of stocks for the past 10 years.

- Experiment with the network architecture. The convolution model has shown much faster convergence than the feed-forward model, but there are a lot of things to optimize: the count of layers, kernel size, residual architecture, attention mechanism, and so on.

Summary

In this chapter, we saw a practical example of RL and implemented the trading agent and custom Gym environment. We tried two different architectures: a feed-forward network with price history on input and a 1D convolution network. Both architectures used the DQN method, with some extensions described in *Chapter 8*, *DQN Extensions*.

This is the last chapter in part two of this book. In part three, we will talk about a different family of RL methods: *policy gradients*. We've touched on this approach a bit, but in the upcoming chapters, we will go much deeper into the subject, covering the REINFORCE method and the best method in the family: A3C.

11
Policy Gradients – an Alternative

In this first chapter of part three of the book, we will consider an alternative way to handle **Markov decision process** (**MDP**) problems, which forms a full family of methods called **policy gradient** methods.

In this chapter, we will:

- Cover an overview of the methods, their motivations, and their strengths and weaknesses in comparison to the already familiar Q-learning

- Start with a simple policy gradient method called **REINFORCE** and try to apply it to our CartPole environment, comparing this with the deep Q-network (DQN) approach

Values and policy

Before we start talking about policy gradients, let's refresh our minds with the common characteristics of the methods covered in part two of this book. The central topic in *value iteration* and *Q-learning* is the **value** of the state (V) or value of the state and action (Q). Value is defined as the discounted total reward that we can gather from this state or by issuing this particular action from the state. If we know the value, our decision on every step becomes simple and obvious: we just act greedily in terms of value, and that guarantees us a good total reward at the end of the episode. So, the values of states (in the case of the value iteration method) or state + action (in the case of Q-learning) stand between us and the best reward. To obtain these values, we have used the Bellman equation, which expresses the value on the current step via the values on the next step.

In *Chapter 1*, *What Is Reinforcement Learning?*, we defined the entity that tells us what to do in every state as the **policy**. As in Q-learning methods, when values are dictating to us how to behave, they are actually defining our policy. Formally, this can be written as $\pi(s) = \arg\max_a Q(s, a)$, which means that the result of our policy π, at every state s, is the action with the largest Q.

This policy-values connection is obvious, so I haven't placed emphasis on the policy as a separate entity and we have spent most of our time talking about values and the way to approximate them correctly. Now it's time to focus on this connection and the policy itself.

Why the policy?

There are several reasons why the policy is an interesting topic to explore. First of all, the policy is what we are looking for when we are solving a reinforcement learning (RL) problem. When the agent obtains the observation and needs to make a decision about what to do next, it needs the policy, not the value of the state or particular action. We do care about the total reward, but at every state, we may have little interest in the exact value of the state.

Imagine this situation: you're walking in the jungle and you suddenly realize that there is a hungry tiger hiding in the bushes. You have several alternatives, such as run, hide, or try to throw your backpack at it, but asking, "What's the exact value of the *run* action and is it larger than the value of the *do nothing* action?" is a bit silly. You don't care much about the value, because you need to make the decision on what to do fast and that's it. Our Q-learning approach tried to answer the policy question indirectly by approximating the values of the states and trying to choose the best alternative, but if we are not interested in values, why do extra work?

Another reason why policies may be more attractive than values is due to environments with lots of actions or, in the extreme case, with a **continuous** action space. To be able to decide on the best action to take having $Q(s, a)$, we need to solve a small optimization problem, finding a, which maximizes $Q(s, a)$. In the case of an Atari game with several discrete actions, this wasn't a problem: we just approximated values of all actions and took the action with the largest Q. If our action is not a small discrete set, but has a scalar value attached to it, such as a steering wheel angle or the speed at which we want to run from the tiger, this optimization problem becomes hard, as Q is usually represented by a highly nonlinear neural network (NN), so finding the argument that maximizes the function's values can be tricky. In such cases, it's much more feasible to avoid values and work with the policy directly.

An extra benefit of policy learning is an environment with **stochasticity**. As you saw in *Chapter 8, DQN Extensions*, in categorical DQN, our agent can benefit a lot from working with the distribution of Q-values, instead of expected mean values, as our network can more precisely capture the underlying probability distribution. As you will see in the next section, the policy is naturally represented as the probability of actions, which is a step in the same direction as the categorical DQN method.

Policy representation

Now that you know the benefits of the policy, let's give it a try. So how do we represent the policy? In the case of Q-values, they were parametrized by the NN that returns values of actions as scalars. If we want our network to parametrize the actions, we have several options. The first and the simplest way could be just returning the identifier of the action (in the case of a discrete set of actions). However, this is not the best way to deal with a discrete set. A much more common solution, which is heavily used in classification tasks, is to return the probability distribution of our actions. In other words, for N mutually exclusive actions, we return N numbers representing the probability of taking each action in the given state (which we pass as an input to the network). This representation is shown in the following diagram:
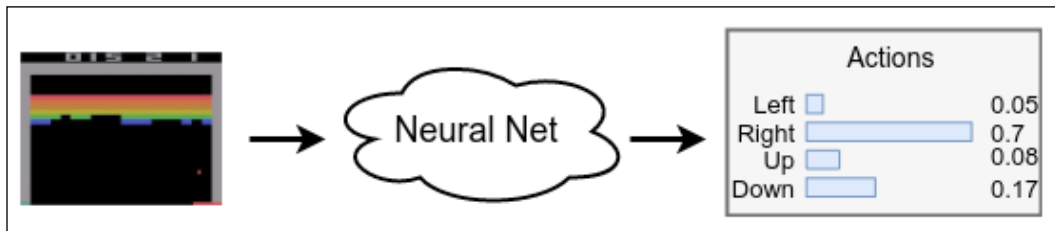
Figure 11.1: Policy approximation with an NN for a discrete set of actions

Such representation of actions as probabilities has the additional advantage of **smooth representation**: if we change our network weights a bit, the output of the network will also change. In the case of a discrete numbers output, even a small adjustment of the weights can lead to a jump to a different action. However, if our output is the probability distribution, a small change of weights will usually lead to a small change in output distribution, such as slightly increasing the probability of one action versus the others. This is a very nice property to have, as gradient optimization methods are all about tweaking the parameters of a model a bit to improve the results. In math notation, the policy is usually represented as $\pi(s)$, so we will use this notation as well.

Policy gradients

We defined our policy representation, but what we haven't seen so far is how we are going to change our network's parameters to improve the policy. If you remember from *Chapter 4*, *The Cross-Entropy Method*, we solved a very similar problem using the cross-entropy method: our network took observations as inputs and returned the probability distribution of the actions. In fact, the cross-entropy method is a younger brother of the methods that we will discuss in this part of the book. To start, we will get acquainted with the method called REINFORCE, which has only minor differences from the cross-entropy method, but first, we need to look at some mathematical notation that we will use in this and the following chapters.

We define the policy gradient as $\nabla J \approx \mathbb{E}[Q(s,a)\nabla \log \pi(a|s)]$. Of course, there is a strong proof of this, but it's not that important. What interests us much more is the meaning of this expression.

The policy gradient defines the direction in which we need to change our network's parameters to improve the policy in terms of the accumulated total reward. The scale of the gradient is proportional to the value of the action taken, which is $Q(s, a)$ in the formula, and the gradient itself is equal to the gradient of log probability of the action taken. This means that we are trying to increase the probability of actions that have given us good total reward and decrease the probability of actions with bad final outcomes. Expectation, E in the formula, just means that we average the gradient of several steps that we have taken in the environment.

From a practical point of view, policy gradient methods could be implemented by performing optimization of this loss function: $\mathcal{L} = -Q(s,a)\log \pi(a|s)$. The minus sign is important, as the loss function is **minimized** during **stochastic gradient descent** (**SGD**), but we want to **maximize** our policy gradient. You will see code examples of policy gradient methods later in this and the following chapters.

The REINFORCE method

The formula of policy gradient that you have just seen is used by most of the policy-based methods, but the details can vary. One very important point is how exactly gradient scales, $Q(s, a)$, are calculated. In the cross-entropy method from *Chapter 4*, *The Cross-Entropy Method*, we played several episodes, calculated the total reward for each of them, and trained on transitions from episodes with a better-than-average reward. This training procedure is a policy gradient method with $Q(s, a) = 1$ for state and action pairs from *good* episodes (with a large total reward) and $Q(s, a) = 0$ for state and action pairs from *worse* episodes.

The cross-entropy method worked even with those simple assumptions, but the obvious improvement will be to use $Q(s, a)$ for training instead of just 0 and 1. Why should it help? The answer is a more fine-grained separation of episodes. For example, transitions of the episode with the total reward of 10 should contribute to the gradient more than transitions from the episode with the reward of 1. The second reason to use $Q(s, a)$ instead of just 0 or 1 constants is to increase probabilities of good actions in the beginning of the episode and decrease the actions closer to the end of the episode (as $Q(s, a)$ incorporates the discount factor, uncertainty for longer sequences of actions is automatically taken into account). That's exactly the idea of the method called REINFORCE. Its steps are as follows:

1. Initialize the network with random weights

2. Play N full episodes, saving their (s, a, r, s') transitions

3. For every step, t, of every episode, k, calculate the discounted total reward for subsequent steps: $Q_{k,t} = \sum_{i=0} \gamma^i r_i$

4. Calculate the loss function for all transitions: $\mathcal{L} = -\sum_{k,t} Q_{k,t} \log \left(\pi \left(s_{k,t}, a_{k,t} \right) \right)$

5. Perform an SGD update of weights, minimizing the loss

6. Repeat from step 2 until converged

The preceding algorithm is different from Q-learning in several important aspects:

*   No explicit exploration is needed. In Q-learning, we used an epsilon-greedy strategy to explore the environment and prevent our agent from getting stuck with a non-optimal policy. Now, with probabilities returned by the network, the exploration is performed automatically. In the beginning, the network is initialized with random weights, and it returns a uniform probability distribution. This distribution corresponds to random agent behavior.

*   No replay buffer is used. Policy gradient methods belong to the on-policy methods class, which means that we can't train on data obtained from the old policy. This is both good and bad. The good part is that such methods usually converge faster. The bad side is that they usually require much more interaction with the environment than off-policy methods such as DQN.

*   No target network is needed. Here, we use Q-values, but they are obtained from our experience in the environment. In DQN, we used the target network to break the correlation in Q-values approximation, but we are not approximating anymore. In the next chapter, you will see that the target network trick can still be useful in policy gradient methods.

The CartPole example

To see the method in action, let's check the implementation of the REINFORCE method on the familiar CartPole environment. The full code of the example is in `Chapter11/02_cartpole_reinforce.py`.

```
GAMMA = 0.99
LEARNING_RATE = 0.01
EPISODES_TO_TRAIN = 4
```

In the beginning, we define hyperparameters (imports are omitted). The `EPISODES_TO_TRAIN` value specifies how many complete episodes we will use for training.

```
class PGN(nn.Module):
    def __init__(self, input_size, n_actions):
        super(PGN, self).__init__()

        self.net = nn.Sequential(nn.Linear(input_size, 128),
                                 nn.ReLU(),
                                 nn.Linear(128, n_actions)
                                 )

    def forward(self, x):
        return self.net(x)
```

The network should also be familiar to you. Note that despite the fact our network returns probabilities, we are not applying softmax nonlinearity to the output. The reason behind this is that we will use the PyTorch `log_softmax` function to calculate the logarithm of the softmax output at once. This method of calculation is much more numerically stable; however, we need to remember that output from the network is not probability, but raw scores (usually called logits).

```
def calc_qvals(rewards):
    res = []
    sum_r = 0.0
    for r in reversed(rewards):
        sum_r *= GAMMA
        sum_r += r
        res.append(sum_r)
    return list(reversed(res))
```

This function is a bit tricky. It accepts a list of rewards for the whole episode and needs to calculate the discounted total reward for every step. To do this efficiently, we calculate the reward from the end of the local reward list. Indeed, the last step of the episode will have a total reward equal to its local reward. The step before the last will have the total reward of $r_{t-1} + \gamma r_t$ (if t is an index of the last step).

Our `sum_r` variable contains the total reward for the previous steps, so to get the total reward for the previous step, we need to multiply `sum_r` by gamma and sum the local reward.

```
if __name__ == "__main__":
    env = gym.make("CartPole-v0")
    writer = SummaryWriter(comment="-cartpole-reinforce")

    net = PGN(env.observation_space.shape[0], env.action_space.n)
    print(net)

    agent = ptan.agent.PolicyAgent(net,
                                   preprocessor=ptan.agent.
                                       float32_preprocessor,
                                   apply_softmax=True)
    exp_source = ptan.experience.ExperienceSourceFirstLast(
                    env, agent, gamma=GAMMA)

    optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE)
```

The preparation steps before the training loop should also be familiar to you. The only new element is the `agent` class from the PTAN library. Here, we are using `ptan.agent.PolicyAgent`, which needs to make a decision about actions for every observation. As our network now returns the policy as the probabilities of the actions, in order to select the action to take, we need to obtain the probabilities from the network and then perform random sampling from this probability distribution.

When we worked with DQN, the output of the network was Q-values, so if some action had the value of 0.4 and another action had 0.5, the second action was preferred 100% of the time. In the case of the probability distribution, if the first action has a probability of 0.4 and the second 0.5, our agent should take the first action with a 40% chance and the second with a 50% chance. Of course, our network can decide to take the second action 100% of the time, and in this case, it returns the probability 0 for the first action and the probability 1 for the second action.

This difference is important to understand, but the change in the implementation is not large. Our `PolicyAgent` internally calls the NumPy `random.choice` function with probabilities from the network. The `apply_softmax` argument instructs it to convert the network output to probabilities by calling softmax first. The third argument preprocessor is a way to get around the fact that the CartPole environment in Gym returns the observation as `float64` instead of the `float32` required by PyTorch.

```
    total_rewards = []
    done_episodes = 0

    batch_episodes = 0
    cur_rewards = []
    batch_states, batch_actions, batch_qvals = [], [], []
```

Before we can start the training loop, we need several variables. The first group of these is used for reporting and contains the total rewards for the episodes and the count of completed episodes. The second group is used to gather the training data. The `cur_rewards` list contains local rewards for the episode being currently played. As this episode reaches the end, we calculate the discounted total rewards from local rewards using the `calc_qvals` function and append them to the `batch_qvals` list. The `batch_states` and `batch_actions` lists contain states and actions that we saw from the last training.

```
for step_idx, exp in enumerate(exp_source):
    batch_states.append(exp.state)
    batch_actions.append(int(exp.action))
    cur_rewards.append(exp.reward)

    if exp.last_state is None:
        batch_qvals.extend(calc_qvals(cur_rewards))
        cur_rewards.clear()
        batch_episodes += 1
```

The preceding code snippet is the beginning of the training loop. Every experience that we get from the experience source contains the state, action, local reward, and next state. If the end of the episode has been reached, the next state will be `None`. For non-terminal experience entries, we just save the state, action, and local reward in our lists. At the end of the episode, we convert the local rewards into Q-values and increment the episodes counter.

```
new_rewards = exp_source.pop_total_rewards()
if new_rewards:
    done_episodes += 1
    reward = new_rewards[0]
    total_rewards.append(reward)
    mean_rewards = float(np.mean(total_rewards[-100:]))
    print("%d: reward: %6.2f, mean_100: %6.2f,"\
          "episodes: %d" % (step_idx, reward,
                            mean_rewards,
                            done_episodes))
    writer.add_scalar("reward", reward, step_idx)
    writer.add_scalar("reward_100", mean_rewards,
                      step_idx)
    writer.add_scalar("episodes", done_episodes, step_idx)
    if mean_rewards > 195:
        print("Solved in %d steps and %d episodes!" %
              (step_idx, done_episodes))
        break
```

This part of the training loop is performed at the end of the episode and is responsible for reporting the current progress and writing metrics to TensorBoard.

```
                if batch_episodes < EPISODES_TO_TRAIN:
                    continue

                optimizer.zero_grad()
                states_v = torch.FloatTensor(batch_states)
                batch_actions_t = torch.LongTensor(batch_actions)
                batch_qvals_v = torch.FloatTensor(batch_qvals)
```

When enough episodes have passed since the last training step, we perform optimization on the gathered examples. As a first step, we convert states, actions, and Q-values into the appropriate PyTorch form.

```
                logits_v = net(states_v)
                log_prob_v = F.log_softmax(logits_v, dim=1)
                log_prob_actions_v = batch_qvals_v * log_prob_v[
                                        range(len(batch_states)),
                                        batch_actions_t]
                loss_v = -log_prob_actions_v.mean()
```

Then we calculate the loss from the steps. To do this, we ask our network to calculate states into logits and calculate the logarithm + softmax of them. On the third line, we select log probabilities from the actions taken and scale them with Q-values. On the last line, we average those scaled values and do negation to obtain the loss to minimize. Once again, this minus sign is very important, as our policy gradient needs to be maximized to improve the policy. As the optimizer in PyTorch does minimization in respect to the loss function, we need to negate the policy gradient.

```
                loss_v.backward()
                optimizer.step()
                batch_episodes = 0
                batch_states.clear()
                batch_actions.clear()
                batch_qvals.clear()

        writer.close()
```

The rest of the code is clear: we perform backpropagation to gather gradients in our variables and ask the optimizer to perform an SGD update. At the end of the training loop, we reset the episodes counter and clear our lists for fresh data to gather.

Results

For reference, I've implemented DQN on the CartPole environment with almost the same hyperparameters as our REINFORCE example. It's in `Chapter11/01_cartpole_dqn.py`.

Neither example requires any command-line arguments, and they should converge in less than a minute.

```
rl_book_samples/chapter11$ ./02_cartpole_reinforce.py
PGN (
  (net): Sequential (
    (0): Linear (4 -> 128)
    (1): ReLU ()
    (2): Linear (128 -> 2)
  )
)
63:  reward: 62.00, mean_100: 62.00, episodes: 1
83:  reward: 19.00, mean_100: 40.50, episodes: 2
99:  reward: 15.00, mean_100: 32.00, episodes: 3
125: reward: 25.00, mean_100: 30.25, episodes: 4
154: reward: 28.00, mean_100: 29.80, episodes: 5
...
27676: reward: 200.00, mean_100: 193.58, episodes: 224
27877: reward: 200.00, mean_100: 194.07, episodes: 225
28078: reward: 200.00, mean_100: 194.07, episodes: 226
28279: reward: 200.00, mean_100: 194.53, episodes: 227
28480: reward: 200.00, mean_100: 195.09, episodes: 228
Solved in 28480 steps and 228 episodes!
```

The convergence dynamics for both DQN and REINFORCE are shown in the following charts. Your training dynamics may vary due to the randomness of the training.
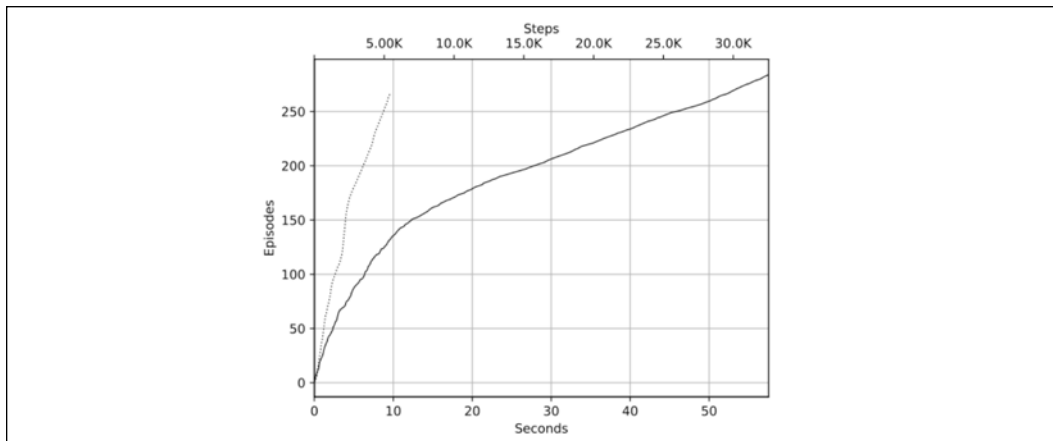
Figure 11.2: The number of episodes before CartPole is solved. The solid line is DQN and the dotted line is REINFORCE
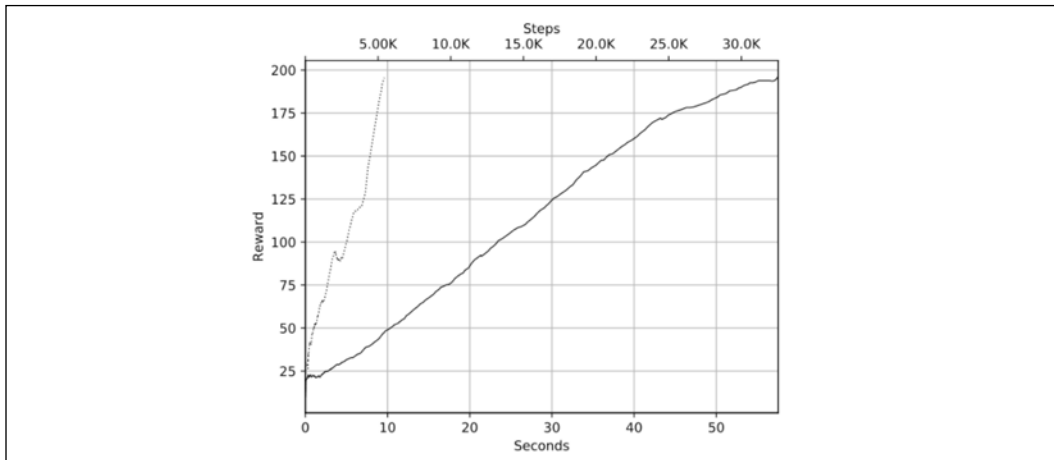
Figure 11.3: Smoothed reward dynamics. The solid line is DQN and the dotted line is REINFORCE

As you can see, REINFORCE converges faster and requires fewer training steps and episodes to solve the CartPole environment. If you remember from *Chapter 4, The Cross-Entropy Method*, the cross-entropy method required about 40 batches of 16 episodes each to solve the CartPole environment, which is 640 episodes in total. The REINFORCE method was able to do the same in fewer than 300 episodes, which is a nice improvement.

Policy-based versus value-based methods

Let's now step back from the code that we have just seen and talk about the differences that both families of methods have:

- Policy methods directly optimize what we care about: our behavior. Value methods, such as DQN, do the same indirectly, learning the value first and providing us with the policy based on this value.

- Policy methods are on-policy and require fresh samples from the environment. Value methods can benefit from old data, obtained from the old policy, human demonstration, and other sources.

- Policy methods are usually less sample-efficient, which means they require more interaction with the environment. Value methods can benefit from large replay buffers. However, sample efficiency doesn't mean that value methods are more computationally efficient, and very often, it's the opposite.

- In the preceding example, during the training, we needed to access our NN only once, to get the probabilities of actions. In DQN, we need to process two batches of states: one for the current state and another for the next state in the Bellman update.

As you can see, there is no strong preference for one family versus another. In some situations, policy methods will be the more natural choice, like in continuous control problems or cases when access to the environment is cheap and fast. However, there are many situations when value methods will shine, for example, the recent state-of-the-art results on Atari games achieved by DQN variants. Ideally, you should be familiar with both families and understand the strong and weak sides of both camps.

In the next section, we will talk about the REINFORCE method's limitations, ways to improve it, and how to apply a policy gradient method to our favorite Pong game.

REINFORCE issues

In the previous section, we discussed the REINFORCE method, which is a natural extension of the cross-entropy method. Unfortunately, both REINFORCE and the cross-entropy method still suffer from several problems, which make both of them limited to simple environments.

Full episodes are required

First of all, we still need to wait for the full episode to complete before we can start training. Even worse, both REINFORCE and the cross-entropy method behave better with more episodes used for training (just from the fact that more episodes mean more training data, which means more accurate policy gradients). This situation is fine for short episodes in the CartPole, when in the beginning, we can barely handle the bar for more than 10 steps; but in Pong, it is completely different: every episode can last for hundreds or even thousands of frames. It's equally bad from the training perspective, as our training batch becomes very large, and from the sample efficiency perspective, as we need to communicate with the environment a lot just to perform a single training step.

The purpose of the complete episode requirement is to get as accurate a Q-estimation as possible. When we talked about DQN, you saw that, in practice, it's fine to replace the exact value for a discounted reward with our estimation using the one-step Bellman equation: $Q(s, a) = r_a + \gamma V(s')$. To estimate $V(s)$, we used our own Q-estimation, but in the case of the policy gradient, we don't have $V(s)$ or $Q(s, a)$ anymore.

To overcome this, two approaches exist. On the one hand, we can ask our network to estimate $V(s)$ and use this estimation to obtain Q. This approach will be discussed in the next chapter and is called the *actor-critic method*, which is the most popular method from the policy gradient family.

On the other hand, we can do the Bellman equation, unrolling *N* steps ahead, which will effectively exploit the fact that the value contribution decreases when gamma is less than 1. Indeed, with gamma=0.9, the value coefficient at the 10th step will be $0.9^{10} = 0.35$. At step 50, this coefficient will be $0.9^{50} = 0.00515$, which is a really small contribution to the total reward. In the case of gamma=0.99, the required count of steps will become larger, but we can still do this.

High gradients variance

In the policy gradient formula, $\nabla J \approx E[Q(s,a)\nabla \log \pi(a|s)]$, we have a gradient proportional to the discounted reward from the given state. However, the range of this reward is heavily environment-dependent. For example, in the CartPole environment, we get the reward of 1 for every timestamp that we are holding the pole vertically. If we can do this for five steps, we get a total (undiscounted) reward of 5. If our agent is smart and can hold the pole for, say, 100 steps, the total reward will be 100. The difference in value between those two scenarios is 20 times, which means that the scale between the gradients of unsuccessful samples will be 20 times lower than for more successful ones. Such a large difference can seriously affect our training dynamics, as one lucky episode will dominate in the final gradient.

In mathematical terms, the policy gradient has high variance, and we need to do something about this in complex environments; otherwise, the training process can become unstable. The usual approach to handling this is subtracting a value called the baseline from the *Q*. The possible choices for the baseline are as follows:

- Some constant value, which is normally the mean of the discounted rewards
- The moving average of the discounted rewards
- The value of the state, *V(s)*

Exploration

Even with the policy represented as the probability distribution, there is a high chance that the agent will converge to some locally optimal policy and stop exploring the environment. In DQN, we solved this using epsilon-greedy action selection: with probability epsilon, the agent took some random action instead of the action dictated by the current policy. We can use the same approach, of course, but policy gradient methods allow us to follow a better path, called the *entropy bonus*.

In information theory, entropy is a measure of uncertainty in some system. Being applied to the agent's policy, entropy shows how much the agent is uncertain about which action to take. In math notation, the entropy of the policy is defined as $H(\pi) = -\sum \pi(a|s) \log \pi(a|s)$. The value of entropy is always greater than zero and has a single maximum when the policy is uniform; in other words, all actions have the same probability. Entropy becomes minimal when our policy has 1 for some action and 0 for all others, which means that the agent is absolutely sure what to do. To prevent our agent from being stuck in the local minimum, we subtract the entropy from the loss function, punishing the agent for being too certain about the action to take.

Correlation between samples

As we discussed in *Chapter 6*, *Deep Q-Networks*, training samples in a single episode are usually heavily correlated, which is bad for SGD training. In the case of DQN, we solved this issue by having a large replay buffer with 100k-1M observations that we sampled our training batch from. This solution is not applicable to the policy gradient family anymore because those methods belong to the on-policy class. The implication is simple: using old samples generated by the old policy, we will get policy gradients for that old policy, not for our current one.

The obvious, but unfortunately wrong, solution would be to reduce the replay buffer size. It might work in some simple cases, but in general, we need fresh training data generated by our current policy. To solve this, parallel environments are normally used. The idea is simple: instead of communicating with one environment, we use several and use their transitions as training data.

Policy gradient methods on CartPole

Nowadays, almost nobody uses the vanilla policy gradient method, as the much more stable actor-critic method exists. However, I still want to show the policy gradient implementation, as it establishes very important concepts and metrics to check the policy gradient method's performance.

Implementation

So, we will start with a much simpler environment of CartPole, and in the next section, we will check its performance on our favorite Pong environment.

The complete code for the following example is available in `Chapter11/04_ cartpole_pg.py`.

```
GAMMA = 0.99
LEARNING_RATE = 0.001
ENTROPY_BETA = 0.01
BATCH_SIZE = 8
REWARD_STEPS = 10
```

Besides the already familiar hyperparameters, we have two new ones: the `ENTROPY_ BETA` value is the scale of the entropy bonus and the `REWARD_STEPS` value specifies how many steps ahead the Bellman equation is unrolled to estimate the discounted total reward of every transition.

```
class PGN(nn.Module):
    def __init__(self, input_size, n_actions):
        super(PGN, self).__init__()

        self.net = nn.Sequential(
                    nn.Linear(input_size, 128),
                    nn.ReLU(),
                    nn.Linear(128, n_actions)
                )

    def forward(self, x):
        return self.net(x)
```

The network architecture is exactly the same as in the previous examples for CartPole: a two-layer network with 128 neurons in the hidden layer. The preparation code is also the same as before, except the experience source is asked to unroll the Bellman equation for 10 steps. The following is the part that differs from `04_ cartpole_pg.py`:

```
exp_source = ptan.experience.ExperienceSourceFirstLast(
    env, agent, gamma=GAMMA, steps_count=REWARD_STEPS)
```

In the training loop, we maintain the sum of the discounted reward for every transition and use it to calculate the baseline for the policy scale:

```
for step_idx, exp in enumerate(exp_source):
    reward_sum += exp.reward
    baseline = reward_sum / (step_idx + 1)
    writer.add_scalar("baseline", baseline, step_idx)
    batch_states.append(exp.state)
    batch_actions.append(int(exp.action))
    batch_scales.append(exp.reward - baseline)
```

In the loss calculation, we use the same code as before to calculate the policy loss (which is the negated policy gradient):

```
optimizer.zero_grad()
logits_v = net(states_v)
log_prob_v = F.log_softmax(logits_v, dim=1)
log_prob_actions_v = batch_scale_v * log_prob_v[
                        range(BATCH_SIZE),
                        batch_actions_t
                        ]
loss_policy_v = -log_prob_actions_v.mean()
```

Then we add the entropy bonus to the loss by calculating the entropy of the batch and subtracting it from the loss. As entropy has a maximum for uniform probability distribution and we want to push the training toward this maximum, we need to subtract from the loss.

```
prob_v = F.softmax(logits_v, dim=1)
entropy_v = -(prob_v * log_prob_v).sum(dim=1).mean()
entropy_loss_v = -ENTROPY_BETA * entropy_v
loss_v = loss_policy_v + entropy_loss_v

loss_v.backward()
optimizer.step()
```

Then, we calculate the Kullback-Leibler (KL) divergence between the new policy and the old policy. KL divergence is an information theory measurement of how one probability distribution diverges from another expected probability distribution. In our example, it is being used to compare the policy returned by the model before and after the optimization step. High spikes in KL are usually a bad sign, showing that our policy was pushed too far from the previous policy, which is a bad idea most of the time (as our NN is a very nonlinear function in a high-dimensional space, such large changes in the model weight could have a very strong influence on the policy).

```
new_logits_v = net(states_v)
new_prob_v = F.softmax(new_logits_v, dim=1)
kl_div_v = -((new_prob_v / prob_v).log() *
                prob_v).sum(dim=1).mean()
writer.add_scalar("kl", kl_div_v.item(), step_idx)
```

Finally, we calculate the statistics about the gradients on this training step. It's usually good practice to show the graph of the maximum and L2 norm of gradients to get an idea about the training dynamics.

```
grad_max = 0.0
grad_means = 0.0
grad_count = 0
for p in net.parameters():
    grad_max = max(grad_max, p.grad.abs().max().item())
    grad_means += (p.grad ** 2).mean().sqrt().item()
    grad_count += 1
```

At the end of the training loop, we dump all the values that we want to monitor in TensorBoard:

```
writer.add_scalar("baseline", baseline, step_idx)
writer.add_scalar("entropy", entropy_v.item(), step_idx)
writer.add_scalar("batch_scales",
                  np.mean(batch_scales),
                  step_idx)
writer.add_scalar("loss_entropy",
                  entropy_loss_v.item(),
                  step_idx)
writer.add_scalar("loss_policy",
                  loss_policy_v.item(),
                  step_idx)
writer.add_scalar("loss_total",
                  loss_v.item(),
                  step_idx)
writer.add_scalar("grad_l2",
                  grad_means / grad_count,
                  step_idx)
writer.add_scalar("grad_max", grad_max, step_idx)

batch_states.clear()
batch_actions.clear()
batch_scales.clear()
```

Results

In this example, we will plot a lot of charts in TensorBoard. Let's start with the familiar one: reward. As you can see in the following chart, the dynamics and performance are not very different from the REINFORCE method.
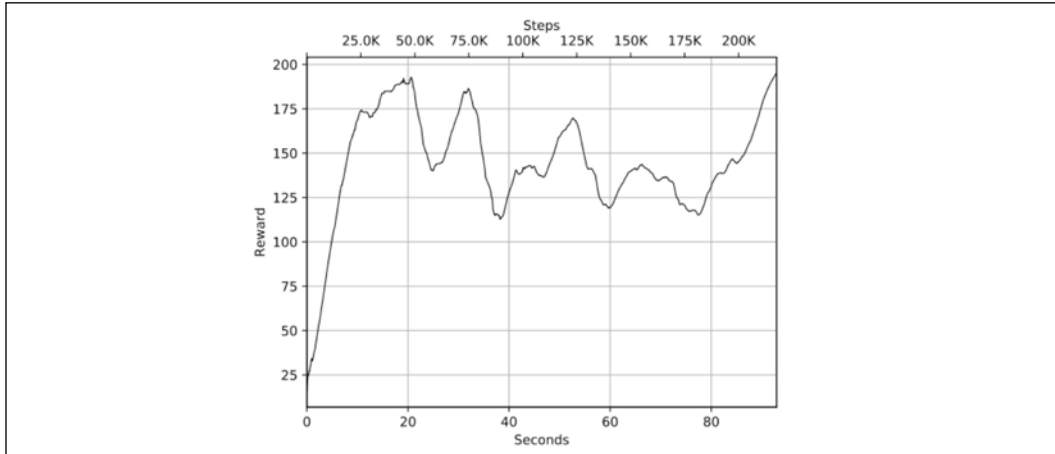
Figure 11.4: The reward dynamics of the policy gradient method

The next two charts are related to our baseline and scales of policy gradients. We expect the baseline to converge to $1 + 0.99 + 0.99^2 + \ldots + 0.99^9$, which is approximately 9.56. Scales of policy gradients should oscillate around zero. That's exactly what we can see in the following graph.
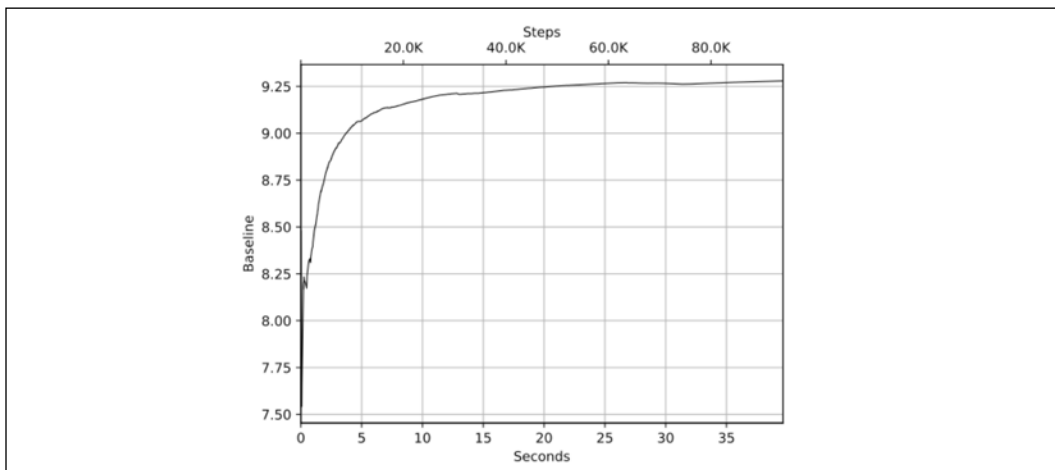
Figure 11.5: The baseline value during the training
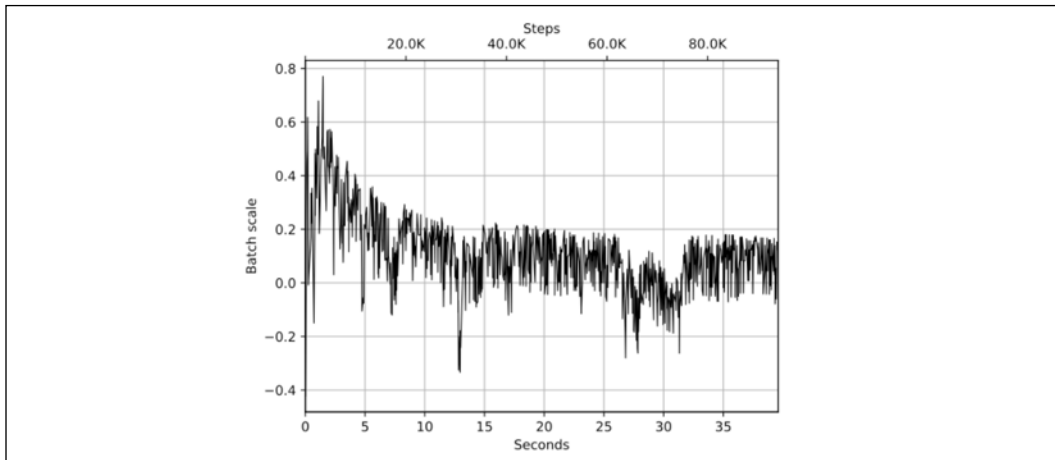
Figure 11.6: Batch scales

The entropy is decreasing over time from 0.69 to 0.52. The starting value corresponds to the maximum entropy with two actions, which is approximately 0.69:

$$H(\pi) = -\sum_{a} \pi(a|s) \log \pi(a|s) = -\left(\frac{1}{2}\log\left(\frac{1}{2}\right) + \frac{1}{2}\log\left(\frac{1}{2}\right)\right) \approx 0.69$$

The fact that the entropy is decreasing during the training, as indicated by the following chart, shows that our policy is moving from uniform distribution to more deterministic actions.
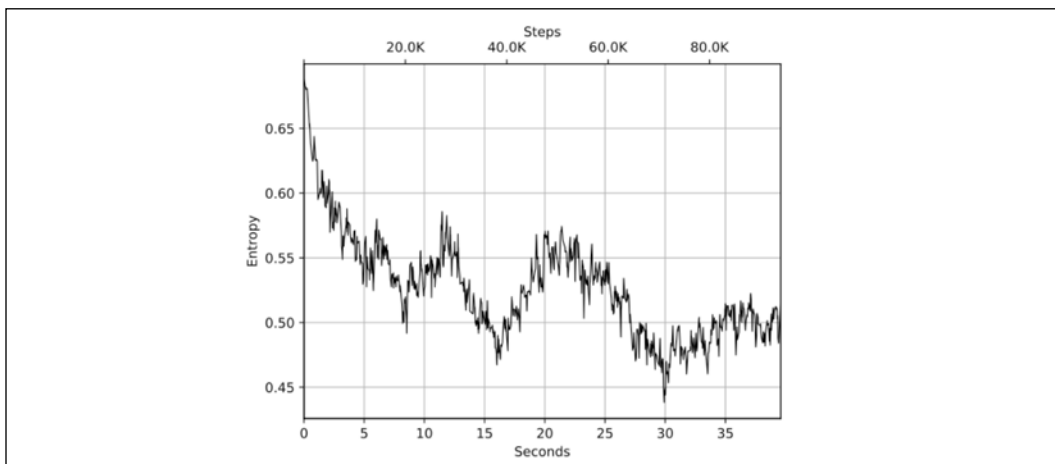
Figure 11.7: Entropy during the training

The next group of plots is related to loss, which includes policy loss, entropy loss, and their sum. The entropy loss is scaled and is a mirrored version of the preceding entropy chart. The policy loss shows the mean scale and direction of the policy gradient computed on the batch. Here, we should check the relative size of both, to prevent entropy loss from dominating too much.
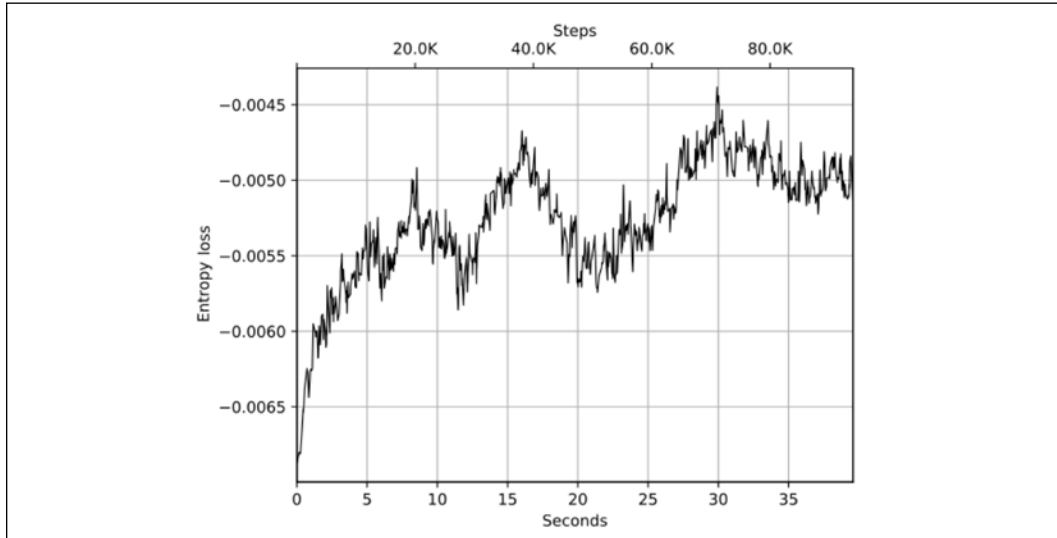
Figure 11.8: Entropy loss
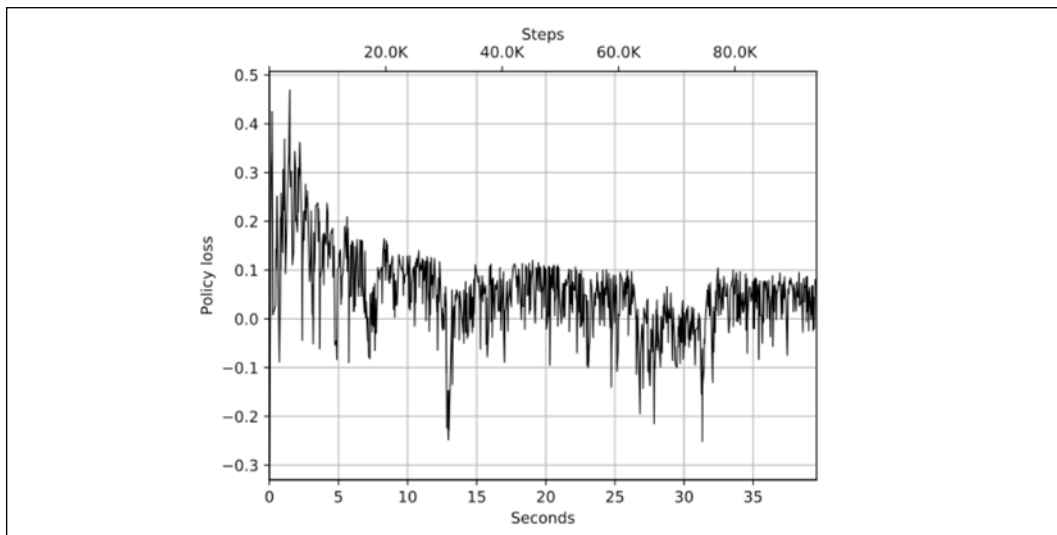
Figure 11.9: Policy loss
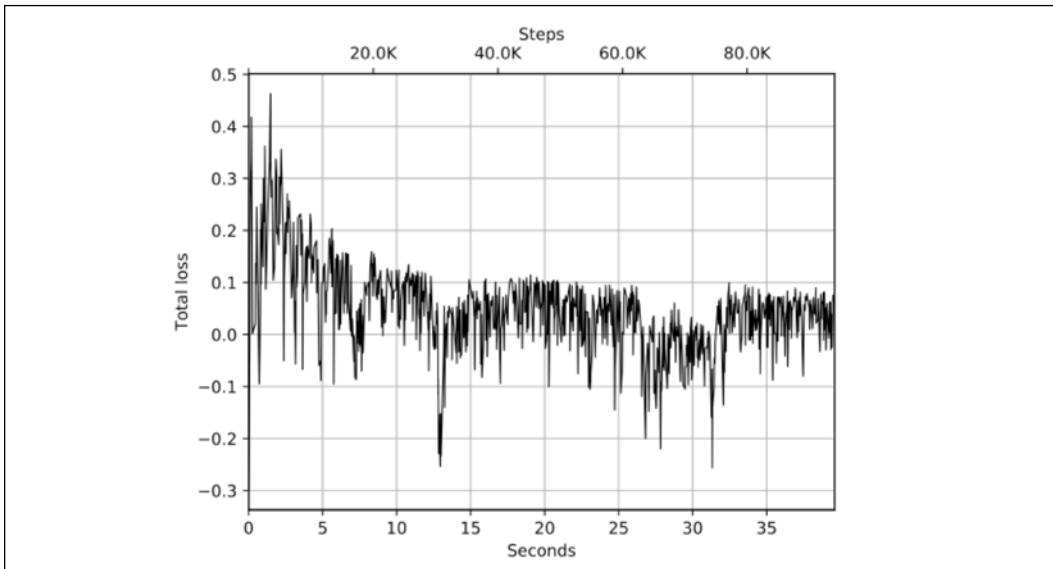
Figure 11.10: Total loss

The final set of charts shows the gradient's L2 values, maximum, and KL. Our gradients look healthy during the whole training: they are not too large and not too small, and there are no huge spikes. The KL charts also look normal, as there are some spikes, but they don't exceed 1e-3.
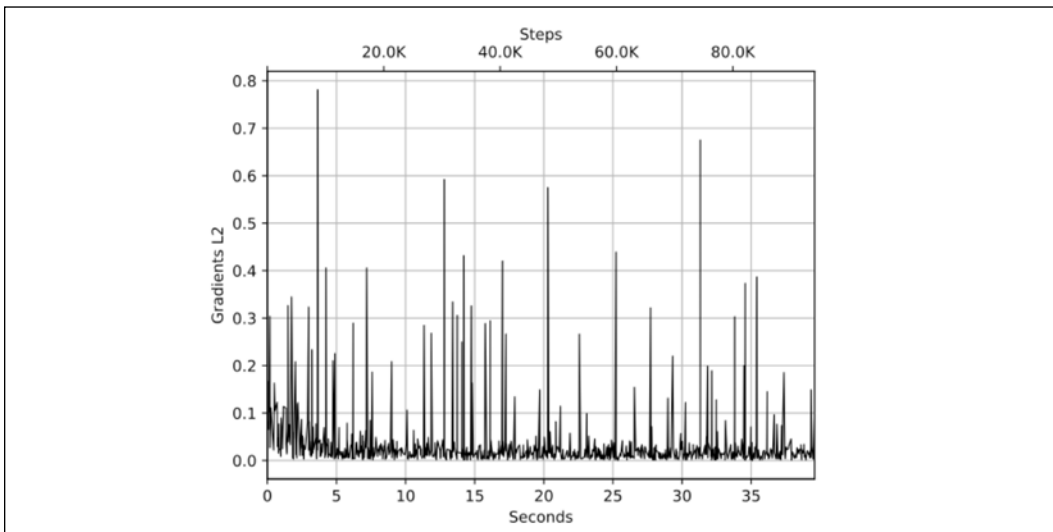
Figure 11.11: The L2 values of gradients during the training
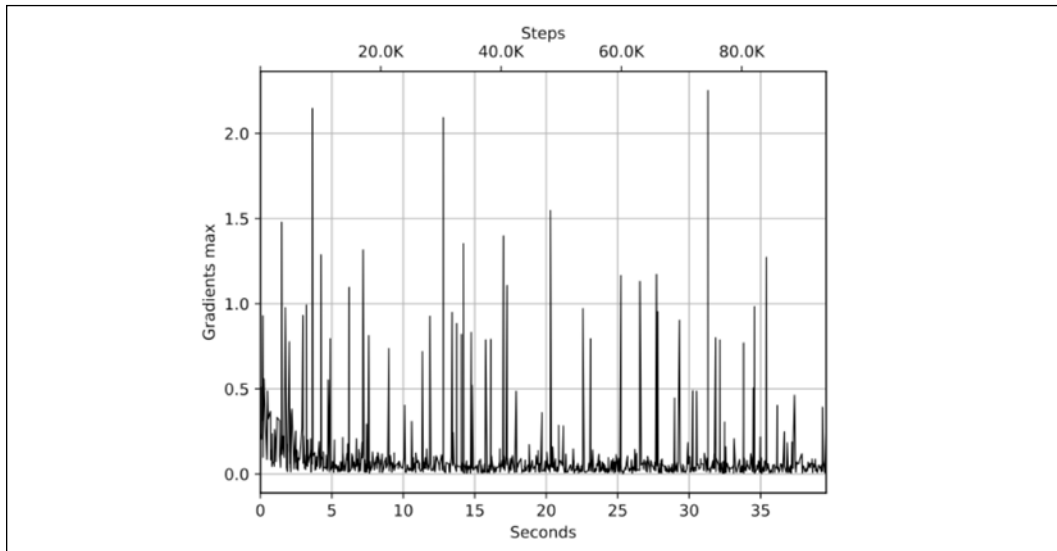
Figure 11.12: The maximum of gradients
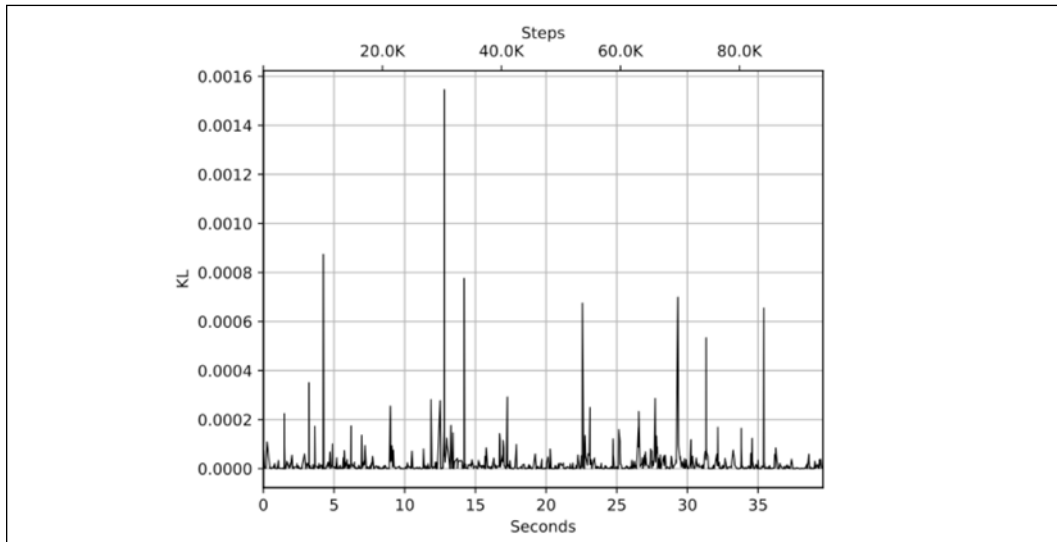
Figure 11.13: The KL of policy updates

Policy gradient methods on Pong

As we covered in the previous section, the vanilla policy gradient method works well on a simple CartPole environment, but it works surprisingly badly on more complicated environments.

For the relatively simple Atari game Pong, our DQN was able to completely solve it in 1M frames and showed positive reward dynamics in just 100k frames, whereas the policy gradient method failed to converge. Due to the instability of policy gradient training, it became very hard to find good hyperparameters and was still very sensitive to initialization.

This doesn't mean that the policy gradient method is bad, because, as you will see in the next chapter, just one tweak of the network architecture to get a better baseline in the gradients will turn the policy gradient method into one of the best methods (the asynchronous advantage actor-critic method). Of course, there is a good chance that my hyperparameters are completely wrong or the code has some hidden bugs or whatever. Regardless, unsuccessful results still have value, at least as a demonstration of bad convergence dynamics.

Implementation

The complete code of the example is in `Chapter11/05_pong_pg.py`.

The three main differences from the previous example's code are as follows:

- The baseline is estimated with a moving average for 1M past transitions, instead of all examples
- Several concurrent environments are used
- Gradients are clipped to improve training stability

To make moving average calculations faster, a `deque`-backed buffer is created:

```python
class MeanBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.deque = collections.deque(maxlen=capacity)
        self.sum = 0.0

    def add(self, val):
        if len(self.deque) == self.capacity:
            self.sum -= self.deque[0]
        self.deque.append(val)
        self.sum += val

    def mean(self):
        if not self.deque:
            return 0.0
        return self.sum / len(self.deque)
```

The second difference in this example is working with multiple environments, and this functionality is supported by the PTAN library. The only action we have to take is to pass the array of `Env` objects to the `ExperienceSource` class. All the rest is done automatically. In the case of several environments, the experience source asks them for transitions in round-robin, providing us with less-correlated training samples. The last difference from the CartPole example is gradient clipping, which is performed using the PyTorch `clip_grad_norm` function from the `torch.nn.utils` package.

The hyperparameters for the best variant are the following:

```
GAMMA = 0.99
LEARNING_RATE = 0.0001
ENTROPY_BETA = 0.01
BATCH_SIZE = 128

REWARD_STEPS = 10
BASELINE_STEPS = 1000000
GRAD_L2_CLIP = 0.1

ENV_COUNT = 32
```

Results

Okay, let's look at one of the best runs of the example. The following are the reward charts. You can see that during the training, the rewards were almost constant for a while, and then some growth started, which was interrupted by flat regions of minimal reward: 21.
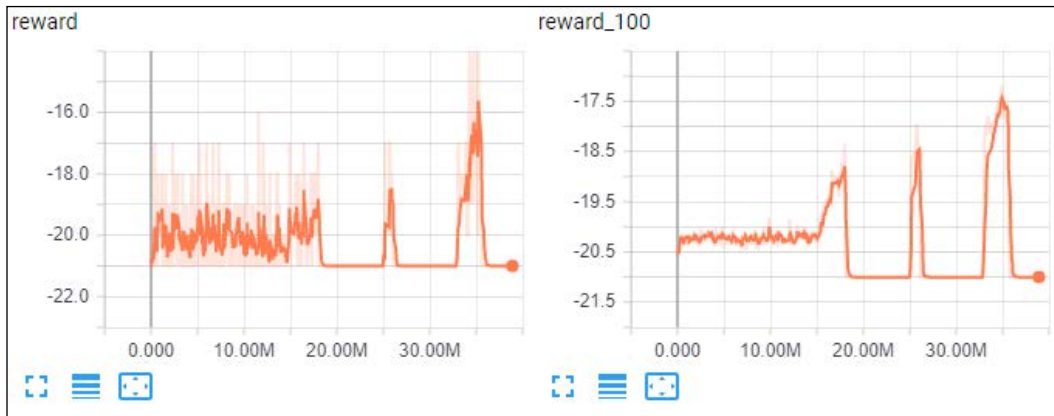
Figure 11.14: Gradients and KL divergence

On the entropy plot, you can see that those flat regions correspond to periods when entropy was zero, which means that our agent had 100% certainty in its actions. During this time interval, gradients were zero, too, so it's quite surprising that our training process was able to recover from those flat regions.
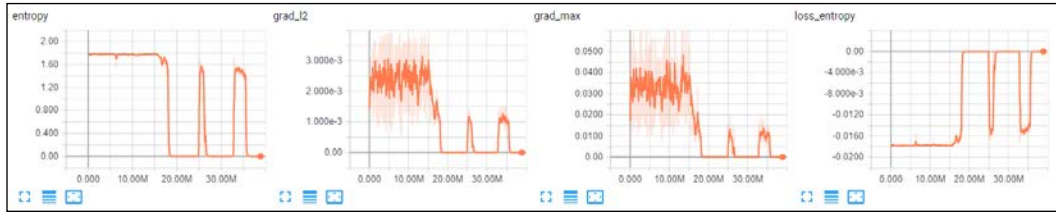
Figure 11.15: Another set of plots for Pong, using the policy gradient method

The chart with the baseline mostly follows the reward and has the same patterns.
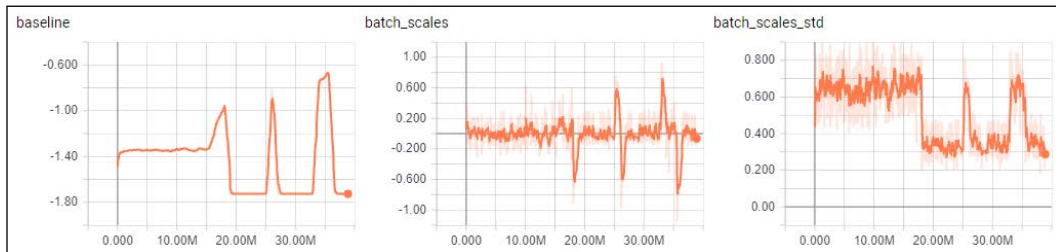
Figure 11.16: Baseline, scales, and standard deviation of scales

The KL plot has large spikes roughly at the moments to and from the zero-entropy transitions, which shows that the policy suffered from heavy jumps in returning distributions.

Figure 11.17: KL divergence and losses during training

Summary

In this chapter, you saw an alternative way of solving RL problems: policy gradient methods, which are different in many ways from the familiar DQN method. We explored a basic method called REINFORCE, which is a generalization of our first method in RL-domain cross-entropy. This policy gradient method is simple, but when applied to the Pong environment, it didn't show good results.

In the next chapter, we will consider ways to improve the stability of policy gradient methods by combining both families of value-based and policy-based methods.

12

The Actor-Critic Method

In *Chapter 11*, *Policy Gradients – an Alternative*, we started to investigate a policy-based alternative to the familiar value-based methods family. In particular, we focused on the method called REINFORCE and its modification, which uses discounted reward to obtain the gradient of the policy (which gives us the direction in which to improve the policy). Both methods worked well for a small CartPole problem, but for a more complicated Pong environment, the convergence dynamics were painfully slow.

Next, we will discuss another extension to the vanilla policy gradient method, which magically improves the stability and convergence speed of that method. Despite the modification being only minor, the new method has its own name, **actor-critic**, and it's one of the most powerful methods in deep reinforcement learning (RL).

In this chapter, we will:

- Explore how the baseline impacts statistics and the convergence of gradients
- Cover an extension of the baseline idea

Variance reduction

In the previous chapter, I briefly mentioned that one of the ways to improve the stability of policy gradient methods is to reduce the variance of the gradient. Now let's try to understand why this is important and what it means to reduce the variance. In statistics, variance is the expected square deviation of a random variable from the expected value of that variable.

$$\text{Var}[x] = \mathbb{E}[(x - \mathbb{E}[x])^2]$$

Variance shows us how far values are dispersed from the mean. When variance is high, the random variable can take values that deviate widely from the mean. On the following plot, there is a normal (Gaussian) distribution with the same value for the mean, $\mu = 10$, but with different values for the variance.
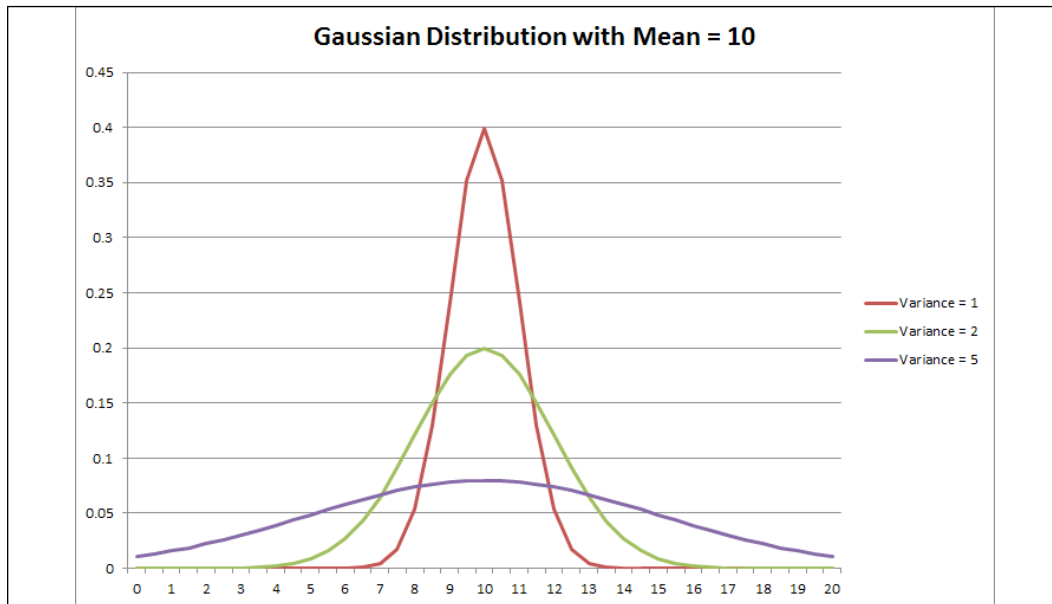
Figure 12.1: The effect of variance on Gaussian distribution

Now let's return to policy gradients. It was stated in the previous chapter that the idea is to increase the probability of good actions and decrease the chance of bad ones. In math notation, our policy gradient was written as $\nabla J \approx \mathbb{E}[Q(s, a)\nabla \log \pi(a|s)]$. The scaling factor $Q(s, a)$ specifies how much we want to increase or decrease the probability of the action taken in the particular state. In the REINFORCE method, we used the discounted total reward as the scaling of the gradient. In an attempt to increase REINFORCE stability, we subtracted the mean reward from the gradient scale. To understand why this helped, let's consider the very simple scenario of an optimization step on which we have three actions with different total discounted rewards: Q_1, Q_2, and Q_3. Now let's consider the policy gradient with regard to the relative values of those Q_s.

As the first example, let both Q_1 and Q_2 be equal to some small positive number and Q_3 be a large negative number. So, actions at the first and second steps led to some small reward, but the third step was not very successful. The resulting **combined** gradient for all three steps will try to push our policy far from the action at step three and slightly toward the actions taken at steps one and two, which is a totally reasonable thing to do.

Now let's imagine that our reward is always positive and only the value is different. This corresponds to adding some constant to each of the rewards: Q_1, Q_2, and Q_3. In this case, Q_1 and Q_2 will become large positive numbers and Q_3 will have a small positive value. However, our policy update will become different! Next, we will try hard to push our policy toward actions at the first and second steps, and slightly push it toward an action at step three. So, strictly speaking, we are no longer trying to avoid the action taken for step three, despite the fact that the relative rewards are the same.

This dependency of our policy update on the constant added to the reward can slow down our training significantly, as we may require many more samples to *average out* the effect of such a shift in the policy gradient. Even worse, as our total discounted reward changes over time, with the agent learning how to act better and better, our policy gradient variance can also change. For example, in the Atari Pong environment, the average reward in the beginning is –21...–20, so all the actions look almost equally bad.

To overcome this in the previous chapter, we subtracted the mean total reward from the Q-value and called this mean **baseline**. This trick normalized our policy gradient: in the case of the average reward being –21, getting a reward of –20 looks like a win for the agent and it pushes its policy toward the taken actions.

CartPole variance

To check this theoretical conclusion in practice, let's plot our policy gradient variance during the training for both the baseline version and the version without the baseline. The complete example is in `Chapter12/01_cartpole_pg.py`, and most of the code is the same as in *Chapter 11*, *Policy Gradients – an Alternative*. The differences in this version are the following:

- It now accepts the command-line option `--baseline`, which enables the mean subtraction from the reward. By default, no baseline is used.

- On every training loop, we gather the gradients from the policy loss and use this data to calculate the variance.

To gather only the gradients from the policy loss and exclude the gradients from the entropy bonus added for exploration, we need to calculate the gradients in two stages. Luckily, PyTorch allows this to be done easily. In the following code, only the relevant part of the training loop is included to illustrate the idea:

```
optimizer.zero_grad()
logits_v = net(states_v)
log_prob_v = F.log_softmax(logits_v, dim=1)
log_p_a_v = log_prob_v[range(BATCH_SIZE), batch_actions_t]
log_prob_actions_v = batch_scale_v * log_p_a_v
loss_policy_v = -log_prob_actions_v.mean()
```

We calculate the policy loss as before, by calculating the log from the probabilities of taken actions and multiplying it by policy scales (which are the total discounted reward if we are not using the baseline or the total reward minus the baseline):

```
loss_policy_v.backward(retain_graph=True)
```

In the next step, we ask PyTorch to backpropagate the policy loss, calculating the gradients and keeping them in our model's buffers. As we have previously performed `optimizer.zero_grad()`, those buffers will contain only the gradients from the policy loss. One tricky thing here is the `retain_graph=True` option when we call `backward()`. It instructs PyTorch to keep the graph structure of the variables. Normally, this is destroyed by the `backward()` call, but in our case, this is not what we want. In general, retaining the graph could be useful when we need to backpropagate the loss multiple times before the call to the optimizer, although this is not a very common situation.

```
grads = np.concatenate([p.grad.data.numpy().flatten()
                        for p in net.parameters()
                        if p.grad is not None])
```

Then, we iterate all parameters from our model (every parameter of our model is a tensor with gradients) and extract their `grad` field in a flattened NumPy array. This gives us one long array with all gradients from our model's variables. However, our parameter update should take into account not only the policy gradient but also the gradient provided by our entropy bonus. To achieve this, we calculate the entropy loss and call `backward()` again. To be able to do this the second time, we need to pass `retain_graph=True`.

On the second `backward()` call, PyTorch will backpropagate our entropy loss and add the gradients to the internal gradients' buffers. So, what we now need to do is just ask our optimizer to perform the optimization step using those combined gradients:

```
prob_v = F.softmax(logits_v, dim=1)
entropy_v = -(prob_v * log_prob_v).sum(dim=1).mean()
entropy_loss_v = -ENTROPY_BETA * entropy_v
entropy_loss_v.backward()
optimizer.step()
```

Later, the only thing we need to do is write statistics that we are interested in into TensorBoard:

```
g_l2 = np.sqrt(np.mean(np.square(grads)))
g_max = np.max(np.abs(grads))
writer.add_scalar("grad_l2", g_l2, step_idx)
writer.add_scalar("grad_max", g_max, step_idx)
writer.add_scalar("grad_var", np.var(grads), step_idx)
```

By running this example twice, once with the `--baseline` command-line option and once without it, we get a plot of variance of our policy gradient. The following are the charts for the reward dynamics:
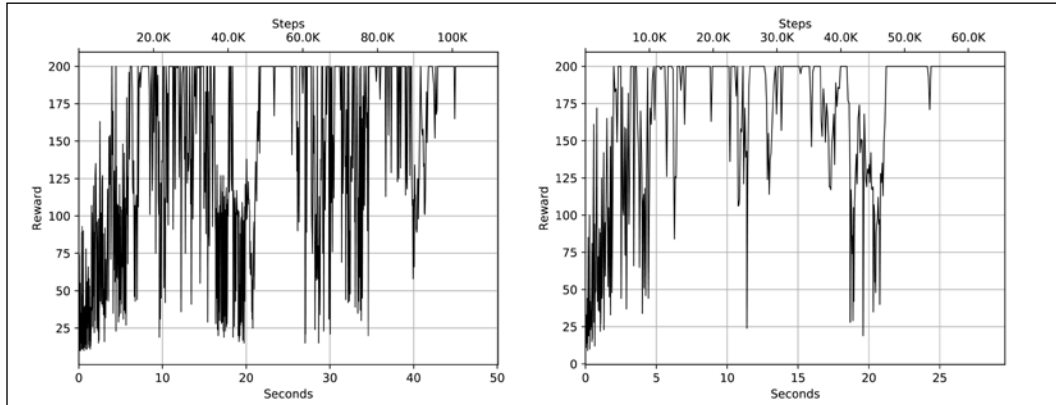
Figure 12.2: The rewards of training episodes without the baseline (left) and with the baseline (right)
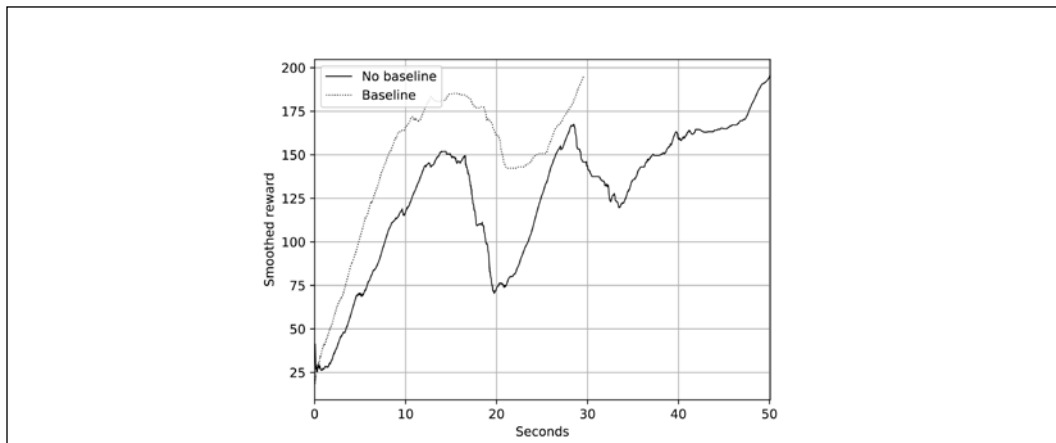
Figure 12.3: The smoothed rewards for both versions

The following three charts show the gradients' magnitude, maximum value, and variance:
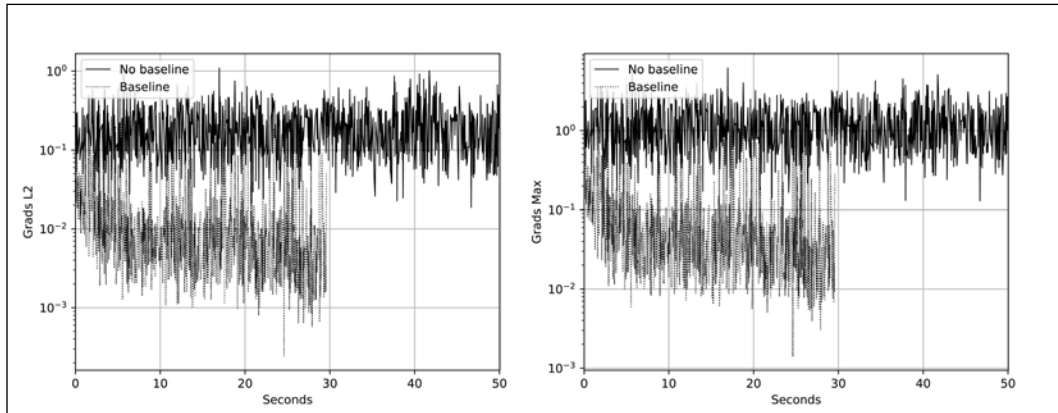
Figure 12.4: Gradient L2 (left) and maximum (right) for both versions
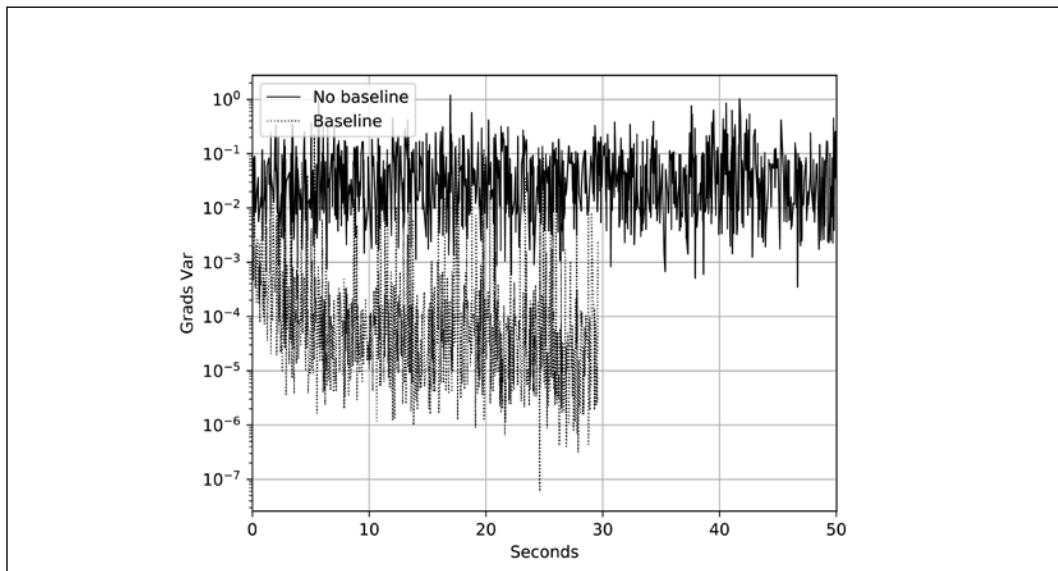
Figure 12.5: Gradients variance for both versions

As you can see, variance for the version with the baseline is two-to-three orders of magnitude lower than the version without one, which helps the system to converge faster.

Actor-critic

The next step in reducing the variance is making our baseline state-dependent (which is a good idea, as different states could have very different baselines). Indeed, to decide on the suitability of a particular action in some state, we use the discounted total reward of the action. However, the total reward itself could be represented as a *value* of the state plus the *advantage* of the action: $Q(s, a) = V(s) + A(s, a)$. You saw this in *Chapter 8*, *DQN Extensions*, when we discussed DQN modifications, particularly dueling DQN.

So, why can't we use $V(s)$ as a baseline? In that case, the scale of our gradient will be just advantage, $A(s, a)$, showing how this taken action is better in respect to the average state's value. In fact, we can do this, and it is a very good idea for improving the policy gradient method. The only problem here is that we don't know the value, $V(s)$, of the state that we need to subtract from the discounted total reward, $Q(s, a)$. To solve this, let's use *another neural network*, which will approximate $V(s)$ for every observation. To train it, we can exploit the same training procedure we used in DQN methods: we will carry out the Bellman step and then minimize the mean square error to improve $V(s)$ approximation.

When we know the value for any state (or at least have some approximation of it), we can use it to calculate the policy gradient and update our policy network to increase probabilities for actions with good advantage values and decrease the chance of actions with bad advantage values. The policy network (which returns a probability distribution of actions) is called the *actor*, as it tells us what to do. Another network is called *critic*, as it allows us to understand how good our actions were. This improvement is known under a separate name, the advantage actor-critic method, which is often abbreviated to A2C. The following is an illustration of its architecture:
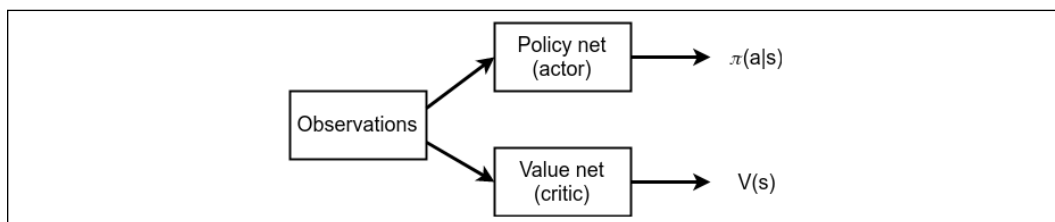
Figure 12.6: The A2C architecture

In practice, the policy and value networks partially overlap, mostly due to efficiency and convergence considerations. In this case, the policy and value are implemented as different heads of the network, taking the output from the common body and transforming it into the probability distribution and a single number representing the value of the state.

This helps both networks to share low-level features (such as convolution filters in the Atari agent), but combine them in a different way. This architecture looks like the following:
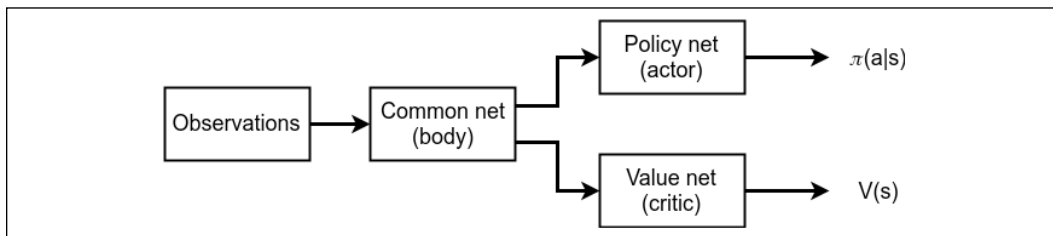
Figure 12.7: The A2C architecture with a shared network body

From a training point of view, we complete these steps:

1. Initialize network parameters, θ, with random values
2. Play N steps in the environment using the current policy, π_θ, and saving the state, s_t, action, a_t, and reward, r_t
3. $R = 0$ if the end of the episode is reached or $V_\theta(s_t)$
4. For $i = t - 1 \dots t_{start}$ (note that steps are processed backward):
   - $R \leftarrow r_i + \gamma R$
   - Accumulate the policy gradients: $\partial \theta_\pi \leftarrow \partial \theta_\pi + \nabla_\theta \log \pi_\theta(a_i|s_i)(R - V_\theta(s_i))$
   - Accumulate the value gradients: $\partial \theta_v \leftarrow \partial \theta_v + \dfrac{\partial (R - V_\theta(s_i))^2}{\partial \theta_v}$
5. Update the network parameters using the accumulated gradients, moving in the direction of the policy gradients, $\partial \theta_\pi$, and in the opposite direction of the value gradients, $\partial \theta_v$
6. Repeat from step 2 until convergence is reached

The preceding algorithm is an outline and similar to those that are usually printed in research papers. In practice, some considerations are as follows:

- An entropy bonus is usually added to improve exploration. It's typically written as an entropy value added to the loss function:

  $\mathcal{L}_H = \beta \sum_i \pi_\theta(s_i) \log \pi_\theta(s_i)$. This function has a minimum when the

  probability distribution is uniform, so by adding it to the loss function, we push our agent away from being too certain about its actions.

- Gradients accumulation is usually implemented as a loss function combining all three components: policy loss, value loss, and entropy loss. You should be careful with signs of these losses, as policy gradients show you the direction of policy improvement, but both value and entropy losses should be minimized.

- To improve stability, it's worth using several environments, providing you with observations concurrently (when you have multiple environments, your training batch will be created from their observations). We will look at several ways of doing this in the next chapter.

The version of the preceding method that uses several environments running in parallel is called advantage asynchronous actor-critic, which is also known as A3C. The A3C method will be the subject of the next chapter, but for now, let's implement A2C.

A2C on Pong

In the previous chapter, you saw a (not very successful) attempt to solve our favorite Pong environment with policy gradient methods. Let's try it again with the actor-critic method at hand.

```
GAMMA = 0.99
LEARNING_RATE = 0.001
ENTROPY_BETA = 0.01
BATCH_SIZE = 128
NUM_ENVS = 50

REWARD_STEPS = 4
CLIP_GRAD = 0.1
```

We start, as usual, by defining hyperparameters (imports are omitted). These values are not tuned, as we will do this in the next section of this chapter. We have one new value here: CLIP_GRAD. This hyperparameter specifies the threshold for gradient clipping, which basically prevents our gradients from becoming too large at the optimization stage and pushing our policy too far. Clipping is implemented using the PyTorch functionality, but the idea is very simple: if the L2 norm of the gradient is larger than this hyperparameter, then the gradient vector is clipped to this value.

The REWARD_STEPS hyperparameter determines how many steps ahead we will take to approximate the total discounted reward for every action.

In the policy gradient methods, we used about 10 steps, but in A2C, we will use our value approximation to get a state value for further steps, so it will be fine to decrease the number of steps.

```python
class AtariA2C(nn.Module):
    def __init__(self, input_shape, n_actions):
        super(AtariA2C, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )
        conv_out_size = self._get_conv_out(input_shape)
        self.policy = nn.Sequential(
            nn.Linear(conv_out_size, 512),
            nn.ReLU(),
            nn.Linear(512, n_actions)
        )

        self.value = nn.Sequential(
            nn.Linear(conv_out_size, 512),
            nn.ReLU(),
            nn.Linear(512, 1)
        )
```

Our network architecture has a shared convolution body and two heads: the first returns the policy with the probability distribution over our actions and the second head returns one single number, which will approximate the state's value. It might look similar to our dueling DQN architecture from *Chapter 8, DQN Extensions*, but our training procedure is different.

```python
    def _get_conv_out(self, shape):
        o = self.conv(torch.zeros(1, *shape))
        return int(np.prod(o.size()))

    def forward(self, x):
        fx = x.float() / 256
        conv_out = self.conv(fx).view(fx.size()[0], -1)
        return self.policy(conv_out), self.value(conv_out)
```

The forward pass through the network returns a tuple of two tensors: `policy` and `value`. Now we have a large and important function, which takes the batch of environment transitions and returns three tensors: the batch of states, batch of actions taken, and batch of Q-values calculated using the formula $Q(s, a) = \sum_{i=0}^{N-1} \gamma^i r_i + \gamma^N V(s_N)$. This Q-value will be used in two places: to calculate **mean squared error** (**MSE**) loss to improve the value approximation in the same way as DQN, and to calculate the advantage of the action.

```python
def unpack_batch(batch, net, device='cpu'):
    states = []
    actions = []
    rewards = []
    not_done_idx = []
    last_states = []
    for idx, exp in enumerate(batch):
        states.append(np.array(exp.state, copy=False))
        actions.append(int(exp.action))
        rewards.append(exp.reward)
        if exp.last_state is not None:
            not_done_idx.append(idx)
            last_states.append(
                np.array(exp.last_state, copy=False))
```

In the first loop, we just walk through our batch of transitions and copy their fields into the lists. Note that the reward value already contains the discounted reward for `REWARD_STEPS`, as we use the `ptan.ExperienceSourceFirstLast` class. We also need to handle episode-ending situations and remember indices of batch entries for non-terminal episodes.

```python
        states_v = torch.FloatTensor(
            np.array(states, copy=False)).to(device)
        actions_t = torch.LongTensor(actions).to(device)
```

In the preceding code, we convert the gathered state and actions into a PyTorch tensor and copy them into the graphics processing unit (GPU) if needed. The extra call to `np.array()` might look redundant, but without it, the performance of tensor creation degrades 5-10x. This issue in PyTorch (`https://github.com/pytorch/pytorch/issues/13918`) hasn't been solved yet, so one solution is to pass a single NumPy array instead of a list of arrays.

The rest of the function calculates Q-values, taking into account the terminal episodes:

```
rewards_np = np.array(rewards, dtype=np.float32)
if not_done_idx:
    last_states_v = torch.FloatTensor(
        np.array(last_states, copy=False)).to(device)
    last_vals_v = net(last_states_v)[1]
    last_vals_np = last_vals_v.data.cpu().numpy()[:, 0]
    last_vals_np *= GAMMA ** REWARD_STEPS
    rewards_np[not_done_idx] += last_vals_np
```

The preceding code prepares the variable with the last state in our transition chain and queries our network for *V(s)* approximation. Then, this value is multiplied by the discount factor and added to the immediate rewards.

```
ref_vals_v = torch.FloatTensor(rewards_np).to(device)
return states_v, actions_t, ref_vals_v
```

At the end of the function, we pack our Q-values into the appropriate form and return it.

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--cuda", default=False,
                        action="store_true", help="Enable cuda")
    parser.add_argument("-n", "--name", required=True,
                        help="Name of the run")
    args = parser.parse_args()
    device = torch.device("cuda" if args.cuda else "cpu")

    make_env = lambda: ptan.common.wrappers.wrap_dqn(
        gym.make("PongNoFrameskip-v4"))
    envs = [make_env() for _ in range(NUM_ENVS)]
    writer = SummaryWriter(comment="-pong-a2c_" + args.name)
```

The preparation code for the training loop is the same as usual, except that we now use the array of environments to gather experience, instead of one environment.

```
    net = AtariA2C(envs[0].observation_space.shape,
                   envs[0].action_space.n).to(device)
    print(net)

    agent = ptan.agent.PolicyAgent(
        lambda x: net(x)[0], apply_softmax=True, device=device)
    exp_source = ptan.experience.ExperienceSourceFirstLast(
        envs, agent, gamma=GAMMA, steps_count=REWARD_STEPS)
    optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE,
                           eps=1e-3)
```

One very important detail here is passing the `eps` parameter to the optimizer. If you're familiar with the **Adam** algorithm, you may know that epsilon is a small number added to the denominator to prevent zero division situations. Normally, this value is set to some small number such as 1e-8 or 1e-10, but in our case, these values turned out to be too small. I have no mathematically strict explanation for this, but with the default value of epsilon, the method does not converge at all. Very likely, the division to a small value of 1e-8 makes the gradients too large, which turns out to be fatal for training stability.

```
batch = []

with common.RewardTracker(writer, stop_reward=18) as tracker:
    with ptan.common.utils.TBMeanTracker(writer,
            batch_size=10) as tb_tracker:
        for step_idx, exp in enumerate(exp_source):
            batch.append(exp)

            new_rewards = exp_source.pop_total_rewards()
            if new_rewards:
                if tracker.reward(new_rewards[0], step_idx):
                    break

            if len(batch) < BATCH_SIZE:
                continue
```

In the training loop, we use two wrappers. The first is already familiar to you: `common.RewardTracker`, which computes the mean reward for the last 100 episodes and tells us when this mean reward exceeds the desired threshold. Another wrapper, `TBMeanTracker`, is from the PTAN library and is responsible for writing into TensorBoard the mean of the measured parameters for the last 10 steps. This is helpful, as training can take millions of steps and we don't want to write millions of points into TensorBoard, but rather write smoothed values every 10 steps. The next code chunk is responsible for our calculation of losses, which is the core of the A2C method.

```
states_v, actions_t, vals_ref_v = \
    unpack_batch(batch, net, device=device)
batch.clear()
optimizer.zero_grad()
logits_v, value_v = net(states_v)
```

In the beginning, we unpack our batch using the function we described earlier and ask our network to return the policy and values for this batch. The policy is returned in unnormalized form, so to convert it into the probability distribution, we need to apply softmax to it.

We postpone this step to use `log_softmax`, as it is more numerically stable.

```
loss_value_v = F.mse_loss(
    value_v.squeeze(-1), vals_ref_v)
```

The value loss part is almost trivial: we just calculate the MSE between the value returned by our network and the approximation we performed using the Bellman equation unrolled four steps forward.

```
log_prob_v = F.log_softmax(logits_v, dim=1)
adv_v = vals_ref_v - value_v.detach()
log_p_a = log_prob_v[range(BATCH_SIZE), actions_t]
log_prob_actions_v = adv_v * log_p_a
loss_policy_v = -log_prob_actions_v.mean()
```

Here, we calculate the policy loss to obtain the policy gradient. The first two steps obtain a log of our policy and calculate the advantage of actions, which is $A(s, a) = Q(s, a) - V(s)$. The call to `value_v.detach()` is important, as we don't want to propagate the policy gradient into our value approximation head. Then, we take the log of probability for the actions taken and scale them with advantage. Our policy gradient loss value will be equal to the negated mean of this scaled log of policy, as the policy gradient directs us toward policy improvement, but loss value is supposed to be minimized.

```
prob_v = F.softmax(logits_v, dim=1)
ent = (prob_v * log_prob_v).sum(dim=1).mean()
entropy_loss_v = ENTROPY_BETA * ent
```

The last piece of our loss function is entropy loss, which is equal to the scaled entropy of our policy, taken with the opposite sign (entropy is calculated as $H(\pi) = -\sum \pi \log \pi$).

```
loss_policy_v.backward(retain_graph=True)
grads = np.concatenate([
    p.grad.data.cpu().numpy().flatten()
    for p in net.parameters()
    if p.grad is not None
])
```

In the preceding code, we calculate and extract gradients of our policy, which will be used to track the maximum gradient, its variance, and the L2 norm.

```
loss_v = entropy_loss_v + loss_value_v
loss_v.backward()
nn_utils.clip_grad_norm_(net.parameters(),
                         CLIP_GRAD)
```

```
                optimizer.step()
                loss_v += loss_policy_v
```

As the final step of our training, we backpropagate the entropy loss and the value loss, clip gradients, and ask our optimizer to update the network.

```
                tb_tracker.track("advantage", adv_v, step_idx)
                tb_tracker.track("values", value_v, step_idx)
                tb_tracker.track("batch_rewards", vals_ref_v,
                                 step_idx)
                tb_tracker.track("loss_entropy", entropy_loss_v,
                                 step_idx)
                tb_tracker.track("loss_policy", loss_policy_v,
                                 step_idx)
                tb_tracker.track("loss_value", loss_value_v,
                                 step_idx)
                tb_tracker.track("loss_total", loss_v, step_idx)
                g_l2 = np.sqrt(np.mean(np.square(grads)))
                tb_tracker.track("grad_l2", g_l2, step_idx)
                g_max = np.max(np.abs(grads))
                tb_tracker.track("grad_max", g_max, step_idx)
                g_var = np.var(grads)
                tb_tracker.track("grad_var", g_var, step_idx)
```

At the end of the training loop, we track all of the values that we are going to monitor in TensorBoard. There are plenty of them and we will discuss them in the next section.

A2C on Pong results

To start the training, run `02_pong_a2c.py` with the `--cuda` and `-n` options (which provides a name for the run for TensorBoard):

```
rl_book_samples/Chapter10$ ./02_pong_a2c.py --cuda -n t2
AtariA2C (
  (conv): Sequential (
    (0): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU ()
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU ()
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (5): ReLU ()
  )
```

```
    (policy): Sequential (
      (0): Linear (3136 -> 512)
      (1): ReLU ()
      (2): Linear (512 -> 6)
    )
    (value): Sequential (
      (0): Linear (3136 -> 512)
      (1): ReLU ()
      (2): Linear (512 -> 1)
    )
  )
  37799: done 1 games, mean reward -21.000, speed 722.89 f/s
  39065: done 2 games, mean reward -21.000, speed 749.92 f/s
  39076: done 3 games, mean reward -21.000, speed 755.26 f/s
  ...
```

As a word of warning, the training process is lengthy. With the original hyperparameters, it requires more than 8 million frames to solve, which is approximately three hours on a GPU. In the next section of the chapter, we will tweak the parameters to improve the convergence speed, but for now, it's three hours. To improve the situation even more, in the next chapter, we will look at the distributed version, which executes the environment in a separate process, but first, let's focus on our plots in TensorBoard.

First of all, the reward dynamics look much better than in the example from the previous chapter:
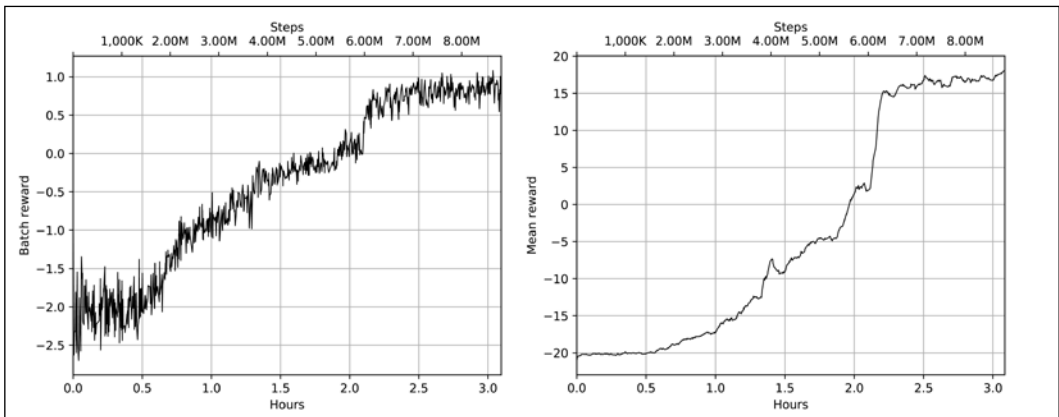
Figure 12.8: The mean batch value (left) and average training reward (right)

The first plot, **batch_rewards**, shows Q-values approximated using the Bellman equation and an overall positive dynamic in Q approximation. The next plot is the mean training episodes reward averaged over the 100 last episodes. This shows that our training process is improving more or less consistently over time.

The next four charts are related to our loss and include the individual loss components and the total loss.
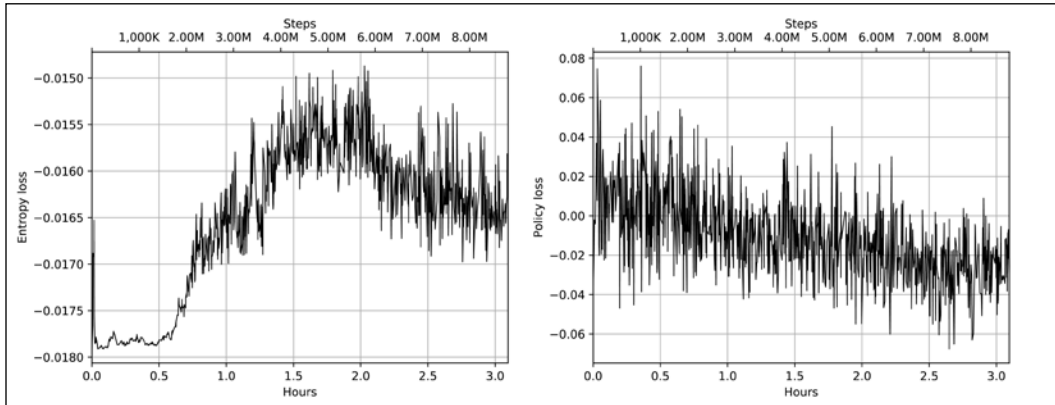
Figure 12.9: Entropy loss (left) and policy loss (right) during the training
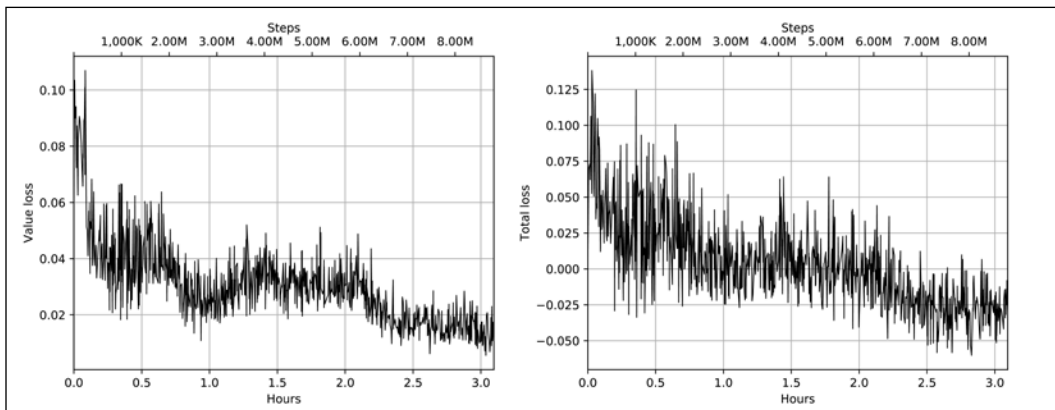
Figure 12.10: Value loss (left) and total loss (right)

Here, we can see various things. First of all, our value loss (*Figure 12.10*, on the left) is decreasing consistently, which shows that our $V(s)$ approximation is improving during the training. The second observation is that our entropy loss (*Figure 12.9*, on the left) is growing, but it doesn't dominate in the total loss. This basically means that our agent becomes more confident in its actions as the policy becomes less uniform.

The last thing to note here is that policy loss (*Figure 12.9*, on the right) is decreasing most of the time and is correlated to the total loss, which is good, as we are interested in the gradients for our policy first of all.

The last set of plots displays the advantage value and policy gradients metrics.
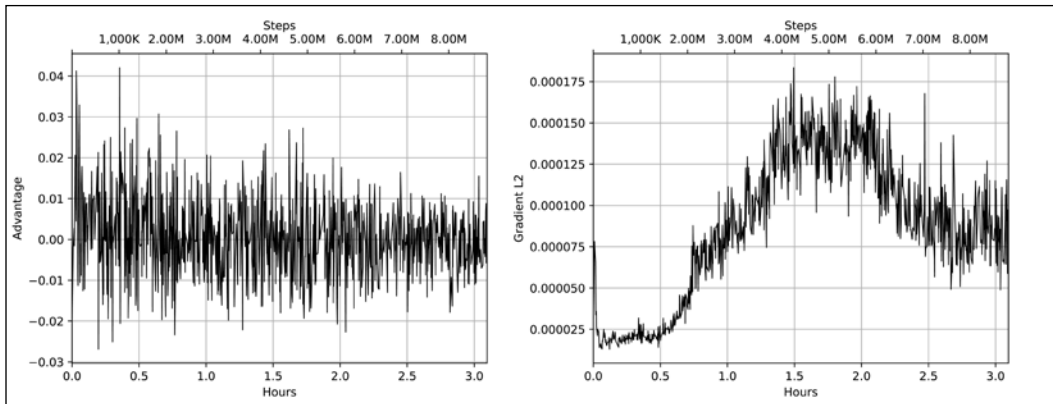
Figure 12.11: The advantage (left) and the gradient L2 magnitude (right) during the training
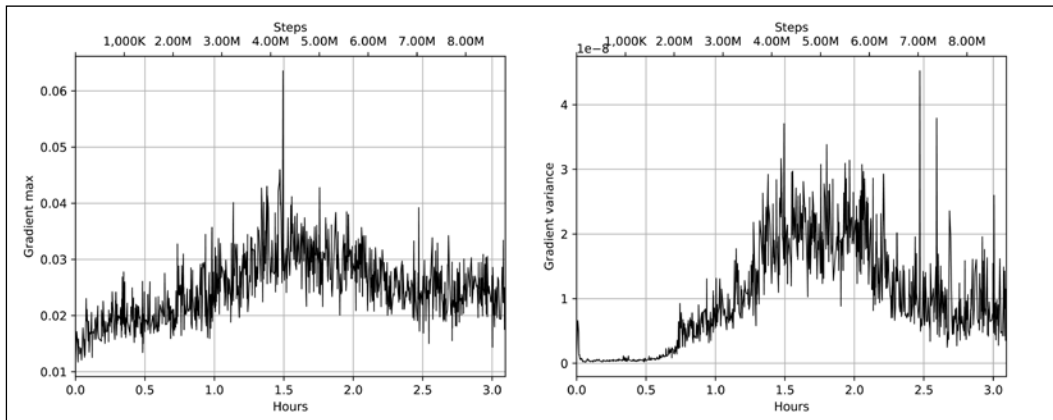
Figure 12.12: The maximum of gradients (left) and variance of gradients (right)

The advantage is a scale of our policy gradients, and it equals to $Q(s, a) - V(s)$. We expect it to oscillate around zero, and the chart meets our expectations. The gradient charts demonstrate that our gradients are not too small and not too large. Variance is very small at the beginning of the training (for 1.5 million frames), but starts to grow later, which means that our policy is changing.

Tuning hyperparameters

In the previous section, we had Pong solved in three hours of optimization and 9 million frames. Now is a good time to tweak our hyperparameters to speed up convergence. The golden rule here is to tweak one option at a time and make conclusions carefully, as the whole process is stochastic.

In this section, we will start with the original hyperparameters and perform the following experiments:

- Increase the learning rate
- Increase the entropy beta
- Change the count of environments that we are using to gather experience
- Tweak the size of the batch

Strictly speaking, the following experiments weren't proper hyperparameter tuning but just an attempt to get a better understanding of how A2C convergence dynamics depend on the parameters. To find the best set of parameters, the full grid search or random sampling of values could give much better results, but they would require much more time and resources.

Learning rate

Our starting **learning rate** (**LR**) is 0.001, and we expect that a larger LR will lead to faster convergence. This turned out to be true in my tests, but only to a certain extent: convergence speed increased up to 0.003, but for larger values, the system didn't converge at all.

The performance results are as follows:

- LR=0.002: 4.8 million frames, 1.5 hours
- LR=0.003: 3.6 million frames, 1 hour
- LR=0.004: hasn't converged
- LR=0.005: hasn't converged

The reward dynamics and the value loss are shown on the following charts. Larger values of LR led to lower value loss, which suggests that using two optimizers for policy and value heads (with different LRs) might lead to more stable learning.
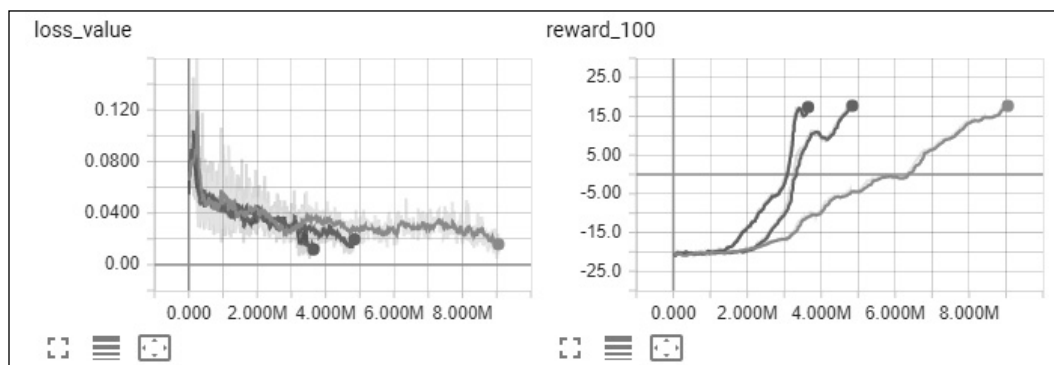
Figure 12.13: Experiments with different LRs (faster convergence corresponds to a large LR)

Entropy beta

I've tried two values for entropy loss scale: 0.02 and 0.03. The first value improved the speed, but the second made it worse again, so the optimal lies somewhere between them. The results were as follows:

- beta=0.02: 6.8 million frames, 2 hours
- beta=0.03: 12 million frames, 4 hours

Count of environments

It's not obvious what count of environments will work best, so I tried several counts both less and greater than our initial value of 50. The results were contradictory, but it appears that with more environments, we get faster convergence:

- Envs=40: 8.6 million frames, 3 hours
- Envs=30: 6.2 million frames, 2 hours (looks like a lucky seed)
- Envs=20: 9.5 million frames, 3 hours
- Envs=10: hasn't converged
- Envs=60: 11.6 million frames, 4 hours (looks like an unlucky seed)
- Envs=70: 7.7 million frames, 2.5 hours

Batch size

The experiments with batch size produced an unexpected result: a smaller batch size leads to faster convergence, but with a very small batch, the reward doesn't grow. This is logical from an RL point of view, as with smaller batches, we perform more frequent updates of the network and require fewer observations, but this is counterintuitive for deep learning, as a larger batch normally brings more training data:

- Batch=64: 4.9 million frames, 1.7 hours
- Batch=32: 3.8 million frames, 1.5 hours
- Batch=16, doesn't converge

Summary

In this chapter, you learned about one of the most widely used methods in deep RL: A2C, which wisely combines the policy gradient update with the value of the state approximation. We analyzed the effect of the baseline on the statistics and convergence of gradients. Then, we checked the extension of the baseline idea: A2C, where a separate network head provides us with the baseline for the current state.

In the next chapter, we will look at ways to perform the same algorithm in a distributed way.

13

Asynchronous Advantage Actor-Critic

This chapter is dedicated to the extension of the **advantage actor-critic** (**A2C**) method that we discussed in detail in *Chapter 12*, *The Actor-Critic Method*. The extension adds true asynchronous environment interaction, and its full name is **asynchronous advantage actor-critic**, which is normally abbreviated to **A3C**. This method is one of the most widely used by reinforcement learning (RL) practitioners.

We will take a look at two approaches for adding asynchronous behavior to the basic A2C method: data-level and gradient-level parallelism. They have different resource requirements and characteristics, which makes them applicable to different situations.

In this chapter, we will:

- Discuss why it is important for policy gradient methods to gather training data from multiple environments
- Implement two different approaches to A3C

Correlation and sample efficiency

One of the approaches to improving the stability of the policy gradient family of methods is using multiple environments in parallel. The reason behind this is the fundamental problem we discussed in *Chapter 6*, *Deep Q-Networks*, when we talked about the correlation between samples, which breaks the **independent and identically distributed** (**i.i.d.**) assumption, which is critical for **stochastic gradient descent** (**SGD**) optimization. The negative consequence of such correlation is very high variance in gradients, which means that our training batch contains very similar examples, all of them pushing our network in the same direction.

However, this may be totally the wrong direction in the global sense, as all those examples may be from one single lucky or unlucky episode.

With our **deep Q-network** (**DQN**), we solved the issue by storing a large number of previous states in the replay buffer and sampling our training batch from this buffer. If the buffer is large enough, the random sample from it will be a much better representation of the states' distribution at large. Unfortunately, this solution won't work for policy gradient methods, as most of them are on-policy, which means that we have to train on samples generated by our current policy, so *remembering old transitions* will not be possible anymore. You can try to do this, but the resulting policy gradient will be for the old policy used to generate the samples and not for your current policy that you want to update.

For several years, this issue was the focus for researchers. Several ways to address it were proposed, but the problem is still far from being solved. The most commonly used solution is gathering transitions using several parallel environments, all of them exploiting the current policy. This breaks the correlation within one single episode, as we now train on several episodes obtained from different environments. At the same time, we are still using our current policy. The one very large disadvantage of this is **sample inefficiency**, as we basically throw away all the experience that we have obtained after one single training.

It's very simple to compare DQN with policy gradient approaches. For example, for DQN, if we use 1 million samples of replay buffer and a training batch size of 32 samples for every new frame, every single transition will be used approximately 32 times before it is pushed from the experience replay. For the priority replay buffer, which was discussed in *Chapter 8*, *DQN Extensions*, this number could be much higher, as the sample probability is not uniform. In the case of policy gradient methods, each experience obtained from the environment can be used only once, as our method requires fresh data, so the data efficiency of policy gradient methods could be an order of magnitude lower than the value-based, off-policy methods.

On the other hand, our A2C agent converged on Pong in 8 million frames, which is just eight times more than 1 million frames for basic DQN in *Chapter 6* and *Chapter 8*. So, this shows us that policy gradient methods are not completely useless; they're just different and have their own specificities that you need to take into account on method selection. If your environment is cheap in terms of the agent interaction (the environment is fast, has a low memory footprint, allows parallelization, and so on), policy gradient methods could be a better choice. On the other hand, if the environment is expensive and obtaining a large amount of experience could slow down the training process, the value-based methods could be a smarter way to go.

Adding an extra A to A2C

From the practical point of view, communicating with several parallel environments is simple. We already did this in the previous chapter, but it wasn't explicitly stated. In the A2C agent, we passed an array of Gym environments into the `ExperienceSource` class, which switched it into round-robin data gathering mode. This means that every time we ask for a transition from the experience source, the class uses the next environment from our array (of course, keeping the state for every environment). This simple approach is equivalent to parallel communication with environments, but with one single difference: communication is not parallel in the strict sense but performed in a serial way. However, samples from our experience source are shuffled. This idea is shown in the following diagram:
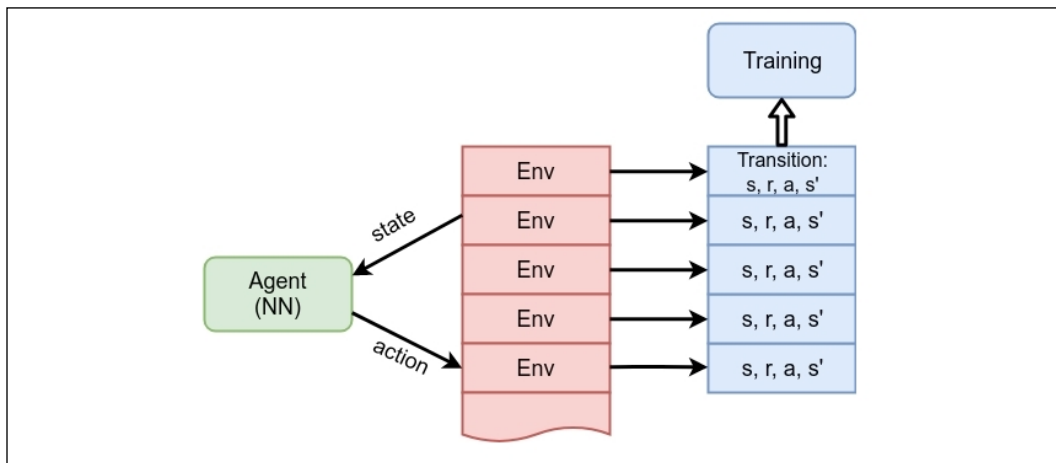
Figure 13.1: An agent training from multiple environments in parallel

This method works fine and helped us to get convergence in the A2C method, but it is still not perfect in terms of computing resource utilization. Even the modest workstation nowadays has several central processing unit (CPU) cores, which can be used for computation, such as training and environment interaction. On the other hand, parallel programming is harder than the traditional paradigm, when you have a clear stream of execution. Luckily, Python is a very expressive and flexible language with lots of third-party libraries, which allows you to do parallel programming without much trouble.

Another piece of good news is that PyTorch natively supports parallel programming in its `torch.multiprocessing` module. Parallel and distributed programming is a very wide topic, which is far beyond the scope of this book. In this chapter, we will just scratch the surface of the large domain of parallelization, but there is much more to learn.

With regard to actor-critic parallelization, two approaches exist:

1. **Data parallelism**: We can have several processes, each of them communicating with one or more environments, and providing us with transitions (s, r, a, s'). All those samples are gathered together in one single training process, which calculates losses and performs an SGD update. Then, the updated neural network (NN) parameters need to be broadcast to all other processes to use in future environment communications.

2. **Gradients parallelism**: As the goal of the training process is the calculation of gradients to update our NN, we can have several processes calculating gradients on their own training samples. Then, these gradients can be summed together to perform the SGD update in one process. Of course, updated NN weights also need to be propagated to all workers to keep data on-policy.

Both approaches are illustrated in the following diagrams:
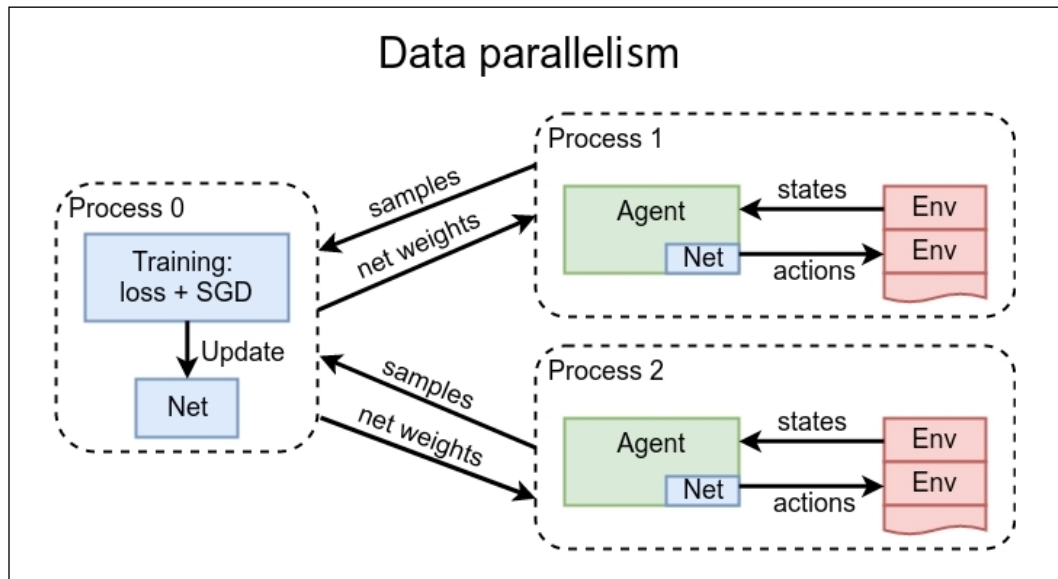
Figure 13.2: The first approach to actor-critic parallelism, based on distributed training samples being gathered
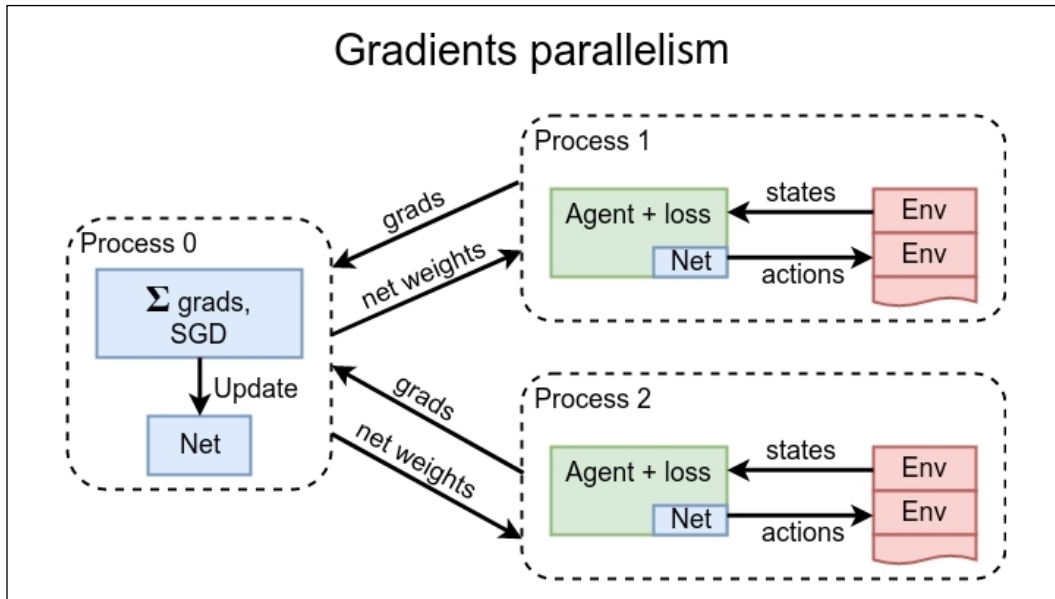
Figure 13.3: The second approach to parallelism, gathering gradients for the model

The difference between the two methods might not look very significant from the diagrams, but you need to be aware of the computation cost. The heaviest operation in A2C optimization is the training process, which consists of loss calculation from data samples (forward pass) and the calculation of gradients with respect to this loss. The SGD optimization step is quite lightweight: basically, just adding the scaled gradients to the NN's weights. By moving the computation of loss and gradients in the second approach from the central process, we eliminated the major potential bottleneck and made the whole process significantly more scalable.

In practice, the choice of the method mainly depends on your resources and your goals. If you have one single optimization problem and lots of distributed computation resources, such as a couple of dozen graphics processing units (GPUs) spread over several machines in the networks, then gradients parallelism will be the best approach to speed up your training.

However, in the case of one single GPU, both methods will provide a similar performance, but the first approach is generally simpler to implement, as you don't need to mess with low-level gradient values. In this chapter, we will implement both methods on our favorite Pong game to see the difference between the approaches and look at PyTorch multiprocessing capabilities.

Multiprocessing in Python

Python includes the `multiprocessing` (most of the time abbreviated to just `mp`) module to support process-level parallelism and the required communication primitives. In our example, we will use the two main classes from this module:

- `mp.Queue`: A concurrent multi-producer, multi-consumer FIFO (first in, first out) queue with transparent serialization and deserialization of objects placed in the queue
- `mp.Process`: A piece of code that is run in the child process and methods to control it from the parent process

PyTorch provides its own thin wrapper around the `multiprocessing` module, which adds the proper handling of tensors and variables on CUDA devices and shared memory. It provides exactly the same functionality as the `multiprocessing` module from the standard library, so all you need to do is use `import torch.multiprocessing` instead of `import multiprocessing`.

A3C with data parallelism

The first version of A3C parallelization that we will check (which was outlined in *Figure 13.2*) has both one main process that carries out training and several child processes communicating with environments and gathering experience to train on.

Implementation

For simplicity and efficiency, the NN weights broadcasting from the trainer process are not implemented. Instead of explicitly gathering and sending weights to child processes, the network is shared between all processes using PyTorch built-in capabilities, allowing us to use the same `nn.Module` instance with all its weights in different processes by calling the `share_memory()` method on NN creation. Under the hood, this method has zero overhead for CUDA (as GPU memory is shared among all the host's processes), or shared memory inter-process communication (IPC) in the case of CPU computation. In both cases, the method improves performance, but limits our example of one single machine using one single GPU card for training and data gathering.

It's not very limiting for our Pong example, but if you need larger scalability, the example should be extended with explicit sharing of NN weights.

Another extra sophistication implemented in this example is how training samples are passed from child processes to the master training process. They could be transferred as NumPy arrays, but this will increase the count of data copy operations, which might become a bottleneck. In our example, we implement different data preparation. The child processes have the responsibility to build mini-batches of data, copy them to GPU, and then pass tensors via the queue. Then, in the main process, we concatenate those tensors and perform the training.

The complete code is in the `Chapter13/01_a3c_data.py` file, and it uses the `Chapter13/lib/common.py` module with the following functionality pieces:

- `class AtariA2C(nn.Module)`: This implements the actor-critic NN module
- `class RewardTracker`: This handles the full episode undiscounted reward, writes it into TensorBoard, and checks for the *game solved* condition
- `unpack_batch(batch, net, last_val_gamma)`: This function converts a batch of transitions (`state`, `reward`, `action`, and `last_state`) for *n* episode steps into data suitable for training

You have already seen the code of those classes and functions in the previous chapters, so we won't repeat them here. Now let's check the code of the main modules, which includes the function for child subprocesses and the main training loop.

```python
#!/usr/bin/env python3
import os
import gym
import ptan
import numpy as np
import argparse
import collections
from tensorboardX import SummaryWriter

import torch.nn.utils as nn_utils
import torch.nn.functional as F
import torch.optim as optim
import torch.multiprocessing as mp

from lib import common
```

In the beginning, we import the required modules. There is nothing new here except that we import the `torch.multiprocessing` library.

```
GAMMA = 0.99
LEARNING_RATE = 0.001
ENTROPY_BETA = 0.01
BATCH_SIZE = 128

REWARD_STEPS = 4
CLIP_GRAD = 0.1

PROCESSES_COUNT = 4
NUM_ENVS = 8
MICRO_BATCH_SIZE = 32

ENV_NAME = "PongNoFrameskip-v4"
NAME = 'pong'
REWARD_BOUND = 18
```

In hyperparameters, we have three new values:

- `PROCESSES_COUNT` specifies the number of child processes that will gather training data for us. This activity is mostly CPU-bound, as the heaviest operation here is the preprocessing of Atari frames, so this value is set equal to the number of CPU cores on my machine.

- `MICRO_BATCH_SIZE` sets the number of training samples that every child process needs to obtain before transferring those samples to the main process.

- `NUM_ENVS` is the number of environments every child process will use to gather data. This number multiplied by the number of processes is the total amount of parallel environments that we will get our training data from.

```
def make_env():
    return ptan.common.wrappers.wrap_dqn(gym.make(ENV_NAME))

TotalReward = collections.namedtuple('TotalReward',
                                     field_names='reward')
```

Before we get to the child process function, we need the environment construction function and a tiny wrapper that we will use to send the total episode reward into the main training process.

```
def data_func(net, device, train_queue):
    envs = [make_env() for _ in range(NUM_ENVS)]
    agent = ptan.agent.PolicyAgent(
```

```
        lambda x: net(x)[0], device=device, apply_softmax=True)
    exp_source = ptan.experience.ExperienceSourceFirstLast(
        envs, agent, gamma=GAMMA, steps_count=REWARD_STEPS)
    micro_batch = []

    for exp in exp_source:
        new_rewards = exp_source.pop_total_rewards()
        if new_rewards:
            data = TotalReward(reward=np.mean(new_rewards))
            train_queue.put(data)

        micro_batch.append(exp)
        if len(micro_batch) < MICRO_BATCH_SIZE:
            continue

        data = common.unpack_batch(
            micro_batch, net, device=device,
            last_val_gamma=GAMMA ** REWARD_STEPS)
        train_queue.put(data)
        micro_batch.clear()
```

The preceding function is very simple, but it is special, as it will be executed in the child process. (We will use the `mp.Process` class to launch those processes in the main code block.) We pass it three arguments: our NN, the device to be used to perform computation (`cpu` or `cuda` string), and the queue we will use to send data from the child process to our master process, which will perform training. The queue is used in the many-producers and one-consumer mode, and can contain two different types of objects:

- `TotalReward`: This is a preceding object that we've defined, which has only one field reward, which is a float value of the total undiscounted reward for the completed episode.
- A tuple with tensors returned by the function `common.unpack_batch()`. Due to `torch.multiprocessing` magic, those tensors will be transferred to the main process without copying physical memory, which might be a costly operation (as an Atari observation is large).

As we get the required number of experience samples for our microbatch, we convert them into training data using the `unpack_batch` function and clear the batch. One thing to note is that as our experience samples represent four-step subsequences (as `REWARD_STEPS` is 4), we need to use a proper discount factor of γ^4 for the last $V(s)$ reward term.

That's it for the child processes, so now let's check the starting code for the main process and the training loop:

```python
if __name__ == "__main__":
    mp.set_start_method('spawn')
    os.environ['OMP_NUM_THREADS'] = "1"
    parser = argparse.ArgumentParser()
    parser.add_argument("--cuda", default=False,
                        action="store_true", help="Enable cuda")
    parser.add_argument("-n", "--name", required=True,
                        help="Name of the run")
    args = parser.parse_args()
    device = "cuda" if args.cuda else "cpu"

    writer = SummaryWriter(comment=f"-a3c-data_pong_{args.name}")
```

In the beginning, we take familiar steps, except for a single call to the `mp.set_start_method`, which instructs the `multiprocessing` module about the kind of parallelism we want to use. The native multiprocessing library in Python supports several ways to start subprocesses, but due to PyTorch multiprocessing limitations, spawn is the only option if you want to use GPU.

Another new line is assignment to the `OMP_NUM_THREADS`, which is an environment variable instructing the OpenMP library about the number of threads it can start. OpenMP (`https://www.openmp.org/`) is heavily used by the Gym and OpenCV libraries to provide a speed-up on multicore systems, which is a good thing most of the time. By default, the process that uses OpenMP starts a thread for every core in the system. But in our case, the effect from OpenMP is the opposite: as we're implementing our own parallelism, by launching several processes, extra threads overload the cores with frequent context switches, which negatively impacts performance. To avoid this, we explicitly set the maximum number of threads OpenMP can start with a single thread. If you want, you can experiment yourself with this parameter. On my system, I experienced a 3-4x performance drop without this code line.

```python
    env = make_env()
    net = common.AtariA2C(env.observation_space.shape,
                          env.action_space.n).to(device)
    net.share_memory()

    optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE,
                           eps=1e-3)
```

After that, we create our NN, move it to the CUDA device, and ask it to share its weights. CUDA tensors are shared by default, but for CPU mode, a call to `share_memory()` is required for multiprocessing to work.

```
train_queue = mp.Queue(maxsize=PROCESSES_COUNT)
data_proc_list = []
for _ in range(PROCESSES_COUNT):
    data_proc = mp.Process(target=data_func,
                           args=(net, device, train_queue))
    data_proc.start()
    data_proc_list.append(data_proc)
```

We now have to start our child processes, but first we create the queue they will use to deliver data to us. The argument to the queue constructor specifies the maximum queue capacity. All attempts to push a new item to the full queue will be blocked, which is very convenient for us for keeping our data samples on-policy. After the queue creation, we start the required number of processes using the `mp.Process` class and keep them for correct shutdown in a list. Right after the `mp.Process.start()` call, our `data_func` function will be executed by the child process.

```
batch_states = []
batch_actions = []
batch_vals_ref = []
step_idx = 0
batch_size = 0

try:
    with common.RewardTracker(writer, REWARD_BOUND) as tracker:
        with ptan.common.utils.TBMeanTracker(
                writer, 100) as tb_tracker:
            while True:
                train_entry = train_queue.get()
                if isinstance(train_entry, TotalReward):
                    if tracker.reward(train_entry.reward,
                                      step_idx):
                        break
                    continue
```

In the beginning of the training loop, we get the next entry from the queue and handle the possible `TotalReward` objects, which we pass to the reward tracker.

```
states_t, actions_t, vals_ref_t = train_entry
batch_states.append(states_t)
batch_actions.append(actions_t)
batch_vals_ref.append(vals_ref_t)
```

```
step_idx += states_t.size()[0]
batch_size += states_t.size()[0]
if batch_size < BATCH_SIZE:
    continue

states_v = torch.cat(batch_states)
actions_t = torch.cat(batch_actions)
vals_ref_v = torch.cat(batch_vals_ref)
batch_states.clear()
batch_actions.clear()
batch_vals_ref.clear()
batch_size = 0
```

As we can have only two types of objects in the queue (`TotalReward` and a tuple with a microbatch), we need to check an entry obtained from the queue only once. After the `TotalReward` entries are handled, we process the tuple with tensors. We accumulate them in lists, and once the required batch size has been reached, we concatenate the tensors using a `torch.cat()` call, which appends tensors along the first dimension.

The rest of the training loop is standard actor-critic loss calculation, which is performed in exactly the same way as in the previous chapter: we calculate the logits of the policy and value estimation using our current network, and calculate the policy, value, and entropy losses.

```
optimizer.zero_grad()
logits_v, value_v = net(states_v)

loss_value_v = F.mse_loss(
    value_v.squeeze(-1), vals_ref_v)

log_prob_v = F.log_softmax(logits_v, dim=1)
adv_v = vals_ref_v - value_v.detach()
size = states_v.size()[0]
log_p_a = log_prob_v[range(size), actions_t]
log_prob_actions_v = adv_v * log_p_a
loss_policy_v = -log_prob_actions_v.mean()

prob_v = F.softmax(logits_v, dim=1)
ent = (prob_v * log_prob_v).sum(dim=1).mean()
entropy_loss_v = ENTROPY_BETA * ent

loss_v = entropy_loss_v + loss_value_v + \
        loss_policy_v
loss_v.backward()
```

```
                    nn_utils.clip_grad_norm_(
                        net.parameters(), CLIP_GRAD)
                    optimizer.step()
```

As the last step, we pass the calculated tensors to the TensorBoard `tracker` class, which will perform the averaging and store the data that we want to monitor:

```
                    tb_tracker.track("advantage", adv_v, step_idx)
                    tb_tracker.track("values", value_v, step_idx)
                    tb_tracker.track("batch_rewards", vals_ref_v,
                                     step_idx)
                    tb_tracker.track("loss_entropy",
                                     entropy_loss_v, step_idx)
                    tb_tracker.track("loss_policy",
                                     loss_policy_v, step_idx)
                    tb_tracker.track("loss_value",
                                     loss_value_v, step_idx)
                    tb_tracker.track("loss_total",
                                     loss_v, step_idx)
        finally:
            for p in data_proc_list:
                p.terminate()
                p.join()
```

In the last `finally` block, which can be executed due to an exception (*Ctrl + C*, for example) or the *game solved* condition, we terminate the child processes and wait for them. This is required to make sure that there are no leftover processes.

Results

Start the example as usual, and after some delay, it should begin writing performance and mean reward data. On a GTX 1080 Ti and 4-core machine, it shows the speed of 2,600…3,000 frames per second (FPS), which is a nice improvement on the 600 FPS we got in the previous chapter.

```
rl_book_samples/Chapter11$ ./01_a3c_data.py --cuda -n final
...
31424: done 22 games, mean reward -20.636, speed 2694.86 f/s
31776: done 23 games, mean reward -20.652, speed 2520.54 f/s
31936: done 24 games, mean reward -20.667, speed 2968.11 f/s
32288: done 25 games, mean reward -20.640, speed 2639.84 f/s
32544: done 26 games, mean reward -20.577, speed 2237.74 f/s
32736: done 27 games, mean reward -20.556, speed 3200.18 f/s
...
```

In terms of convergence dynamics, the new version is similar to A2C with parallel environments and solves Pong in 6…7 million observations from the environment. However, those 7 million frames are processed in slightly less than one hour, instead of waiting for three hours with the version from the previous chapter.
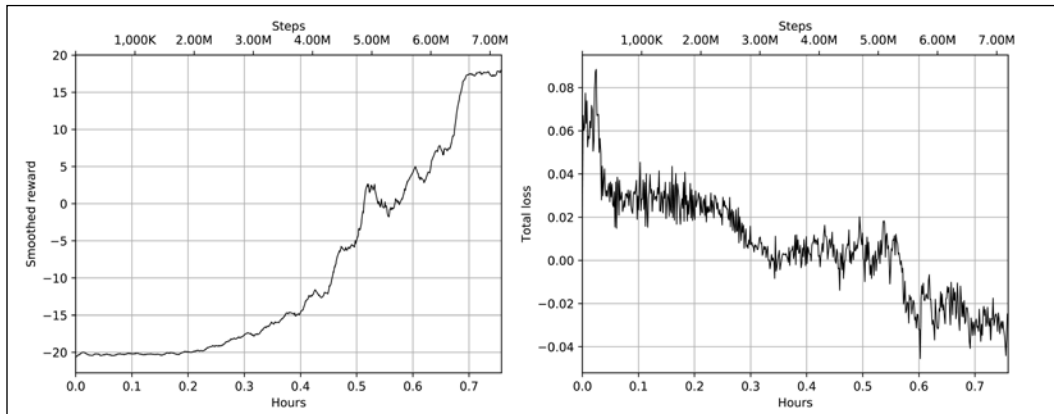
Figure 13.4: The reward (left) and total loss (right) of the data-parallel version
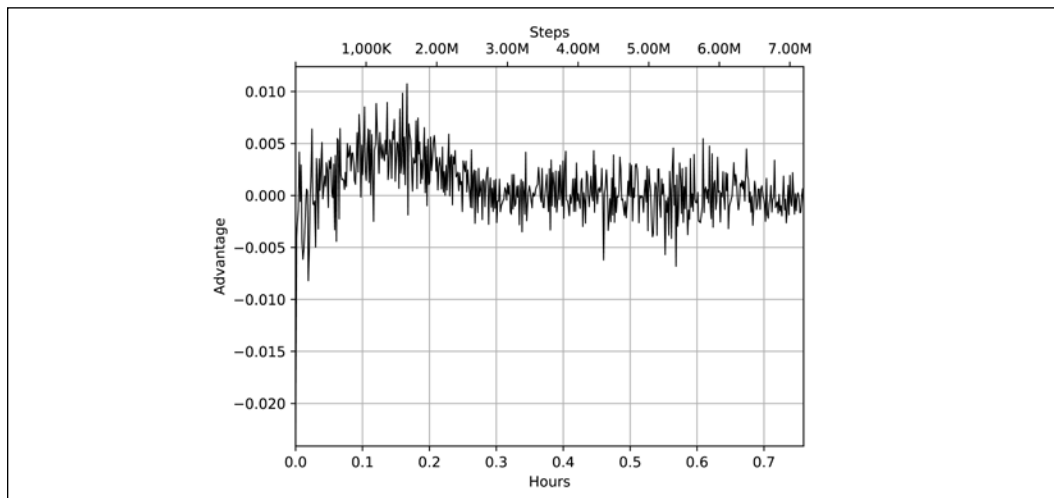
Figure 13.5: Advantage during the training

A3C with gradients parallelism

The next approach that we will consider to parallelize A2C implementation will have several child processes, but instead of feeding training data to the central training loop, they will calculate the gradients using their local training data, and send those gradients to the central master process.

This process is responsible for combining those gradients together (which is basically just summing them) and performing an SGD update on the shared network.

The difference might look minor, but this approach is much more scalable, especially if you have several powerful nodes with multiple GPUs connected with the network. In this case, the central process in the data-parallel model quickly becomes a bottleneck, as the loss calculation and backpropagation are computationally demanding. Gradient parallelization allows for the spreading of the load on several GPUs, performing only a relatively simple operation of gradient combination in the central place.

Implementation

The complete example is in the `Chapter13/02_a3c_grad.py` file, and it uses the same `Chapter13/lib/common.py` file as in our previous example:

```
GAMMA = 0.99
LEARNING_RATE = 0.001
ENTROPY_BETA = 0.01

REWARD_STEPS = 4
CLIP_GRAD = 0.1

PROCESSES_COUNT = 4
NUM_ENVS = 8

GRAD_BATCH = 64
TRAIN_BATCH = 2
ENV_NAME = "PongNoFrameskip-v4"
REWARD_BOUND = 18
```

As usual, we define the hyperparameters, which are mostly the same as in the previous example, except `BATCH_SIZE` is replaced by two parameters: `GRAD_BATCH` and `TRAIN_BATCH`. The value of `GRAD_BATCH` defines the size of the batch used by every child process to compute the loss and get the value of the gradients. The second parameter, `TRAIN_BATCH`, specifies how many gradient batches from the child processes will be combined on every SGD iteration. Every entry produced by the child process has the same shape as our network parameters, and we sum up `TRAIN_BATCH` values of them together. So, for every optimization step, we use the `TRAIN_BATCH` * `GRAD_BATCH` training samples. As the loss calculation and backpropagation are quite heavy operations, we use large `GRAD_BATCH` to make them more efficient.

Due to this large batch, we should keep `TRAIN_BATCH` relatively low to keep our network update on-policy.

```python
def make_env():
    return ptan.common.wrappers.wrap_dqn(gym.make(ENV_NAME))



def grads_func(proc_name, net, device, train_queue):
    envs = [make_env() for _ in range(NUM_ENVS)]

    agent = ptan.agent.PolicyAgent(
        lambda x: net(x)[0], device=device, apply_softmax=True)
    exp_source = ptan.experience.ExperienceSourceFirstLast(
        envs, agent, gamma=GAMMA, steps_count=REWARD_STEPS)

    batch = []
    frame_idx = 0
    writer = SummaryWriter(comment=proc_name)
```

The preceding is the function executed by the child process, which is much more complicated than in our data-parallel example. As a compensation, the training loop in the main process becomes almost trivial. On the creation of the child process, we pass several arguments to the function:

- The name of the process, which is used to create the TensorBoard writer. In this example, every child process writes its own TensorBoard dataset
- The shared NN
- A device to perform computations (a `cpu` or `cuda` string)
- The queue used to deliver the calculated gradients to the central process

Our child process function looks very similar to the main training loop in the data-parallel version, which is not surprising, as the responsibilities of our child process increased. However, instead of asking the optimizer to update the network, we gather gradients and send them to the queue. The rest of the code is almost the same:

```python
    with common.RewardTracker(writer, REWARD_BOUND) as tracker:
        with ptan.common.utils.TBMeanTracker(
                writer, 100) as tb_tracker:
            for exp in exp_source:
                frame_idx += 1
                new_rewards = exp_source.pop_total_rewards()
                if new_rewards and tracker.reward(
                        new_rewards[0], frame_idx):
                    break
```

```
batch.append(exp)
if len(batch) < GRAD_BATCH:
    continue
```

Up to this point, we've gathered the batch with transitions and handled the end-of-episode rewards.

```
data = common.unpack_batch(
    batch, net, device=device,
    last_val_gamma=GAMMA**REWARD_STEPS)
states_v, actions_t, vals_ref_v = data

batch.clear()

net.zero_grad()
logits_v, value_v = net(states_v)
loss_value_v = F.mse_loss(
    value_v.squeeze(-1), vals_ref_v)

log_prob_v = F.log_softmax(logits_v, dim=1)
adv_v = vals_ref_v - value_v.detach()
log_p_a = log_prob_v[range(GRAD_BATCH), actions_t]
log_prob_actions_v = adv_v * log_p_a
loss_policy_v = -log_prob_actions_v.mean()

prob_v = F.softmax(logits_v, dim=1)
ent = (prob_v * log_prob_v).sum(dim=1).mean()
entropy_loss_v = ENTROPY_BETA * ent

loss_v = entropy_loss_v + loss_value_v + \
         loss_policy_v
loss_v.backward()
```

In the preceding section, we calculate the combined loss from the training data and perform backpropagation of the loss, which effectively stores gradients in the `tensor.grad` field for every network parameter. This could be done without bothering with synchronization with other workers, as our network's parameters are shared, but the gradients are locally allocated by every process.

```
tb_tracker.track("advantage", adv_v, frame_idx)
tb_tracker.track("values", value_v, frame_idx)
tb_tracker.track("batch_rewards", vals_ref_v,
                 frame_idx)
tb_tracker.track("loss_entropy", entropy_loss_v,
```

```
                                        frame_idx)
                    tb_tracker.track("loss_policy", loss_policy_v,
                                        frame_idx)
                    tb_tracker.track("loss_value", loss_value_v,
                                        frame_idx)
                    tb_tracker.track("loss_total", loss_v, frame_idx)
```

In the preceding code, we send our intermediate values that we're going to monitor during the training to TensorBoard.

```
                    nn_utils.clip_grad_norm_(
                        net.parameters(), CLIP_GRAD)
                    grads = [
                        param.grad.data.cpu().numpy()
                        if param.grad is not None else None
                        for param in net.parameters()
                    ]
                    train_queue.put(grads)
```

At the end of the loop, we need to clip the gradients and extract them from the network's parameters into a separate buffer (to prevent them from being corrupted by the next iteration of the loop).

```
            train_queue.put(None)
```

The last line in `grads_func` puts `None` into the queue, signaling that this child process has reached the *game solved* state and training should be stopped.

```
    if __name__ == "__main__":
        mp.set_start_method('spawn')
        os.environ['OMP_NUM_THREADS'] = "1"
        parser = argparse.ArgumentParser()
        parser.add_argument("--cuda", default=False,
                            action="store_true", help="Enable cuda")
        parser.add_argument("-n", "--name", required=True,
                            help="Name of the run")
        args = parser.parse_args()
        device = "cuda" if args.cuda else "cpu"

        env = make_env()
        net = common.AtariA2C(env.observation_space.shape,
                            env.action_space.n).to(device)
        net.share_memory()
```

The main process starts with the creation of the network and sharing of its weights. As in the previous section, we need to set a start method for `torch.` `multiprocessing` and limit the number of threads started by OpenMP.

```
optimizer = optim.Adam(net.parameters(),
                       lr=LEARNING_RATE, eps=1e-3)


train_queue = mp.Queue(maxsize=PROCESSES_COUNT)
data_proc_list = []
for proc_idx in range(PROCESSES_COUNT):
    proc_name = f"-a3c-grad_pong_{args.name}#{proc_idx}"
    p_args = (proc_name, net, device, train_queue)
    data_proc = mp.Process(target=grads_func, args=p_args)
    data_proc.start()
    data_proc_list.append(data_proc)
```

Then, as before, we create the communication queue and spawn the required count of child processes.

```
batch = []
step_idx = 0
grad_buffer = None

try:
    while True:
        train_entry = train_queue.get()
        if train_entry is None:
            break
```

The major difference from the data-parallel version of A3C lies in the training loop, which is much simpler here, as child processes have done all the heavy calculations for us. In the beginning of the loop, we handle the situation when one of the processes has reached the required mean reward to stop the training. In this case, we just exit the loop.

```
step_idx += 1

if grad_buffer is None:
    grad_buffer = train_entry
else:
    for tgt_grad, grad in zip(grad_buffer,
                              train_entry):
        tgt_grad += grad
```

To average the gradients from different children, we call the optimizer's `step()` function for every `TRAIN_BATCH` gradient obtained. For intermediate steps, we just sum the corresponding gradients together.

```
if step_idx % TRAIN_BATCH == 0:
    for param, grad in zip(net.parameters(),
                           grad_buffer):
        param.grad = torch.FloatTensor(grad).to(device)

    nn_utils.clip_grad_norm_(
        net.parameters(), CLIP_GRAD)
    optimizer.step()
    grad_buffer = None
```

When we have accumulated enough gradient pieces, we convert the sum of the gradients into the PyTorch `FloatTensor` and assign them to the `grad` field of the network parameters. After that, all we need to do is call the optimizer's `step()` method to update the network parameters using the accumulated gradients.

```
finally:
    for p in data_proc_list:
        p.terminate()
        p.join()
```

On the exit from the training loop, we stop all child processes to make sure that we terminated them, even if *Ctrl + C* was pressed to stop the optimization. This is needed to prevent zombie processes from occupying GPU resources.

Results

This example can be started the same way as the previous example, and after a while, it should start displaying speed and mean reward. However, you need to be aware that displayed information is local for every child process, which means that speed, the count of games completed, and the number of frames need to be multiplied by the number of processes. My benchmarks have shown speed to be around 450-550 FPS for every child, which gives 1,800-2,200 FPS in total.

```
rl_book_samples/Chapter11$ ./02_a3c_grad.py --cuda -n final
11278: done 1 games, mean reward -21.000, speed 520.23 f/s
11640: done 2 games, mean reward -21.000, speed 610.54 f/s
11773: done 3 games, mean reward -21.000, speed 485.09 f/s
11803: done 4 games, mean reward -21.000, speed 359.42 f/s
11765: done 1 games, mean reward -21.000, speed 519.08 f/s
11771: done 2 games, mean reward -21.000, speed 531.22 f/s
...
```

Convergence dynamics are also very similar to the previous version. The total number of observations is about 8…10 million, which requires one and a half hours to complete.
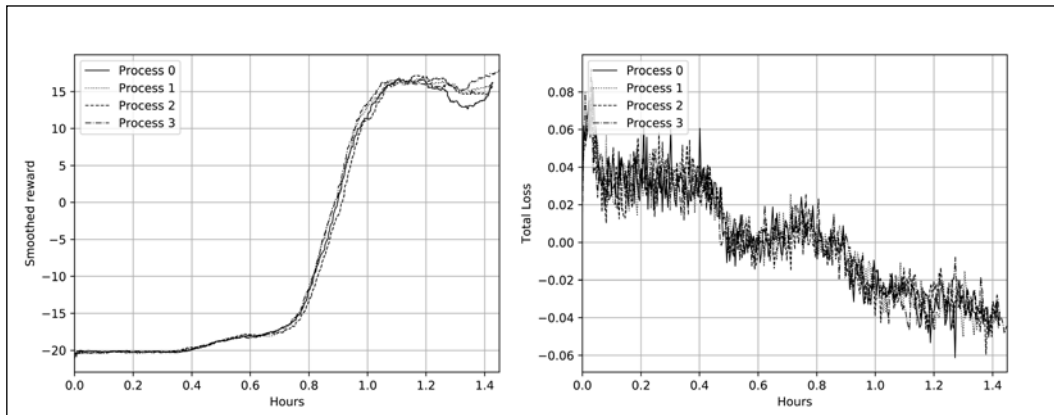
Figure 13.6: The mean reward (left) and total loss (right) for the gradient-parallel version
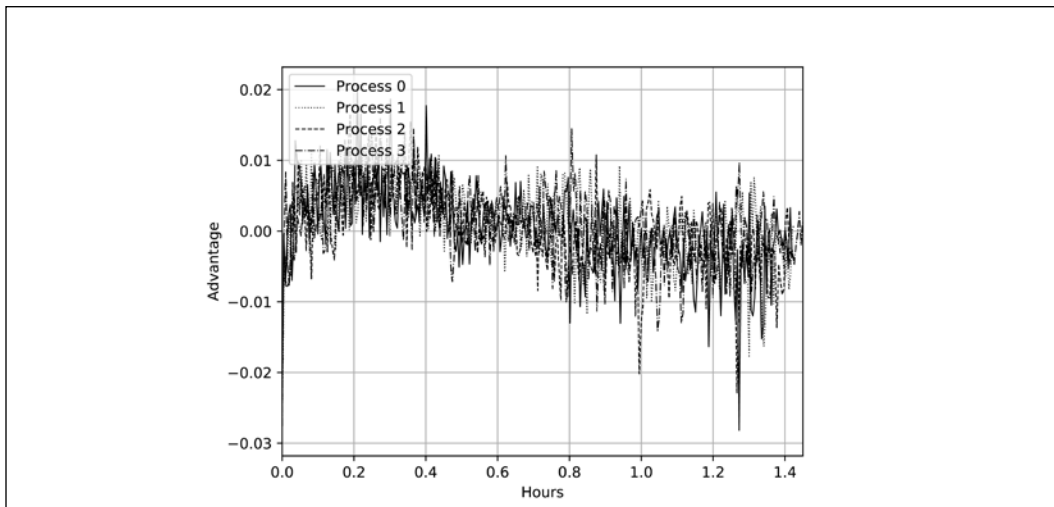
Figure 13.7: Advantage during the training

Summary

In this chapter, we discussed why it is important for policy gradient methods to gather training data from multiple environments, due to their on-policy nature. We also implemented two different approaches to A3C, in order to parallelize and stabilize the training process. Parallelization will come up once again in this book, when we discuss black-box methods (*Chapter 20*, *Black-Box Optimization in RL*).

In the next three chapters, we will take a look at practical problems that can be solved using policy gradient methods, which will wrap up the policy gradient methods part of the book.

14
Training Chatbots with RL

In this chapter, we will take a look at another practical application of deep reinforcement learning (RL), which has become popular over the past several years: the training of natural language models with RL methods. It started with a paper called *Recurrent Models of Visual Attention* (`https://arxiv.org/abs/1406.6247`), which was published in 2014, and has been successfully applied to a wide variety of problems from the **natural language processing** (**NLP**) domain.

In this chapter, we will:

- Begin with a brief introduction to the NLP basics, including **recurrent neural networks** (**RNNs**), **word embedding**, and the **seq2seq** (sequence-to-sequence) model
- Discuss similarities between NLP and RL problems
- Take a look at original ideas on how to improve NLP seq2seq training using RL methods

The core of the chapter is a dialogue system trained on a movie dialogues dataset: the Cornell Movie-Dialogs Corpus.

An overview of chatbots

A trending topic in recent years has been AI-driven **chatbots**. There are various opinions on the subject, ranging from chatbots being completely useless to being an absolutely brilliant idea, but one thing is hard to question: chatbots open up new ways for people to communicate with computers that are much more human-like and natural than the old-style interfaces that we are all used to.

At its core, a chatbot is a computer program that uses natural language to communicate with other parties (humans or other computer programs) in a form of *dialogue*.

Such scenarios can take many different forms, such as one chatbot talking to a user, many chatbots talking to each other, and so on. For example, there might be a technical support chatbot that can answer free-text questions from users. However, usually chatbots share common properties of a dialogue interaction (the user asks a question, but the chatbot can ask clarifying questions to get the missing information) and a free form of natural language (which is different from phone menus, for example, when you have a fixed *press N to get to X category*, or *enter your bank account number to check the balance* option given to the user).

Chatbot training

Natural language understanding was the stuff of science fiction for a long time. In science fiction, you can just chat with your starship's computer to get useful and relevant information about the recent alien invasion, without pressing any button. This scenario was exploited by authors and filmmakers for decades, but in real life, such interactions with computers started to become a reality only recently. You still can't talk to your starship, but you can, at least, switch your toaster on and off without pushing buttons, which is undoubtedly a major step forward!

The reason why it took computers so long to understand language is simply due to the complexity of language itself. Even in trivial scenarios—like saying, "Toaster, switch on!"—you can imagine several ways to formulate your command, and it's usually very hard to capture all those ways and corner cases in advance using normal computer programming techniques. Unfortunately, traditional computer programming requires you to give computers exact, explicit instructions, and one single corner case, or fuzziness in the input can make your super-sophisticated code fail.

The recent advances in machine learning (ML) and deep learning (DL), with all their applications, are the first real step in the direction of breaking this strictness in computer programming by replacing it with a different idea: letting computers find patterns in data by themselves. This approach has turned out to be quite successful in some domains and now everybody is very excited about the new methods and their potential applications. The DL resurrection started in computer vision and then continued in the NLP domain, and you can definitely expect more and more new, successful, and useful applications in the future.

So, returning to our chatbots, natural language complexity was the major blocker in practical applications. For a long time, chatbots were mostly toy examples created by bored engineers for their entertainment. One of the oldest examples of such a system, and definitely the most popular, is ELIZA (`https://en.wikipedia.org/wiki/ELIZA`), which was created in the 1960s.

Although ELIZA was quite successful at mimicking a psychotherapist, it had zero understanding of the user's phrases and just included a small set of manually created patterns and typical replies given to the user's input.

This approach was a major way forward in implementing such systems in a pre-DL era. It was a common belief that we just needed to add more patterns capturing the real language corner cases, and after some time, computers would be able to understand the human language. Unfortunately, this idea turned out to be impractical, as the number of rules and contradicting examples that humans need to handle is too large and complex to be created manually.

The ML approach allows you to attack the complexity of the problem from a different direction. Instead of manually creating many rules to handle the user's input, you gather a lot of training data and allow the ML algorithm to find the best way to solve the problem. This approach has its own specific details related to the NLP domain, and we will cover an overview of them in the next section. For now, what is much more important is that software developers have discovered the ability to work with natural language in the same way as other, much more computer-friendly and formal things, such as document formats, network protocols, or computer language grammars. This is still not trivial: it requires a lot of work, and sometimes you step into uncharted territory, but at least this approach works sometimes and doesn't require you to lock hundreds of linguists into one room for a decade to gather NLP rules!

Chatbots are still very new and experimental, but the overall idea is to allow computers to communicate with the user in the form of free-text dialogue, instead of more formal ways. Let's take Internet shopping as an example. When you want to buy something, you go to an online store, such as Amazon or eBay, and look through categories or use the website search function to find the product that you want. However, this scenario has several problems. First of all, large web stores could have millions of items falling into thousands of categories in a very non-deterministic way. A simple child's toy can belong to several categories, like *puzzles*, *educational games*, and *5-10 years old*, at the same time. On the other hand, if you don't know exactly what you want, or at least which category it belongs to, you can easily spend hours browsing endless lists of similar items, hoping that you will find what you're looking for. The search engines of websites solve this issue only partially, as you can get a meaningful number of results only if you know some unique combination of words or the right brand name to search for.

A different view of the problem would be a chatbot that can ask the user questions about their intentions and price range to limit the search. Of course, this approach is not universal and shouldn't be seen as a complete replacement for modern websites with search functions, catalogs, and other user interface (UI) solutions developed over time, but it can provide a nice alternative in some use cases and serve some percentage of users better than the old-style interaction methods.

The deep NLP basics

Hopefully, you're excited about chatbots and their potential applications, so let's now get to the boring details of NLP building blocks and standard approaches. As with almost everything in ML, there is a lot of hype around deep NLP and it is evolving at a fast pace, so this section will just scratch the surface and cover the most common and standard building blocks. For a more detailed description, Richard Socher's online course *CS224d* (`http://cs224d.stanford.edu`) is a really good starting point.

RNNs

NLP has its own specifics that make it different from computer vision or other domains. One such feature is processing variable-length objects. At various levels, NLP deals with objects that could have different lengths; for example, a word in a language could contain several characters. Sentences are formed from variable-length word sequences. Paragraphs or documents consist of varying numbers of sentences. Such variability is not NLP-specific and can arise in different domains, like in signal processing or video processing. Even standard computer vision problems could be seen as a sequence of some objects, like an image captioning problem when a neural network (NN) can focus on various amounts of regions of the same image to better describe the image.

RNNs provide one of the standard building blocks. An RNN is a network with fixed input and output that is applied to a sequence of objects and can pass information along this sequence. This information is called the **hidden state**, and it is normally just a vector of numbers of some size.

In the following diagram, we have an RNN with one input, which is a vector of numbers; the output is another vector. What makes it different from a standard feed-forward or convolutional NN is two extra gates: one input and one output. The extra input feeds the hidden state from the previous item into the RNN unit, and the extra output provides a transformed hidden state to the next sequence.
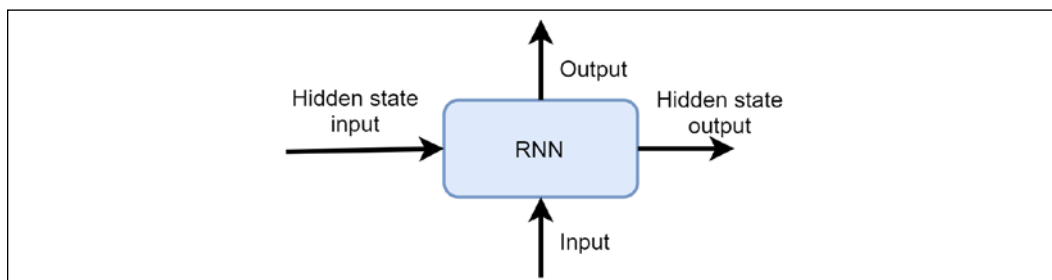
Figure 14.1: The structure of an RNN building block

This is supposed to solve our variable-length issue, and, in fact, it does. As an RNN has two inputs, it can be applied to input sequences of any length, just by passing the hidden state produced by the previous entry to the next one. In *Figure 14.2*, an RNN is applied to the sentence *this is a cat*, producing the output for every word in the sequence. During the application, we have the same RNN applied to every input item, but by having the hidden state, it can now pass information along the sequence. This is similar to convolutional NNs, when we have the same set of filters applied to various locations of the image, but the difference is that a convolutional NN can't pass the hidden state.
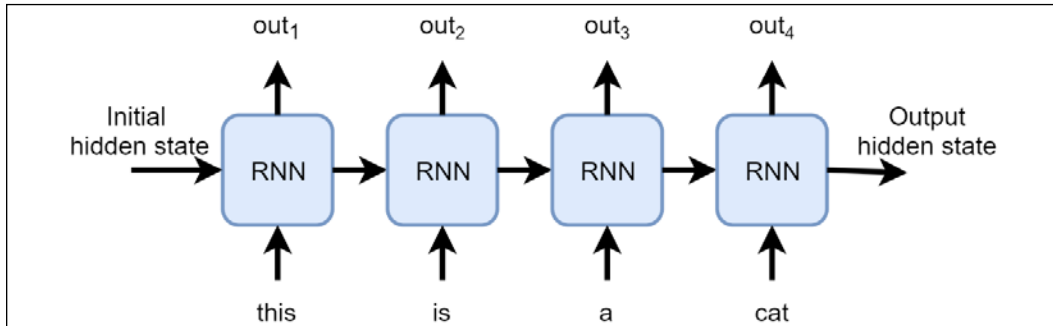
Figure 14.2: How an RNN is applied to a sentence

Despite the simplicity of this model, it adds an extra degree of freedom to the standard feed-forward NN model. The feed-forward NNs are determined by their input and always produce the same output for some fixed input (in testing mode, of course, and not during the training). An RNN's output depends not only on the input but on the hidden state, which could be changed by the NN itself. So, the NN could pass some information from the beginning of the sequence to the end and produce different output for the same input in different contexts. This context dependency is very important in NLP, as in natural language, a single word could have a completely different meaning in different contexts, and the meaning of a whole sentence could be changed by a single word.

Of course, such flexibility comes with its own cost. RNNs usually require more time to train and can produce some weird behavior, like loss oscillations or sudden amnesia during the training. However, the research community has already done a lot of work and is still working hard to make RNNs more practical and stable, so RNNs can be seen as a standard building block of the systems that need to process variable-length input.

Word embedding

Another standard building block of modern DL-driven NLP is **word embedding**, which is also called **word2vec** by one of the most popular training methods. The idea comes from the problem of representing our language sequences in NNs. Normally, NNs work with fixed-sized vectors of numbers, but in NLP, we normally have words or characters as input to the model.

One possible solution might be *one-hot encoding* our dictionary, which is when every word has its own position in the input vector and we set this number to 1 when we encounter this word in the input sequence. This is a standard approach for NNs when you have to deal with some relatively small discrete set of items and want to represent them in an NN-friendly way.

Unfortunately, one-hot encoding doesn't work very well for several reasons. First of all, our input set is usually not small. If we want to encode only the most commonly used English dictionary, it will contain at least several thousand words. The *Oxford English Dictionary* has 170,000 commonly used words and 50,000 obsolete and rare words. This is only established vocabulary and doesn't count slang, new words, scientific terms, abbreviations, typos, jokes, Twitter memes, and so on. And this is only for the English language!

The second problem related to the one-hot representation of words is the uneven frequency of vocabulary. There are relatively small sets of very frequent words, like *a* and *cat*, but a very large set of much more rarely used words, like *covfefe* or *bibliopole*, and those rare words can occur only once or twice in a very large text corpus. So, our one-hot representation is very inefficient in terms of space.

Another issue with simple one-hot representation is not capturing a word's relations. For example, some words are synonyms and have the same meaning, but they will be represented by different vectors. Some words are used very frequently together, like *United Nations* or *fair trade*, and this fact is also not captured in one-hot representation.

To overcome all this, we can use word embeddings, which map every word in some vocabulary into a dense, fixed-length vector of numbers. These numbers are not random but trained on a large corpus of text to capture the context of words. A detailed description of word embeddings is beyond the scope of this book, but this is a really powerful and widely used NLP technique to represent words, characters, and other objects in some sequence. For now, you can think about them as just mapping words into number vectors, and this mapping is convenient for the NN to be able to distinguish words from each other.

To obtain this mapping, two methods exist. First of all, you can download pretrained vectors for the language that you need. There are several sources of embeddings available; just search on Google for *GloVe pretrained vectors* or *word2vec pretrained*. (GloVe and word2vec are different methods used to train such vectors, which produce similar results.)

Another way to obtain embeddings is to train them on your own dataset. To do this, you can either use special tools, such as `fasttext` (`https://fasttext.cc/`, an open source utility from Facebook), or just initialize embeddings randomly and allow your model to adjust them during normal training.

The Encoder-Decoder architecture

Another model that is widely used in NLP is called **Encoder-Decoder**, or seq2seq. It originally comes from machine translation, when your system needs to accept a sequence of words in the source language and produce another sequence in the target language. The idea behind seq2seq is to use an RNN to process an input sequence and *encode* this sequence into some fixed-length representation. This RNN is called an **encoder**. Then you feed the encoded vector into another RNN, called a **decoder**, which has to produce the resulting sequence in the target language. An example of this idea is shown next, where we are translating an English sentence into Russian:
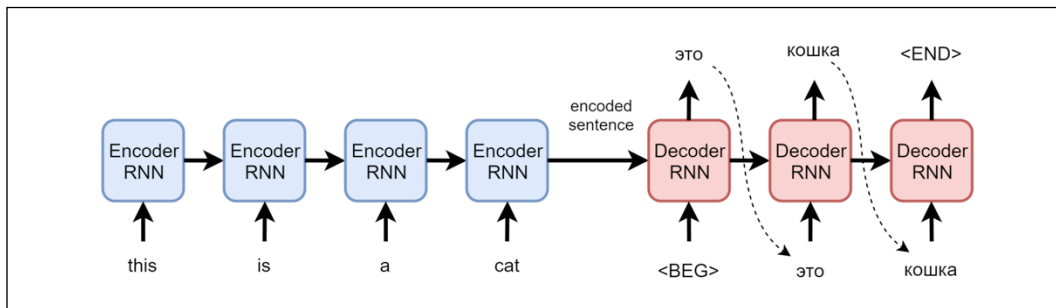
Figure 14.3: The Encoder-Decoder architecture in machine translation

This model (with a lot of modern tweaks and extensions) is still a major workhorse of machine translation, but is general enough to be applicable to a much wider set of domains, for example, audio processing, image annotation, and video captioning. In our chatbot example, we will use it to generate reply phrases when given the input sequence of words.

Seq2seq training

That's all very interesting, but how is it related to RL? The connection lies in the training process of the seq2seq model, but before we come to the modern RL approaches to the problem, I need to say a couple of words about the standard way of carrying out the training.

Log-likelihood training

Imagine that we need to create a machine translation system from one language (say, French) into another language (English) using the seq2seq model. Let's assume that we have a good, large dataset of sample translations with French-English sentences that we're going to train our model on. How do we do this?

The encoding part is obvious: we just apply our encoder RNN to the first sentence in the training pair, which produces an encoded representation of the sentence. The obvious candidate for this representation is the hidden state returned from the last RNN application. At the encoding stage, we ignore the RNN's outputs, taking into account only the hidden state from the last RNN application. We also extend our sentence with the special word <END>, which signals to the encoder the end of the sentence. (In NLP, the elements of the sequence we're dealing with are called **tokens**. Most of the time, they are words, but they might be letters, for example.) This encoding process is shown in the following diagram:
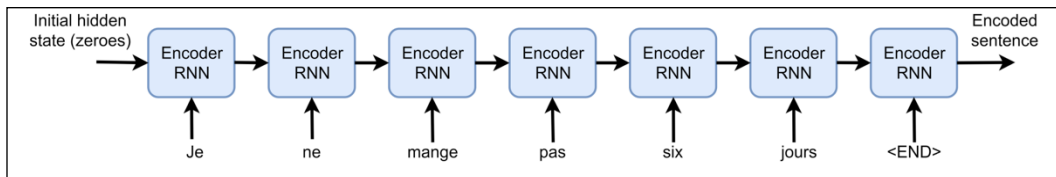
Figure 14.4: The encoding step

To start decoding, we pass the encoded representation to the decoder's input hidden state and pass the token <BEG> as a signal to begin decoding. In this step, the decoder RNN has to return the first token of the translated sentence. However, in the beginning of training, when both the encoder and decoder RNNs are initialized with random weights, the decoder's output will be random; our goal will be to push it toward the correct translation using stochastic gradient descent (SGD).

The traditional approach is to treat this problem as classification, when our decoder needs to return the probability distribution over the tokens in the current position of the decoded sentence.

Normally, this is done by transforming the decoder's output using the shallow feed-forward NN and producing a vector whose length is the size of our dictionary. Then, we use this probability distribution and the standard loss for classification problems: cross-entropy loss (also known as **log-likelihood loss**).

That's clear with the first token in the decoded sequence, which should be produced by the `<BEG>` token given on the input, but what about the rest of the sequence? There are two options here. The first alternative is to feed tokens from the reference sentence. For example, if we have the training pair *Je ne mange pas six jours → I haven't eaten for six days*, we feed tokens (*I, haven't, eaten...*) to the decoder and then use cross-entropy loss between the RNN's output and the next token in the sentence. This training mode is called **teacher forcing**, and at every step we feed a token from the correct translation, asking the RNN to produce the correct next token. This process is shown in the following diagram:
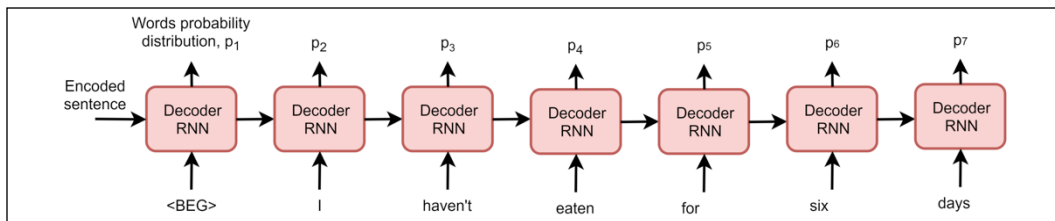
Figure 14.5: How the encoded vector is decoded in the teacher-forcing mode

The loss expression for the preceding example will be calculated as follows:

```
L = xentropy(p1,"I")       + xentropy(p2,"haven't")
  + xentropy(p3,"eaten")  + xentropy(p4,"for")
  + xentropy(p5,"six")     + xentropy(p6,"days")
  + xentropy(p7,"<END>")
```

As both the decoder and encoder are differentiable NNs, we can just backpropagate the loss to push both of them toward the better classification of this example in the future, in the same way that we train the image classifier, for example. Unfortunately, this procedure doesn't solve the seq2seq training problem completely, and the issue is related to the way the model was used. During the training, we know both the input and the desired output sequences, so we can feed the valid output sequence to the decoder, which is being asked only to produce the next token of the sequence.

After the model has been trained, we won't have a target sequence (as this sequence is supposed to be produced by the model). So, the simplest way to use the model will be to encode the input sequence using the encoder and then ask the decoder to generate one item of the output at a time, feeding the produced token into the input of the decoder.

Passing the previous result into the input might look natural, but there is a danger here. During the training, we haven't asked our decoder RNN to use its own output as input, so one single mistake during the generation may confuse the decoder and lead to garbage output.

To overcome this, a second approach to seq2seq training exists, called **curriculum learning**. This method uses the same log-likelihood loss, but instead of passing the full target sequence as the decoder's input, we just ask the decoder to decode the sequence in the same way as we're going to use it after training. This process is illustrated in the following diagram. This adds robustness to the decoder, which gives a better result on the practical application of the model. As a downside, this mode may lead to very long training, as our decoder will learn how to produce the desired output token by token. To compensate for this in practice, we usually train a model using both teacher and curriculum learning, just randomly choosing between those two for every batch.
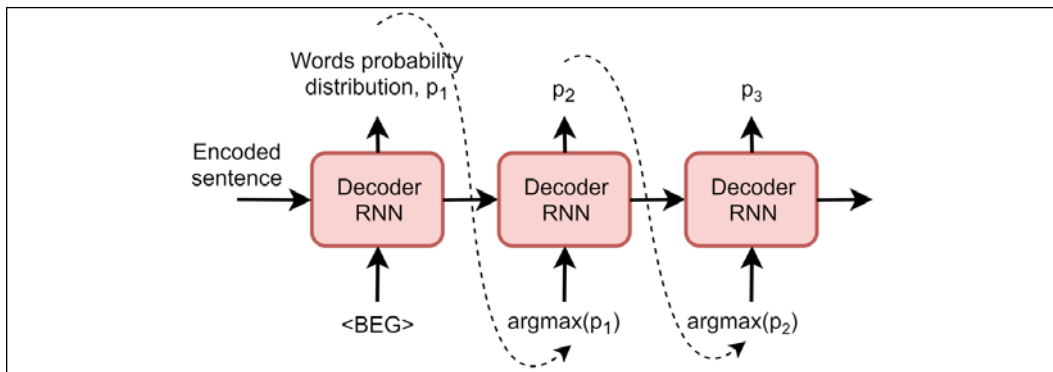
Figure 14.6: How decoding in the curriculum learning mode is performed

The bilingual evaluation understudy (BLEU) score

Before we get into the main topic of this chapter (RL for seq2seq), I need to introduce the metric used to compare the quality of machine translation output in NLP problems. The metric, called **BLEU**, is one of the standard ways to compare the output sequence produced by the machine with some set of reference outputs. It allows multiple reference outputs to be used (one sentence could be translated in various ways), and at its core, it calculates the ratio of unigrams, bigrams, and so on shared between the produced output and reference sentences. Other alternatives exist, such as CIDEr and ROUGE. In this example, we will use BLEU implemented in the `nltk` Python library (the `nltk.translate.bleu_score` package).

RL in seq2seq

RL and text generation might look very different, but there are connections, which could be used to improve the quality of the trained seq2seq models. The first thing to note is that our decoder outputs the probability distribution at every step, which is very similar to policy gradient models. From this perspective, our decoder could be seen as an agent trying to decide which token to produce at every step.

There are several advantages of such an interpretation of the decoding process. First of all, by treating our decoding process as stochastic, we can automatically take into account multiple target sequences. For example, there are many possible replies to the *hello, how are you?* phrase, and all of them are correct. By optimizing the log-likelihood objective, our model will try to learn some average of all those replies, but the average of the phrases *I'm fine, thanks!* and *not very good* will not necessarily be a meaningful phrase. By returning the probability distribution and sampling the next token from it, our agent potentially could learn how to produce all possible variants, instead of learning some averaged answer.

The second benefit is optimizing the objective that we care about. In log-likelihood training, we're minimizing the cross-entropy between the produced tokens and tokens from the reference, but in machine translation, and many other NLP problems, we don't really care about log-likelihood: we want to maximize the BLEU score of the produced sequence. Unfortunately, the BLEU score is not differentiable, so we can't backpropagate on it. However, policy gradient methods, such as REINFORCE (from *Chapter 11*, *Policy Gradients – an Alternative*) work even when the reward is not differentiable: we just push up the probabilities of successful episodes and decrease the worse ones.

The third advantage we can exploit is in the fact that our sequence generation process is defined by us and we know its internals. By introducing stochasticity into the process of decoding, we can repeat the decoding process several times, gathering different decoding scenarios from the single training sample. This can be beneficial when our training dataset is limited, which is almost always the case, unless you're a Google or Facebook employee.

To understand how to switch our training from the log-likelihood objective to an RL scenario, let's look at both from the mathematical point of view. Log-likelihood estimation means maximizing the sum $\sum_{i=1}^{N} \log p_{model}(y_i|x_i)$ by tweaking the model's parameter, which is exactly the same as the minimization of **Kullback–Leibler** (**KL**) divergence between the data probability distribution and the probability distribution parameterized by the model, which could be written as a maximization of $\mathbb{E}_{x \sim p_{data}} \log p_{model}(x)$.

On the other hand, the REINFORCE method has the objective of maximizing $\mathbb{E}_{s \sim data, a \sim \pi(a|s)} Q(s, a) \log \pi(a|s)$. The connection is obvious and the difference between the two is just the scale before the logarithm and the way we're selecting actions (which are tokens in our dictionary).

In practice, REINFORCE for seq2seq training could be written as the following algorithm:

1. For every sample in the dataset, obtain the encoded representation, *E*, using the encoder RNN
2. Initialize the current token with the special begin token: *T* = '<BEG>'
3. Initialize the output sequence with the empty sequence: *Out* = []
4. While *T* != '<END>':
    - Get the probability distribution of the tokens and the new hidden state, passing the current token and the hidden state:
      p, *H* = Decoder(*T*, *E*)
    - Sample the output token, T_{out}, from the probability distribution
    - Remember the probability distribution, *p*
    - Append T_{out} to the output sequence: *Out* += T_{out}
    - Set the current token: $T \leftarrow T_{out}, E \leftarrow H$
5. Calculate BLEU or another metric between Out and the reference sequences:
    $Q = BLEU(Out, Out_{ref})$
6. Estimate the gradients: $\nabla J = \sum_{T} Q \nabla \log p(T)$
7. Update the model using SGD
8. Repeat until converged

Self-critical sequence training

The described approach, despite its positive sides, also has several complications. First of all, it's almost useless to train from scratch. Even for simple dialogues, the output sequence usually has at least five words, each taken from a dictionary of several thousand words. The number of different phrases of size five with a dictionary of 1,000 words equals 5^{1000}, which is slightly less than 10^{700}. So, the probability of obtaining the correct reply in the beginning of the training (when our weights for both the encoder and decoder are random) is negligibly small.

To overcome this, we can combine both log-likelihood and RL approaches and pretrain our model with the log-likelihood objective first (switching between teacher forcing and curriculum learning). After the model gets to some level of quality, we can switch to the REINFORCE method to fine-tune the model. In general, this could be seen as a uniform approach to complex RL problems, when a large action space makes it infeasible to start with a randomly behaving agent, as the chance of such an agent randomly reaching the goal is negligible. There is a lot of research happening around the incorporation of externally generated samples into the RL training process, and using log-likelihood pretraining on correct actions is one of the approaches.

Another issue with the vanilla REINFORCE method is the high variance of the gradients that we discussed in *Chapter 12*, *The Actor-Critic Method*. As you might remember, to solve the issue, we used the A2C method, which used the dedicated estimation of the state's value as a variance. We can apply the A2C method that way, of course, by extending our decoder with another head and returning BLEU score estimation given the decoded sequence, but there is a better approach. In the paper *Self-critical Sequence Training for Image Captioning* [1], published by *S. Rennie*, *E. Marcherett*, and others in 2016, a better baseline was proposed.

To obtain the baseline, the authors of the paper used the decoder in argmax mode to generate a sequence, which then was used to calculate the similarity metric, like BLEU, for example. Switching to argmax mode makes the decoder process fully deterministic and provides the baseline for the REINFORCE policy gradient in this formula:

$$\nabla J = \mathbb{E}\big[\big(Q(s) - b(s)\big)\nabla \log p(a|s)\big]$$

In the following section, we will implement and train a simple chatbot from the movie dialogues dataset.

Chatbot example

In the beginning of this chapter, we talked a bit about chatbots and NLP, so let's try to implement something simple using seq2seq and RL training. There are two large groups of chatbots: **entertainment human-mimicking** and **goal-oriented** chatbots. The first group is supposed to entertain a user by giving human-like replies to their phrases, without fully understanding them. The latter category is much harder to implement and is supposed to solve a user's problem, such as providing information, changing reservations, or switching on and off your home toaster.

Most of the latest efforts in the industry are focused on the goal-oriented group, but the problem is far from being fully solved yet. As this chapter is supposed to give a short example of the methods described, we will focus on training an entertainment bot using an online dataset with phrases extracted from movies.

Despite the simplicity of this problem, the example is large in terms of its code and the new concepts presented, so the whole code is not included in the book. We will focus only on the central modules responsible for model training and usage, but a lot of functions will be covered in an overview.

The example structure

The complete example is in the `Chapter14` folder and contains the following parts:

- `data`: A directory with the `get_data.sh` script to download and unpack the dataset that we will use in the example. The dataset archive is 10 MB, contains structured dialogues extracted from various sources, and is known as the **Cornell Movie-Dialogs Corpus**, which is available here: `https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html`.

- `libbots`: A directory with Python modules shared between various example's components. Those modules are described in the next section.

- `tests`: A directory with unit tests for library modules.

- The root folder contains two programs to train the model: `train_crossent.py`, which is used to train the model in the beginning, and `train_scst.py`, which is used to fine-tune the pretrained model using the REINFORCE algorithm.

- A script to display various statistics and data from the dataset: `cor_reader.py`.

- A script to apply the trained model to the dataset, displaying quality metrics: `data_test.py`.

- A script to use the model against a user-provided phrase: `use_model.py`.

- A bot for Telegram messenger, which uses the pretrained model: `telegram_bot.py`.

We will start with the data-related parts of the example and then look at both training scripts, before finishing by covering the model usage.

Modules: cornell.py and data.py

The two library modules working with the dataset used to train the model are `cornell.py` and `data.py`. Both are related to data processing and are used to transform the dataset into a form suitable for training, but working on different layers.

The `cornell.py` file includes low-level functions to parse data in the Cornell Movie-Dialogs Corpus format and represent it in a form suitable for later processing. The main goal of the module is to load a list of dialogues from movies. As the dataset contains metadata about the movies, we can filter the dialogues to be loaded by various criteria, but only a genre filter is implemented. In the list of dialogues returned, every dialogue is represented as a list of phrases and every phrase is a list of lowercase words (tokens). For example, a phrase could be `["hi", "!", "how", "are", "you", "?"]`.

The conversion of sentences to the list of tokens is called *tokenization* in NLP, and even this step by itself can be a tricky process, as you need to handle punctuation, abbreviations, quotes, apostrophes, and other natural language specifics. Luckily, the `nltk` library includes several tokenizers, so going from a sentence to a list of tokens is just a matter of calling the appropriate function, which significantly simplifies our task. The main function in `cornell.py` used outside of it is the function `load_dialogues()`, which is supposed to load dialogue data with an optional genre filter.

The `data.py` module works on a higher level and doesn't include any dataset-specific knowledge. It provides the following functionality, which is used almost everywhere in the example:

- Working with mappings from tokens in their integer IDs: saving and loading from the file (the `save_emb_dict()` and `load_emb_dict()` functions), encoding the list of tokens into a list of IDs (`encode_words()`), decoding the list of integer IDs into tokens (`decode_words()`), and generating the dictionary mapping from the training data (`phrase_pairs_dict()`)

- Working with the training data: iterating batches of a given size (`iterate_batches()`) and splitting data into training/testing parts (`split_train_test()`)

- Loading dialogue data and converting it into phrase-reply pairs suitable for training: the `load_data()` function

On data loading and dictionary creation, we also add special tokens with the predefined IDs:

- A token for unknown words, `#UNK`, used for all out-of-dictionary tokens
- A token for the beginning of the sequence, `#BEG`, prepended to all sequences
- A token for the end of the sequence, `#END`

Besides the optional genre filter, which could be used to limit data size during experiments, several other filters are applied to the loaded data. The first filter limits the maximum number of tokens in training pairs. RNN training can be expensive in terms of the number of operations and memory usage, so I left only the training pairs with 20 tokens for the first and the second training entry. This also helps for convergence speed, as the variability of dialogues with short sentences is much less, so it's easier for our RNN to train on this data. This also has a downside of our model producing only short replies.

The second filter applied to the data is related to the dictionary. The number of words in the dictionary has a significant influence on the performance and graphics processing unit (GPU) memory needed, as both our embeddings matrix (keeping embeddings vectors for every `dict` token) and decoder output projection matrix (which converts the decoder RNN's output into the probability distribution) have dictionary size as one of their dimensions. So, by reducing the number of words in our dictionary, we can reduce memory and improve the training speed. To get this during the data loading, we calculate the count of occurrences for every word in the dictionary and map all words that have been encountered fewer than 10 times to an unknown token. Later, all the training pairs with an unknown token are removed from the training set.

BLEU score and utils.py

The `nltk` library is used to calculate the BLEU score, but to make the BLEU calculation a bit more convenient, two wrapper functions are implemented: `calc_bleu(candidate_seq, reference_seq)`, which calculates the score when we have one candidate and one reference sequence, and `calc_bleu_many(candidate_seq, reference_sequences)`, which is used to get the score when we have several reference sequences to compare against our candidate. In the case of several candidates, the best BLEU score is calculated and returned.

Also, to reflect the short phrases in our dataset, BLEU is calculated for unigrams and bigrams only. The following is the code of the `utils.py` module, which is responsible for BLEU calculations and an extra two functions used to tokenize the sentences and convert the list of tokens back into a string:

```
import string
from nltk.translate import bleu_score
from nltk.tokenize import TweetTokenizer


def calc_bleu_many(cand_seq, ref_sequences):
    sf = bleu_score.SmoothingFunction()
    return bleu_score.sentence_bleu(ref_sequences, cand_seq,
                                    smoothing_function=sf.method1,
                                    weights=(0.5, 0.5))


def calc_bleu(cand_seq, ref_seq):
    return calc_bleu_many(cand_seq, [ref_seq])


def tokenize(s):
    return TweetTokenizer(preserve_case=False).tokenize(s)


def untokenize(words):
    to_pad = lambda t: not t.startswith("'") and \
                       t not in string.punctuation
    return "".join([
        (" " + i) if to_pad(i) else i
        for i in words
    ]).strip()
```

Model

The functions related to the training process and the model itself are defined in the `libbots/model.py` file. It's important for understanding the training process, so the code with comments is shown as follows:

```
HIDDEN_STATE_SIZE = 512
EMBEDDING_DIM = 50
```

The first hyperparameter (`HIDDEN_STATE_SIZE`) defines the size of the hidden state used by both the encoder and decoder RNNs. In the PyTorch implementation of RNNs, this value defines three parameters at once:

- The dimension of the hidden state expected on the input and returned as the output of the RNN unit.

- The dimension of the output returned from the RNN. Despite being the same dimension, the output of the RNN is different from the hidden state.

- The internal count of neurons used for RNN transformation.

The second hyperparameter, `EMBEDDING_DIM`, defines the dimensionality of our embeddings, which is a set of vectors used to represent every token in our dictionary. For this example, we're not using the pretrained embeddings, like GloVe or word2vec, but we will train them alongside the model. As our encoder and decoder both accept tokens on the input, this dimensionality of embeddings also defines the size of the RNN's input.

```python
class PhraseModel(nn.Module):
    def __init__(self, emb_size, dict_size, hid_size):
        super(PhraseModel, self).__init__()

        self.emb = nn.Embedding(
            num_embeddings=dict_size, embedding_dim=emb_size)
        self.encoder = nn.LSTM(
            input_size=emb_size, hidden_size=hid_size,
            num_layers=1, batch_first=True)
        self.decoder = nn.LSTM(
            input_size=emb_size, hidden_size=hid_size,
            num_layers=1, batch_first=True)
        self.output = nn.Linear(hid_size, dict_size)
```

In the model's constructor, we create embedding, encoder, decoder, and output projection components. As with RNN implementation, long short-term memory (LSTM) is used. Internally, LSTM has a more complicated structure than the simple RNN layer we've discussed, but the idea is the same: we have the input, output, input state, and output state. The `batch_first` argument specifies that the batch will be provided as the first dimension of the input tensor to the RNN. The projection layer is a linear transformation, which converts the output from the decoder into the dictionary probability distribution.

The rest of the model consists of methods that are used to perform different transformations of the data using our seq2seq model. Strictly speaking, this class breaks PyTorch convention to override the `forward` method to apply the network to the data. This is intentional, to emphasize the fact that the seq2seq model can't be interpreted as a single transformation of input data to output. In our example, we will use the model in different ways, for example, processing the target sequence in teacher-forcing mode, decoding the sequence one by one using argmax, or performing one single decoding step.

In *Chapter 3, Deep Learning with PyTorch*, it was mentioned that subclasses of the `nn.Module` have to override the `forward()` method and use the call convention to transform the data with the custom `nn.Module`. In our implementation, we're violating this rule, but in current versions of PyTorch, this is fine, as the `__call__()` method of `nn.Module` is responsible for calling the PyTorch hooks we're not using.

But still, this is a violation of PyTorch conventions. We're doing so for our convenience.

```
def encode(self, x):
    _, hid = self.encoder(x)
    return hid
```

The preceding method performs the simplest operation in our model: it encodes the input sequence and returns the hidden state from the last step of the encoder RNN. In PyTorch, all RNN classes return the tuple of two objects as a result. The first component of the tuple is the output of the RNN for every application of the RNN, and the second is the hidden state from the last item in the input sequence. We're not interested in the encoder's output, so we just return the hidden state.

```
def get_encoded_item(self, encoded, index):
    # For RNN
    # return encoded[:, index:index+1]
    # For LSTM
    return encoded[0][:, index:index+1].contiguous(),\
            encoded[1][:, index:index+1].contiguous()
```

The preceding function is a utility method used to get access to the hidden state of the individual component of the input batch. It's required because we're encoding the whole batch of sequences in one call (using the `encode()` method), but decoding is performed for every batch sequence individually. This method is used to extract the hidden state of a specific element of the batch (given in the `index` argument). The details of this extraction are RNN implementation-dependent. LSTM, for example, has the hidden state represented as a tuple of two tensors: the cell state and the hidden state. However, a simple RNN implementation, like the vanilla `torch.nn.RNN` class or the more complicated `torch.nn.GRU`, has the hidden state as a single tensor. So, this knowledge is encapsulated in this method, which should be adjusted if you switch the encoder and decoder underlying the RNN type.

The rest of the methods are solely related to the decoding process in its different forms. Let's start with the teacher-forcing mode:

```
def decode_teacher(self, hid, input_seq):
    # Method assumes batch of size=1
    out, _ = self.decoder(input_seq, hid)
    out = self.output(out.data)
    return out
```

This is the simplest and most efficient way to perform decoding. In this mode, we simply apply the decoder RNN to the reference sequence (reply phrase of the training sample).

In teacher-forcing mode, the input for every step is known in advance; the only dependency that the RNN has between steps is its hidden state, which allows us to perform RNN transformation very efficiently, without transferring data from and to the GPU, implemented in the underlying CuDNN library.

This is not the case for other decoding methods, when the output of every decoder step defines the input for the next step. This connection between output and input is done in Python code, so decoding is performed step by step, not necessarily transferring the data (as all our tensors are already in GPU memory). However, the control is defined by Python code and not by the highly optimized CuDNN library.

```python
def decode_one(self, hid, input_x):
    out, new_hid = self.decoder(input_x.unsqueeze(0), hid)
    out = self.output(out)
    return out.squeeze(dim=0), new_hid
```

The preceding method performs one single decoding step for one example. We pass the hidden state for the decoder (which is set to the encoded sequence at the first step) and input the tensor with the embeddings vector for the input token. Then, the result from the decoder is passed through the output projection to obtain the raw scores for every token in the dictionary. It's not a probability distribution, as we don't pass the output through the softmax function, just the raw scores (also called logits). The result of the function is those logits and the new hidden state returned by the decoder.

```python
def decode_chain_argmax(self, hid, begin_emb, seq_len,
                        stop_at_token=None):
    res_logits = []
    res_tokens = []
    cur_emb = begin_emb
```

The `decode_chain_argmax()` method performs decoding of an encoded sequence, using `argmax` as a transition from the probability distribution to a token index produced. The function arguments are as follows:

- `hid`: The hidden state returned by the encoder for the input sequence.
- `begin_emb`: The embedding vector for the #BEG token used to start decoding.
- `seq_len`: The maximum length of the decoded sequence. The resulting sequence could be shorter if the decoder returns the #END token, but could never be longer. It helps to stop decoding when the decoder starts to repeat itself infinitely, which might happen at the beginning of training.
- `stop_at_token`: An optional token ID (normally #END token) that stops the decoding process.

This function is supposed to return two values: a tensor with resulting logits returned by the decoder on every step and the list of token IDs produced. The first value is used for training, as we need the output tensors to calculate the loss, while the second value is passed to the quality metric function, which is the BLEU score in this case.

```
        for _ in range(seq_len):
            out_logits, hid = self.decode_one(hid, cur_emb)
            out_token_v = torch.max(out_logits, dim=1)[1]
            out_token = out_token_v.data.cpu().numpy()[0]

            cur_emb = self.emb(out_token_v)

            res_logits.append(out_logits)
            res_tokens.append(out_token)
            if stop_at_token is not None:
                if out_token == stop_at_token:
                    break
        return torch.cat(res_logits), res_tokens
```

At every decoding loop iteration, we apply the decoder RNN to one single token, passing the current hidden state for the decoder (in the beginning, it equals the encoded vector) and the embedding vector for the current token. The output from the decoder RNN is a tuple with logits (unnormalized probabilities for every word in the dictionary) and the new hidden state. To go from logits to the decoded token ID, we use the `argmax` function as the name of the method. Then, we obtain embeddings for the decoded token, save the logits and token ID in the resulting lists, and check for stop conditions.

```
    def decode_chain_sampling(self, hid, begin_emb, seq_len,
                              stop_at_token=None):
        res_logits = []
        res_actions = []
        cur_emb = begin_emb

        for _ in range(seq_len):
            out_logits, hid = self.decode_one(hid, cur_emb)
            out_probs_v = F.softmax(out_logits, dim=1)
            out_probs = out_probs_v.data.cpu().numpy()[0]
            action = int(np.random.choice(
                out_probs.shape[0], p=out_probs))
            action_v = torch.LongTensor([action])
            action_v = action_v.to(begin_emb.device)
            cur_emb = self.emb(action_v)
```

```
            res_logits.append(out_logits)
            res_actions.append(action)
            if stop_at_token is not None:
                if action == stop_at_token:
                    break
        return torch.cat(res_logits), res_actions
```

The next and the last function to perform decoding of the sequence does almost the same as `decode_chain_argmax()`, but, instead of argmax, it performs the random sampling from the returned probability distribution. The rest of the logic is the same.

Besides the `PhraseModel` class, the `model.py` file contains several functions used to prepare the input to the model, which has to be in a tensor form for PyTorch RNN machinery to work properly.

```
def pack_batch_no_out(batch, embeddings, device="cpu"):
    assert isinstance(batch, list)
    # Sort descending (CuDNN requirements)
    batch.sort(key=lambda s: len(s[0]), reverse=True)
    input_idx, output_idx = zip(*batch)
```

This function packs the input batch (which is a list of `(phrase, replay)` tuples) into a form suitable for encoding and `decode_chain_*` functions. As the first step, we sort the batch by the first phrase's length in decreasing order. This is a requirement of the CuDNN library used by PyTorch as a CUDA backend.

```
    # create padded matrix of inputs
    lens = list(map(len, input_idx))
    input_mat = np.zeros((len(batch), lens[0]), dtype=np.int64)
    for idx, x in enumerate(input_idx):
        input_mat[idx, :len(x)] = x
```

Then we create a matrix with `[batch, max_input_phrase]` dimensions and copy our input phrases there. This form is called **padded sequence**, because our sequences of variable length are padded with zeros to the longest sequence.

```
    input_v = torch.tensor(input_mat).to(device)
    input_seq = rnn_utils.pack_padded_sequence(
        input_v, lens, batch_first=True)
```

As the next step, we wrap this matrix into a PyTorch tensor and use a special function from the PyTorch RNN module to convert this matrix from the padded form into the so-called **packed form**. In the packed form, our sequences are stored column-wise (that is, in a transposed form), keeping the length of every column.

For example, in the first row, we have all first tokens from all sequences; in the second row, we have tokens from the second position for sequences longer than 1; and so on. This representation allows CuDNN to perform RNN processing very efficiently, handling our batch of sequences at once.

```
# lookup embeddings
r = embeddings(input_seq.data)
emb_input_seq = rnn_utils.PackedSequence(
    r, input_seq.batch_sizes)
return emb_input_seq, input_idx, output_idx
```

At the end of the function, we convert our data from the integer token IDs into embeddings, which can be done in one step, as our token IDs have already been packed into the tensor. Then we return the resulting tuple with three items: the packed sequence to be passed to the encoder, and two lists of lists with the integer token IDs for the input and output sequences.

```
def pack_input(input_data, embeddings, device="cpu"):
    input_v = torch.LongTensor([input_data]).to(device)
    r = embeddings(input_v)
    return rnn_utils.pack_padded_sequence(
        r, [len(input_data)], batch_first=True)
```

The preceding function is used to convert the encoded phrase (as the list of token IDs) into the packed sequence suitable to pass to the RNN.

```
def pack_batch(batch, embeddings, device="cpu"):
    emb_input_seq, input_idx, output_idx = pack_batch_no_out(
        batch, embeddings, device)

    # prepare output sequences, with end token stripped
    output_seq_list = []
    for out in output_idx:
        s = pack_input(out[:-1], embeddings, device)
        output_seq_list.append(s)
    return emb_input_seq, output_seq_list, input_idx, output_idx
```

The next function uses the `pack_batch_no_out()` method, but, in addition, it converts the output indices into the list of packed sequences to be used in the teacher-forcing mode of training. Those sequences have the #END token stripped.

```
def seq_bleu(model_out, ref_seq):
    model_seq = torch.max(model_out.data, dim=1)[1]
    model_seq = model_seq.cpu().numpy()
    return utils.calc_bleu(model_seq, ref_seq)
```

Finally, the last function in `model.py`, which calculates the BLEU score from the tensor with logits, is produced by the decoder in teacher-forcing mode. Its logic is simple: it just calls argmax to obtain sequence indices and then uses the BLEU calculation function from the `utils.py` module.

Dataset exploration

It's always a good idea to look at your dataset from various angles, like counting statistics, plotting various characteristics of data, or just eyeballing your data to get a better understanding of your problem and potential issues. The tool `cor_reader.py` supports the minimalistic functionality for data analysis. By running it with the `--show-genres` option, you will get all genres from the dataset with a number of movies in each, sorted by the count of movies in order of decreasing size. The top 10 of them are shown as follows:

```
$ ./cor_reader.py --show-genres
Genres:
drama: 320
thriller: 269
action: 168
comedy: 162
crime: 147
romance: 132
sci-fi: 120
adventure: 116
mystery: 102
horror: 99
```

The `--show-dials` option displays dialogues from the movies without any preprocessing, in the order they appear in the database. The number of dialogues is large, so it's worth passing the `-g` option to filter by genre. For example, let's look at two dialogues from the comedy movies:

```
$ ./cor_reader.py -g comedy --show-dials | head -n 10
Dialog 0 with 4 phrases:
can we make this quick?
roxanne korrine and andrew barrett are having an incredibly horrendous
public break - up on the quad . again .
well , i thought we'd start with pronunciation , if that's okay with you
.
```

```
not the hacking and gagging and spitting part . please .
```

```
okay ... then how ' bout we try out some french cuisine . saturday ?
night ?
```

```
Dialog 1 with two phrases:
```

```
you're asking me out . that's so cute . what's your name again ? forget
it .
```

By passing the `--show-train` option, you can check the training pairs, grouped by the first phrase and sorted in descending order by the count of replies. This data already has the frequency of words (at least 10 occurrences) and phrase length (at most 20 tokens) filters applied. The following is a part of the output for the family genre:

```
$ ./cor_reader.py -g family --show-train | head -n 20
Training pairs (558 total)
0: #BEG yes . #END
 : #BEG but you will not ... be safe ... #END
 : #BEG oh ... oh well then , one more won't matter . #END
 : #BEG vada you've gotta stop this , there's absolutely nothing wrong
with you ! #END
 : #BEG good . #END
 : #BEG he's getting big . vada , come here and sit down for a minute .
#END
 : #BEG who's that with your dad ? #END
 : #BEG for this . #END
 : #BEG didn't i tell you ? i'm always right , you know , my dear ...
aren't i ? #END
 : #BEG oh , i hope we got them in time . #END
 : #BEG oh - - now look at him ! this is terrible ! #END
1: #BEG no . #END
 : #BEG were they pretty ? #END
 : #BEG it's there . #END
 : #BEG why do you think she says that ? #END
 : #BEG come here , sit down . #END
 : #BEG what's wrong with your eyes ? #END
 : #BEG maybe we should , just to see what's the big deal . #END
 : #BEG why not ? #END
```

As you can see, even in a small subset of the data, there are phrases with multiple reply candidates. The last option, supported by `cor_reader.py`, is `--show-dict-freq`, which counts the frequency of words and shows them sorted by the count of occurrences:

```
$ ./cor_reader.py -g family --show-dict-freq | head -n 10
Frequency stats for 772 tokens in the dict
.: 1907
,: 1175
?: 1148
you: 840
!: 758
i: 653
-: 578
the: 506
a: 414
```

Training: cross-entropy

To train the first approximation of the model, the cross-entropy method is used and implemented in `train_crossent.py`. During the training, we randomly switch between the teacher-forcing mode (when we give the target sequence on the decoder's input) and argmax chain decoding (when we decode the sequence one step at a time, choosing the token with the highest probability in the output distribution). The decision between those two training modes is taken randomly with the fixed probability of 50%. This allows for combining the characteristics of both methods: fast convergence from teacher forcing and stable decoding from curriculum learning.

Implementation

What follows is the implementation of the cross-entropy method training from `train_crossent.py`.

```
SAVES_DIR = "saves"

BATCH_SIZE = 32
LEARNING_RATE = 1e-3
MAX_EPOCHES = 100

log = logging.getLogger("train")

TEACHER_PROB = 0.5
```

In the beginning, we define hyperparameters specific to the cross-entropy method training step. The value of TEACHER_PROB defines the probability of teacher-forcing training being chosen randomly for every training sample.

```
def run_test(test_data, net, end_token, device="cpu"):
    bleu_sum = 0.0
    bleu_count = 0
    for p1, p2 in test_data:
        input_seq = model.pack_input(p1, net.emb, device)
        enc = net.encode(input_seq)
        _, tokens = net.decode_chain_argmax(
            enc, input_seq.data[0:1], seq_len=data.MAX_TOKENS,
            stop_at_token=end_token)
        bleu_sum += utils.calc_bleu(tokens, p2[1:])
        bleu_count += 1
    return bleu_sum / bleu_count
```

The run_test method is called every epoch to calculate the mean BLEU score for the held-out test dataset, which is 5% of loaded data by default.

```
if __name__ == "__main__":
    fmt = "%(asctime)-15s %(levelname)s %(message)s"
    logging.basicConfig(format=fmt, level=logging.INFO)
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--data", required=True,
        help="Category to use for training. Empty "
            "string to train on full dataset")
    parser.add_argument(
        "--cuda", action='store_true', default=False,
        help="Enable cuda")
    parser.add_argument(
        "-n", "--name", required=True, help="Name of the run")
    args = parser.parse_args()
    device = torch.device("cuda" if args.cuda else "cpu")

    saves_path = os.path.join(SAVES_DIR, args.name)
    os.makedirs(saves_path, exist_ok=True)
```

The program allows specifying the genre of films that we want to train on and the name of the current training, which is used in a TensorBoard comment and as a name of the directory for the periodical model checkpoints.

```
    phrase_pairs, emb_dict = data.load_data(
        genre_filter=args.data)
    log.info("Obtained %d phrase pairs with %d uniq words",
```

```
            len(phrase_pairs), len(emb_dict))
        data.save_emb_dict(saves_path, emb_dict)
        end_token = emb_dict[data.END_TOKEN]
        train_data = data.encode_phrase_pairs(phrase_pairs, emb_dict)
```

When the arguments have been parsed, we load the dataset for the provided genre, save the embeddings dictionary (which is a mapping from the token's string to the integer ID of the token), and encode the phrase pairs. At this point, our data is a list of tuples with two entries, and every entry is a list of token integer IDs.

```
        rand = np.random.RandomState(data.SHUFFLE_SEED)
        rand.shuffle(train_data)
        log.info("Training data converted, got %d samples",
                len(train_data))
        train_data, test_data = data.split_train_test(train_data)
        log.info("Train set has %d phrases, test %d",
                len(train_data), len(test_data))
```

After the data has been loaded, we split it into train/test parts, and then we shuffle the data using a fixed random seed (to be able to repeat the same shuffle at the RL training stage).

```
        net = model.PhraseModel(
            emb_size=model.EMBEDDING_DIM, dict_size=len(emb_dict),
            hid_size=model.HIDDEN_STATE_SIZE).to(device)
        log.info("Model: %s", net)
        writer = SummaryWriter(comment="-" + args.name)
        optimiser = optim.Adam(net.parameters(), lr=LEARNING_RATE)
        best_bleu = None
```

Then, we create the model, passing it the dimensionality of the embeddings, the size of the dictionary, and the hidden size of the encoder and decoder.

```
        for epoch in range(MAX_EPOCHES):
            losses = []
            bleu_sum = 0.0
            bleu_count = 0
            for batch in data.iterate_batches(train_data, BATCH_SIZE):
                optimiser.zero_grad()
                input_seq, out_seq_list, _, out_idx = \
                    model.pack_batch(batch, net.emb, device)
                enc = net.encode(input_seq)
```

Our training loop performs the fixed number of epochs, and each of them is an iteration over the batches of pairs of the encoded phrases.

For every batch, we pack it using `model.pack_batch()`, which returns the packed input sequence, packed output sequence, and two lists of input and output token indices of the batch. To get the encoded representation for every input sequence in the batch, we call `net.encode()`, which just passes the input sequence through our encoder and returns the hidden state from the last RNN application. This hidden state has a shape of (`batch_size`, `model.HIDDEN_STATE_SIZE`), which is (16, 512) by default.

```
net_results = []
net_targets = []
for idx, out_seq in enumerate(out_seq_list):
    ref_indices = out_idx[idx][1:]
    enc_item = net.get_encoded_item(enc, idx)
```

Then, we decode every sequence in our batch individually. Maybe it is possible to parallelize this loop somehow, but it will make the example less readable. For every sequence in the batch, we get a reference sequence of token IDs (without training the #BEG token) and the encoded representation of the input sequence created by the encoder.

```
if random.random() < TEACHER_PROB:
    r = net.decode_teacher(enc_item, out_seq)
    bleu_sum += model.seq_bleu(r, ref_indices)
else:
    r, seq = net.decode_chain_argmax(
        enc_item, out_seq.data[0:1],
        len(ref_indices))
    bleu_sum += utils.calc_bleu(seq, ref_indices)
```

In the preceding code, we randomly decide which method of decoding to use: teacher forcing or curriculum learning. They differ only in the model's method called and the way we compute the BLEU score. For teacher-forcing mode, the `decode_teacher()` method returns the tensor of logits of size `[out_seq_len, dict_size]`, so to calculate the BLEU score, we need to use the function from the `model.py` module. In the case of curriculum learning, implemented by the `decode_chain_argmax()` method, it returns both a logits tensor and list of token IDs of the output sequence. This allows us to calculate the BLEU score directly.

```
net_results.append(r)
net_targets.extend(ref_indices)
bleu_count += 1
```

At the end of sequence processing, we append the resulting logits and reference indices to be used later in the loss calculation.

```
results_v = torch.cat(net_results)
```

```
        targets_v = torch.LongTensor(net_targets).to(device)
        loss_v = F.cross_entropy(results_v, targets_v)
        loss_v.backward()
        optimiser.step()
        losses.append(loss_v.item())
```

To calculate the cross-entropy loss, we convert the list of logits tensors into one single tensor and convert the list with reference token IDs into a PyTorch tensor, putting it in GPU memory. Then, we just need to calculate the cross-entropy loss, perform the backpropagation, and ask the optimizer to adjust the model. This ends the processing of one batch.

```
    bleu = bleu_sum / bleu_count
    bleu_test = run_test(test_data, net, end_token, device)
    log.info("Epoch %d: mean loss %.3f, mean BLEU %.3f, "
             "test BLEU %.3f", epoch, np.mean(losses),
             bleu, bleu_test)
    writer.add_scalar("loss", np.mean(losses), epoch)
    writer.add_scalar("bleu", bleu, epoch)
    writer.add_scalar("bleu_test", bleu_test, epoch)
```

When all the batches have been processed, we calculate the mean BLEU score from the training, run a test on the held-out dataset, and report our metrics.

```
    if best_bleu is None or best_bleu < bleu_test:
        if best_bleu is not None:
            out_name = os.path.join(
                saves_path, "pre_bleu_%.3f_%02d.dat" % (
                    bleu_test, epoch))
            torch.save(net.state_dict(), out_name)
            log.info("Best BLEU updated %.3f", bleu_test)
        best_bleu = bleu_test

    if epoch % 10 == 0:
        out_name = os.path.join(
            saves_path, "epoch_%03d_%.3f_%.3f.dat" % (
                epoch, bleu, bleu_test))
        torch.save(net.state_dict(), out_name)
```

To be able to fine-tune the model, we save the model's weights with the best test BLEU score seen so far. We also save the checkpoint file every 10 iterations.

Results

That's it for our training code. To start it, you need to pass the name of the run in the command line and provide the genre filter. The complete dataset is quite large (617 movies in total) and may require lots of time to train, even on a GPU. For example, on the GTX 1080 Ti, every epoch takes about 16 minutes, which is 18 hours for 100 epochs.

By applying the genre filter, you can train on a subset of the movies; for example, the genre *comedy* includes 159 movies, bringing us 22,000 training phrase pairs, which is smaller than the 150,000 phrase pairs from the complete dataset. The dictionary size with the *comedy* filter is also much smaller (4,905 words versus 11,131 words in the complete data). This decreases the epoch time from 16 minutes to three minutes. To make the training set even smaller, you can use the *family* genre, which has only 16 movies with 3,000 phrase pairs and 772 words. In this case, 100 epochs take only 30 minutes.

As an example, however, here's how to start training for the *comedy* genre. This process writes checkpoints into the directory `saves/crossent-comedy`, while TensorBoard metrics are written in the `runs` directory.

```
$ ./train_crossent.py --cuda --data comedy -n crossent-comedy
2019-11-14 21:39:38,670 INFO Loaded 159 movies with genre comedy
2019-11-14 21:39:38,670 INFO Read and tokenise phrases...
2019-11-14 21:39:42,496 INFO Loaded 93039 phrases
2019-11-14 21:39:42,714 INFO Loaded 24716 dialogues with 93039 phrases,
generating training pairs
2019-11-14 21:39:42,732 INFO Counting freq of words...
2019-11-14 21:39:42,924 INFO Data has 31774 uniq words, 4913 of them
occur more than 10
2019-11-14 21:39:43,054 INFO Obtained 47644 phrase pairs with 4905 uniq
words
2019-11-14 21:39:43,291 INFO Training data converted, got 26491 samples
2019-11-14 21:39:43,291 INFO Train set has 25166 phrases, test 1325
2019-11-14 21:39:45,874 INFO Model: PhraseModel(
  (emb): Embedding(4905, 50)
  (encoder): LSTM(50, 512, batch_first=True)
  (decoder): LSTM(50, 512, batch_first=True)
  (output): Linear(in_features=512, out_features=4905, bias=True)
)
2019-11-14 21:41:30,358 INFO Epoch 0: mean loss 4.992, mean BLEU 0.161,
```

```
test BLEU 0.067
2019-11-14 21:43:15,275 INFO Epoch 1: mean loss 4.674, mean BLEU 0.168,
test BLEU 0.074
2019-11-14 21:43:15,286 INFO Best BLEU updated 0.074
2019-11-14 21:45:01,627 INFO Epoch 2: mean loss 4.548, mean BLEU 0.176,
test BLEU 0.092
2019-11-14 21:45:01,637 INFO Best BLEU updated 0.092
```

For the cross-entropy training, there are three metrics written into the TensorBoard: loss, the train BLEU score, and the test BLEU score. The following are the plots for the *comedy* genre, which took about three hours to train.
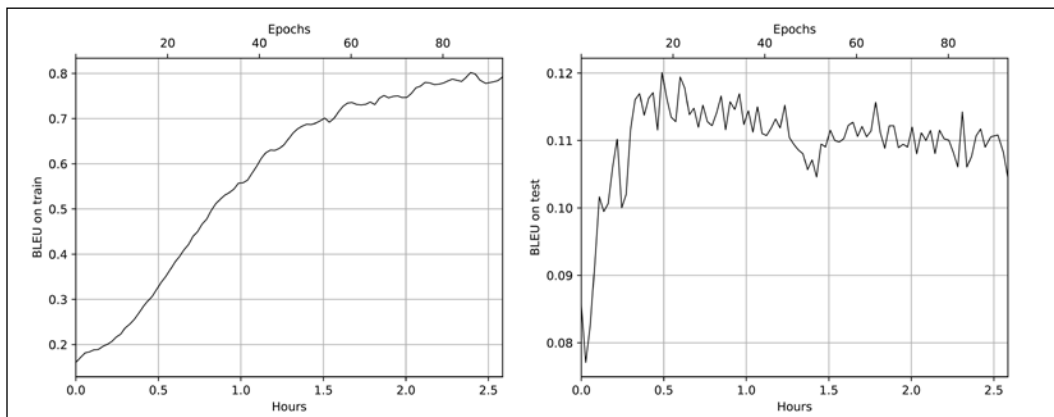
Figure 14.7: BLEU scores on the train (left) and test (right) datasets during training
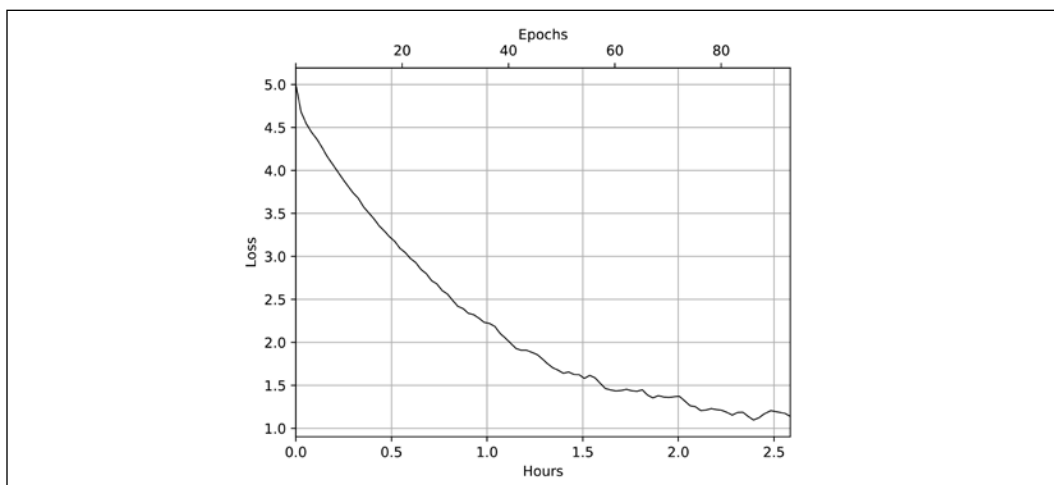
Figure 14.8: Loss during training the cross-entropy objective on the *comedy* genre

As you can see, the BLEU score for the training data is consistently growing, saturating around 0.83, but the BLEU score for the test dataset stopped improving after the 25th epoch and is much less impressive than the training data BLEU score. There are two reasons for this. First of all, our dataset is not large and representative enough for our training process to generalize reply phrases to get a good score on the test dataset. In the *comedy* genre, we have 25,166 training pairs and 1,325 testing pairs, so there is a high chance of testing pairs containing phrases that are totally new and not related to the training pairs. This happens due to the high variability of the dialogues that we have. We will take a look at our data in the next section.

The second possible reason for the low test BLEU score could be the fact that cross-entropy training doesn't take into account phrases with several possible replies. As you will see in the next section, our data contains phrases with several alternatives for replies. Cross-entropy tries to find the model's weights, which will produce output sequences matching the desired output, but if your desired output is random, there is not much that the model can do about that.

Another reason for the low test score could be lack of proper regularization in the model, which should help to prevent overfitting. This is left as an exercise for you to check the effect.

Training: SCST

As we've already discussed, RL training methods applied to the seq2seq problem can potentially improve the final model. The main reasons are:

- Better handling of multiple target sequences. For example, *hi* could be replied with *hi*, *hello*, *not interested*, or something else. The RL point of view is to treat our decoder as a process of selecting actions when every action is a token to be generated, which fits better to the problem.

- Optimizing the BLEU score directly instead of cross-entropy loss. Using the BLEU score for the generated sequence as a gradient scale, we can push our model toward the successful sequences and decrease the probability of unsuccessful ones.

- By repeating the decoding process, we can generate more episodes to train on, which will lead to better gradient estimation.

- Additionally, using the self-critical sequence training approach, we can get the baseline almost for free, without increasing the complexity of our model, which could improve the convergence even more.

All this looks quite promising, so let's check it.

Implementation

RL training is implemented as a separate training step in the `train_scst.py` tool. It requires the model file saved by `train_crossent.py` to be passed in a command line.

```
SAVES_DIR = "saves"

BATCH_SIZE = 16
LEARNING_RATE = 5e-4
MAX_EPOCHES = 10000
```

As usual, we start with hyperparameters (imports were omitted). This training script has the same set of hyperparameters; the only difference is a smaller batch size, as the GPU memory requirements for SCST are higher and have a smaller learning rate.

```
log = logging.getLogger("train")


def run_test(test_data, net, end_token, device="cpu"):
    bleu_sum = 0.0
    bleu_count = 0
    for p1, p2 in test_data:
        input_seq = model.pack_input(p1, net.emb, device)
        enc = net.encode(input_seq)
        _, tokens = net.decode_chain_argmax(
            enc, input_seq.data[0:1], seq_len=data.MAX_TOKENS,
            stop_at_token=end_token)
        ref_indices = [
            indices[1:]
            for indices in p2
        ]
        bleu_sum += utils.calc_bleu_many(tokens, ref_indices)
        bleu_count += 1
    return bleu_sum / bleu_count
```

The preceding is the function, which is run every epoch to calculate the BLEU score on the test dataset. It is almost the same as in `train_crossent.py`, as the only difference is in the test data, which is now grouped by the first phrase. So, the shape of the data is now `[(first_phrase, [second_phrases])]`. As before, we need to strip the #BEG token from every second phrase, but the BLEU score is now calculated by another function, which accepts several reference sequences and returns the best score from them.

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    fmt = "%(asctime)-15s %(levelname)s %(message)s"
    logging.basicConfig(format=fmt, level=logging.INFO)
    parser.add_argument(
        "--data", required=True,
        help="Category to use for training. Empty "
            "string to train on full dataset")
    parser.add_argument(
        "--cuda", action='store_true', default=False,
        help="Enable cuda")
    parser.add_argument(
        "-n", "--name", required=True,
        help="Name of the run")
    parser.add_argument(
        "-l", "--load", required=True,
        help="Load model and continue in RL mode")
    parser.add_argument(
        "--samples", type=int, default=4,
        help="Count of samples in prob mode")
    parser.add_argument(
        "--disable-skip", default=False, action='store_true',
        help="Disable skipping of samples with high argmax BLEU")
    args = parser.parse_args()
    device = torch.device("cuda" if args.cuda else "cpu")
```

The tool now accepts three new command-line arguments: option `-l` is passed to provide the file name with the model to load, while option `--samples` is used to change the number of decoding iterations performed for every training sample. Using more samples leads to more accurate policy gradient estimation but increases GPU memory requirements. The last new option, `--disable-skip`, could be used to disable the skipping of training samples with a high BLEU score (by default, the threshold is 0.99). This skipping functionality significantly increases the training speed, as we train only on training samples with bad sequences generated in argmax mode, but my experiments have shown that disabling this skipping leads to better model quality.

```
    saves_path = os.path.join(SAVES_DIR, args.name)
    os.makedirs(saves_path, exist_ok=True)

    phrase_pairs, emb_dict = \
        data.load_data(genre_filter=args.data)
    log.info("Obtained %d phrase pairs with %d uniq words",
            len(phrase_pairs), len(emb_dict))
    data.save_emb_dict(saves_path, emb_dict)
```

```
end_token = emb_dict[data.END_TOKEN]
train_data = data.encode_phrase_pairs(phrase_pairs, emb_dict)
rand = np.random.RandomState(data.SHUFFLE_SEED)
rand.shuffle(train_data)
train_data, test_data = data.split_train_test(train_data)
log.info("Training data converted, got %d samples",
         len(train_data))
```

Then, we load the training data the same way we did in the cross-entropy training. The extra code is in the following two lines, which are used to group the training data by the first phrase.

```
train_data = data.group_train_data(train_data)
test_data = data.group_train_data(test_data)
log.info("Train set has %d phrases, test %d",
         len(train_data), len(test_data))

rev_emb_dict = {idx: word for word, idx in emb_dict.items()}

net = model.PhraseModel(
    emb_size=model.EMBEDDING_DIM, dict_size=len(emb_dict),
    hid_size=model.HIDDEN_STATE_SIZE).to(device)
log.info("Model: %s", net)

writer = SummaryWriter(comment="-" + args.name)
net.load_state_dict(torch.load(args.load))
log.info("Model loaded from %s, continue "
         "training in RL mode...", args.load)
```

When the data is loaded, we create the model and load its weights from the given file.

```
beg_token = torch.LongTensor([emb_dict[data.BEGIN_TOKEN]])
beg_token = beg_token.to(device)
```

Before we start the training, we need a special tensor with the ID of #BEG token. It will be used to look up the embeddings and pass the result to the decoder.

```
with ptan.common.utils.TBMeanTracker(
        writer, 100) as tb_tracker:
    optimiser = optim.Adam(
        net.parameters(), lr=LEARNING_RATE, eps=1e-3)
    batch_idx = 0
    best_bleu = None
    for epoch in range(MAX_EPOCHES):
        random.shuffle(train_data)
```

```
                 dial_shown = False

                 total_samples = 0
                 skipped_samples = 0
                 bleus_argmax = []
                 bleus_sample = []
```

For every epoch, we count the total number of samples and count the skipped samples (due to the high BLEU score). To track the BLEU change during the training, we keep arrays with the BLEU score of argmax-generated sequences and sequences generated by doing sampling.

```
            for batch in data.iterate_batches(
                    train_data, BATCH_SIZE):
                batch_idx += 1
                optimiser.zero_grad()
                input_seq, input_batch, output_batch = \
                    model.pack_batch_no_out(batch, net.emb, device)
                enc = net.encode(input_seq)

                net_policies = []
                net_actions = []
                net_advantages = []
                beg_embedding = net.emb(beg_token)
```

In the beginning of every batch, we pack the batch and encode all first sequences of the batch by calling net.encode(). Then, we declare several lists, which will be populated during the individual decoding of batch entries.

```
                for idx, inp_idx in enumerate(input_batch):
                    total_samples += 1
                    ref_indices = [
                        indices[1:]
                        for indices in output_batch[idx]
                    ]
                    item_enc = net.get_encoded_item(enc, idx)
```

In the preceding loop, we start processing individual entries in a batch: we strip the #BEG token from reference sequences and obtain an individual entry of the encoded batch.

```
                    r_argmax, actions = net.decode_chain_argmax(
                        item_enc, beg_embedding, data.MAX_TOKENS,
                        stop_at_token=end_token)
                    argmax_bleu = utils.calc_bleu_many(
                        actions, ref_indices)
                    bleus_argmax.append(argmax_bleu)
```

As a next step, we decode the batch entry in argmax mode and calculate its BLEU score. This score will be used as a baseline in the REINFORCE policy gradient estimation later.

```
if not args.disable_skip:
    if argmax_bleu > 0.99:
        skipped_samples += 1
        continue
```

In case we have sample skipping enabled, and argmax BLEU is higher than the threshold (a threshold of 0.99 means a near-to-perfect match of the sequences), we stop with this batch entry and go to the next.

```
if not dial_shown:
    w = data.decode_words(
        inp_idx, rev_emb_dict)
    log.info("Input: %s", utils.untokenize(w))
    ref_words = [
        utils.untokenize(
            data.decode_words(
                ref, rev_emb_dict))
        for ref in ref_indices
    ]
    ref = " ~~|~~ ".join(ref_words)
    log.info("Refer: %s", ref)
    w = data.decode_words(
        actions, rev_emb_dict)
    log.info("Argmax: %s, bleu=%.4f",
             utils.untokenize(w), argmax_bleu)
```

The preceding code piece is executed once every epoch and provides a random sample input sequence, reference sequences, and the result of the decoder (the sequence and BLEU score). It's useless for the training process but provides us with information during the training.

Then, we need to perform several rounds of decoding of the batch entry using random sampling. By default, the count of such rounds is 4, but this is tunable using the command-line option.

```
for _ in range(args.samples):
    r_sample, actions = \
        net.decode_chain_sampling(
            item_enc, beg_embedding,
            data.MAX_TOKENS,
            stop_at_token=end_token)
    sample_bleu = utils.calc_bleu_many(
        actions, ref_indices)
```

The sampling decoding call has the same set of arguments as for argmax decoding, followed by the same call to the `calc_bleu_many()` function to obtain the BLEU score.

```
if not dial_shown:
    w = data.decode_words(
        actions, rev_emb_dict)
    log.info("Sample: %s, bleu=%.4f",
             utils.untokenize(w),
             sample_bleu)

net_policies.append(r_sample)
net_actions.extend(actions)
adv = sample_bleu - argmax_bleu
net_advantages.extend(
    [adv]*len(actions))
bleus_sample.append(sample_bleu)
```

At the rest of the decoding loop, we show the decoded sequence if we need to and populate our lists. To get the advantage of the decoding round, we subtract the BLEU score obtained by the argmax method from the result of the random sampling decoding.

```
    dial_shown = True
if not net_policies:
    continue
```

When we're done with a batch, we have several lists: the list with logits from every step of the decoder, the list of taken actions for those steps (which are, in fact, tokens chosen), and the list of advantages for every step.

```
policies_v = torch.cat(net_policies)
actions_t = torch.LongTensor(
    net_actions).to(device)
adv_v = torch.FloatTensor(
    net_advantages).to(device)
```

The returned logits are already in GPU memory, so we can use the `torch.cat()` function to combine them into the single tensor. The other two lists need to be converted and copied on the GPU.

```
log_prob_v = F.log_softmax(policies_v, dim=1)
lp_a = log_prob_v[range(len(net_actions)),
                  actions_t]
log_prob_actions_v = adv_v * lp_a
loss_policy_v = -log_prob_actions_v.mean()

loss_v = loss_policy_v
```

```
            loss_v.backward()
            optimiser.step()
```

When everything is ready, we can calculate the policy gradient by applying `log(softmax())` and choose values from the chosen actions, scaled by their advantages. The negative mean of those scaled logarithms will be our loss that we ask the optimizer to minimize.

```
            tb_tracker.track("advantage", adv_v, batch_idx)
            tb_tracker.track("loss_policy", loss_policy_v,
                             batch_idx)
            tb_tracker.track("loss_total", loss_v, batch_idx)
```

As the last steps in the batch processing loop, we send the advantage and the loss value to TensorBoard.

```
            bleu_test = run_test(test_data, net,
                                 end_token, device)
            bleu = np.mean(bleus_argmax)
            writer.add_scalar("bleu_test", bleu_test, batch_idx)
            writer.add_scalar("bleu_argmax", bleu, batch_idx)
            writer.add_scalar("bleu_sample",
                              np.mean(bleus_sample), batch_idx)
            writer.add_scalar("skipped_samples",
                              skipped_samples / total_samples,
                              batch_idx)
            writer.add_scalar("epoch", batch_idx, epoch)
            log.info("Epoch %d, test BLEU: %.3f",
                     epoch, bleu_test)
```

The preceding code is executed at the end of every epoch and calculates the BLEU score for the test dataset and reports it, together with the BLEU scores obtained during the training, to TensorBoard.

```
            if best_bleu is None or best_bleu < bleu_test:
                best_bleu = bleu_test
                log.info("Best bleu updated: %.4f", bleu_test)
                torch.save(net.state_dict(), os.path.join(
                    saves_path, "bleu_%.3f_%02d.dat" % (
                        bleu_test, epoch)))
            if epoch % 10 == 0:
                torch.save(net.state_dict(), os.path.join(
                    saves_path, "epoch_%03d_%.3f_%.3f.dat" % (
                        epoch, bleu, bleu_test)))
```

As before, we write a model checkpoint every time that the test BLEU score updates the maximum or every 10 epochs.

Results

To run the training, you need the model saved by the cross-entropy training passed with the `-l` argument. The genre your model was trained on has to match the flag passed to the SCST training.

```
$ ./train_scst.py --cuda --data comedy -l saves/crossent-comedy/
epoch_020_0.342_0.120.dat -n sc-comedy-test2
2019-11-14 23:11:41,206 INFO Loaded 159 movies with genre comedy
2019-11-14 23:11:41,206 INFO Read and tokenise phrases...
2019-11-14 23:11:44,352 INFO Loaded 93039 phrases
2019-11-14 23:11:44,522 INFO Loaded 24716 dialogues with 93039 phrases,
generating training pairs
2019-11-14 23:11:44,539 INFO Counting freq of words...
2019-11-14 23:11:44,734 INFO Data has 31774 uniq words, 4913 of them
occur more than 10
2019-11-14 23:11:44,865 INFO Obtained 47644 phrase pairs with 4905 uniq
words
2019-11-14 23:11:45,101 INFO Training data converted, got 25166 samples
2019-11-14 23:11:45,197 INFO Train set has 21672 phrases, test 1253
2019-11-14 23:11:47,746 INFO Model: PhraseModel(
  (emb): Embedding(4905, 50)
  (encoder): LSTM(50, 512, batch_first=True)
  (decoder): LSTM(50, 512, batch_first=True)
  (output): Linear(in_features=512, out_features=4905, bias=True)
)
2019-11-14 23:11:47,755 INFO Model loaded from saves/crossent-comedy/
epoch_020_0.342_0.120.dat, continue training in RL mode...
2019-11-14 23:11:47,771 INFO Input: #BEG no, sir. i'm not going to deny
it. but if you'd just let me explain-- #END
2019-11-14 23:11:47,771 INFO Refer: you better. #END
2019-11-14 23:11:47,771 INFO Argmax: well, you. #END, bleu=0.3873
2019-11-14 23:11:47,780 INFO Sample: that's. i day right. i don't want to
be disturbed first. i don't think you'd be more, bleu=0.0162
2019-11-14 23:11:47,784 INFO Sample: you ask out clothes, don't-? #END,
bleu=0.0527
2019-11-14 23:11:47,787 INFO Sample: it's all of your work. #END,
bleu=0.2182
2019-11-14 23:11:47,790 INFO Sample: come time must be very well. #END,
bleu=0.1890
```

From my experiments, RL fine-tuning is able to improve both the test BLEU score and the training BLEU score. For example, the following charts show the cross-entropy training on the *comedy* genre.
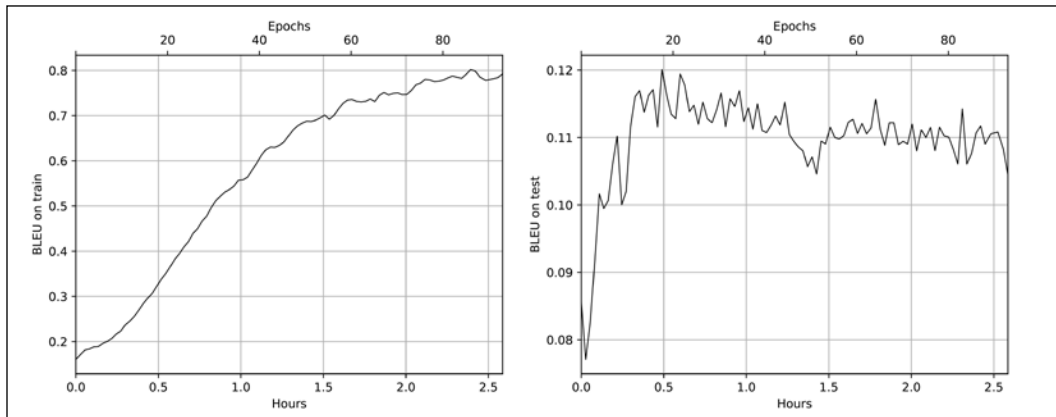
Figure 14.9: Cross-entropy BLEU scores

From those charts, you can see that the best test BLEU score was 0.124, and the training BLEU score stopped improving at 0.73. By fine-tuning the model, saved at epoch 20 (with training BLEU 0.72 and test BLEU 0.121), it was able to improve the train BLEU from 0.124 to 0.14. BLEU scores obtained from training samples are also increasing and look like they can increase even more with more training. The comparison charts follow:
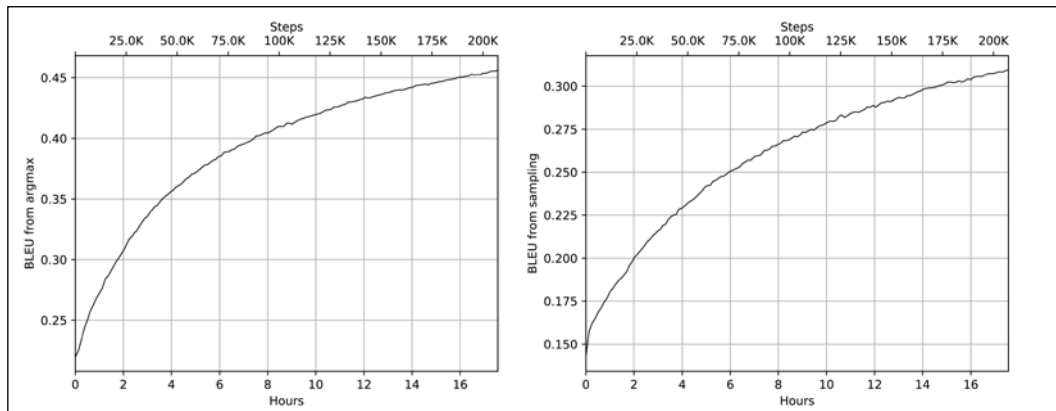
Figure 14.10: BLEU scores during the training: argmax (left) and sampling (right)
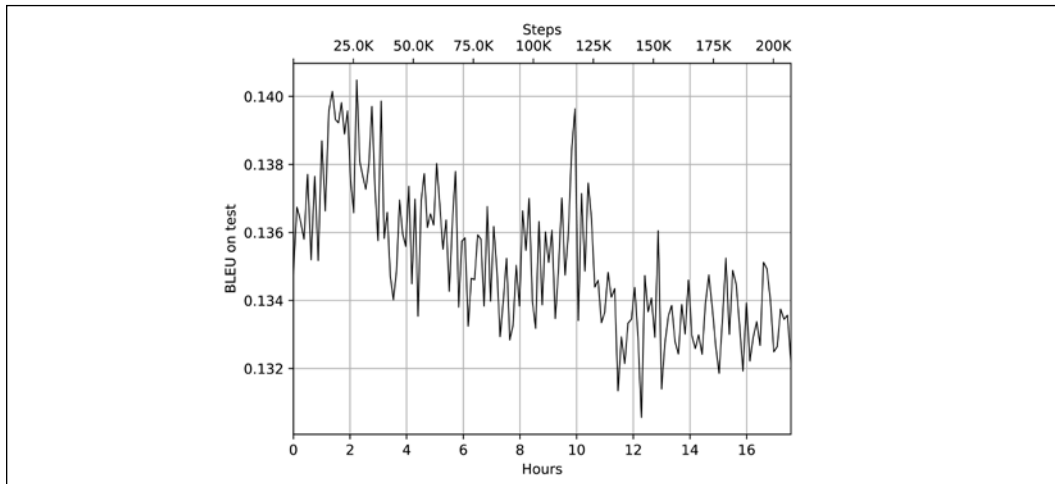
Figure 14.11: BLEU scores on the test set
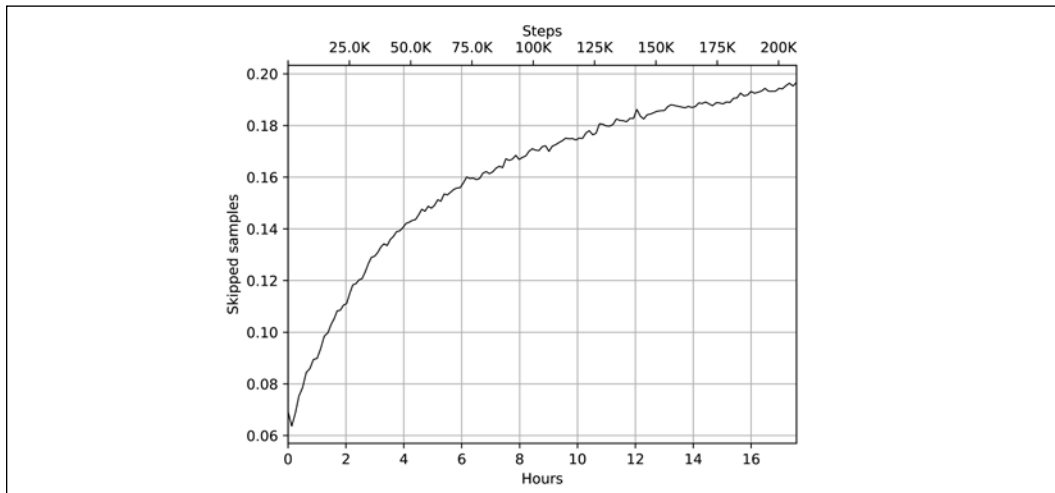
Figure 14.12: Skipped training samples due to high BLEU scores

Separating training from the same model, but without skipping training samples with a high BLEU score from argmax decoding (with option `--disable-skip`), I was able to reach 0.140 BLEU on the test set, which is not very impressive; but, as already explained, it's hard to get good generalization on so few dialogue samples.
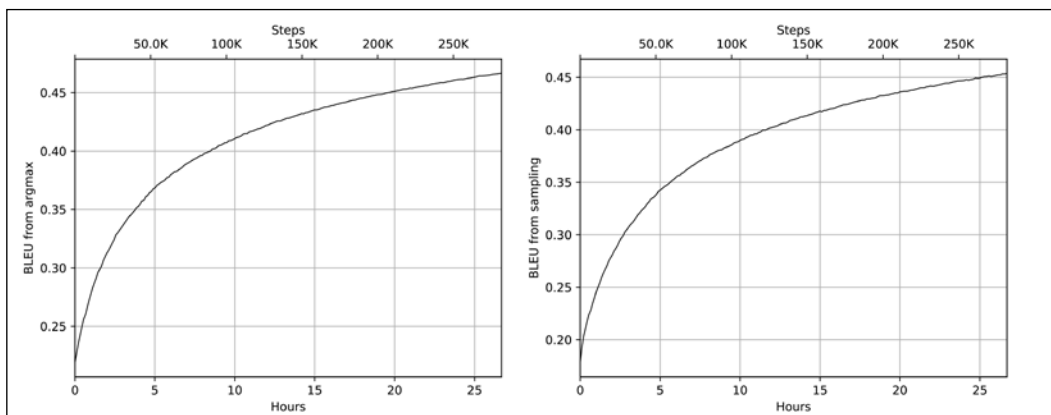
Figure 14.13: BLEU scores from training without skipping samples
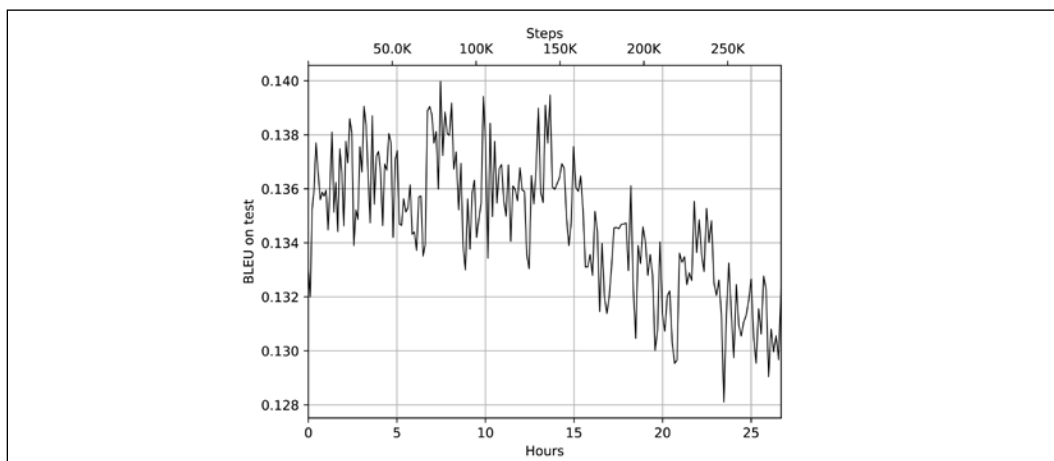
Figure 14.14: The BLEU test score without skipping samples

Models tested on data

Once we've got our models ready, we can check them against our dataset and free-form sentences. During the training, both training tools (`train_crossent.py` and `train_scst.py`) periodically save the model, which is done in two different situations: when the BLEU score on the test dataset updates the maximum and every 10 epochs. Both kinds of models have the same format (produced by the `torch.save()` method) and contain the model's weights. Except the weights, I save the token to integer ID mapping, which will be used by tools to preprocess the phrases.

To experiment with models, two utilities exist: `data_test.py` and `use_model.py`. `data_test.py` loads the model, applies it to all phrases from the given genre, and reports the average BLEU score. Before the testing, phrase pairs are grouped by the first phrase. For example, the following is the result for two models, trained on the *comedy* genre. The first one was trained by the cross-entropy method and the second one was fine-tuned by RL methods.

```
$ ./data_test.py --data comedy -m saves/xe-comedy/epoch_060_0.112.dat

2019-11-14 22:26:40,650 INFO Loaded 159 movies with genre comedy

2019-11-14 22:26:40,650 INFO Read and tokenise phrases...

2019-11-14 22:26:43,781 INFO Loaded 93039 phrases

2019-11-14 22:26:43,986 INFO Loaded 24716 dialogues with 93039 phrases,
generating training pairs

2019-11-14 22:26:44,003 INFO Counting freq of words...

2019-11-14 22:26:44,196 INFO Data has 31774 uniq words, 4913 of them
occur more than 10

2019-11-14 22:26:44,325 INFO Obtained 47644 phrase pairs with 4905 uniq
words

2019-11-14 22:30:06,915 INFO Processed 22767 phrases, mean BLEU = 0.2175
```

```
$ ./data_test.py --data comedy -m saves/sc-comedy2-lr5e-4/
epoch_150_0.456_0.133.dat

2019-11-14 22:34:17,035 INFO Loaded 159 movies with genre comedy

2019-11-14 22:34:17,035 INFO Read and tokenise phrases...

2019-11-14 22:34:20,162 INFO Loaded 93039 phrases

2019-11-14 22:34:20,374 INFO Loaded 24716 dialogues with 93039 phrases,
generating training pairs

2019-11-14 22:34:20,391 INFO Counting freq of words...

2019-11-14 22:34:20,585 INFO Data has 31774 uniq words, 4913 of them
occur more than 10

2019-11-14 22:34:20,717 INFO Obtained 47644 phrase pairs with 4905 uniq
words

2019-11-14 12:37:42,205 INFO Processed 22767 phrases, mean BLEU = 0.4413
```

The second way to experiment with a model is the script `use_model.py`, which allows you to pass any string to the model and ask it to generate the reply:

```
rl_book_samples/Chapter12$ ./use_model.py -m saves/sc-comedy-e40-no-skip/
epoch_080_0.841_0.124.dat -s 'how are you?'

very well. thank you.
```

By passing a number to the `--self` option, you can ask the model to process its own reply as input, in other words, to generate the dialogue.

```
rl_book_samples/Chapter12$ ./use_model.py -m saves/sc-comedy-e40-no-
skip/epoch_080_0.841_0.124.dat -s 'how are you?' --self 10
very well. thank you.
 okay ... it's fine.
 hey ...
shut up.
 fair enough
 so?
so, i saw my draw
what are you talking about?
just one.
i have a car.
```

By default, the generation is performed using argmax, so the model's output is always defined by the input tokens. It's not always what we want, so we can add randomness to the output by passing the `--sample` option. In that case, on every decoder step, the next token will be sampled from the returned probability distribution.

```
rl_book_samples/Chapter12$ ./use_model.py -m saves/sc-comedy-e40-no-
skip/epoch_080_0.841_0.124.dat -s 'how are you?' --self 2 --sample very
well.
very well.
rl_book_samples/Chapter12$ ./use_model.py -m saves/sc-comedy-e40-no-
skip/epoch_080_0.841_0.124.dat -s 'how are you?' --self 2 --sample very
well. thank you.
ok.
```

Telegram bot

As a final step, the Telegram chatbot using the trained model was implemented. To be able to run it, you need to install the `python-telegram-bot` extra package into your virtual environment using `pip install`.

Another step you need to take to start the bot is to obtain the API token by registering the new bot. The complete process is described in the documentation, `https://core.telegram.org/bots#6-botfather`. The resulting token is a string of the form `110201543:AAHdqTcvCH1vGWJxfSeofSAs0K5PALDsaw`.

The bot requires this string to be placed in a configuration file in `~/.config/`
`rl_Chapter14_bot.ini`, and the structure of this file is shown in the Telegram bot
source code as follows. The logic of the bot is not very different from the other two
tools used to experiment with the model: it receives the phrase from the user and
replies with the sequence generated by the decoder.

```python3
#!/usr/bin/env python3
# This module requires python-telegram-bot
import os
import sys
import logging
import configparser
import argparse

try:
    import telegram.ext
except ImportError:
    print("You need python-telegram-bot package installed "
          "to start the bot")
    sys.exit()

from libbots import data, model, utils

import torch

# Configuration file with the following contents
# [telegram]
# api=API_KEY
CONFIG_DEFAULT = "~/.config/rl_Chapter14_bot.ini"

log = logging.getLogger("telegram")


if __name__ == "__main__":
    fmt = "%(asctime)-15s %(levelname)s %(message)s"
    logging.basicConfig(format=fmt, level=logging.INFO)
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--config", default=CONFIG_DEFAULT,
        help="Configuration file for the bot, default=" +
            CONFIG_DEFAULT)
    parser.add_argument(
        "-m", "--model", required=True, help="Model to load")
    parser.add_argument(
        "--sample", default=False, action='store_true',
        help="Enable sampling mode")
    prog_args = parser.parse_args()
```

The bot supports two modes of operations: argmax decoding, which is used by default, and sample mode. In argmax, the bot's replies on the same phrase are always the same. When the sampling is enabled, during decoding, we sample from the returned probability distribution on every step, which increases the variability of the bot's replies.

```
conf = configparser.ConfigParser()
if not conf.read(os.path.expanduser(prog_args.config)):
    log.error("Configuration file %s not found",
              prog_args.config)
    sys.exit()

emb_dict = data.load_emb_dict(
    os.path.dirname(prog_args.model))
log.info("Loaded embedded dict with %d entries",
         len(emb_dict))
rev_emb_dict = {
    idx: word for word, idx in emb_dict.items()
}
end_token = emb_dict[data.END_TOKEN]

net = model.PhraseModel(
    emb_size=model.EMBEDDING_DIM, dict_size=len(emb_dict),
    hid_size=model.HIDDEN_STATE_SIZE)
net.load_state_dict(torch.load(prog_args.model))
```

In the preceding code, we parse the configuration file to get the Telegram API token, load embeddings, and initialize the model with weights. We don't need to load the dataset, as no training is needed.

```
def bot_func(bot, update, args):
    text = " ".join(args)
    words = utils.tokenize(text)
    seq_1 = data.encode_words(words, emb_dict)
    input_seq = model.pack_input(seq_1, net.emb)
    enc = net.encode(input_seq)
```

This function is called by the `python-telegram-bot` library to notify it about the user sending a phrase to the bot. Here, we obtain the phrase, tokenize it, and convert it to the form suitable for the model. Then, the encoder is used to obtain the initial hidden state for the decoder.

```
if prog_args.sample:
    _, tokens = net.decode_chain_sampling(
        enc, input_seq.data[0:1], seq_len=data.MAX_TOKENS,
        stop_at_token=end_token)
else:
    _, tokens = net.decode_chain_argmax(
        enc, input_seq.data[0:1], seq_len=data.MAX_TOKENS,
        stop_at_token=end_token)
```

Next, we call one of the decoding methods, depending on the program command-line arguments. The result that we get in both cases is the sequence of integer token IDs of the decoded sequence.

```
if tokens[-1] == end_token:
    tokens = tokens[:-1]
reply = data.decode_words(tokens, rev_emb_dict)
if reply:
    reply_text = utils.untokenize(reply)
    bot.send_message(chat_id=update.message.chat_id,
                     text=reply_text)
```

When we've got the decoded sequence, we need to decode it back in text form using our dictionary and send the reply to the user.

```
updater = telegram.ext.Updater(conf['telegram']['api'])
updater.dispatcher.add_handler(
    telegram.ext.CommandHandler('bot', bot_func,
                                pass_args=True))

log.info("Bot initialized, started serving")
updater.start_polling()
updater.idle()
```

The last piece of code is the `python-telegram-bot` machinery to register our bot function. To trigger it, you need to use the command /`bot` phrase in the telegram chat.

The following is an example of a conversation generated by the trained model:
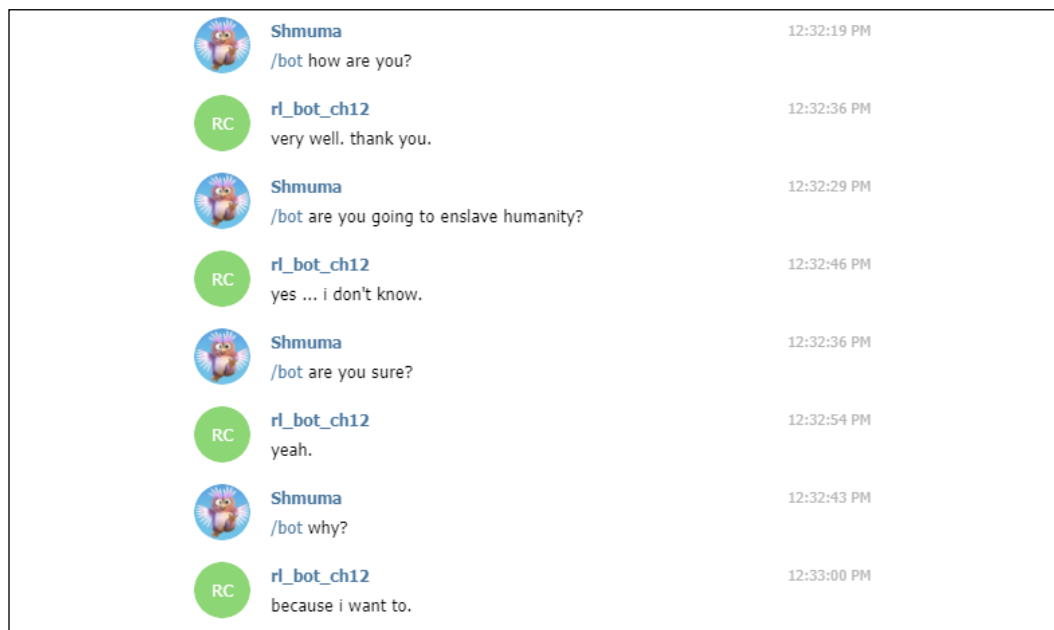
Figure 14.15: A generated conversation

Summary

Despite its simplicity, and the toy-like example in this chapter, seq2seq is a very widely used model in NLP and other domains, so the alternative RL approach could potentially be applicable to a wide range of problems. In this chapter, we've just scratched the surface of deep NLP models and ideas, which go well beyond the scope of this book. We covered the basics of NLP models, such as RNNs and the seq2seq model, along with different ways that it could be trained.

In the next chapter, we will take a look at another example of the application of RL methods in another domain: *automating web navigation tasks*.

15
The TextWorld Environment

In the previous chapter, you saw how reinforcement learning (RL) methods can be applied to natural language processing (NLP) problems, in particular, to improve the chatbot training process. Continuing our journey into the NLP domain, in this chapter, we will now use RL to solve text-based **interactive fiction** games, using the environment published by Microsoft Research called **TextWorld**.

In this chapter, we will:

- Cover a brief historical overview of interactive fiction
- Study the TextWorld environment
- Implement the simple baseline deep Q-network (DQN) method, and then try to improve it by implementing a command generator using recurrent neural networks (RNNs). This will provide a good illustration of how RL can be applied to complicated environments with a rich observation space

Interactive fiction

As you have already seen, computer games are not only entertaining for humans, but also provide challenging problems for RL researchers due to the complicated observations and action spaces, long sequences of decisions to be made during the gameplay, and natural reward systems.

Arcade games like Atari 2600 are just one of many genres that the gaming industry has. From a historical perspective, the Atari 2600 platform peaked in popularity during the late 70s and early 80s. Then followed the era of Z80 and clones, which evolved into the period of the PC-compatible platforms and consoles we have now.

Over time, computer games continually become more complex, colorful, and detailed in terms of graphics, which inevitably increases hardware requirements. This trend makes it harder for RL researchers and practitioners to apply RL methods to the more recent games; for example, almost everybody can train an RL agent to solve an Atari game, but for StarCraft II, DeepMind had to burn electricity for weeks, leveraging clusters of graphics processing unit (GPU) machines. Of course, this activity is needed for future research, as it allows us to check ideas and optimize methods, but the complexity of StarCraft II and Dota, for example, makes them prohibitively expensive for most people.

There are several ways of solving this problem. The first one is to take games that are "in the middle" of the complexities of Atari and StarCraft. Luckily, there are literally thousands of games from the Z80, NES, Sega, and C64 platforms.

Another way is to take a challenging game but make a simplification to the environment. There are several Doom environments (available in Gym), for example, that use the game engine as a platform, but the goal is much simpler than in the original game, like navigating the corridor, gathering weapons, or shooting enemies. Those microgames are also available on StarCraft II.

The third, and completely different, approach is to take some game that may not be very complex in terms of observation, but requires long-term planning, has complex exploration of the state space, and has challenging interactions between objects. An example of this family is the famous Montezuma's Revenge from the Atari suite, which is still challenging even for modern RL methods. This last approach is quite appealing, due to the accessibility of resources, combined with still having a complexity that reaches the edge of RL methods' limits.

Another example of this is text-based games, which are also known as interactive fiction. This genre is almost dead now, being made obsolete by modern games and hardware progress, but at the time of Atari and Z80, interactive fiction was provided concurrently with traditional games. Instead of using rich graphics to show the game state (which was tricky for hardware from the 70s), these games relied on the players' minds and imagination.

The gaming process was communicated via text when the description of the current game state was given to the player. An example is *you are standing at the end of a road before a small brick building. Around you is a forest. A small stream flows out of the building and down a gully.*

As you can see in *Figure 15.1*, this is the very beginning of the Adventure game from 1976, which was the first game of this kind. Actions in the game were given in the form of free-text commands, which normally had a simple structure and limited set of words, for example "verb + noun."
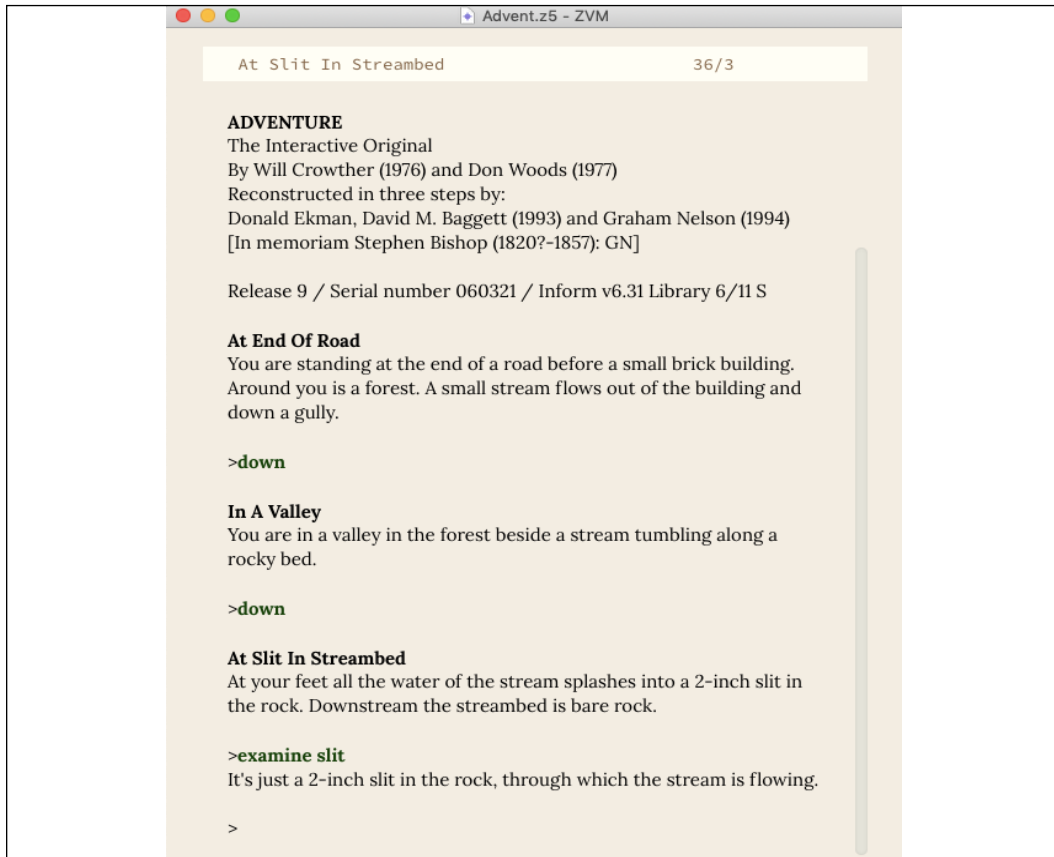
Figure 15.1: An example of the interactive fiction game process

Despite the simplistic descriptions, in the 80s and early 90s, hundreds of large and small games were developed by individual developers and commercial studios. Those games sometimes required many hours of gameplay, contained thousands of locations, and had a lot of objects to interact with.

For example, the following figure shows part of the Zork I game map published by Infocom in 1980.
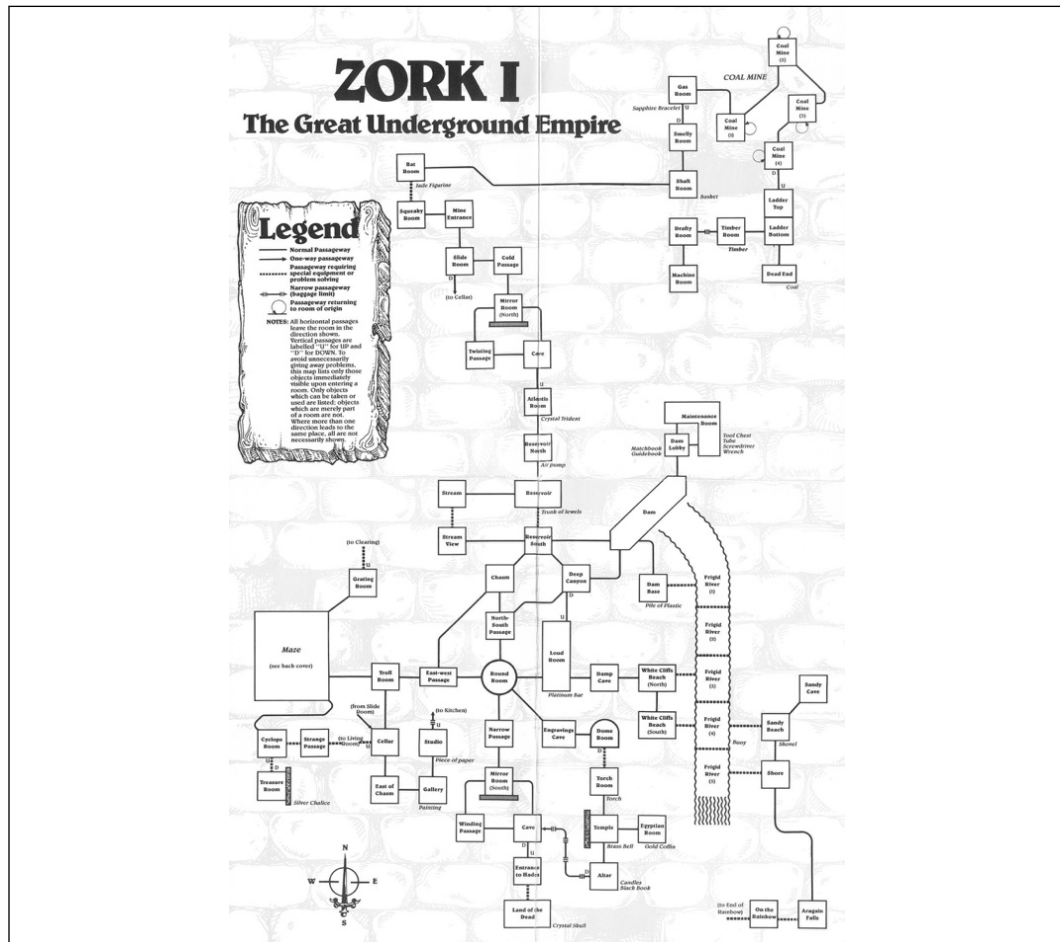
Figure 15.2: The Zork I map (underground part)

As you can imagine, the challenges of such games could be increased almost infinitely, as complex interactions between objects, exploration of game states, communication with other characters, and other real-life scenarios could be included. There are many such games available on the Interactive Fiction Archive website: `http://ifarchive.org`.

In June 2018, Microsoft Research released an open source project that aimed to provide researchers and RL enthusiasts with a simple way to experiment with text-based games using familiar tools.

Their project, called TextWorld, is available on GitHub (`https://github.com/microsoft/TextWorld`) and provides the following functionality:

- A Gym environment for text-based games. It supports games in two formats: Z-machine bytecode (versions 1-8 are supported) and Glulx games.

- A game generator that allows you to produce randomly generated quests with requirements like the number of objects, the description complexity, and the quest length.

- The capability to tune (for generated games) the complexity of the environment by peeking into the game state. For example, intermediate rewards could be enabled, which will give a positive reward to the agent every time it makes a step in the right direction. Several such factors will be described in the next section.

- Various utility functions to deal with the observation and action space, generated games, and so forth.

In this chapter, we will explore the environment's capabilities and implement several versions of training code to solve the generated games. The complexity of the games will not be very high, but you can use them as a basis for your own experiments and idea validation.

The environment

At the time of writing, the TextWorld environment supports only Linux and macOS platforms and internally relies on the Inform 7 system (`http://inform7.com`). There are two webpages for the project: one is the Microsoft Research webpage: `https://www.microsoft.com/en-us/research/project/textworld/`, which contains general information about the environment, and the another is on GitHub (`https://github.com/microsoft/TextWorld`) and describes installation and usage. Let's start with installation.

Installation

The installation instructions suggest that you can install the package by just typing `pip install textworld` in your Python virtual environment, but at the time of writing, this step is broken by a changed URL for the Inform 7 engine. Hopefully, this will be fixed on the next TextWorld release, but if you experience any issues, you can set up a version that I've tested for this example by running `pip install git+https://github.com/microsoft/TextWorld@f1ac489fefeb6a48684ed1f89422b84b7b4a6e4b`.

Once installed, the package can be imported in Python code, and it also provides two command-line utilities for game generation and gameplay: `tw-make` and `tw-play`. They are not needed if you have ambitious plans to solve full-featured interactive fiction games from `http://ifarchive.org`, but in our case, we will start with artificially generated quests for simplicity.

Game generation

The `tw-make` utility allows you to generate games with the following characteristics:

- **Game scenario**: For example, you can choose a classic quest with the aim of using objects and following some sequence of actions, or a "coin collection" scenario, when the player needs to navigate the scenes and find coins

- **Game theme**: You can set up the interior of the game, but at the moment, only the "house" and "basic" themes exist

- **Object properties**: You can include adjectives with objects; for instance, it might be the "green key" that opens the box, not just the "key"

- **The number of parallel quests that the game can have**: By default, there is only one sequence of actions to be found, but you can change this and allow the game to have subgoals and alternative paths

- **The length of the quest**: You can define how many steps the player needs to take before reaching the end of the game

- **Random seeds**: You can use these to generate reproducible games

The resulting game generated could be in Glulx or Z-machine format, which are standard portable virtual machine instructions that are widely used for normal games and supported by several interactive fiction interpreters, so you can play the generated games in the same way as normal interactive fiction games.

Let's generate some games and check what they bring us:

```
$ tw-make tw-coin_collector --output t1 --seed 10 --level 5 --force-
entity-numbering
Global seed: 10
Game generated: t1.ulx
```

The command generates three files: `t1.ulx`, `t1.ni`, and `t1.json`. The first one contains bytecode to be loaded into the interpreter, and the others are extended data that could be used by the environment to provide extra information during the gameplay.

To play the game in interactive mode, you can use any interactive fiction interpreter supporting the Glulx format, or use the provided utility `tw-play`, which might not be the most convenient way to play interactive fiction games, but will enable you to check the result.

```
$ tw-play t1.ulx
Using GitGlulxMLEnvironment.
```

```
 _____  _____  __     __  _____
|        \|        \|  \   |  \|        \
 \$$$$$$$$| $$$$$$$$| $$   | $$ \$$$$$$$$
   | $$   | $$__     \$$\ /  $$   | $$
   | $$   | $$  \     >$$  $$    | $$
   | $$   | $$$$$    /  $$$$\    | $$
   | $$   | $$_____ |  $$ \$$\   | $$
   | $$   | $$     \| $$  | $$   | $$
    \$$    \$$$$$$$$ \$$   \$$    \$$

 __       __   _____   _____   __       _____
|  \  _  |  \ /      \ |       \ |  \     |       \
| $$ / \ | $$|  $$$$$$\| $$$$$$$\| $$     | $$$$$$$\
| $$/  $\| $$| $$  | $$| $$__| $$| $$     | $$  | $$
| $$  $$$\ $$| $$  | $$| $$    $$| $$     | $$  | $$
| $$ $$\$$\$$| $$  | $$| $$$$$$$\| $$     | $$  | $$
| $$$$  \$$$$| $$__/ $$| $$  | $$| $$_____ | $$__/ $$
| $$$    \$$$ \$$    $$| $$  | $$| $$     \| $$    $$
 \$$      \$$  \$$$$$$  \$$   \$$ \$$$$$$$$ \$$$$$$$
```

Get ready to pick stuff up and put it in places, because you've just entered

TextWorld! First step, try to take a trip east. And then, head south. Next, make

an effort to take a trip south. If you can accomplish that, try to take a trip

west. With that accomplished, recover the coin from the floor of the chamber.

Got that? Good!

-= Spare Room =-

You are in a spare room. An usual one.

```
You don't like doors? Why not try going east, that entranceway is
unblocked.


>
```

In this chapter, we will experiment with several games. You need to generate them by using the provided script: `Chapter15/game/make_games.sh`. It generates 21 games of length 5, using different seed values to ensure variability between the games.

Observation and action spaces

Generating and playing a game might be fun, but the core value of TextWorld is in its ability to provide an RL interface for generated or existing games. Let's check what we can do with the game we just generated in the previous section:

```
In [1]: import textworld
In [2]: from textworld.gym import register_game
In [3]: import gym
In [4]: env_id = register_game("t1.ulx")
In [5]: env_id
Out[5]: 'tw-v0'
In [6]: env = gym.make(env_id)
In [7]: env
Out[7]: <TimeLimit<TextworldGamesEnv<tw-v0>>>
In [8]: r = env.reset()
In [9]: r
Out[9]:
("\n\n\n                    _____  _____  __     __  _____
\n                   |        \\|        \\|  \\   |  \\|        \\
\n                    \\$$$$$$$$| $$$$$$$$| $$   | $$ \\$$$$$$$$
\n                      | $$   | $$__      \\$$\\/  $$   | $$
\n                      | $$   | $$  \\      >$$  $$    | $$
\n                      | $$   | $$$$    /  $$$$\\     | $$
\n                      | $$   | $$_____ |  $$ \\$$\\    | $$
\n                      | $$   | $$     \\| $$  | $$   | $$
\n                       \\$$    \\$$$$$$$$ \\$$   \\$$    \\$$
\n                     __        __   _____   _____   __       _____
```

```
\n                | \\  _  |  \\ /      \\ |       \\ |  \\       |
\\ \n              | $$ / \\ | $$|  $$$$$\\| $$$$$$\\| $$      |
$$$$$$\\\n          | $$/  $\\| $$| $$   | $$| $$__| $$| $$      |
$$   | $$\n            | $$  $$$\\ $$| $$   | $$| $$    $$| $$      |
$$   | $$\n            | $$ $$\\$$\\$$| $$   | $$| $$$$$$\\| $$
| $$   | $$\n             | $$$$  \\$$$$| $$__/ $$| $$   | $$| $$_____
| $$__/ $$\n             | $$$      \\$$$ \\$$    $$| $$   | $$| $$
\\| $$    $$\n             \\$$        \\$$  \\$$$$$$  \\$$    \\$$
\\$$$$$$$$ \\$$$$$$$ \n\nGet ready to pick stuff up and put it in places,
because you've just entered TextWorld! First step, try to take a trip
east. And then, head south. Next, make an effort to take a trip south. If
you can accomplish that, try to take a trip west. With that accomplished,
recover the coin from the floor of the chamber. Got that? Good!\n\n-=
Spare Room =-\nYou are in a spare room. An usual one.\n\n\n\nYou don't
like doors? Why not try going east, that entranceway is unblocked.\n\n\
n\n",
 {})
```

So, we have registered the generated game in the Gym games registry, created the environment, and got the initial observation. Yes, all this text mess is our observation, from which our agent has to select its next action. If you check the result of the `reset()` method more carefully, you may notice that the result is not a string, but a tuple with a string and a dict.

This is one of the TextWorld specifics that make it incompatible with the Gym API. Indeed, `Env.reset()` has to return just the observation, but TextWorld returns the observation and *extended information*, which is a dict. We will discuss why it was implemented this way in the upcoming section, when we talk about the extended game information. For now, just note the extra dict returned from `reset()`, and we will continue our exploration of the environment.

```
In [10]: env.observation_space
Out[10]: Word(L=200, V=1250)
In [11]: env.action_space
Out[11]: Word(L=8, V=1250)
In [12]: env.action_space.sample()
Out[12]: array([ 575,  527,   84, 1177,  543,  702, 1139,  178])
In [13]: type(env.action_space)
Out[13]: textworld.gym.spaces.text_spaces.Word
```

As you can see, both the observation and action spaces are represented by the custom class `Word`, which is provided by the TextWorld library. This class represents a variable-length sequence of words from some vocabulary. In the observation space, the maximum length of the sequence equals 200 (which is shown as `L=200` in the preceding code), and the vocabulary is 1,250 tokens (`V=1250`). For the action space, the vocabulary is the same, but sequences are shorter: up to eight tokens.

As you saw in the previous chapter, the normal representation of sequences in NLP is a list of token IDs from the vocabulary. Access to this vocabulary is provided by the fields of the `Word` class. In particular, it has:

- The dictionary `id2w`, which maps the token ID into the word
- The dictionary `w2id` with reverse mapping
- The `tokenize()` method, which takes the string and returns the sequence of tokens

This functionality provides a convenient and efficient way to deal with observation and action sequences:

```
In [16]: env.action_space.tokenize("go north")
Out[16]: array([  3, 466, 729,    2])
In [17]: act = env.action_space.sample()
In [18]: act
Out[18]: array([1090,  996,  695, 1195,  447, 1244,   61,  734])
In [19]: [env.action_space.id2w[a] for a in act]
Out[19]: ['this', 'spork', 'modest', "weren't", 'g', "you've", 'amazing', 'noticed']
```

This text is not very meaningful, as it was randomly sampled from the vocabulary. Another thing you might notice is that the tokenized version of the "go north" sequence has four tokens instead of two. That's due to extra tokens representing the beginning and the end of the sequence, which are added automatically. We used the same approach in the previous chapter.

Extra game information

Before we start planning our first training code, we need to discuss one additional functionality of TextWorld that we will use. As you might guess, even a simple problem might be too challenging for us. Note that observations are text sequences of up to 200 tokens from the vocabulary of size 1,250. Actions could be up to eight tokens long. Generated games have five actions to be executed in the correct order.

So, our chance of randomly finding the proper sequence of 8 × 5 = 40 tokens is something around $\frac{1}{1250^{40}} \approx \frac{1}{10^{123}}$. This is not very promising, even with the fastest GPUs. Of course, we have start- and end-sequence tokens, which we can take into account to increase our chances; still, the probability of finding the correct sequence of actions with random exploration is tiny.

Another challenge is the partially observable Markov decision process (POMDP) nature of the environment, which comes from the fact that our inventory in the game is usually not shown. It is a normal practice in interactive fiction games to display the objects your character possesses only after some explicit command, like "inventory." But our agent has no idea about the previous state. So, from its point of view, the situation after the command `"take apple"` is exactly the same as before (with the difference that the apple is no longer mentioned in the scene description). We can deal with that by stacking states as we did in Atari games, but we need to do it explicitly, and the amount of information the agent needs to process will increase significantly.

With all this being said, we should make some simplifications in the environment. Luckily, TextWorld provides us with convenient means for such cheating. During the game registration, we can pass extra flags to enrich the observation space with extra pieces of more structured information. Here is the list of internals that we can peek into:

- A separate description of the current room, as it will be given by the `"look"` command
- The current inventory
- The name of the current location
- The facts of the current world state
- The last action and the last command performed
- The list of admissible commands in the current state
- The sequence of actions to execute to win the game

In addition, besides extra structured observations provided on every step, we can ask TextWorld to give us intermediate rewards every time we move in the right direction in the quest. As you might guess, this is extremely helpful for speeding up the convergence.

The most useful features in the additional information we can add are admissible commands, which enormously decrease our action space from 1250^{40} to just a dozen, and intermediate rewards, which guide the training in the right direction.

To enable this extra information, we need to pass an optional argument to the `register_game()` method:

```
In [1]: from textworld.gym import register_game

In [2]: from textworld.envs.wrappers.filter import EnvInfos

In [3]: import gym

In [4]: env_id = register_game("t1.ulx", request_
infos=EnvInfos(inventory=True, intermediate_reward=True, admissible_
commands=True))

In [6]: env = gym.make(env_id)

In [7]: env.reset()

Out[7]:
("\n\n\n                         _____  _____  __      __  _____
\n                        |        \\\\|        \\\\|  \\    |  \\|        \\\\
\n                         \\\\$$$$$$$$| $$$$$$$$| $$   | $$ \\\\$$$$$$$$
\n                           | $$    | $$__     \\\\$$\\\\/  $$   | $$
\n                           | $$    | $$  \\\\     >$$  $$    | $$
\n                           | $$    | $$$$   /  $$$$\\\\     | $$
\n                           | $$    | $$_____  |  $$ \\\\$$\\\\    | $$
\n                           | $$    | $$     \\\\| $$  | $$    | $$
\n                            \\\\$$     \\\\$$$$$$$$ \\\\$$   \\\\$$     \\\\$$
\n
\n              __        __  _____   _____   __        _____
\\\\ \n           |  \\\\  _  |  \\\\ /      \\\\ |       \\\\ |  \\\\      |       \\\\
$$$$$$$\\\\\n           | $$ / \\\\ | $$|  $$$$$$\\\\| $$$$$$$\\\\| $$      | $$ | $$\n           | $$/  $\\\\| $$| $$  | $$| $$__| $$| $$      |
$$ | $$\n           | $$  $$$\\\\ $$| $$  | $$| $$    $$| $$      |
$$ | $$\n           | $$ $$\\\\$$\\\\$$| $$  | $$| $$$$$$$\\\\| $$
| $$ | $$\n           | $$$$  \\\\$$$$| $$__/ $$| $$ | $$| $$_____
| $$__/ $$\n           | $$$    \\\\$$$ \\\\$$    $$| $$ | $$| $$
\\\\| $$    $$\n            \\\\$$      \\\\$$ \\\\$$$$$$ \\\\$$   \\\\$$
\\\\$$$$$$$ \\\\$$$$$$ \n\nGet ready to pick stuff up and put it in places,
because you've just entered TextWorld! First step, try to take a trip
east. And then, head south. Next, make an effort to take a trip south. If
you can accomplish that, try to take a trip west. With that accomplished,
recover the coin from the floor of the chamber. Got that? Good!\n\n-=
Spare Room =-\nYou are in a spare room. An usual one.\n\n\n\nYou don't
like doors? Why not try going east, that entranceway is unblocked.\n\n\
n\n",
 {'inventory': 'You are carrying nothing.\n\n\n',
```

```
    'admissible_commands': ['go east', 'inventory', 'look'],
    'intermediate_reward': 0})
```

As you can see, the environment now provides us with extra information in the dictionary that was empty before. In this state, only three commands make sense. Let's try the first one.

```
In [8]: env.step("go east")
Out[8]:
("\n-= Attic =-\nYou make a grand eccentric entrance into an attic.\n\
n\n\nYou need an unblocked exit? You should try going south. You don't
like doors? Why not try going west, that entranceway is unblocked.\n\
n\n",
 0,
 False,
 {'inventory': 'You are carrying nothing.\n\n\n',
  'admissible_commands': ['go south', 'go west', 'inventory', 'look'],
  'intermediate_reward': 1})
```

The command was accepted, and we were given an intermediate reward of 1. Okay, that's great. Now we have everything needed to implement our first baseline DQN agent to solve TextWorld problems!

Baseline DQN

In this problem, the major challenge lies in inconvenient observation and action spaces. Text sequences might be problematic on their own, as we discussed in the previous chapter. The variability of sequence lengths might cause vanishing and exploding gradients in RNNs, slow training, and convergence issues. In addition to that, our TextWorld environment provides us with several such sequences that we need to handle separately. Our scene description string, for example, might have a completely different meaning to the agent than the inventory string, which describes our possessions.

As mentioned, another obstacle is the action space. As you have seen in the previous section, TextWorld might provide us with a list of commands that we can execute in every state. It significantly reduces the action space we need to choose from, but there are other complications. One of them is that the list of admissible commands changes from state to state (as different locations might allow different commands to be executed). Another issue is that every entry in the admissible commands list is a sequence of words.

Potentially, we might get rid of both of those variabilities by building a dictionary of all possible commands and using it as a discrete, fixed-size action space. In simple games, this might work, as the number of locations and objects is not that large. You can try this as an exercise, but we will follow a different path.

Thus far, you have seen only discrete action spaces having a small number of predefined actions, and this influenced the architecture of the DQN: the output from the network predicted Q-values for all actions in one run, which was convenient both during the training and model application (as we need all Q-values for all actions to find argmax anyway). But this choice of DQN architecture is not something dictated by the method, so if needed, we can tweak it. And our issue with a variable number of actions might be solved this way. To get a better understanding of how, let's check the architecture of our TextWorld baseline DQN, as shown in the following figure.

Figure 15.3: The architecture of the TextWorld baseline DQN

The major part of the diagram is occupied by preprocessing blocks. On input to the network (blocks on the left), we get variable sequences of individual parts of observations ("Raw text", "Description", and "Inventory") and the sequence of one action command to be evaluated. This command will be taken from the admissible commands list, and the goal of our network will be to predict a single Q-value for the current game state and this particular command. This approach is different from the DQNs we have used before, but as we don't know in advance which commands will be evaluated in every state, we will evaluate every command individually.

Those four input sequences (which are lists of token IDs in our vocabulary) will be passed through an embeddings layer and then fed into separate long short-term memory (LSTM) RNNs. We discussed embeddings and RNNs in the previous chapter, so if you're not familiar with those NLP concepts, you can check the corresponding sections there.

The goal of LSTM networks (which are called "Encoders" in the figure) is to convert variable-length sequences into fixed-size vectors. Every input piece is processed by its own LSTM with separated weights, which will allow the network to capture different data from different input sequences.

The output from the encoders is concatenated into one single vector and passed to the main DQN network. As our variable-length sequences have been transformed into fixed-size vectors, the DQN network is simple: just several feed-forward layers producing one single Q-value. This is less efficient computationally, but for the baseline it is fine.

The complete source code is in the `Chapter15` directory and it includes the following modules:

- `train_basic.py`: A baseline training program
- `lib/common.py`: Common utilities to set up the Ignite engine and hyperparameters
- `lib/preproc.py`: The preprocessing pipeline, including embeddings and encoder classes
- `lib/model.py`: The DQN model and DQN agent with helper functions

For the sake of space in this chapter, the full source code won't be given. Only the most important or tricky parts will be explained in the subsequent sections.

Observation preprocessing

Let's start with the leftmost part of our pipeline. On the input, we're going to get several lists of tokens, both for the individual state observation and for our command that we're going to evaluate. But as you have already seen, the TextWorld environment produces the string and a dict with the extended information, so we need to tokenize the strings and get rid of nonrelevant information. That's the responsibility of the `TextWorldPreproc` class, which is defined in the `lib/preproc.py` module:

```
class TextWorldPreproc(gym.Wrapper):

    log = logging.getLogger("TextWorldPreproc")

    def __init__(self, env: gym.Env, encode_raw_text: bool = False,
                 encode_extra_fields: Iterable[str] = (
                         'description', 'inventory'),
                 use_admissible_commands: bool = True,
                 use_intermediate_reward: bool = True,
```

```
                tokens_limit: Optional[int] = None):
        super(TextWorldPreproc, self).__init__(env)
        if not isinstance(env.observation_space, tw_spaces.Word):
            raise ValueError(
                "Env should expose textworld obs, "
                "got %s instead" % env.observation_space)
        self._encode_raw_text = encode_raw_text
        self._encode_extra_field = tuple(encode_extra_fields)
        self._use_admissible_commands = use_admissible_commands
        self._use_intermedate_reward = use_intermediate_reward
        self._num_fields = len(self._encode_extra_field) + \
                            int(self._encode_raw_text)
        self._last_admissible_commands = None
        self._last_extra_info = None
        self._tokens_limit = tokens_limit
        self._cmd_hist = []
```

The class implements the `gym.Wrapper` interface, so it will transform the TextWorld environment observations and actions in the way we need. The constructor accepts several flags, which simplifies future experiments. For example, you can disable the usage of admissible commands or intermediate reward, set the limit of tokens, or change the set of observation fields to be processed.

```
    @property
    def num_fields(self):
        return self._num_fields

    def _encode(self, obs: str, extra_info: dict) -> dict:
        obs_result = []
        if self._encode_raw_text:
            tokens = self.env.observation_space.tokenize(obs)
            if self._tokens_limit is not None:
                tokens = tokens[:self._tokens_limit]
            obs_result.append(tokens)
        for field in self._encode_extra_field:
            extra = extra_info[field]
            tokens = self.env.observation_space.tokenize(extra)
            if self._tokens_limit is not None:
                tokens = tokens[:self._tokens_limit]
            obs_result.append(tokens)
        result = {"obs": obs_result}
        if self._use_admissible_commands:
            adm_result = []
            for cmd in extra_info['admissible_commands']:
                cmd_tokens = self.env.action_space.tokenize(cmd)
                adm_result.append(cmd_tokens)
```

```
        result['admissible_commands'] = adm_result
        self._last_admissible_commands = \
            extra_info['admissible_commands']
    self._last_extra_info = extra_info
    return result
```

The `_encode` method is the heart of the observation transformation. It takes the observation string and the extended information dictionary and returns a single dictionary with the following keys:

- `obs`: The list of lists with the token IDs of input sequences.
- `admissible_commands`: A list with commands available from the current state. Every command is tokenized and converted into the list of token IDs.

In addition, the method remembers the extra information dictionary and raw admissible commands list. This is not needed for training, but will be useful during the model application, to be able to get back the command text from the index of the command.

```
    def reset(self):
        res = self.env.reset()
        self._cmd_hist = []
        return self._encode(res[0], res[1])


    def step(self, action):
        if self._use_admissible_commands:
            action = self._last_admissible_commands[action]
            self._cmd_hist.append(action)
        obs, r, is_done, extra = self.env.step(action)
        if self._use_intermedate_reward:
            r += extra.get('intermediate_reward', 0)
        if self._reward_wrong_last_command is not None:
            # that value is here if we gave a nonsense command
            if extra.get('last_command', '') == 'None':
                r += self._reward_wrong_last_command
        new_extra = dict(extra)
        fields = list(self._encode_extra_field)
        fields.append('admissible_commands')
        fields.append('intermediate_reward')
        for f in fields:
            if f in new_extra:
                new_extra.pop(f)
        return self._encode(obs, extra), r, is_done, new_extra
```

Two public methods, `reset()` and `step()`, use the `_encode()` method to do the transformation of the observation.

The admissible commands are enabled, and the action is expected to be the index in the list of commands, instead of the command text.

```
@property
def last_admissible_commands(self):
    if self._last_admissible_commands:
        return tuple(self._last_admissible_commands)
    return None


@property
def last_extra_info(self):
    return self._last_extra_info
```

Finally, there are two properties that give access to the remembered state. To illustrate how this class is supposed to be applied and what it does with the observation, let's check the following small interactive session:

```
In [1]: from textworld.gym import register_game

In [2]: import gym

In [3]: from lib import preproc

In [4]: from textworld.envs.wrappers.filter import EnvInfos

In [5]: env_id = register_game("games/simple1.ulx", request_
infos=EnvInfos(inventory=True, intermediate_reward=True, admissible_
commands=True, description=True))

In [6]: env = gym.make(env_id)

In [7]: env.reset()

Out[7]:
("\n\n\n                     _____  _____  __     __  _____
\n                    |        \\|        \\|  \\   |  \\|        \\
\n                     \\$$$$$$$$| $$$$$$$$| $$   | $$ \\$$$$$$$$
\n                       | $$   | $$__     \\$$\\/  $$   | $$
\n                       | $$   | $$  \\     >$$  $$    | $$
\n                       | $$   | $$$$$    /  $$$$\\    | $$
\n                       | $$   | $$_____ |  $$ \\$$\\   | $$
\n                       | $$   | $$     \\| $$  | $$   | $$
\n                        \\$$    \\$$$$$$$$ \\$$   \\$$    \\$$
\n          __        __   _____   _____   __         _____
\n         |  \\      |  \\ /      \\ /      \\ |  \\       |        \\
\\ \n         | $$ /  \\ | $$|  $$$$$$\\|  $$$$$$\\| $$        |
$$$$$$\\\\n         | $$/  $\\| $$| $$   | $$| $$__| $$| $$        |
```

```
$$   |  $$\n                    | $$  $$$\\ $$| $$   | $$| $$    $$| $$       |
$$   |  $$\n                    | $$ $$\\$$\\$$| $$   | $$| $$$$$$$\\| $$
| $$   |  $$\n                    | $$$$  \\$$$$| $$__/ $$| $$   | $$| $$_____
| $$__/  $$\n                    | $$$    \\$$$ \\$$    $$| $$   | $$| $$
\\|  $$    $$\n                     \\$$        \\$$ \\$$$$$$  \\$$    \\$$
```

\\$$$$$$$$ \\$$$$$$$ \n\nWelcome to TextWorld! Here is your task for
today. First off, if it's not too much trouble, I need you to take a trip
east. Then, take a trip north. And then, take the insect from the case.
After that, go to the south. Then, sit the insect on the stand inside the
studio. And if you do that, you're the winner!\n\n-= Spare Room =-\nYou
find yourself in a spare room. A typical kind of place.\n\nYou rest your
hand against a wall, but you miss the wall and fall onto a rectangular
locker.\n\nThere is an unguarded exit to the east. You don't like doors?
Why not try going south, that entranceway is unblocked.\n\nThere is a
rectangular key on the floor.\n\n\n\n",

{'description': "-= Spare Room =-\nYou find yourself in a spare room. A
typical kind of place.\n\nYou rest your hand against a wall, but you miss
the wall and fall onto a rectangular locker.\n\nThere is an unguarded
exit to the east. You don't like doors? Why not try going south, that
entranceway is unblocked.\n\nThere is a rectangular key on the floor.\
n\n",

  'inventory': 'You are carrying:\n a pair of headphones\n a whisk\n\
n\n',

  'admissible_commands': ['drop pair of headphones',
   'drop whisk',
   'examine pair of headphones',
   'examine rectangular key',
   'examine rectangular locker',
   'examine whisk',
   'go east',
   'go south',
   'inventory',
   'look',
   'open rectangular locker',
   'take rectangular key'],
  'intermediate_reward': 0})

So, that's our raw observation obtained from the TextWorld environment. Let's apply
our preprocessor:

```
In [13]: pr_env = preproc.TextWorldPreproc(env)
In [14]: pr_env.reset()
```

```
Out[14]:
{'obs': [array([<values hidden>]),
  array([ 3, 1240, 78, 1, 35, 781, 747, 514, 35, 1203,  2])],
 'admissible_commands': [array([ 3, 342, 781, 747, 514,  2]),
  array([  3,  342, 1203,  2]),
<more commands here>
  array([  3, 1053,  871,  588,  2])]}
```

Let's try to execute some action. Zero action corresponds to the first entry in the admissible commands list, which is "drop pair of headphones" in our case.

```
In [15]: pr_env.step(0)
Out[15]:
({'obs': [array([<values hidden>]),
   array([  3, 1240,  78,  1,  35, 1203,  2])],
  'admissible_commands': [array([  3,  342, 1203,  2]),
   array([ 3, 383, 781, 747, 514,  2]),
<more commands here>
   array([  3, 1053,  871,  588,  2])]},
 0,
 False,
 {})
In [17]: pr_env.last_extra_info['inventory']
Out[17]: 'You are carrying:\n  a whisk\n\n\n'
```

As you can see, we no longer have the headphones, but it wasn't the right action to take, as an intermediate reward was not given.

Okay, this representation still can't be fed into neural networks (NNs), but it is much closer to what we want.

Embeddings and encoders

The next step in the preprocessing pipeline is implemented in two classes:

- `Encoder`: A wrapper around the LSTM unit that transforms one single sequence (after embeddings have been applied) into a fixed-size vector
- `Preprocessor`: This is responsible for the application of embeddings and the transformation of individual sequences with corresponding encoder classes

The `Encoder` class is simpler, so let's start with it:

```
class Encoder(nn.Module):
    def __init__(self, emb_size: int, out_size: int):
        super(Encoder, self).__init__()
        self.net = nn.LSTM(
            input_size=emb_size, hidden_size=out_size,
            batch_first=True)

    def forward(self, x):
        self.net.flatten_parameters()
        _, hid_cell = self.net(x)
        return hid_cell[0].squeeze(0)
```

The logic is very simple: we apply the LSTM layer and return its hidden state after processing the sequence. We did the same in the previous chapter when we implemented the seq2seq architecture.

The `Preprocessor` class is a bit more complicated, as it combines several `Encoder` instances and is also responsible for embeddings:

```
class Preprocessor(nn.Module):
    def __init__(self, dict_size: int, emb_size: int,
                 num_sequences: int, enc_output_size: int):
        super(Preprocessor, self).__init__()

        self.emb = nn.Embedding(num_embeddings=dict_size,
                                embedding_dim=emb_size)
        self.encoders = []
        for idx in range(num_sequences):
            enc = Encoder(emb_size, enc_output_size)
            self.encoders.append(enc)
            self.add_module(f"enc_{idx}", enc)
        self.enc_commands = Encoder(emb_size, enc_output_size)
```

In the constructor, we create an embeddings layer, which will map every token in our dictionary into a fixed-size dense vector. Then we create `num_sequences` instances of `Encoder` for every input sequence and one additional instance to encode command tokens.

```
    def _apply_encoder(self, batch: List[List[int]],
                       encoder: Encoder):
        dev = self.emb.weight.device
        batch_t = [self.emb(torch.tensor(sample).to(dev))
                   for sample in batch]
        batch_seq = rnn_utils.pack_sequence(
```

```
            batch_t, enforce_sorted=False)
        return encoder(batch_seq)
```

The internal method `_apply_encoder()` takes the batch of sequences (every sequence is a list of token IDs) and transforms it with an encoder. If you remember, in the previous chapter, we needed to sort a batch of variable-length sequences before RNN application. Since PyTorch 1.0, this is no longer needed, as this sorting and transformation is handled by the `PackedSequence` class internally. To enable this functionality, we need to pass the `enforce_sorted=False` parameter.

```
        def encode_sequences(self, batches):
            data = []
            for enc, enc_batch in zip(self.encoders, zip(*batches)):
                data.append(self._apply_encoder(enc_batch, enc))
            res_t = torch.cat(data, dim=1)
            return res_t


        def encode_commands(self, batch):
            return self._apply_encoder(batch, self.enc_commands)
```

Two public methods of the class are just calling the `_apply_encoder()` method to transform input sequences and commands. In the case of sequences, we concatenate the batch along the second dimension to combine fixed-size vectors produced by individual encoders into a single vector.

Now, let's see how the `Preprocessor` class is supposed to be applied:

```
In [1]: from textworld.gym import register_game
In [2]: import gym
In [3]: from lib import preproc
In [4]: from textworld.envs.wrappers.filter import EnvInfos
In [5]: env_id = register_game("games/simple1.ulx", request_
infos=EnvInfos(inventory=True, intermediate_reward=True, admissible_
commands=True, description=True))
In [6]: env = gym.make(env_id)
In [7]: env = preproc.TextWorldPreproc(env)
In [8]: prep = preproc.Preprocessor(dict_size=env.observation_space.
vocab_size, emb_size=10, num_sequences=env.num_fields, enc_output_
size=20)
In [9]: prep
Out[9]:
Preprocessor(
  (emb): Embedding(1250, 10)
```

```
  (enc_0): Encoder(
    (net): LSTM(10, 20, batch_first=True)
  )
  (enc_1): Encoder(
    (net): LSTM(10, 20, batch_first=True)
  )
  (enc_commands): Encoder(
    (net): LSTM(10, 20, batch_first=True)
  )
)
```

We have created a `Preprocessor` instance, which includes embeddings of size 10 and encodes two input sequences and the command text into vectors of size 20. To apply it, we need to pass input batches to one of its public methods.

```
In [10]: obs = env.reset()
In [11]: prep.encode_sequences([obs['obs']])
Out[11]: torch.Size([1, 40])
In [12]: prep.encode_commands(obs['admissible_commands']).size()
Out[12]: torch.Size([12, 20])
```

The dimensions of the outputs meet our expectations, as a single observation of two sequences is 2 × 20 = 40 numbers, and a list of 12 commands is encoded as a tensor of the shape (12, 20).

The DQN model and the agent

With all those preparations made, the DQN model is quite obvious: it should accept vectors of num_sequences × encoder_size and produce a single scalar value. But there is one difference from the other DQN models covered, which is in the way we apply the model.

```
class DQNModel(nn.Module):
    def __init__(self, obs_size: int, cmd_size: int,
                 hid_size: int = 256):
        super(DQNModel, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(obs_size + cmd_size, hid_size),
            nn.ReLU(),
            nn.Linear(hid_size, 1)
        )
```

```
def forward(self, obs, cmd):
    x = torch.cat((obs, cmd), dim=1)
    return self.net(x)

@torch.no_grad()
def q_values(self, obs_t, commands_t):
    result = []
    for cmd_t in commands_t:
        qval = self(obs_t, cmd_t.unsqueeze(0))[0].cpu().item()
        result.append(qval)
    return result
```

The `forward()` method accepts two batches—observations and commands—producing the batch of Q-values for every pair. Another method, `q_values()`, takes one observation produced by the `Preprocessor` class and the tensor of encoded commands, then applies the model and returns a list of Q-values for every command.

One final class worth showing is the PTAN agent, which combines all the logic together and converts environment observations (after the `TextWorldPreproc` class) into the list of actions to be taken.

```
class DQNAgent(ptan.agent.BaseAgent):
    def __init__(self, net: DQNModel,
                 preprocessor: preproc.Preprocessor,
                 epsilon: float = 0.0, device="cpu"):
        self.net = net
        self._prepr = preprocessor
        self._epsilon = epsilon
        self.device = device

    @property
    def epsilon(self):
        return self._epsilon

    @epsilon.setter
    def epsilon(self, value: float):
        if 0.0 <= value <= 1.0:
            self._epsilon = value

    @torch.no_grad()
    def __call__(self, states, agent_states=None):
        if agent_states is None:
            agent_states = [None] * len(states)

        # for every state in the batch, calculate
```

```
actions = []
for state in states:
    commands = state['admissible_commands']
    if random.random() <= self.epsilon:
        actions.append(random.randrange(len(commands)))
    else:
        obs_t = self._prepr.encode_sequences(
            [state['obs']]).to(self.device)
        commands_t = self._prepr.encode_commands(commands)
        commands_t = commands_t.to(self.device)
        q_vals = self.net.q_values(obs_t, commands_t)
        actions.append(np.argmax(q_vals))
return actions, agent_states
```

The logic is quite straightforward, but it is still illustrative of how the preprocessing pipeline and DQN model are combined.

Training code

With all the preparations and preprocessing in place, the rest is almost the same as we have covered before, so I won't repeat the training code; I will just describe the training logic.

To train the model, the `Chapter15/train_basic.py` utility has to be used. It allows several command-line arguments to change the training behavior:

- `-g` or `--game`: This is the prefix of the game files in the `games` directory. The provided script generates several games named `simpleNN.ulx`, where NN is the game seed.

- `-s` or `--suffices`: This is the count of games to be used during the training. If you specify 1 (which is the default), the training will be performed only on the file `simple1.ulx`. If option `-s 10` is given, 10 games with indices 1…10 will be registered and used for training. This option is used for generalization experiments, which will be explained in the following section.

- `-v` or `--validation`: This is the suffix of the game to be used for validation. It is `-val` by default and defines the game file that will be used to check the generalization of our trained agent.

- `--params`: This means the hyperparameters to be used. Two sets are defined in `lib/common.py`: `small` and `medium`. The first one has a small number of embeddings and encoder vectors, which is great for solving a couple of games quickly; however, this set struggles with converging when many games are used for training.

- `--cuda`: This enables CUDA training.
- `-r` or `--run`: This is the name of the run and is used in the name of the save directory and TensorBoard.

During the training, validation is performed every 100 training iterations and runs the validation game on the current network. The reward and the number of steps are recorded in TensorBoard and help us to understand the generalization capabilities of our agent. Generalization in RL is known to be a large issue. With a limited set of trajectories, the training process has a tendency to overfit to some states, which doesn't guarantee good behavior on unseen games. In comparison to Atari games, where the gameplay normally doesn't change much, the variability of interactive fiction games might be high, due to different quests, objects, and the way they communicate. So, it's an interesting experiment to check how our agent is able to generalize between games.

Training results

By default, the script `games/make_games.sh` generates 20 games with names from `simple1.ulx` to `simple20.ulx`, plus a game for validation: `simple-val.ulx`.

To begin, let's train the agent on one game, using the `small` hyperparameters set:

```
$ ./train_basic.py  -s 1 --cuda -r t1
```

Option `-s` specifies the number of game indices that will be used for training. In this case, only one will be used. The training stops when the average number of steps in the game drops below 10, which means the agent has found the proper sequence of steps and can reach the end of the game in an efficient way.

In the case of one game, it takes just half an hour and about 300 episodes to solve the game. The following charts show the reward and number of steps dynamics during the training.
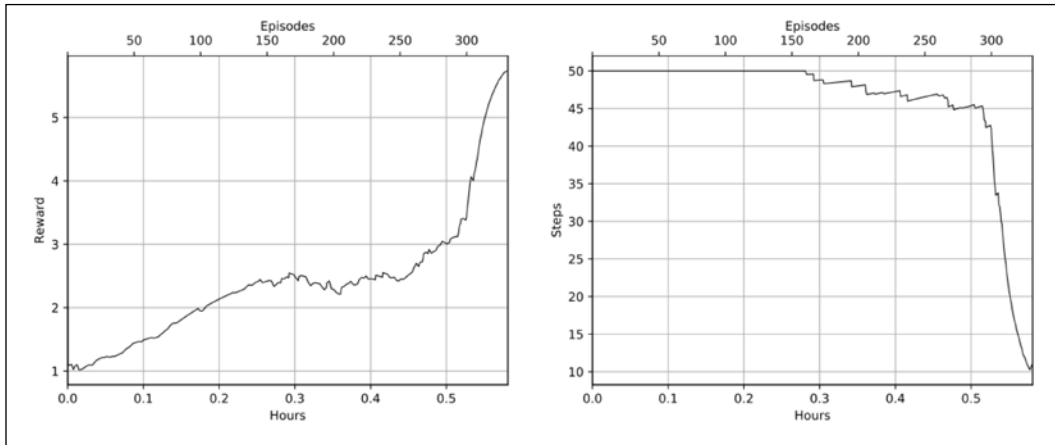
Figure 15.4: The reward and steps dynamics for training on one game

But if we check the validation reward (which is a reward obtained on the game `simple-val.ulx`), we see zero improvement over time. In my case, which is shown in *Figure 15.5*, validation got a negative reward by doing something weird.
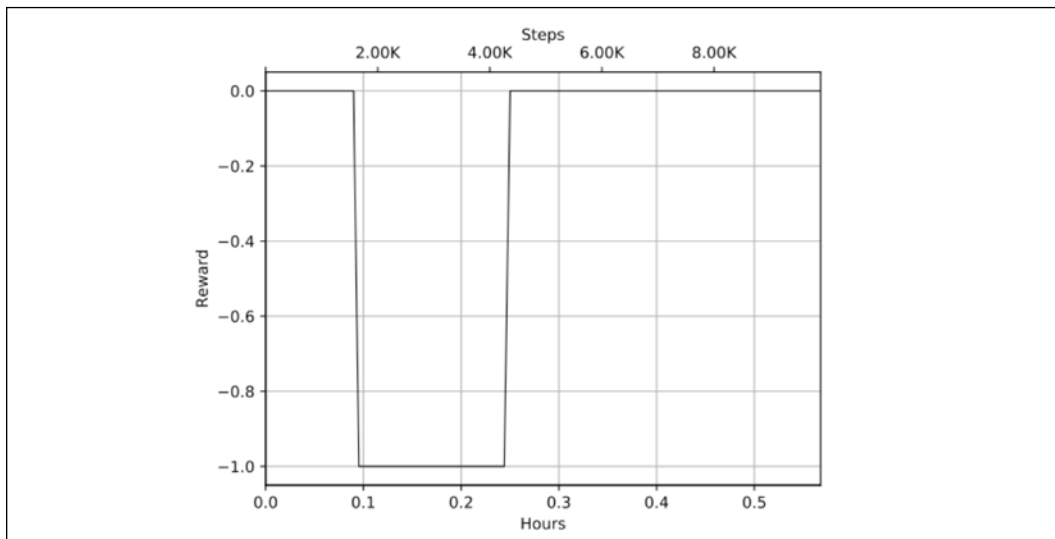
Figure 15.5: The validation reward for one game

If we try to increase the number of games used for the training, the convergence will require more time, as the network needs to discover more sequences of actions in different states. The following are the same charts for reward and steps for 10 games (with option `-s 10` passed).
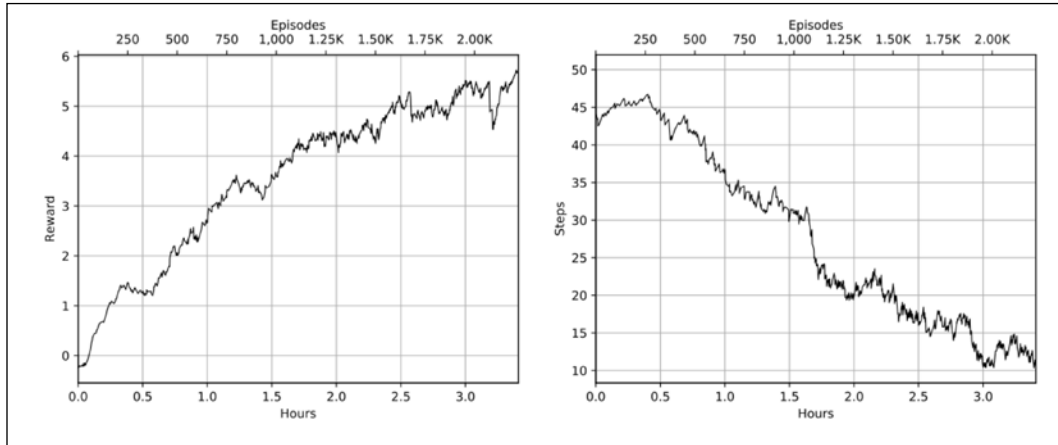
Figure 15.6: The reward and steps dynamics on 10 games

As you can see, it takes more than three hours to converge, but still our small hyperparameter set is able to improve the performance on 10 games played during the training. Unfortunately, the validation metrics are still not that great. In the middle of the training, the network was able to do one right action in the sequence (out of a total of five), but there was no improvement over time.
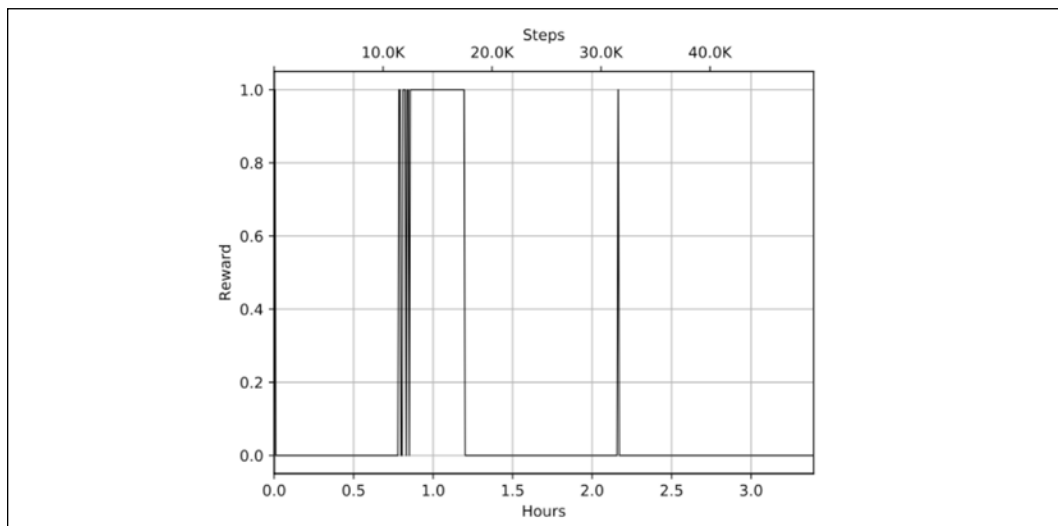
Figure 15.7: The reward obtained on the validation game during training on 10 games

Increasing the number of games doesn't improve the situation much: the agent is still able to improve the reward on all the games, but it just takes longer to converge. Yet the validation game still took only one proper step.

To increase the network capacity, I experimented with a larger set of hyperparameters, which has both larger embeddings dimensionality (128 versus 20) and a larger encoder vector size (256 versus 20). It also has a larger replay size and anneals the epsilon parameter for longer. To switch to this set of hyperparameters, the command-line option `--params medium` needs to be used. This option specifies the entry in the dictionary defined in `lib/common.py`, so you can add your own set of hyperparameters for experimentation.

To compare the effect of increasing the model capacity, the following charts were obtained from training on 25 games in the `medium` and `small` hyperparameters sets.
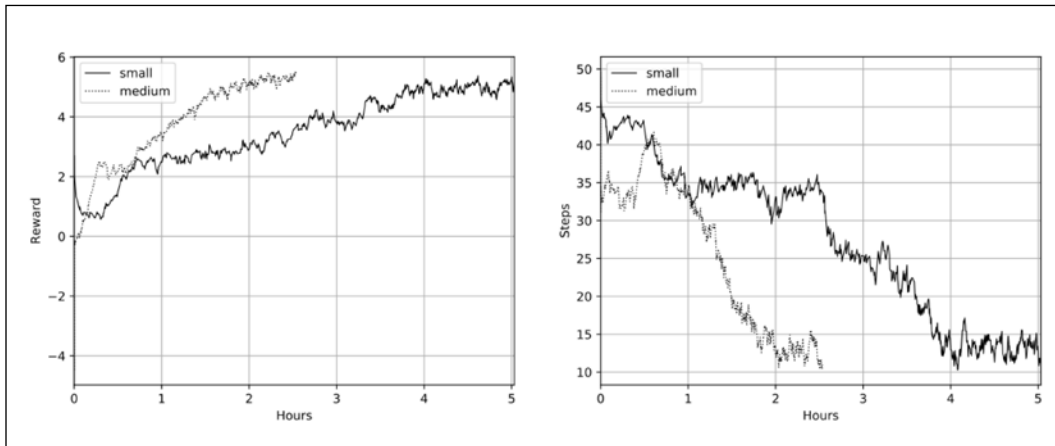
Figure 15.8: Reward and steps for different hyperparameters trained on 25 games

As you can see, the larger network was able to solve the games almost two times faster than the small one. But the following validation chart shows the opposite result. The small network got a better validation score than the medium one, which is also not surprising, as the larger network has more capacity to overfit to the trained episodes.

The small network needs to find the efficient representation of all the states it sees, which is more likely to be useful in unseen games.
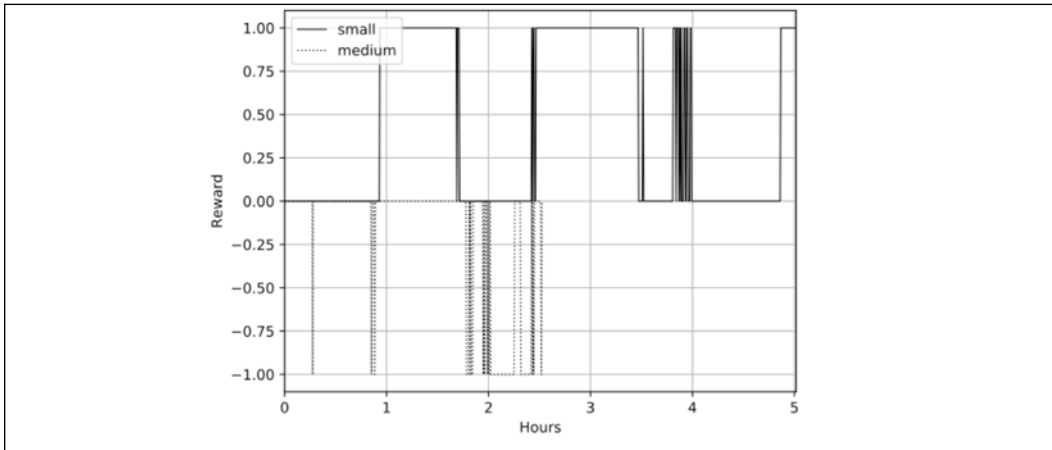
Figure 15.9: The validation reward for different hyperparameters on 25 games

The practical takeaway from this is that if your setup allows you to train on all regimes of your environment, you can consider increasing the network size to learn faster. But if there is the chance of something new and unusual, it might be safer to train a smaller network for a longer time, to finally get a more robust network. Of course, it's not a universal solution to this tricky problem of RL generalization and handling unseen data, but it's just something to keep in mind.

Figure 15.10 shows the results of my final extreme experiment, with 200 games used for training on `medium` hyperparameters. It was able to solve all of them, but took about 12 hours to converge.
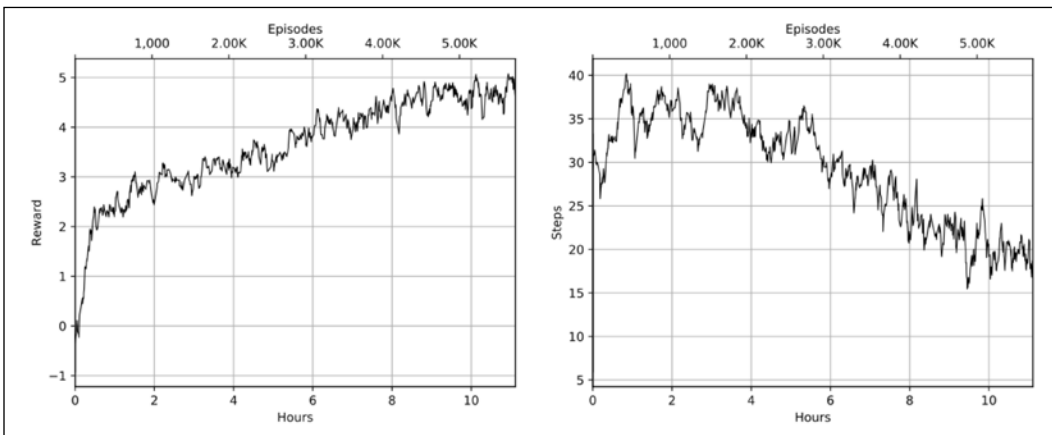
Figure 15.10: The results on 200 games used for training

But, unfortunately, even with the large number of games, the validation wasn't able to get above reward=1 per episode. Maybe even more games are needed, some bug is present in the code, or other improvements are needed. Overall, there is a lot of room for experimentation. For example, if, during validation, the agent gets stuck in some state and makes a choice to take the same action over and over again, we can push it from this local optimum and select the second largest Q-value.
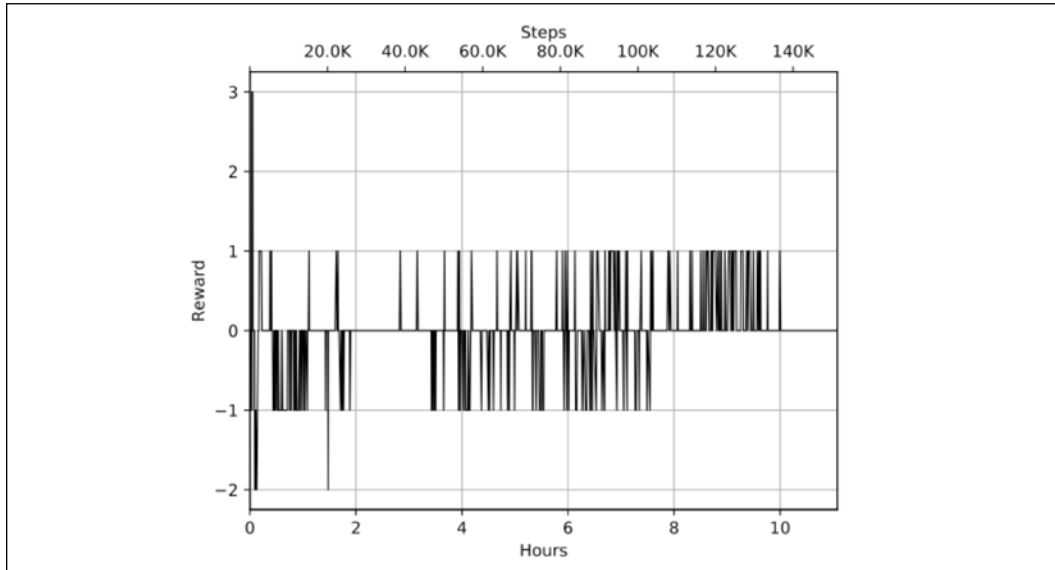
Figure 15.11: Validation results for 200 games

The command generation model

In this part of the chapter, we will extend our baseline model with an extra submodule that will generate commands that our DQN network should evaluate. In the baseline model, commands were taken from the admissible commands list, which was taken from the extended information from the environment. But maybe we can generate commands from the observation using the same techniques that we covered in the previous chapter.

The architecture of our new model is shown in *Figure 15.12*.
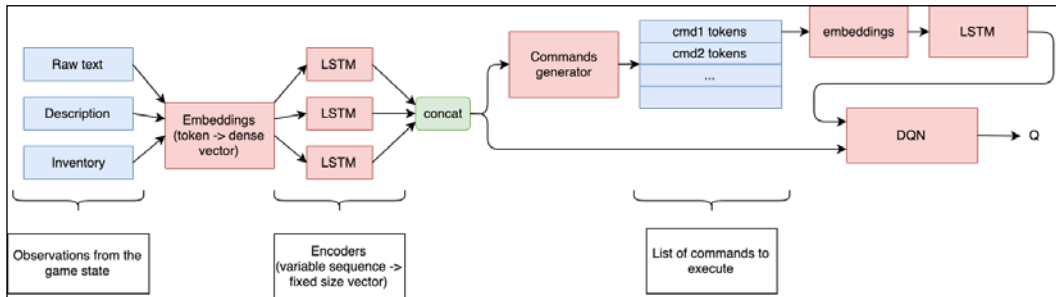
Figure 15.12: The architecture of the DQN with command generation

In comparison with *Figure 15.3* from earlier in the chapter, there are several changes here. First of all, our preprocessor pipeline no longer accepts a command sequence in the input. The second difference is that the preprocessor's output now not only gets passed to the DQN model, but it also forks to the "Commands generator" submodule.

The responsibility of this new submodule is to produce several commands to be evaluated by the DQN network for potential reward. Once commands are generated, they are processed by the LSTM encoder in the same way they were before to convert the sequence into a fixed-size command representation.

Zooming in, the "Command generator" submodule includes the already familiar LSTM network, accompanied by a single-layer feed-forward network to convert the LSTM output into our vocabulary probability logits. The structure of the "Command generator" block is shown in the following figure:
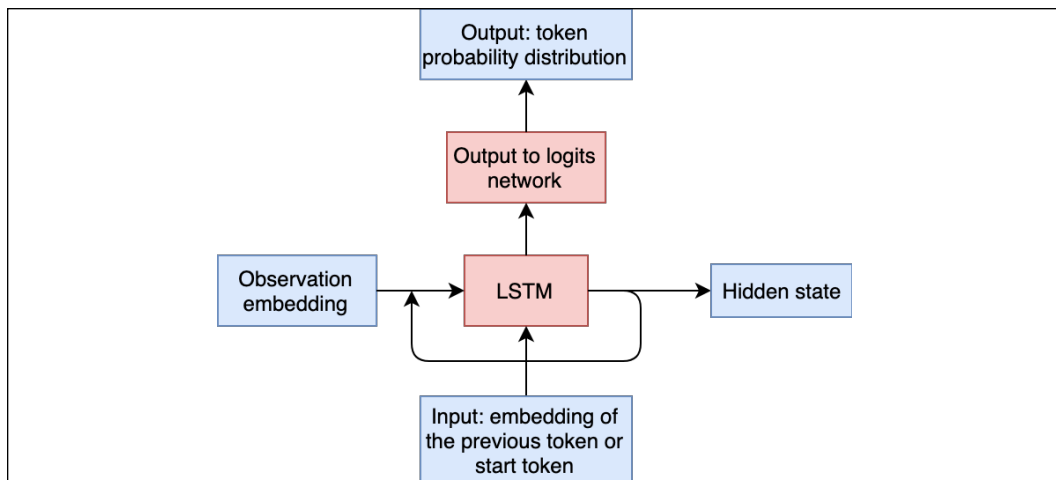
Figure 15.13: The command generator internal structure and connections

The model is very similar to the one we used in our example in *Chapter 14*, *Training Chatbots with RL*. At the first step of the command generator application, we feed an observation vector produced by encoders into the hidden state of the LSTM. On input, it gets a special *start of the sequence* token. The output from the LSTM is transformed by the feed-forward network, which produces a probability distribution over the token vocabulary. To get the actual produced token, we sample from this distribution, then this token is passed as an input to the subsequent step. Some output tokens are treated specially; for example, an *end of the sequence* token is used as a separator between commands. So, basically, that's just a classical left-to-right language model applied to our game commands.

The training of our example will be done in two steps:

1. First, we will do pretraining of the language model and preprocessor using the game states and admissible commands provided by the environment. As an objective, classical cross-entropy loss between commands produced by the command generator and the true list of admissible commands will be used. When this step reaches a point at which we can generate commands that are understood by the game, we will go to the next training step.

2. As the second and final step, we will freeze the weights of the preprocessor and command generator networks and train the command encoder and DQN network.

That's not the only option, and I'm not even sure that's the optimal way to do it. It was chosen for simplicity, and you can always experiment and try your own ideas.

As the code is complex and lengthy, I have included only the important parts of the implementation in the following section. The complete example can be found in `Chapter15/train_lm.py` and dependent modules.

Implementation

Let's start with the command generator, as it is the most complicated new piece in this example. It resides in `lib/model.py`, class `CommandModel`.

```
class CommandModel(nn.Module):
    def __init__(self, obs_size: int, dict_size: int,
                 embeddings: nn.Embedding, max_tokens: int,
                 max_commands: int, start_token: int,
                 sep_token: int):
        super(CommandModel, self).__init__()

        self.emb = embeddings
        self.max_tokens = max_tokens
```

```
        self.max_commands = max_commands
        self.start_token = start_token
        self.sep_token = sep_token

        self.rnn = nn.LSTM(
            input_size=embeddings.embedding_dim,
            hidden_size=obs_size, batch_first=True)
        self.out = nn.Linear(in_features=obs_size,
                             out_features=dict_size)
```

The constructor accepts a lot of arguments, defining the dimensionality of the input observation vector, the count of tokens, the embeddings to be used, and the token IDs for the start and separator tokens. The arguments `max_tokens` and `max_commands` specify the limits of tokens in each command generator and the count of commands we want to produce. Those limits are needed, as our command model can easily go overboard and start to generate very long commands, which doesn't make much sense. As our games normally have quite short commands, we can set this limitation in advance. In the constructor's body, we create both the LSTM and output transformation layers.

```
        def forward(self, input_seq, obs_t):
            hid_t = obs_t.unsqueeze(0)
            output, _ = self.rnn(input_seq, (hid_t, hid_t))
            return self.out(output)
```

The `forward` method is quite simple, but this is offset by the class' complex and lengthy `commands()` method. In the forward pass, we apply LSTM and pass the observations vector in the hidden state. The LSTM layer has two tensors in the hidden state: one is the *hidden state* and the other is the *cell state*. For simplicity, I just pass the observation to both inputs, but it might not be the optimal thing to do. Zero tensors as the cell state might work better.

Now we will cover the last method in the class, which generates commands for the batch of observations. This method is complicated, as we generate the commands token by token and need to track the end-of-sequence condition.

```
        def commands(self, obs_batch):
            batch_size = obs_batch.size(0)
            commands = [[] for _ in range(batch_size)]
            cur_commands = [[] for _ in range(batch_size)]

            inp_t = torch.full((batch_size, ), self.start_token,
                               dtype=torch.long)
            inp_t = inp_t.to(obs_batch.device)
            inp_t = self.emb(inp_t)
```

```
# adding time dimension (dim=1, as batch_first=True)
inp_t = inp_t.unsqueeze(1)
# hidden state is inserted on first dim
p_hid_t = obs_batch.unsqueeze(0)
hid = (p_hid_t, p_hid_t)
```

In the beginning, we create the list for completed commands and the list with commands currently being built. Then, we create a tensor, `inp_t`, which at the first step has the same token for all entries in the batch: the start token. The LSTM layer's hidden state is constructed from observations provided on the input.

```
while True:
    out, hid = self.rnn(inp_t, hid)
    out = out.squeeze(1)
    out_t = self.out(out)

    cat = t_distr.Categorical(logits=out_t)
    tokens = cat.sample()
```

On every loop iteration, we apply LSTM to the current token, convert the output from LSTM into a probability distribution, construct the utility class `torch.distributions.Categorical`, and sample the next token from this distribution. The output is a `LongTensor` with tokens for every sample in the batch.

```
for idx, token in enumerate(tokens):
    token = token.item()
    cur_commands[idx].append(token)
    if token == self.sep_token or \
            len(cur_commands[idx]) >= self.max_tokens:
        if cur_commands[idx]:
            l = len(commands[idx])
            if l < self.max_commands:
                commands[idx].append(
                    cur_commands[idx])
            cur_commands[idx] = []
        if token != self.sep_token:
            tokens[idx] = self.sep_token
```

Then, we iterate for every token in the produced batch and populate our commands lists. We need to process situations when LSTM generates commands longer than our limits and stop the command sequence.

```
if min(map(len, commands)) == self.max_commands:
    break
# convert tokens into input tensor
inp_t = self.emb(tokens)
```

```
                inp_t = inp_t.unsqueeze(1)
        return commands
```

At the end of the loop, we check that we have got enough commands, and if that's the case, we return them from the function.

The next new class is the agent that uses the command generator to convert observations into actions. If you remember, at the beginning of the chapter, you saw the class `TextWorldPreproc`, one of the responsibilities of which was to transform the action space from a free-text string into the index in the admissible commands list. In this part of the chapter, our commands are generated, rather than chosen from the admissible commands list, so this transformation to the environment's action space is disabled using the `use_admissible_commands=False` option in the `TextWorldPreproc` constructor. In addition to that, this preprocessor class is extended to check for situations when generated commands are nonsense for the environment. In this case, we give a small negative reward to the agent. (Check how the `reward_wrong_last_command` option in `TextWorldPreproc` is handled.)

The following code is the `CmdAgent` class, which is used in the pretraining phase. Its responsibility is to generate an observation vector using the preprocessor pipeline, generate the list of commands, and return the randomly chosen command in the environment.

```
class CmdAgent(ptan.agent.BaseAgent):
    def __init__(self, env, cmd: CommandModel,
                 preprocessor: preproc.Preprocessor,
                 device = "cpu"):
        self.env = env
        self.cmd = cmd
        self.prepr = preprocessor
        self.device = device

    @torch.no_grad()
    def __call__(self, states, agent_states=None):
        if agent_states is None:
            agent_states = [None] * len(states)

        actions = []
        for state in states:
            obs_t = self.prepr.encode_sequences(
                [state['obs']]).to(self.device)
            commands = self.cmd.commands(obs_t)[0]
            cmd = random.choice(commands)
            tokens = [
                self.env.action_space.id2w[t]
```

```
                for t in cmd
                if t not in {self.cmd.sep_token,
                             self.cmd.start_token}
            ]
            action = " ".join(tokens)
            actions.append(action)
        return actions, agent_states
```

The code is not very complicated, so I will leave it without much explanation.

The final piece in our pretraining phase is the way we calculate loss. That's the responsibility of the `pretrain_loss` function in `lib/model.py`.

```
def pretrain_loss(cmd: CommandModel, commands: List,
                  observations_t: torch.Tensor):
    commands_batch = []
    target_batch = []
    min_length = None

    for cmds in commands:
        inp = [cmd.start_token]
        for c in cmds:
            inp.extend(c[1:])
        commands_batch.append(inp[:-1])
        target_batch.append(inp[1:])
        if min_length is None or len(inp) < min_length:
            min_length = len(inp)

    commands_batch = [c[:min_length-1]
                        for c in commands_batch]
    target_batch = [c[:min_length-1]
                      for c in target_batch]
```

In the first half of the function, we take the input batch, every item of which is a list of commands, and linearize this list into the single sequence that we expect our command generator to learn. As every command includes start and stop sequence tokens, we get rid of them in our target and input sequences. Then, we trim the batch on the shortest sequence.

```
        commands_t = torch.tensor(commands_batch, dtype=torch.long)
        commands_t = commands_t.to(observations_t.device)
        target_t = torch.tensor(target_batch, dtype=torch.long)
        target_t = target_t.to(observations_t.device)
        input_t = cmd.emb(commands_t)
        logits_t = cmd(input_t, observations_t)
```

```
logits_t = logits_t.view(-1, logits_t.size()[-1])
target_t = target_t.view(-1)
return F.cross_entropy(logits_t, target_t)
```

When the preparation steps have been completed, we convert our sequences into the tensor, apply embeddings, and convert the result into the logits using the command generator. Then, we apply cross-entropy loss to drive the command generator toward the desired sequences.

Pretraining results

By default, the program `train_lm.py` starts the pretraining of the command generator mode, and once it reaches the average reward of –0.5, it switches to DQN training. During the training, the initial reward starts from values of –5, as there are 50 steps in the environment, and for every bad command that is not understood by the environment, the agent gets a reward of –0.1. So, getting an average reward of –0.5 means our command generator is good enough to generate valid commands, which don't necessarily give a positive reward, but at least make sense for the TextWorld environment.
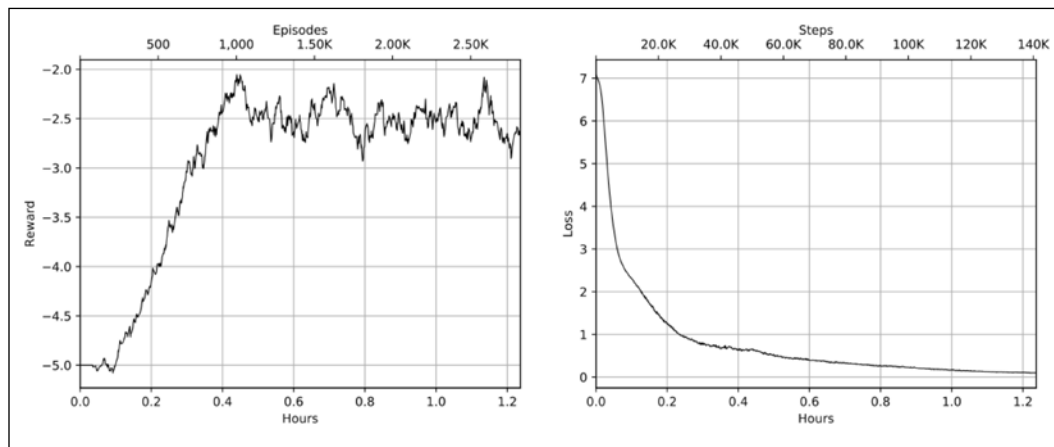
Figure 15.14: Reward and loss for pretraining on one game with a "small" hyperparameter set
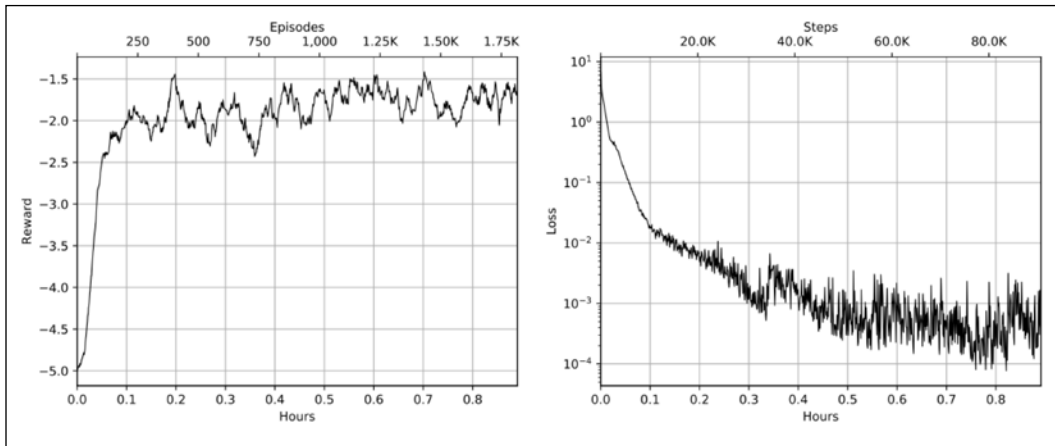
Figure 15.15: Reward and loss for pretraining on one game with "medium" hyperparameters

From these results, it's obvious that the `medium` hyperparameter set has a much larger capacity for learning the data, which suggests a further increase of the model's complexity (which is left for the reader to explore). Another, less obvious, conclusion is that pretraining is much faster than baseline training. From the metrics, the speed of pretraining is ~35 observations per second, whereas the baseline is ~5.

With a larger number of games used for training, the difference between small and medium models is much more vivid. For example, *Figure 15.16* shows the comparison between two setups on five games.
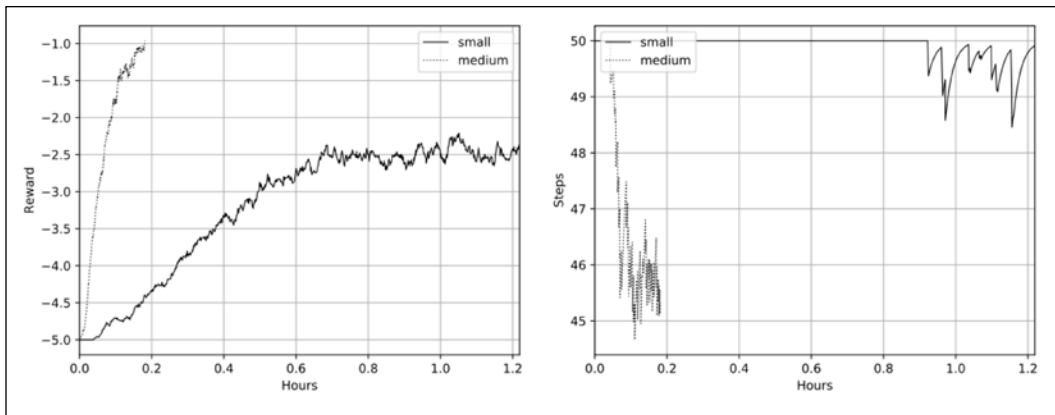
Figure 15.16: Reward and steps for different model sizes with five games

DQN training code

The second phase of the training comes after the command generator has mastered generating command sequences that are good enough not to surprise the environment too much. At this stage, we freeze the preprocessor pipeline and the command generator, and start training the DQN and command encoder networks the same way we did for our baseline version. But the code is different, as our architecture has been modified. For instance, the agent now needs to ask the command generator to produce the list of commands to be evaluated by the DQN network. The new agent has been implemented in the `CmdDQNAgent` class in `lib/model.py`. The logic is quite straightforward, so I won't include it here.

What might be interesting to look at is how the loss is calculated. It is implemented in two functions: `unpack_batch_dqncmd` and `calc_loss_dqncmd` in `lib/model.py`.

```python
@torch.no_grad()
def unpack_batch_dqncmd(batch, prep: preproc.Preprocessor,
                        cmd: CommandModel,
                        cmd_encoder: preproc.Encoder,
                        net: DQNModel, env: gym.Env):
    observations, taken_actions, rewards = [], [], []
    not_done_indices, next_observations = [], []

    for idx, exp in enumerate(batch):
        observations.append(exp.state['obs'])
        taken_actions.append(
            env.action_space.tokenize(exp.action))
        rewards.append(exp.reward)
        if exp.last_state is not None:
            not_done_indices.append(idx)
            next_observations.append(exp.last_state['obs'])

    observations_t = prep.encode_sequences(observations)
    next_q_vals = [0.0] * len(batch)
    if next_observations:
        next_observations_t = \
            prep.encode_sequences(next_observations)
        next_commands = cmd.commands(next_observations_t)

        for idx, next_obs_t, next_cmds in \
                zip(not_done_indices,
                    next_observations_t,
                    next_commands):
            next_embs_t = prep._apply_encoder(
                next_cmds, cmd_encoder)
```

```
            q_vals = net.q_values_cmd(next_obs_t, next_embs_t)
            next_q_vals[idx] = max(q_vals)

    return observations_t, taken_actions, rewards, next_q_vals
```

The responsibility of the `unpack_batch_dqncmd` function is to prepare the observations tensor, the list of tokenized commands executed, the immediate reward, and (what is most complicated to get) the best Q-values for the next state of the transition. To get the best Q-values, we generate commands for the next state, evaluate them using the DQN, and then take the maximum Q-value.

```
def calc_loss_dqncmd(batch, preprocessor, cmd,
                     cmd_encoder, tgt_cmd_encoder,
                     net, tgt_net, gamma, env, device="cpu"):
    obs_t, commands, rewards, next_best_qs = unpack_batch_dqncmd(
        batch, preprocessor, cmd, tgt_cmd_encoder, tgt_net, env)
    cmds_t = preprocessor._apply_encoder(commands, cmd_encoder)
    q_values_t = net(obs_t, cmds_t)
    tgt_q_t = torch.tensor(rewards) + \
            gamma * torch.tensor(next_best_qs)
    tgt_q_t = tgt_q_t.to(device)
    return F.mse_loss(q_values_t.squeeze(-1), tgt_q_t)
```

To calculate the loss, we unpack the batch, apply the encoder to the executed commands, and calculate the mean squared error between the value produced by the network and the Q-values approximation using the Bellman equation. There isn't anything new here, but you need to be careful with gradients. For example, it might be tempting to apply the command encoder in `unpack_batch_dqncmd`, but that would be wrong, as it will block gradients in our command encoder, which is not what we want.

The result of DQN training

After the pretraining step reaches the average reward of –0.5 (or, if you've become bored waiting, you can press *Ctrl+C* in the console), it switches to DQN training. Before that, it saves the weights of the command generator and preprocessor, so they can be loaded later by passing the save directory in the `--load-cmd` command-line option.

The following figures show the results of my experiments with models of various sizes.
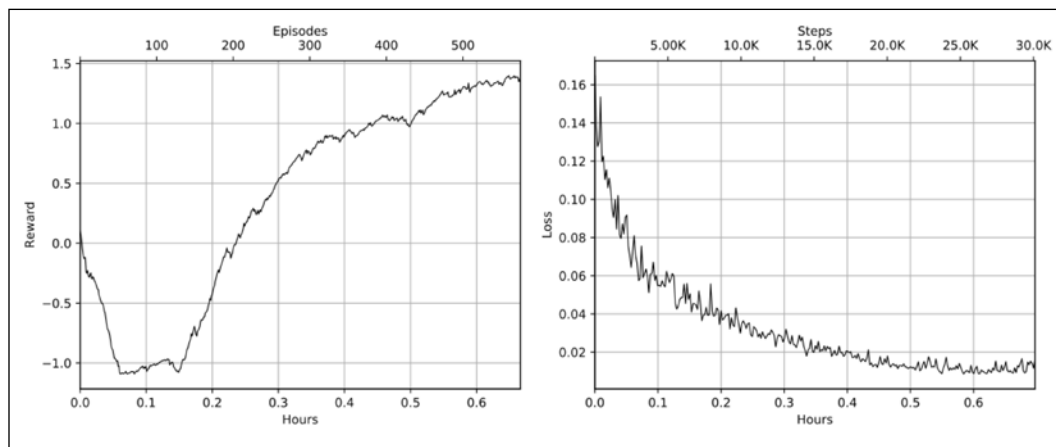
Figure 15.17: The reward and loss of DQN training on "small" hyperparameters with one game
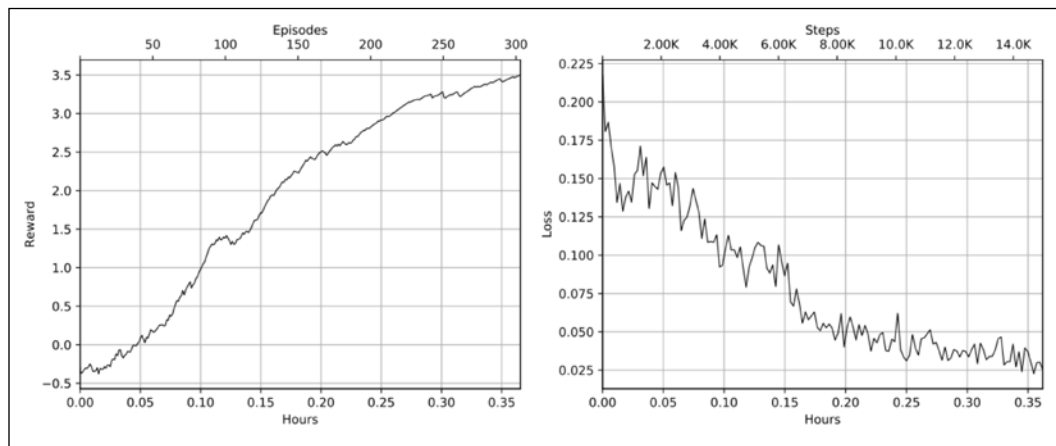
Figure 15.18: The reward and loss of DQN training on "medium" hyperparameters with one game

From these figures, you can see the same results as before: the learning capacity of the larger network is higher (it took the larger network only 30 minutes to get to 3.5 average reward). The performance of the DQN step is at ~12 observations per second.

In terms of generalization, the results are still not great. The small network showed some progress in the middle of the training, but it wasn't consistent. The following chart shows the validation reward for both networks.
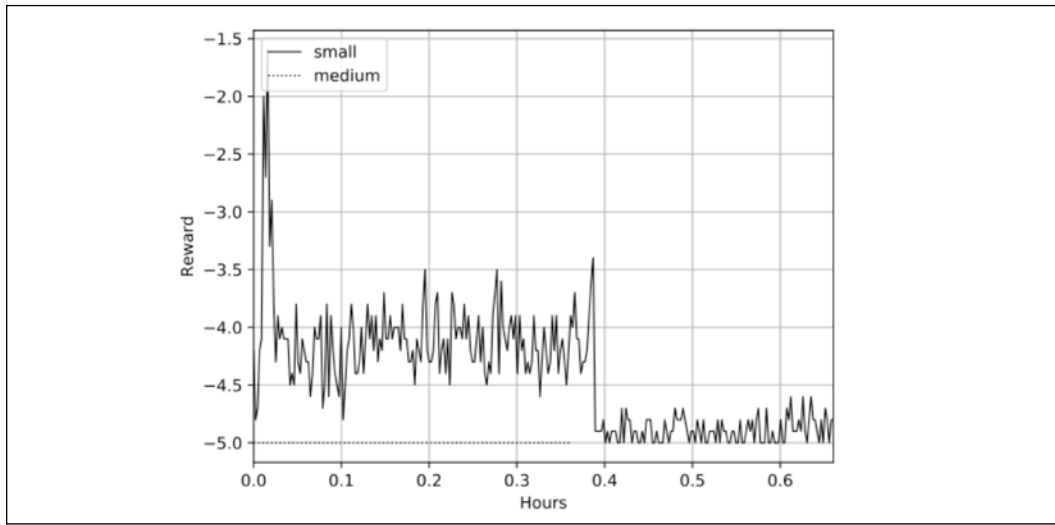
Figure 15.19: The validation reward for small and medium networks trained on one game

Summary

In this chapter, you have seen how DQN can be applied to interactive fiction games, which is an interesting and challenging domain at the intersection of RL and NLP. You learned how to handle complex textual data with NLP tools and experimented with fun and challenging interactive fiction environments, with lots of opportunities for future experimentations.

In the next chapter, we will continue our exploration of "RL in the wild" and check the applicability of methods in web automation.

16
Web Navigation

We will now take a look at another practical application of reinforcement learning (RL): web navigation and browser automation.

In this chapter, we will:

- Discuss **web navigation** in general and the practical application of browser automation
- Explore how web navigation can be solved with an RL approach
- Take a deep look at one very interesting, but commonly overlooked and a bit abandoned, RL benchmark that was implemented by OpenAI, called **Mini World of Bits** (**MiniWoB**)

Web navigation

When the web was invented, it started as several text-only web pages interconnected by hyperlinks. If you're curious, here is the home of the first web page: `http://info.cern.ch/`, with text and links. The only thing you can do is read the text and click on links to go between pages.

Several years later, in 1995, the Internet Engineering Task Force (IETF) published the HTML 2.0 specification, which had a lot of extensions to the original version invented by Tim Berners-Lee. Among these extensions were forms and form elements that allowed web page authors to add activity to their websites. Users could enter and change text, toggle checkboxes, select drop-down lists, and push buttons. The set of controls was similar to a minimalistic set of graphical user interface (GUI) application controls. The difference was that this happened inside the browser's window, and both the data and user interface (UI) controls that users interacted with were defined by the server's page, but not by the local installed application.

Fast forward 22 years, and now we have JavaScript, HTML5 canvas, and office applications working inside our browsers. The boundary between the desktop and the web is so thin and blurry that you may not even know whether the app you're using is an HTML page or a native app. However, it is still the browser that understands HTML and communicates with the outside world using HTTP.

At its core, web navigation is defined as a process of a user interacting with the website or websites. The user can click on links, type text, or do any other actions to reach some goal, such as sending an email, finding out the exact dates of the French Revolution, or checking recent Facebook notifications. All this will be done using web navigation, so that leaves a question: can our program learn how to do the same?

Browser automation and RL

For a long time, automating website interaction focused on the very practical tasks of **website testing** and **web scraping**. Website testing is especially critical when you have some complicated website that you (or other people) have developed and you want to ensure that it does what it is supposed to do. For example, if you have a login page that was redesigned and is ready to be deployed on a live website, then you will want to be sure that this new design does sane things in case a wrong password is entered, the user clicks on **I forgot my password**, and so on. A complex website could potentially include hundreds or thousands of use cases that should be tested on every release, so all such functions should be automated.

Web scraping solves the problem of extracting data from websites at scale. For example, if you want to build a system that aggregates all prices for all pizza places in your town, you will potentially need to deal with hundreds of different websites, which could be problematic to build and maintain. Web scraping tools try to solve the problem of interacting with websites. Their functionality can vary from simple HTTP requests and subsequent HTML parsing to full emulation of the user moving the mouse, clicking buttons, thinking, and so on.

The standard approach to browser automation normally allows you to control the real browser, such as Chrome or Firefox, with your program, which can observe the web page data, like the Document Object Model (DOM) tree and an object's location on the screen, and issue the actions, like moving the mouse, pressing some keys, pushing the **Back** button or just executing some JavaScript code. The connection to the RL problem setup is obvious: our agent interacts with the web page and browser by issuing actions and observing some state. The reward is not that clear and should be task-specific, like successfully filling some form in or reaching the page with the desired information.

Practical applications of a system that could learn browser tasks are related to the previous use cases. For example, in web testing for very large websites, it's extremely tedious to define the testing process using low-level browser actions like "move the mouse five pixels to the left, then press the left button." What you want to do is give the system some demonstrations and let it generalize and repeat the shown actions in all similar situations, or at least make it robust enough for UI redesign, button text change, and so on. Additionally, there are many cases when you don't know the problem in advance, for example, when you want the system to explore the weak points of the website, like security vulnerabilities. In that case, the RL agent could try a lot of weird actions very quickly, much faster than humans could. Of course, the action space for security testing is enormous, so random clicking won't be very competitive with experienced human testers. In that case, the RL-based system could, potentially, combine the prior knowledge and experience of humans but still keep the ability to explore and learn from this exploration.

Another potential domain that could benefit from RL browser automation is scraping and web data extraction in general. For example, you might want to extract some data from hundreds of thousands of different websites, like hotel websites, car rental agents, or other businesses around the world. Very often, before you get to the desired data, a form with parameters needs to be filled out, which becomes a very nontrivial task given the different websites' design, layout, and natural language flexibility. With such a task at hand, an RL agent can save tons of time and effort by extracting the data reliably and at scale.

The MiniWoB benchmark

Potential practical applications of browser automation with RL are attractive but have one very serious drawback: they're too large to be used for research and the comparison of methods. In fact, the implementation of a full-sized web scraping system could take months of effort from a team, and most of the issues would not be directly related to RL, like data gathering, browser engine communication, input and output representation, and lots of other questions that real production system development consists of.

By solving all those issues, we can easily miss the forest by looking at the trees. That's why researchers love benchmark datasets, like MNIST, ImageNet, the Atari suite, and many others. However, not every problem makes a good benchmark. On the one hand, it should be simple enough to allow quick experimentations and comparison between methods. On the other hand, the benchmark has to be challenging and leave room for improvements. For example, Atari benchmarks consist of a wide variety of games, from very simple games, which could be solved in half an hour (like Pong), to quite complex games that haven't been properly solved yet (like Montezuma's Revenge, which requires the complex planning of actions).

To the best of my knowledge, there is only one such benchmark for the browser automation domain, which makes it even worse that this benchmark was undeservedly forgotten by the RL community. As an attempt to fix this issue, we will take a look at the benchmark in this chapter. Let's talk about its history first.

In December 2016, OpenAI published a dataset called MiniWoB that contains 80 browser-based tasks. These tasks are observed on a pixel level (strictly speaking, besides pixels, a text description of tasks is given to the agent) and are supposed to be communicated with the mouse and keyboard actions using the Virtual Network Computing (VNC) client (`https://en.wikipedia.org/wiki/Virtual_Network_Computing`). VNC is a standard remote-desktop protocol by which a VNC server allows clients to connect and work with a server's GUI applications using the mouse and keyboard via the network.

The 80 tasks vary a lot in terms of complexity and the actions required from the agent. Some tasks are very simple, even for RL, like "click on the dialog's close button," or "push the single button," but some require multiple steps, for example, "open collapsed groups and click on the link with some text," or "select a specific date using the date picker tool" (and this date is randomly generated every episode). Some of the tasks are simple for humans, but require character recognition, for example, "mark checkboxes with this text" (and the text is generated randomly). Screenshots of some MiniWoB problems are shown in the following figure:
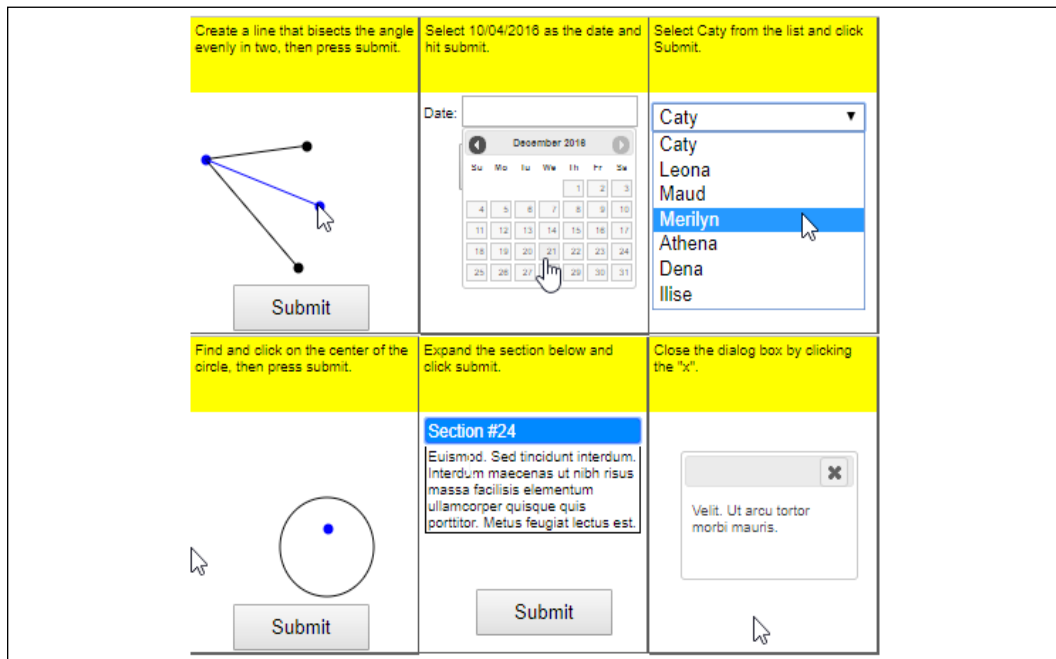
Figure 16.1: MiniWoB environments

Unfortunately, despite the brilliant idea and challenging nature of MiniWoB, it was almost abandoned by OpenAI right after the initial release. As an attempt to right a wrong, in this chapter, we will take a closer look at this benchmark and learn how to write an agent solving some of the tasks. We will also discuss how to extract, preprocess, and incorporate human demonstrations into the training process, and check their effect on the final performance of the agents.

Before jumping into the RL part of the agent, we need to understand how MiniWoB works. To do this, we need to take a closer look at OpenAI Gym's extension, called OpenAI Universe.

OpenAI Universe

The core idea underlying OpenAI Universe (available at `https://github.com/openai/universe`) is to wrap general GUI applications into an RL environment using the same core classes provided by Gym. To achieve this, it uses the VNC protocol to connect with the VNC server running inside the Docker (a standard method running lightweight containers) container, exposing the mouse and keyboard actions to the RL agent and providing the GUI application image as an observation.

The reward is provided by an external small rewarder daemon running inside the same container and giving the agent the scalar reward value based on this rewarder judgement. It is possible to launch several containers locally, or over the network, to gather episodes data in parallel, in the same way that we started several Atari emulators to increase the convergence of the asynchronous advantage actor-critic (A3C) method in *Chapter 13*, *Asynchronous Advantage Actor-Critic*. The architecture is illustrated in the following diagram:
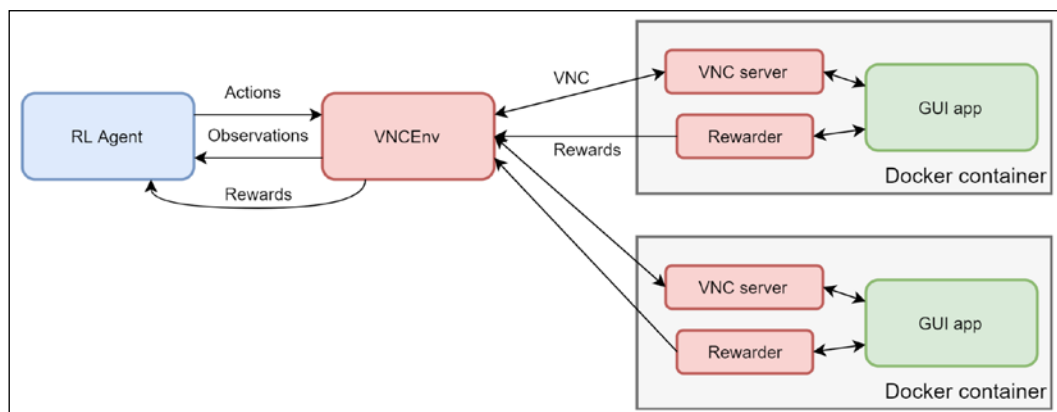
Figure 16.2: The OpenAI Universe architecture

This architecture allows quick integration of third-party applications into the RL framework, as you don't need to make any changes in the app itself; you just need to package it as a Docker container and write a relatively small rewarder daemon that uses a simple text protocol to communicate. On the other side, this approach is much more resource-intensive compared with, for example, Atari games, for which the emulator is relatively lightweight and works completely inside the RL agent's process. The VNC approach requires a VNC server to be started side by side with the application, and the rate of the RL agent's communication with the application is defined by the VNC server speed and network throughput (in the case of remote Docker containers).

Installation

As both Universe and MiniWoB were abandoned, their dependencies weren't updated for a long time, so the main environment you have created for this book's examples won't work here. In particular, Universe depends on the old Gym version, which has a set of incompatible changes. To solve the issue, you can create a separate Python environment with Universe and the old Gym, but with the latest PTAN and PyTorch versions. To do this, you can create a fresh Python 3.6 virtual environment (later versions of Python won't work), set up Universe with `pip install universe`, and only then set up other dependencies, like PTAN and PyTorch.

If you're using Anaconda (which I strongly recommend), you can create the environment with one command, `conda env create -f Chapter13/environment.yml`, which will make a new environment named `rl_book_ch16` with all dependencies installed.

In any case, that's a bit of a hassle caused by the discontinuation of MiniWoB support. There is an attempt to continue the development of MiniWoB, called MiniWoB++ (`https://github.com/stanfordnlp/miniwob-plusplus`) from the Stanford NLP Group, but due to a major change in the underlying platform (MiniWoB++ uses Selenium (`https://selenium.dev/`) for the browser automation) part of the examples in this chapter require a major rewrite.

Another component required by Universe is Docker, which is available on most modern operating systems. To install it, refer to Docker's website: `https://www.docker.com`. Universe gives you the flexibility of where and how to start the containers, so your agent can connect to one or many remote machines with Docker installed. To check that Docker is up and running, try the command `docker ps`, which shows running containers.

Actions and observations

In contrast to Atari games, or other Gym environments that we have worked with so far, Universe exposes a much more generic action space. Atari games used six-to-seven discrete actions, corresponding to the controller's buttons and joystick directions. CartPole's action space is even smaller, with just two actions available. However, VNC gives our agent much more flexibility in terms of what it can do. First of all, the full keyboard, with control keys and the up/down state of every key, is exposed. So, your agent can decide to press 10 buttons simultaneously and it will be totally fine from a VNC point of view. The second part of the action space is the mouse: you can move the mouse to any coordinate and control the state of its buttons. This significantly increases the dimensionality of the action space that the agent needs to learn how to handle.

Besides the larger action space, the Universe environment has slightly different environment semantics compared to the Gym environments. The difference is in two aspects. The first is the so-called *vectorized* representation of the observations, actions, and reward. As you can see in *Figure 16.2*, one environment could be connected to several Docker containers running the same application and gather parallel experience from them. Such parallel communication allows policy gradient methods to obtain more diverse training samples, but now we need to specify which exact application we need to send the action to with the `env.step()` call. To solve this, the Universe environment's `step()` method requires not a single action, but a list of actions for every connected container. The return from this function is also vectorized and now consists of a tuple of lists: `(observations, rewards, done_flags, infos)`.

The second difference is dictated by the VNC protocol's asynchronous nature of observations and actions. In the Atari environment, every call to `step()` triggers the request to the emulator to move forward one clock tick (which is 1/25 of second), so our agent could block the emulator for a while and it would be completely transparent for the running game. With VNC, this is not the case. As the GUI application is running in parallel to our client, we can't block it anymore. If our agent decides to think for a while, it can miss observations that happened during that time.

Another implication of the asynchronous nature of the observations is the situation when the container is not ready yet or is in the middle of resetting. In that case, the specific observation can be `None`, and those situations need to be handled by the agent.

Environment creation

To create the Universe environment, you need, as before, to call `gym.make()` with the environment ID. For example, a very simple problem from the MiniWoB set is `wob.mini.ClickDialog-v0`, which requires you to close the dialog by clicking on the × button. However, before the environment can be used, you need to configure it by specifying where and how many Docker instances you want. There is a special method of the environment called `configure()`. This method needs to be called before any other methods of the environment. It accepts several arguments.

The most important arguments are as follows:

- `remotes`: A parameter, which could be a number or a string. If it's specified as a number, it gives the number of local containers that need to be started for the environment. As a string, this parameter can specify the URL of already-running containers that the environment needs to connect in the form of `vnc://host1:port1+port2,host2:port1+port2`. The first port is a VNC protocol port (`5900` by default). The second port is a port of the rewarder daemon, which is, by default, `15900`. Both ports could be redefined on the Docker container launch.

- `fps`: An argument giving the expected frames per second (FPS) for the agent's observations.

- `vnc_kwargs`: An argument that has to be a dict, with extra VNC protocol parameters, defining the compression level and the quality of the image to be transferred to the agent. Those parameters are very important for performance, especially for containers running in the cloud.

To illustrate this, let's consider a very simple program that starts a single container with the `ClickDialog` problem and obtains its first observation as an image. This example is available in `Chapter16/adhoc/wob_create.py`.

```python
#!/usr/bin/env python3
import gym
import universe
import time

from PIL import Image
```

This example is very simple, so we need only a very small set of packages. Importing the `universe` package is required despite it not being used, as with this import it registers its environments in Gym.

```
if __name__ == "__main__":
    env = gym.make("wob.mini.ClickDialog-v0")

    env.configure(remotes=1, fps=5, vnc_kwargs={
        'encoding': 'tight', 'compress_level': 0,
        'fine_quality_level': 100, 'subsample_level': 0
    })
    obs = env.reset()
```

Next, we create our environment and ask it to configure itself. The passed arguments specify that only one local container will be started (five FPS) and the VNC connection will run without image compression. This will mean that a large amount of traffic is passed between the VNC server and VNC client, which will prevent compression artefacts from appearing in the image. This could be required for MiniWoB problems showing text in a relatively small font size.

```
while obs[0] is None:
    a = env.action_space.sample()
    obs, reward, is_done, info = env.step([a])
    print("Env is still resetting...")
    time.sleep(1)
```

While our single observation is `None` (we expect only one observation in a returned list, as we have asked only for one remote container), we pass random actions to the environment, waiting for the image to appear.

```
print(obs[0].keys())
im = Image.fromarray(obs[0]['vision'])
im.save("image.png")
env.close()
```

When, finally, we have got the image from the server, we save it as a PNG file, shown in *Figure 16.3*. In MiniWoB problems, the image is not the only observation we get. In fact, the observation from the environment is a dict with two entries: `vision`, containing a NumPy array with screen pixels, and `text`, containing the text description of the problem. For some problems, only the image is required, but for some tasks from the MiniWoB suite, the text includes essential information for solving the problem, like which color area to click on or what dates need to be selected.

The following image was cropped, as the original resolution of the observation is 1024×768.
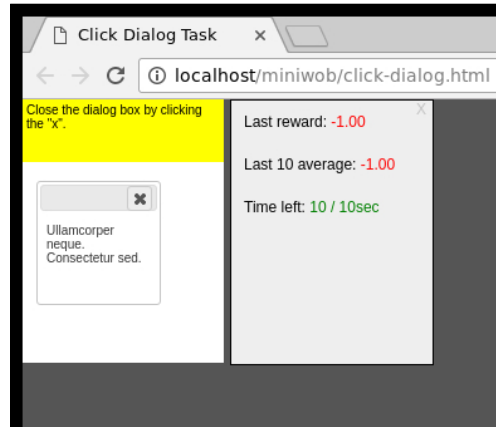
Figure 16.3: Part of a MiniWoB observation image

MiniWoB stability

My experiments with the original MiniWoB Docker image, published by OpenAI, have shown one serious issue: sometimes the server-side Python script, which controls the browser inside the container, crashes. This leads to training problems, as our environment loses the connection to the container and the training stops. The solution for this issue is a one-line change, but it is complicated by the fact that OpenAI doesn't support MiniWoB and doesn't accept the fixes. To resolve the issue, I had to apply the patch inside the container. There is another small patch related to human demonstration that fixes the issue with recording files being overwritten between episodes. The patched image with both fixes was pushed into my Docker Hub repository and is available as the `shmuma/miniwob:v2` label, so you can use it instead of the original `quay.io/openai/universe.world-of-bits:0.20.0` image. If you're curious, I've placed the patches and instructions on how to apply them in the code sample repository: `Chapter16/wob_fixes`.

The simple clicking approach

As the first demo, let's implement a simple A3C agent that decides where it should click given the image observation. This approach can solve only a small subset of the full MiniWoB suite, and we will discuss restrictions of this approach later. For now, it will allow us to get a better understanding of the problem.

As with the previous chapter, due to its size, I won't put the complete source code here. We will focus on the most important functions and I will provide the rest as an overview. The complete source code is available in the GitHub repository.

Grid actions

When we talked about Universe's architecture and organization, it was mentioned that the richness and flexibility of the action space creates a lot of challenges for the RL agent. MiniWoB's active area inside the browser is just 160×210 (exactly the same dimension that the Atari emulator has), but even with such a small area, our agent could be asked to move the mouse, perform clicks, drag objects, and so on. Just the mouse alone could be problematic to master, as, in the extreme case, there could be an almost infinite number of different actions that the agent could perform, like pressing the mouse button at some point and dragging the mouse to a different location. In our example, we will simplify our problem a lot by just considering clicks at some fixed grid points inside the active webpage area. The sketch of our action space is as follows:
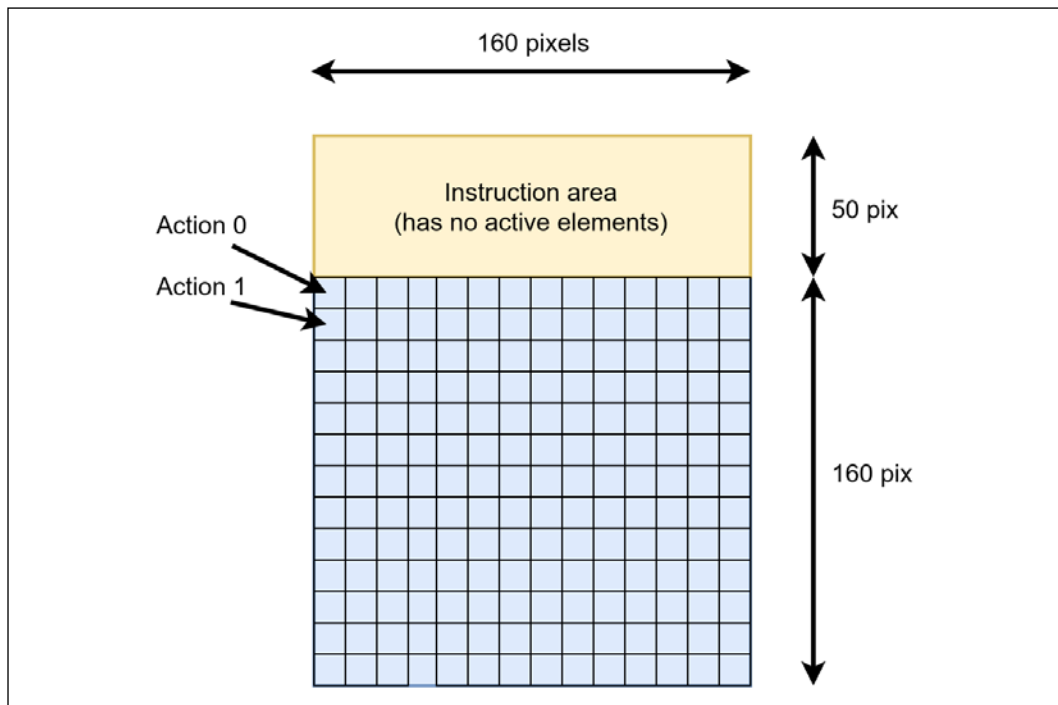
Figure 16.4: A grid action space

This approach is already implemented in Universe as an action wrapper: `universe.wrappers.experimental.action_space.SoftmaxClickMouse`. It has all the defaults preset for MiniWoB environments: a 160×210 region shifted 10 pixels to the right and 75 pixels down (to get rid of the browser's frames). The grid of actions is 10×10, which gives 256 final actions to choose from.

Besides the action preprocessor, we definitely need an observation preprocessor, as the input image from the VNC environment is a 1024×768×3 tensor, but the active area of MiniWoB is just 210×160. There is no suitable cropper defined, so I implemented it myself as a class, `lib.wob_vnc.MiniWoBCropper`, which is in the `Chapter16/lib/wob_vnc.py` library module. Its code is very simple and is shown here:

```python
WIDTH = 160
HEIGHT = 210
X_OFS = 10
Y_OFS = 75


class MiniWoBCropper(vectorized.ObservationWrapper):
    def __init__(self, env, keep_text=False):
        super(MiniWoBCropper, self).__init__(env)
        self.keep_text = keep_text

    def _observation(self, observation_n):
        res = []
        for obs in observation_n:
            if obs is None:
                res.append(obs)
                continue
            img = obs['vision'][Y_OFS:Y_OFS+HEIGHT,
                                X_OFS:X_OFS+WIDTH, :]
            img = np.transpose(img, (2, 0, 1))
            if self.keep_text:
                t_fun = lambda d: d.get('instruction', '')
                text = " ".join(map(t_fun, obs.get('text', [{}])))
                res.append((img, text))
            else:
                res.append(img)
        return res
```

The optional `keep_text` argument in the constructor enables the mode to preserve the text description of the problem. We don't need it at the moment, and our first version of the agent will always keep it disabled. In this mode, `MiniWoBCropper` returns the NumPy array with the shape (3, 210, 160).

Example overview

With decisions about actions and observations made, our next steps are straightforward. We will use the A3C method to train the agent, which should decide from the 160×210 observation which grid cell to click. Besides the policy, which is a probability distribution over 256 grid cells, our agent estimates the value of the state, which will be used as a baseline in policy gradient estimation.

There are several modules in this example:

- `Chapter16/lib/common.py`: Methods shared among examples in this chapter, including the already-familiar `RewardTracker` and `unpack_batch` functions

- `Chapter16/lib/model_vnc.py`: Includes a definition of the model, which will be shown in the next section

- `Chapter16/lib/wob_vnc.py`: Includes MiniWoB-specific code, like the observation cropper, environment configuration method, and other utility functions

- `Chapter16/wob_click_train.py`: The script used to train the model

- `Chapter16/wob_click_play.py`: The script that loads the model weights and uses them against the single environment, recording observations and counting statistics about the reward

The model

The model is very straightforward and uses the same patterns that you have seen in other A3C examples. I haven't spent much time optimizing and fine-tuning the architecture and hyperparameters, so it's likely that the final result could be improved significantly. The following is the model definition with two convolution layers, a single-layered policy, and value heads.

```
class Model(nn.Module):
    def __init__(self, input_shape, n_actions):
        super(Model, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 64, 5, stride=5),
            nn.ReLU(),
            nn.Conv2d(64, 64, 3, stride=2),
            nn.ReLU(),
        )

        conv_out_size = self._get_conv_out(input_shape)
```

```
        self.policy = nn.Linear(conv_out_size, n_actions)
        self.value = nn.Linear(conv_out_size, 1)

    def _get_conv_out(self, shape):
        o = self.conv(torch.zeros(1, *shape))
        return int(np.prod(o.size()))

    def forward(self, x):
        fx = x.float() / 256
        conv_out = self.conv(fx).view(fx.size()[0], -1)
        return self.policy(conv_out), self.value(conv_out)
```

The training code

The training script is in `Chapter16/wob_click_train.py` and should be very familiar, but I put it here because it contains several Universe- and MiniWoB-specific pieces. This script can work in two modes: *with* and *without* human demonstrations. Currently we're considering only training from scratch, but some code is related to demonstrations and should be ignored for now. We will look at it in the appropriate section later.

```
import os
import gym
import random
import universe
import argparse
import numpy as np
from tensorboardX import SummaryWriter

from lib import wob_vnc, model_vnc, common, vnc_demo

import ptan

import torch
import torch.nn.utils as nn_utils
import torch.nn.functional as F
import torch.optim as optim
```

There is not much to say about the used modules, except the new `universe`. It might look unused, but you still need to import it. As mentioned earlier, on import it registers new environments in Gym's repository so they become available on the `gym.make()` call.

```
REMOTES_COUNT = 8
ENV_NAME = "wob.mini.ClickDialog-v0"
```

```
GAMMA = 0.99
REWARD_STEPS = 2
BATCH_SIZE = 16
LEARNING_RATE = 0.0001
ENTROPY_BETA = 0.001
CLIP_GRAD = 0.05
DEMO_PROB = 0.5
SAVES_DIR = "saves"
```

The hyperparameters section is also mostly the same, except that a couple of hyperparameters are new. First of all, `REMOTES_COUNT` specifies the number of Docker containers that we will try to connect. By default, our training script assumes that those containers have already started on one machine and we can connect to them on predefined ports (`5900..5907` for VNC connection and `15900..15907` for the rewarder daemon). We will look at the details of starting containers in the next section.

The parameter `ENV_NAME` specifies the problem that we will try to attack, and it could be redefined with the command-line arguments. The problem `ClickDialog` is very simple and gives the reward to the agent for clicking on the dialog's close button.

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-n", "--name", required=True,
                        help="Name of the run")
    parser.add_argument("--cuda", default=False,
                        action='store_true', help="CUDA mode")
    parser.add_argument("--port-ofs", type=int, default=0,
                        help="Offset for container's ports, "
                             "default=0")
    parser.add_argument("--env", default=ENV_NAME,
                        help="Environment name to solve, "
                             "default=" + ENV_NAME)
    parser.add_argument("--demo", help="Demo dir to load. "
                                       "Default=No demo")
    parser.add_argument("--host", default='localhost',
                        help="Host with docker containers")
    args = parser.parse_args()
    device = torch.device("cuda" if args.cuda else "cpu")
```

We have many command-line options, and you can use them to tweak the training behavior. There is only one required option to pass the name of the run, which will be used for TensorBoard and the directory to save the model's weights.

The parameter `--demo` should be ignored for now, as it is related to human demonstrations.

```
env_name = args.env
if not env_name.startswith('wob.mini.'):
    env_name = "wob.mini." + env_name

name = env_name.split('.')[-1] + "_" + args.name

writer = SummaryWriter(comment="-wob_click_" + name)

saves_path = os.path.join(SAVES_DIR, name)

os.makedirs(saves_path, exist_ok=True)
```

After arguments are parsed, we normalize the environment name (all MiniWoB environments start with the `wob.mini.` prefix, so we don't require it to be specified in the command line), start the TensorBoard writer, and create the directory for the models.

```
demo_samples = None
if args.demo:
    demo_samples = vnc_demo.load_demo(args.demo, env_name)
    if not demo_samples:
        demo_samples = None
    else:
        print("Loaded %d demo samples, will use them "
              "during training" % len(demo_samples))
```

The preceding piece of code is related to demonstrations and should be ignored for now.

```
env = gym.make(env_name)
env = universe.wrappers.experimental.SoftmaxClickMouse(env)
env = wob_vnc.MiniWoBCropper(env)
wob_vnc.configure(env, wob_vnc.remotes_url(
    port_ofs=args.port_ofs, hostname=args.host,
    count=REMOTES_COUNT))
```

To prepare the environment, we ask Gym to create it, wrap it into the `SoftmaxClickMouse` wrapper described before, and then apply our cropper. However, this environment is not ready to be used yet. To complete the initialization, we need to configure it using utility functions in the `wob_vnc` module. Their goal is to call the `env.configure()` method with arguments specifying VNC connection parameters, like image quality, compression level, and the addresses of Docker containers that we want to connect. Those connection endpoints are specified in a URL of a special form, generated by the function `wob_vnc.remotes_url()`.

This URL has the form of `vnc://host:port1+port2,host:port1+port2` and allows a single environment to communicate with any number of Docker containers running on multiple hosts.

```
net = model_vnc.Model(input_shape=wob_vnc.WOB_SHAPE,
                      n_actions=env.action_space.n).to(device)
print(net)
optimizer = optim.Adam(net.parameters(),
                       lr=LEARNING_RATE, eps=1e-3)

agent = ptan.agent.PolicyAgent(
    lambda x: net(x)[0], device=device, apply_softmax=True)
exp_source = ptan.experience.ExperienceSourceFirstLast(
    [env], agent, gamma=GAMMA, steps_count=REWARD_STEPS,
    vectorized=True)
```

Before the training can be started, we create the model, the agent, and the experience source from the PTAN library. The only new thing here is the argument `vectorized=True`, which tells the experience source that our environment is vectorized and returns multiple results in one call.

```
best_reward = None
with common.RewardTracker(writer) as tracker:
    with ptan.common.utils.TBMeanTracker(
            writer, batch_size=10) as tb_tracker:
        batch = []
        for step_idx, exp in enumerate(exp_source):
            rewards_steps = exp_source.pop_rewards_steps()
            if rewards_steps:
                rewards, steps = zip(*rewards_steps)
                tb_tracker.track("episode_steps",
                                 np.mean(steps), step_idx)

                mean_reward = tracker.reward(np.mean(rewards),
                                             step_idx)
                if mean_reward is not None:
                    if best_reward is None or \
                            mean_reward > best_reward:
                        if best_reward is not None:
                            name = "best_%.3f_%d.dat" % (
                                mean_reward, step_idx)
                            fname = os.path.join(
                                saves_path, name)
                            torch.save(net.state_dict(), fname)
                            print("Best reward updated: %.3f "
```

```
                        "-> %.3f" % (
                    best_reward, mean_reward))
                best_reward = mean_reward
        batch.append(exp)
        if len(batch) < BATCH_SIZE:
            continue
```

In the beginning of the training loop, we ask our experience source for new experience objects and pack them into the batch. In the meantime, we track the average undiscounted reward, and if it updates the maximum, we save the model's weight.

```
        if demo_samples and random.random() < DEMO_PROB:
            random.shuffle(demo_samples)
            demo_batch = demo_samples[:BATCH_SIZE]
            model_vnc.train_demo(
                net, optimizer, demo_batch, writer,
                step_idx, device=device)
```

The preceding piece of code is relevant to demonstrations and should be ignored for now.

```
        states_v, actions_t, vals_ref_v = \
            common.unpack_batch(
                batch, net, device=device,
                last_val_gamma=GAMMA ** REWARD_STEPS)
        batch.clear()
```

When the batch is complete, we unpack it into individual tensors and perform the A3C training procedure: calculate the value loss to improve the value head estimation and calculate the policy gradient using the value as a baseline for the advantage.

```
        optimizer.zero_grad()
        logits_v, value_v = net(states_v)

        loss_value_v = F.mse_loss(
            value_v.squeeze(-1), vals_ref_v)

        log_prob_v = F.log_softmax(logits_v, dim=1)
        adv_v = vals_ref_v - value_v.detach()
        lpa = log_prob_v[range(BATCH_SIZE), actions_t]
        log_prob_actions_v = adv_v * lpa
        loss_policy_v = -log_prob_actions_v.mean()

        prob_v = F.softmax(logits_v, dim=1)
```

```
ent_v = prob_v * log_prob_v
entropy_loss_v = ENTROPY_BETA * ent_v
entropy_loss_v = entropy_loss_v.sum(dim=1).mean()
```

To improve exploration, we add the entropy loss calculated as a scaled negative entropy of the policy.

```
loss_v = loss_policy_v + entropy_loss_v + \
        loss_value_v
loss_v.backward()
nn_utils.clip_grad_norm_(
    net.parameters(), CLIP_GRAD)
optimizer.step()

tb_tracker.track("advantage", adv_v, step_idx)
tb_tracker.track("values", value_v, step_idx)
tb_tracker.track("batch_rewards", vals_ref_v,
                    step_idx)
tb_tracker.track("loss_entropy", entropy_loss_v,
                    step_idx)
tb_tracker.track("loss_policy", loss_policy_v,
                    step_idx)
tb_tracker.track("loss_value", loss_value_v,
                    step_idx)
tb_tracker.track("loss_total", loss_v, step_idx)
```

Then, we track the key quantities with TensorBoard to be able to monitor them during the training.

Starting containers

Before the training can be started, you need to have Docker containers with MiniWoB started. Universe provides an option to start them automatically. To do this, you need to pass the integer value to the `env.configure()` call; for example, `env.configure(remotes=4)` will start four Docker containers with MiniWoB locally.

Despite the simplicity of this start mode, it has several disadvantages:

- You have no control over the containers' location, so all containers will be started locally. This is not convenient when you want them to be started on a remote machine or multiple machines.

- By default, Universe starts the container published in `quay.io` (at the time of writing, it's image `quay.io/openai/universe.world-of-bits` with version 0.20.0), which has a serious bug in the reward calculation. Due to this, your training process can crash from time to time, which is not good when training can take days. There is an option to `env.configure()`, called `docker_image`, which allows you to redefine the image used to start, but you need to hard-code the image into the code.

- The starting tuple of containers has an overhead, so your training has to wait before all the containers start.

As an alternative, I find it much more flexible to start Docker containers in advance. In that case, you need to pass to `env.configure()` a URL pointing the environment to the hosts and ports that it has to be connected with. To start the container, you need to run the following command:

```
docker run -d -p 5900:5900 -p 15900:15900--privileged --ipc host --cap-
add SYS_ADMIN <CONTAINER_ID> <ARGS>
```

The meanings of the arguments are as follows:

1. `-d` starts the container in detach mode. To be able to see the container's logs, you can replace this option with `-t`. In that case, the container will be started interactively and can be stopped with *Ctrl + C*.

2. `-p SRC_PORT:TGT_PORT` forwards the source port from the container's host to the target port inside the container. This option allows you to start several MiniWoB containers on one machine. Every container starts the VNC server listening on port `5900` and the rewarder daemon on port `15900`. Argument `-p 5900:5900` makes the VNC server available on port `5900` on the host machine (the machine running the container). For the second container, you should pass `-p 5901:5900`, which makes it available on port `5901`, instead of the occupied `5900`. The same is true for rewarder: inside the container, it listens on port `15900`. By providing the `–p` option, you can forward connections from your host machine to the container's port.

3. `--privileged` allows the container to access the host's devices. (The reason MiniWoB gets started with this option may be because there are some VNC server requirements.)

4. `--ipc host` enables containers to share the interprocess communications (IPC) namespace with the host.

5. `--cap-add SYS_ADMIN` extends the container's capabilities to perform extended configuration of the host's settings.

6. `<CONTAINER_ID>` is the identifier of the container. This should be `shmuma/miniwob:v2`, which is a patched version of the original `quay.io/openai/universe.world-of-bits:0.20.0`. More details were given in the preceding section of *MiniWoB stability*.

7. `<ARGS>` enables you to pass extra arguments to the container to change its mode of operation. We will need them later for recording human demonstrations. For now, it can be empty.

That's it! Our training script expects eight containers to be running, sitting on ports `5900-5907` and `15900-15907`. For example, to start them, I use the following commands (also available as `Chapter16/adhoc/start_docker.sh`):

```bash
#!/usr/bin/env bash
for i in 'seq 0 7'; do
    p1=$((5900+i))
    p2=$((15900+i))
    docker run -d -p $p1:5900 -p $p2:15900 --privileged \
        --ipc host --cap-add SYS_ADMIN shmuma/miniwob run -f 20
done
```

All of them will be started in the background and can be seen with the `docker ps` command:

Figure 16.5: The docker ps output

The training process

When the containers have started and are ready to be used, you can start the training. In the beginning, it shows messages about the connection status, but, finally, it should start reporting about the episodes' statistics.

```
$ ./wob_click_train.py -n t2 --cuda
[2018-01-29 14:27:48,545] Making new env: wob.mini.ClickDialog-v0
[2018-01-29 14:27:48,547] Using SoftmaxClickMouse with action_region=
(10, 125, 170, 285), noclick_regions=[]
[2018-01-29 14:27:48,547] SoftmaxClickMouse noclick regions removed
0 of 256 actions
[2018-01-29 14:27:48,548] Writing logs to file: /tmp/universe-9018.log
```

```
[2018-01-29 14:27:48,548] Using the golang VNC implementation
[2018-01-29 14:27:48,548] Using VNCSession arguments: {'compress_level':
0, 'subsample_level': 0, 'encoding': 'tight', 'start_timeout': 21,
'fine_quality_level': 100}. (Customize by running "env.configure (vnc_
kwargs={...})"
[2018-01-29 14:27:48,579] [0] Connecting to environment: vnc://
localhost:5900 password=openai.

[2018-01-29 14:27:52,218] Throttle fell behind by 1.06s; lost 5.32 frames
[2018-01-29 14:27:52,955] [1:localhost:5901] Initial reset complete:
episode_id=17803
37: done 1 games, mean reward 0.686, speed 11.77 f/s
52: done 2 games, mean reward 0.447, speed 28.29 f/s
72: done 3 games, mean reward -0.035, speed 33.24 f/s
98: done 4 games, mean reward -0.130, speed 25.92 f/s
125: done 5 games, mean reward -0.015, speed 33.64 f/s
146: done 6 games, mean reward 0.137, speed 26.18 f/s
```

If desired, you can manually connect to the container's VNC server using a VNC client, such as TurboVNC. Most environments provide a convenient in-browser VNC client: `http://localhost:15900/viewer/?password=openai`.

By default, the training process starts the `ClickDialog-v0` environment, which should take 100k-200k to reach the mean reward of 0.8-0.99. The convergence dynamics are shown in the following charts:
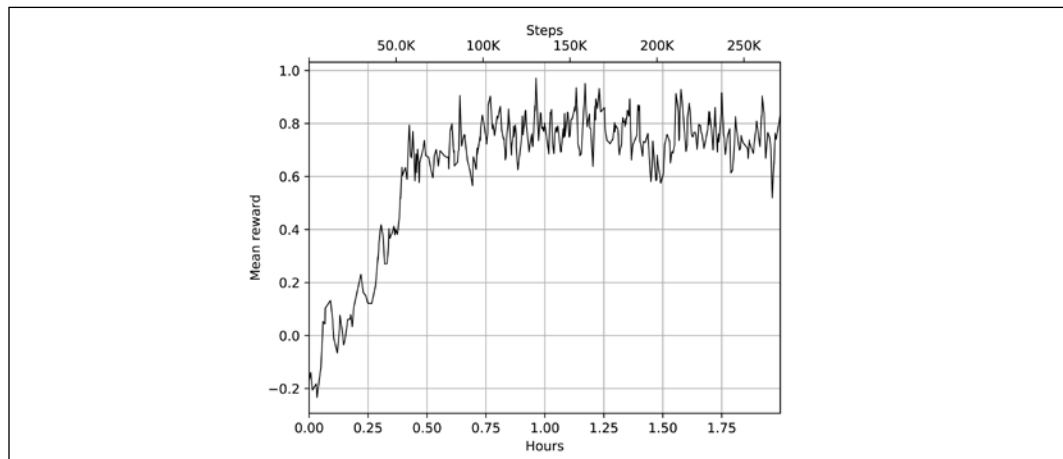
Figure 16.6: Reward dynamics in the ClickDialog problem
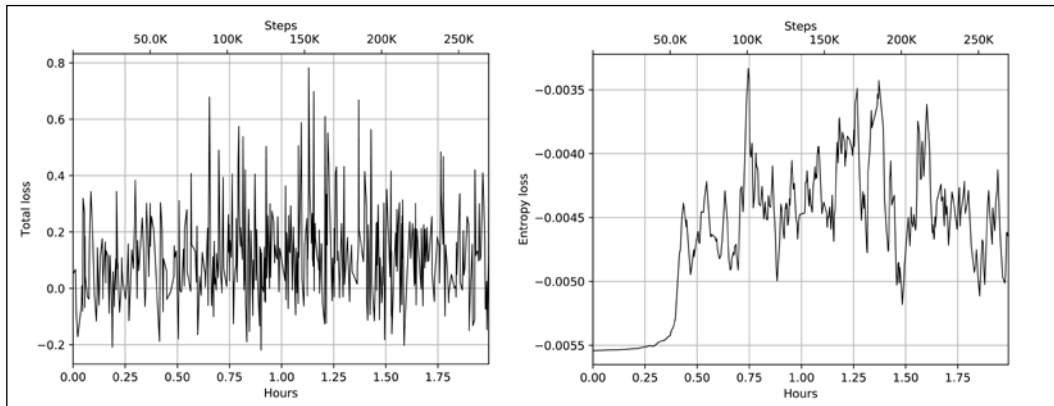
Figure 16.7: Total loss (left) and entropy loss (right) during the training
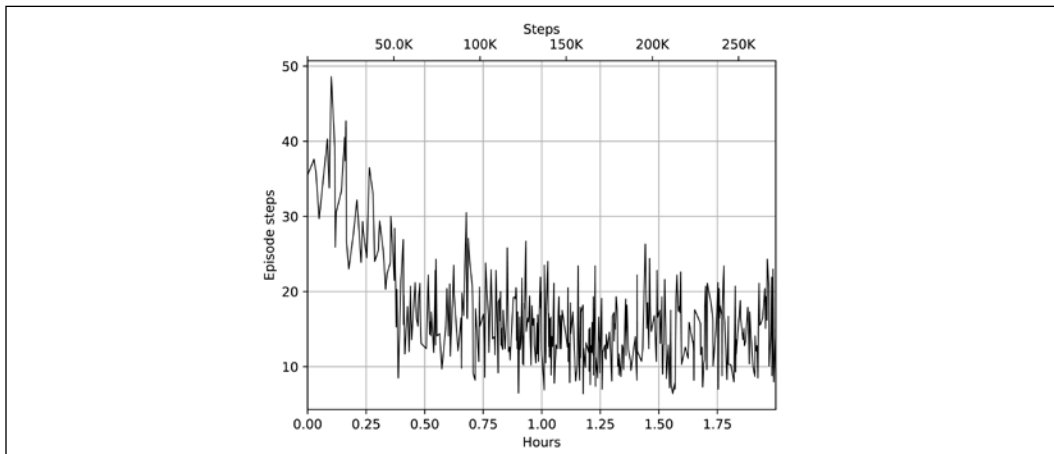
Figure 16.8: The number of steps in the training episodes

The *episode_steps* chart shows the mean count of actions that the agent should carry out before the end of the episode. Ideally, for this problem the count should be 1, as the only action that the agent needs to take is to click on the dialog's close button. However, in fact, the agent sees seven to nine frames before the episode ends. This happens for two reasons: the cross on the dialog close button can appear with some delay, and the browser inside the container adds a time gap before the agent clicks and the rewarder notices this. Anyway, in approximately 100k frames (which is about half an hour with eight containers), the training procedure converged to quite a good policy, which can close the dialog most of the time.

Checking the learned policy

To be able to peek inside the agent's activity, there is a tool that loads the model weights from the file and runs several episodes, recording the screenshots with the agent's observations and actions selected. The tool is called `Chapter16/wob_click_play.py`, and it connects to the first container (port `5900` and `15900`), running on the local machine and accepting the following arguments:

- `-m`: The filename with the model to be loaded.
- `--save <IMG_PREFIX>`: If specified, this saves every observation in a separate file. The argument is the path prefix.
- `--count`: This sets the count of episodes to run.
- `--env`: This sets the environment name to be used, which by default is `ClickDialog-v0`.
- `--verbose`: This shows every step with the reward, `done`, and internal info.

This is useful for examining (or even debugging) the agent's behavior for different states during training. For example, checking the best model trained on `ClickDialog` shows us this:

```
rl_book_samples/Chapter16$ ./wob_click_play.py -m saves/ClickDialog-v0_
t1/best_1.047_209563.dat --count 5

[2018-01-29 15:43:57,188] [0:localhost:5900] Sending reset for env_
id=wob.mini.ClickDialog-v0 fps=60 episode_id=0

[2018-01-29 15:44:01,223] [0:localhost:5900] Initial reset complete:
episode_id=288

Round 0 done
Round 1 done
Round 2 done
Round 3 done
Round 4 done
Done 5 rounds, mean steps 6.40, mean reward 0.734
```

To check the agent's actions, you can pass the `--save` option with the prefix of the images to be written. The actions that the agent performs are shown as a circle at the click point. The area on the right contains the technical information about the last reward and the time left before timeout. For example, one of the saved images is shown in the following figure:
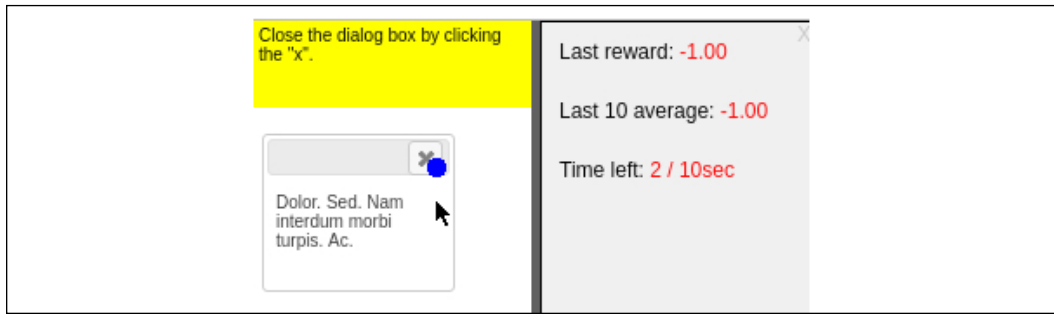
Figure 16.9: A screenshot of the agent in action

Issues with simple clicking

Unfortunately, the demonstrated approach can only be used to solve relatively simple problems, like `ClickDialog`. If you try to use it for more complicated tasks, convergence is unlikely. There could be multiple reasons for this. First of all, our agent is stateless, which basically means that it makes the decision about the action only from the observation, without taking into account its previous actions. You may remember in *Chapter 1*, *What Is Reinforcement Learning?*, that we discussed the Markov property of the Markov decision process (MDP) and that this Markov property allowed us to discard all previous history, keeping only the current observation. Even in relatively simple problems from MiniWoB, this Markov property could be violated. For example, there is a problem called `ClickButtonSequence-v0` (the screenshot is shown in the following figure), which requires our agent to first click on button **ONE** and then on button **TWO**. Even if our agent is lucky enough to randomly click in the required order, it won't be able to distinguish from the single image what button needs to be clicked next.
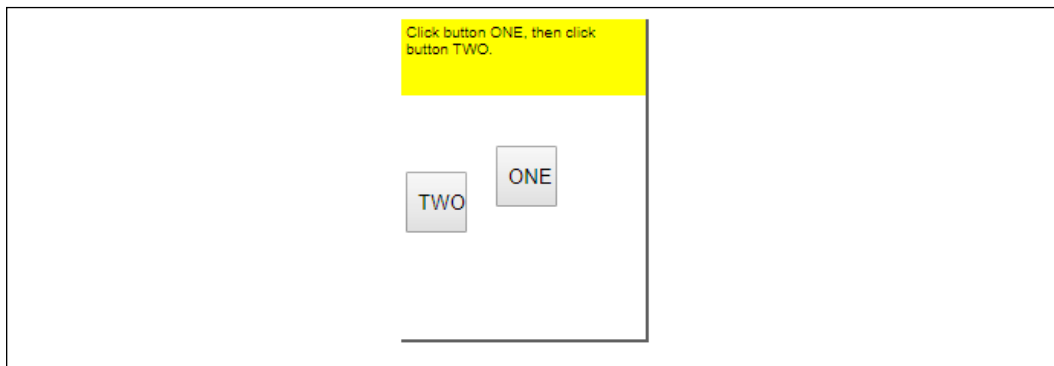
Figure 16.10: An example of an environment that the stateless agent could struggle to solve

Despite the simplicity of this problem, we cannot use our RL methods to solve it, because MDP formalism is not applicable anymore. Such problems are called partially observable MDPs, or POMDPs, and the usual approach for them is allowing the agent to keep some kind of state. The challenge here is to find the balance between keeping only minimal relevant information and overwhelming the agent with nonrelevant information by adding everything into the observation.

Another issue that we can face with our example is that the data required to solve the problem might not be available in the image or could just be in an inconvenient form. For example, there are two problems: ClickTab and ClickCheckboxes. In the first one, you need to click on one of three tabs, but every time, the tab that needs to be clicked is randomly chosen. Which tab needs to be clicked is shown in a description (provided with an in-text field of observation and shown on the top of the environment's page), but our agent sees only pixels, which makes it complicated to connect the tiny number on the top with the outcome of the random click result. The situation is even worse with the ClickCheckboxes problem, when several checkboxes with randomly generated text need to be clicked. One of the possible options to prevent overfitting to the problem is to use some kind of **optical character recognition** (**OCR**) network to convert the image in the observation into text form. The following figure shows examples of the two problem environments:
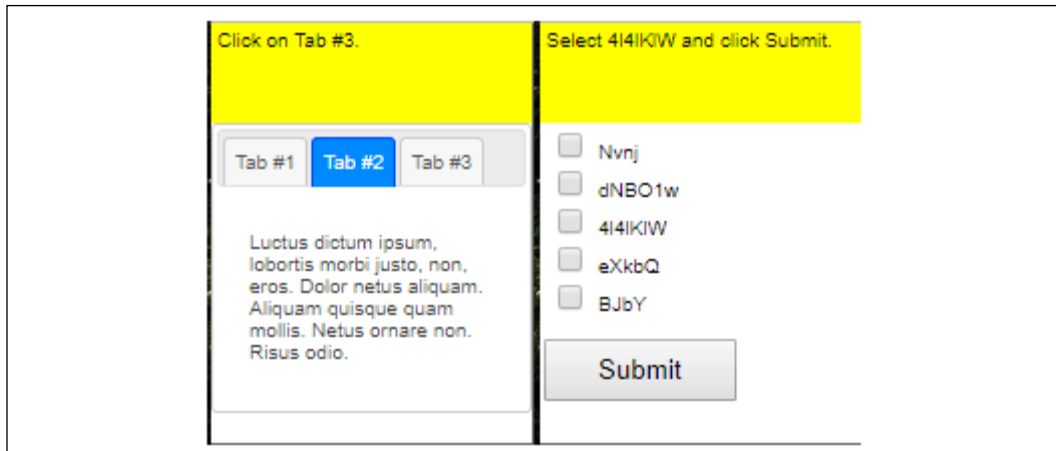
Figure 16.11: An example of environments where the text description is important

Yet another issue could be related just to the dimensionality of the action space that the agent needs to explore. Even for single-click problems, the number of actions could be very large, so it can take a long time for the agent to discover how to behave. One of the possible solutions here is incorporating demonstrations into the training. For example, in the following image, there is a problem called CountSides-v0. The goal there is to click on the button that corresponds to the number of sides of the shape shown.
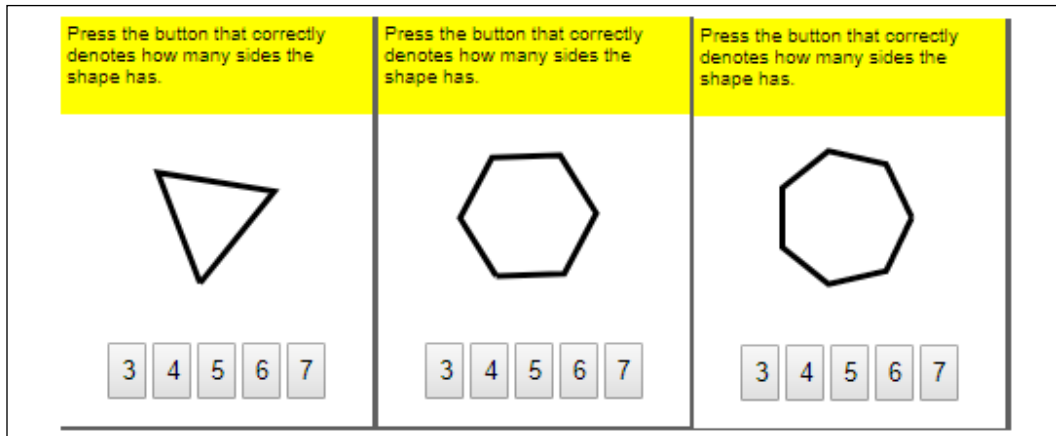
Figure 16.12: A screenshot of the CountSides environment

I tried to train the agent from scratch, and after a day of training, it showed almost zero progress. However, after adding a couple of dozen examples of correct clicks, it successfully solved the problem in 15 minutes of training. Of course, maybe my hyperparameters were bad, but, still, the effect of the demonstrations is quite impressive. In the next example of this chapter, we will take a look at how we can record and inject human demonstrations to improve the convergence.

Human demonstrations

The idea behind demonstrations is simple: to help our agent to discover the best way to solve the task, we show it some examples of actions that we think are required for the problem. Those examples could be not the best solution or not 100% accurate, but they should be good enough to show the agent promising directions to explore.

In fact, this is a very natural thing to do, as all human learning is based on some prior examples given by a teacher in class, parents, or other people. Those examples could be in a written form (for example, recipe books) or given as demonstrations that you need to repeat several times to get right (for example, dance classes). Such forms of training are much more effective than random searches. Just imagine how complicated and lengthy it would be to learn how to clean your teeth by trial and error alone. Of course, there is a danger from learning how to follow demonstrations, which could be wrong or not the most efficient way to solve the problem; but overall, it's much more effective than a random search.

All our previous examples used zero prior knowledge and started with random weights' initialization, which caused random actions to be performed at the beginning of the training.

After some iterations, the agent discovered that some actions in some states give more promising results (via the Q-value or policy with the higher advantage) and started to prefer those actions over the others. Finally, this process led to a more or less optimal policy, which gave the agent a high reward at the end. It worked well when our action space dimensionality was low and the environment's behavior wasn't very complex, but just doubling the action count caused at least twice the observations needed. In the case of our clicker agent, we have 256 different actions corresponding to 10×10 grids in the active area, which is 128 times more actions than we had in the CartPole environment. It is not surprising that the training process is lengthy and may fail to converge at all.

This issue of dimensionality can be addressed in various ways, like smarter exploration methods, training with better sampling efficiency (one-shot training), incorporating prior knowledge (transfer learning), and other means. There is a lot of research activity focused on making RL better and faster, and we can be sure that many breakthroughs are ahead. In this section, we will try the more traditional approach of incorporating the demonstration recorded by humans into the training process.

You might remember our discussion about on-policy and off-policy methods (which were discussed in *Chapter 4*, *The Cross-Entropy Method*, and *Chapter 8*, *DQN Extensions*). This is very relevant to our human demonstrations because, strictly speaking, we can't use off-policy data (human observation-actions pairs) with an on-policy method (A3C in our case). That is due to the nature of on-policy methods: they estimate the policy gradients using the samples gathered from the current policy. If we just push human-recorded samples into the training process, the estimated gradient will be relevant for a human policy, but not our current policy given by the neural network (NN). To solve this issue, we need to cheat a bit and look at our problem from the supervised learning angle. To be concrete, we will use the log-likelihood objective to push our NN toward taking actions from demonstrations.

Before we can go into the implementation details, we need to address a very important question: how do we obtain the demonstrations in the most convenient form?

Recording the demonstrations

There is no universal recipe for how to record demonstrations, as they depend on the observation and action space details. However, from a higher perspective, we should save the information available for a human or another agent, whose actions we want to record, as well as actions taken by this agent.

For example, if we want to obtain Atari game sessions played by somebody, we need to save the screen image, plus the button pressed on this screen.

In our case of the OpenAI Universe environment, there is an elegant solution, based on the VNC protocol and used as a universal transport. To save the demonstrations, we need to capture the screen sent by the server to the VNC client, as well as mouse and keyboard actions sent by the client to the server. MiniWoB provides built-in functionality for this based on the VNC protocol proxy, which is illustrated in the following scheme:
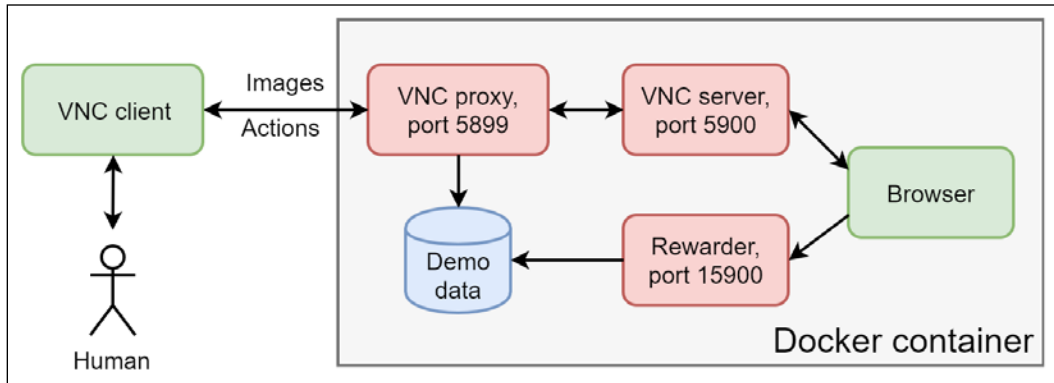
Figure 16.13: Demonstrating the recording architecture

By default, the VNC proxy is not started on the container's start, as there is a separate *demo mode*. To start the container with proxy enabled, you need to pass the arguments `demonstration -e ENV_NAME` to the container. You also will need to pass the port forwarding options to make port `5899` (that the VNC proxy is listening to) available from outside. The whole command line used to start the container in recording mode for env `ClickTest2` is as follows (and is also available as `Chapter16/adhoc/start_docker_demo.sh`):

```
docker run -e TURK_DB='' -p 5899:5899 --privileged --ipc host --cap-add
SYS_ADMIN shmuma/miniwob:v2 demonstration -e wob.mini.ClickTest2-v0
```

The argument `TURK_DB` is required, which is probably related to **Mechanical Turk**, which is used by OpenAI to gather human demonstrations for internal experiments. Unfortunately, OpenAI hasn't released those demonstrations, despite its promise to do so. So, the only way to get the demonstrations is to record them yourself.

Once the container is started, you can connect to it using any VNC client that you prefer. There are several alternatives available for Linux, Windows, and Mac. You should connect to the host of your container on port `5899`. The connection password is `openai`. After connection, you should see the browser window with an environment that you have specified on the start of the container.

Now you can start solving the problem, but don't forget that all your actions will be recorded and used later during the training; your actions should be efficient and not include any nonrelevant actions, like clicking at the wrong place and so on. Of course, you can always do an experiment checking the robustness of the training in such noisy demonstrations. The time given for solving the problem is also limited, as for most of the environments it's 10 seconds. On expiration, the problem will start over, and you will be given the reward of –1. If you don't see the mouse pointer, you should enable **local mouse render** mode in your VNC client.

Once you have recorded some demonstrations, you can disconnect from the server and copy the recorded data. Remember that your recording will be preserved only while the container is alive. The recorded data is placed in the `/tmp/demo` folder inside the container's filesystem, but you can see the files using the `docker exec` command (the following `80daf4b8f257` is the ID of a container started in demo mode):

```
$ docker exec -t 80daf4b8f257 ls -laR /tmp/demo
/tmp/demo:
total 20
drwxr-xr-x 3 root root 4096 Jan 30 17:06 .
drwxrwxrwt 19 root root 4096 Jan 30 17:07 ..
drwxr-xr-x 2 root root 4096 Jan 30 17:07
1517332006-fprnte8qiy3af3-0
-rw-r--r-- 1 nobody nogroup 20 Jan 30 17:09 env_id.txt
-rw-r--r-- 1 root root 531 Jan 30 17:09 rewards.demo


/tmp/demo/1517332006-fprnte8qiy3af3-0:

total 35132

drwxr-xr-x 2 root root 4096 Jan 30 17:07 .
drwxr-xr-x 3 root root 4096 Jan 30 17:06 ..
-rw-r--r-- 1 root root 51187 Jan 30 17:07 client.fbs
-rw-r--r-- 1 root root 20 Jan 30 17:07 env_id.txt
-rw-r--r-- 1 root root 5888 Jan 30 17:07 rewards.demo
-rw-r--r-- 1 root root 35900918 Jan 30 17:07 server.fbs
```

One individual VNC session is saved inside the `/tmp/demo` folder in a separate subdirectory, so you can use the same container for several recording sessions. To copy the data, you can use the `docker cp` command:

```
docker cp 80daf4b8f257:/tmp/demo .
```

Once you've got the raw data files, you can use them for training, but first, let's talk about the data format.

The recording format

For every client connection, the VNC proxy records four files:

- `env_id.txt`: A text file with the ID of the environment used to record the demonstration. This is very convenient for filtering when you have several demonstration data directories.

- `rewards.demo`: A JSON file with events recorded by the rewarder daemon. This includes timestamped events from the environment, like a text description change, the reward obtained, and others.

- `client.fbs`: A binary format with events sent by the client to the VNC server. Inside it contains timestamps of raw VNC protocol messages (called the **remote framebuffer protocol** or RFP).

- `server.fbs`: A binary format with data sent by the VNC server to the client. It has the same format as `client.fbs`, but the set of messages is different.

The trickiest files here are `client.fbs` and `server.fbs`, as they are binary, and the format has no convenient reader (at least I'm not aware of such a library). The protocol of VNC is standardized in request for comments (RFC) 6143 and called RFP, which is available on the IETF website `https://tools.ietf.org/html/rfc6143`. This protocol defines the set of messages that the VNC client and server can exchange to provide a remote desktop to a user. The client can send the keyboard or mouse events, and the server is responsible for sending the image of the desktop to allow the client to see an up-to-date view of the applications. To improve user experience over slow network links, the server optimizes the transfer by optionally compressing the image and sending only relevant (modified) parts of the GUI desktop.

To make demo recordings usable for RL agent training, we need to convert this VNC format into a set of images and user events issued at the time of the image. To achieve this, I implemented a small VNC protocol parser using the Kaitai Struct binary parser language (the project website is `http://kaitai.io/`), which provides a convenient way to parse complex binary file formats using a declarative YAML-formatted language. If you're curious, the source files for the client and server messages are in the `Chapter16/ksy` directory.

Python code related to the demo format is placed in the module `Chapter16/lib/vnc_demo.py`, which contains a high-level function loader for the demo directory and the set of lower-level methods used to interpret the internal binary format. The result returned by the loader function, `vnc_demo.load_demo()`, is the list of tuples. Every tuple contains a NumPy array with the observation used by the MiniWoB model and the index of the mouse action performed.

To check the demo data, there is a small utility, `Chapter16/adhoc/demo_dump.py`, that loads the demo directory with `client.fbs` and `server.fbs` and dumps demo samples as image files. The example of the command line used to convert the demonstrations that I recorded into images is as follows:

```
rl_book_samples/Chapter16$ ./adhoc/demo_dump.py -d data/demo-CountSides/
-e wob.mini.CountSides-v0 -o count
[2018-01-30 12:44:11,794] Making new env: wob.mini.CountSides-v0
[2018-01-30 12:44:11,796] Using SoftmaxClickMouse with action_region=
(10, 125, 170, 285), noclick_regions=[]
[2018-01-30 12:44:11,797] SoftmaxClickMouse noclick regions removed
0 of 256 actions Loaded 64 demo samples
[2018-01-30 12:44:12,191] Making new env: wob.mini.CountSides-v0
[2018-01-30 12:44:12,192] Using SoftmaxClickMouse with action_region=
(10, 125, 170, 285), noclick_regions=[]
[2018-01-30 12:44:12,192] SoftmaxClickMouse noclick regions removed
0 of 256 actions
```

This command produced 64 image files with the prefix `count`. What follows are examples of images produced by this utility, showing demonstrations that I recorded.
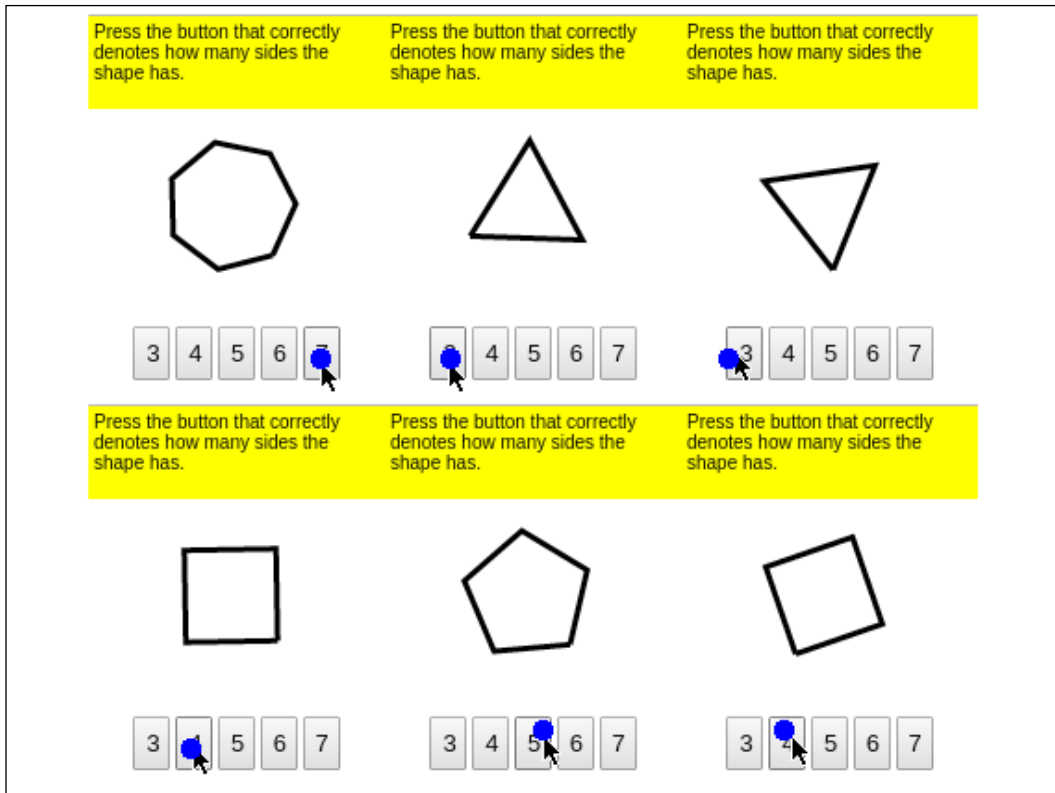
Figure 16.14: On every image, the click point is shown as a circle

The binary data for this recording is available in `Chapter16/demos/demo-CountSides.tar.gz`, and you need to unpack it before usage. It also needs to be said that my implementation of the VNC protocol reading is experimental, works only with files produced by the VNC proxy used in the MiniWoB image 0.20.0, and does not pretend to fully comply with VNC protocol RFC. Furthermore, the reading process is hard-coded for our action space transformation and doesn't produce examples of mouse movements, keys pressed, and other events. If you think it should be extended for a more generic case, you're always welcome to contribute.

Training using demonstrations

Now that we know how to record and load the demonstration data, we have only one question unanswered: how does our training process need to be modified to incorporate human demonstrations? The simplest solution, which nevertheless works surprisingly well, is to use the log-likelihood objective that we used in training our chatbot in *Chapter 14*, *Training Chatbots with RL*. To do so, we need to look at our A3C model as a classification problem producing the classification of input observations in its policy head. In its simplest form, the value head will be left untrained, but, in fact, it won't be hard to train it: we know the rewards obtained during the demonstrations, so what is needed is to calculate the discounted reward from every observation to the end of the episode.

To check how it was implemented, let's return to the code pieces we skipped during the description of `Chapter16/wob_click_train.py`. First of all, we can pass the directory with the demonstration data by passing the `--demo <DIR>` option in the command line. This will enable the branch shown on the following code block, where we load the demonstration samples from the directory specified. Function `vnc_demo.load_demo()` is smart enough to automatically load demonstrations from any level of subdirectories, so you just need to pass the directory where your demonstrations are placed.

```
demo_samples = None
if args.demo:
    demo_samples = vnc_demo.load_demo(args.demo, env_name)
    if not demo_samples:
        demo_samples = None
    else:
        print("Loaded %d demo samples, will use them "
              "during training" % len(demo_samples))
```

The second piece of code relevant to demonstration training is inside the training loop and is executed before any normal batch. The training from demonstrations is performed with some probability (by default, it is 0.5) and specified by the `DEMO_PROB` hyperparameter.

```
if demo_samples and random.random() < DEMO_PROB:
    random.shuffle(demo_samples)
    demo_batch = demo_samples[:BATCH_SIZE]
    model_vnc.train_demo(
        net, optimizer, demo_batch, writer,
        step_idx, device=device)
```

The logic is simple: with `DEMO_PROB` chance, we sample `BATCH_SIZE` samples from our demonstration data and perform the round of training of our network in the batch. The actual training is performed by the `model_vnc.train_demo()` function, which is shown in the following:

```
def train_demo(net, optimizer, batch, writer, step_idx,
               preprocessor=ptan.agent.default_states_preprocessor,
               device="cpu"):
    batch_obs, batch_act = zip(*batch)
    batch_v = preprocessor(batch_obs)
    if torch.is_tensor(batch_v):
        batch_v = batch_v.to(device)
    optimizer.zero_grad()
    ref_actions_v = torch.LongTensor(batch_act).to(device)
    policy_v = net(batch_v)[0]
    loss_v = F.cross_entropy(policy_v, ref_actions_v)
    loss_v.backward()
    optimizer.step()
    writer.add_scalar("demo_loss", loss_v.item(), step_idx)
```

The training code is also very simple and straightforward. We split our batch on the observation and the actions list, preprocess the observations to convert them into a PyTorch tensor, and place them on the GPU. We then ask our A3C network to return the policy and calculate the cross-entropy loss between the result and the desired actions. From an optimization point of view, we're pushing our network toward the actions taken in the demonstrations.

Results

To check the effect of demonstrations, I performed two sets of training on the `CountSides` problem with the same hyperparameters: one was done without demonstrations, and another used 64 demonstration clicks.

To start the training without human demonstrations, this command line was used:

$ ./wob_click_train.py --cuda -n v1-nodemo --env CountSides-v0

To incorporate recorded demonstrations into the training process, the training needed to be started like this:

$ ./wob_click_train.py --cuda -n v1-demo --env CountSides-v0 --demo demos

The archive `demos/demo-CountSides.tar.gz` needed to be extracted into the `demos` directory.

The difference was dramatic. Training performed *from scratch* reached the best mean reward of –0.4 after 15 hours of training and 2.2M frames without any significant improvement in the training dynamics. The following are charts for training without demonstrations.
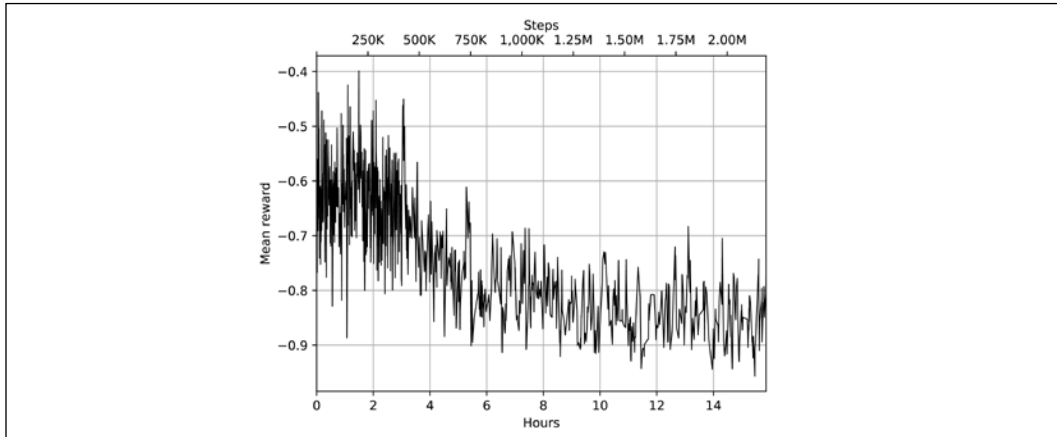
Figure 16.15: The reward dynamics of the CountSides environment without demonstrations
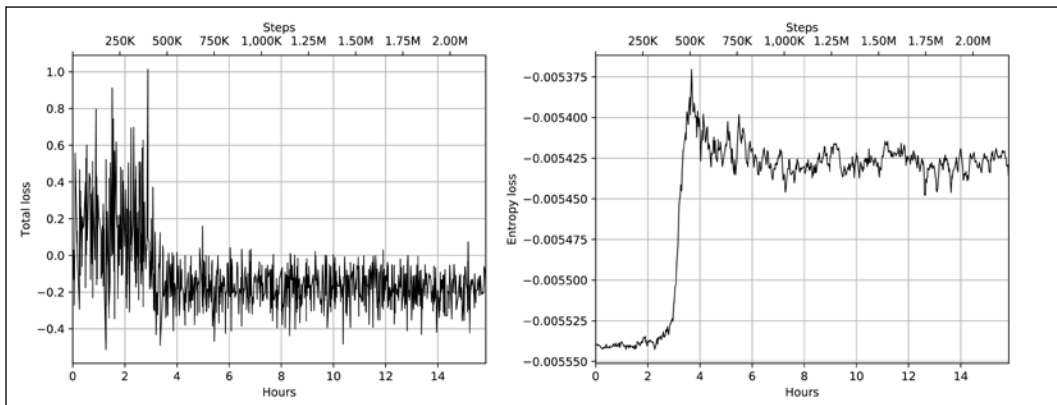
Figure 16.16: The total loss (left) and entropy loss (right) during the training
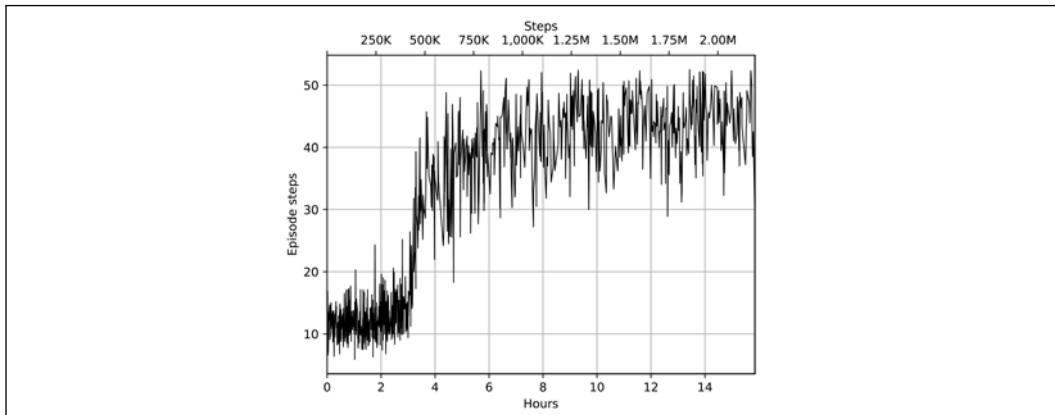
Figure 16.17: Lengths of training episodes

With just 64 demonstration samples added and in only 20k frames, the training was able to reach the mean reward of 1.65. High entropy loss, shown as follows, demonstrates that the agent became very sure about its actions.
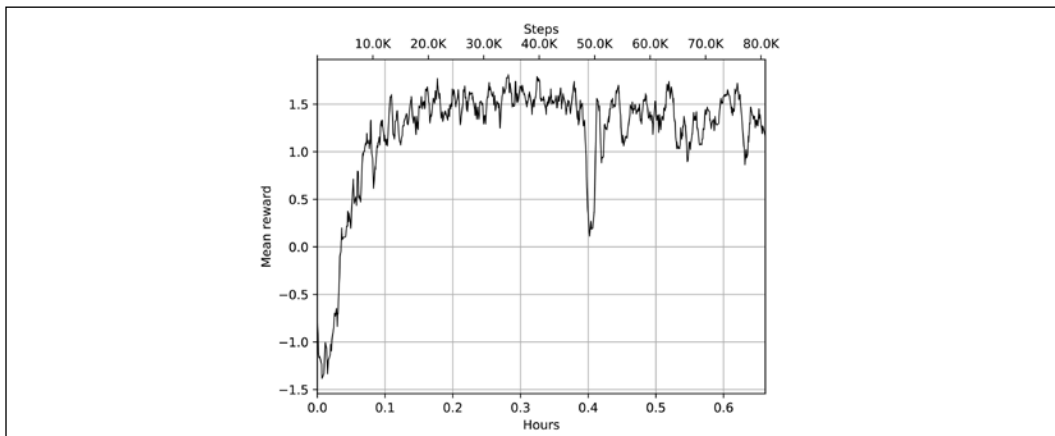
Figure 16.18: The mean reward dynamics during the training with demonstrations
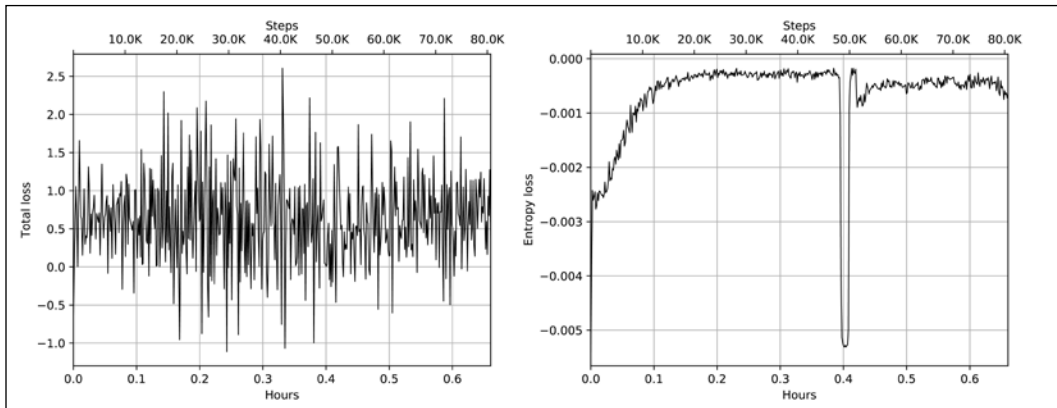
Figure 16.19: The total loss (left) and entropy loss (right) during the training
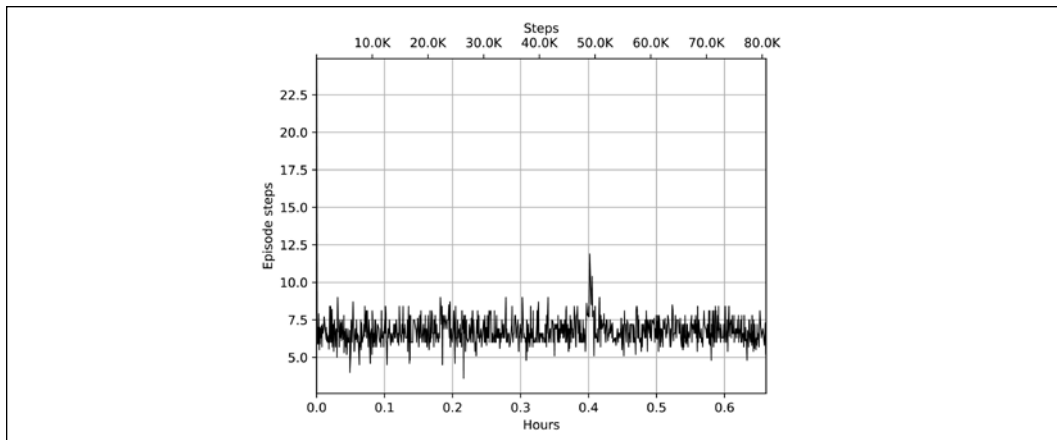
Figure 16.20: Lengths of training episodes

The lengths of training episodes, shown in *Figure 16.20*, dropped from 20 to six so quickly after the training that this decrease is not visible on the plot.

To put things into perspective, the following are the same charts combined.
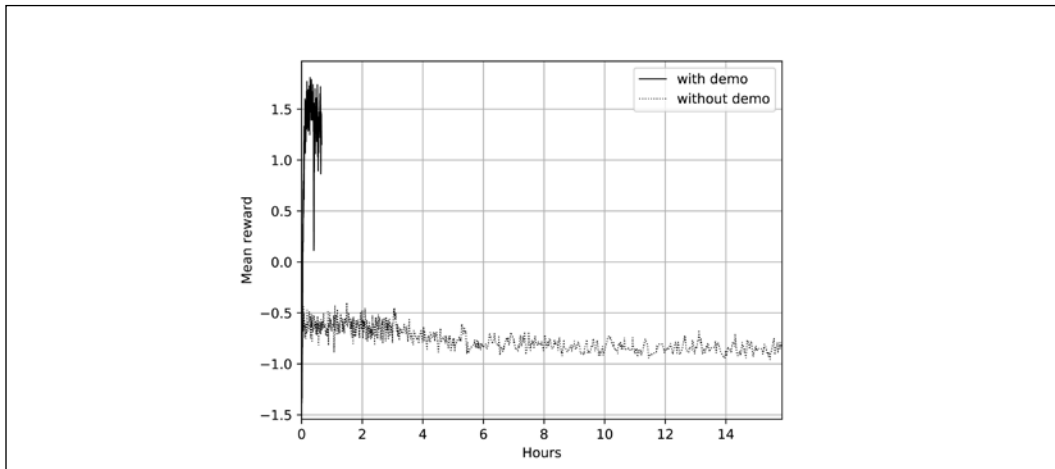
Figure 16.21: The mean reward with and without demonstrations
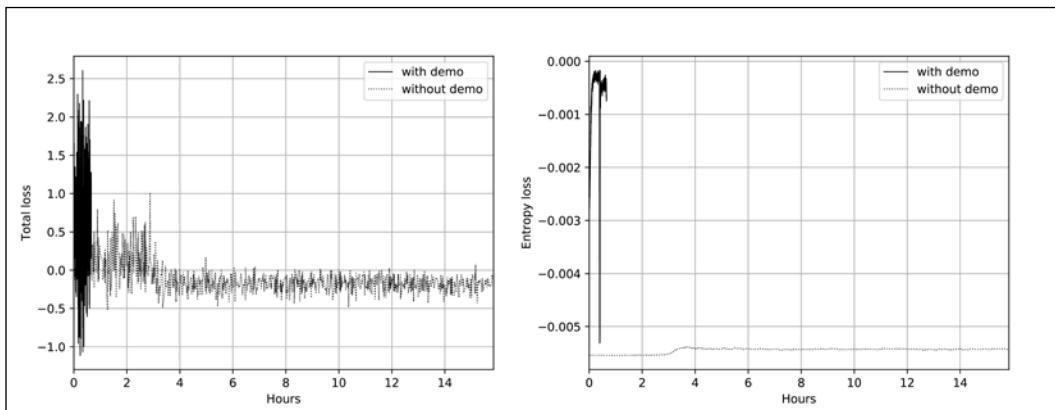
Figure 16.22: The total loss (left) and entropy loss (right) with and without demonstrations
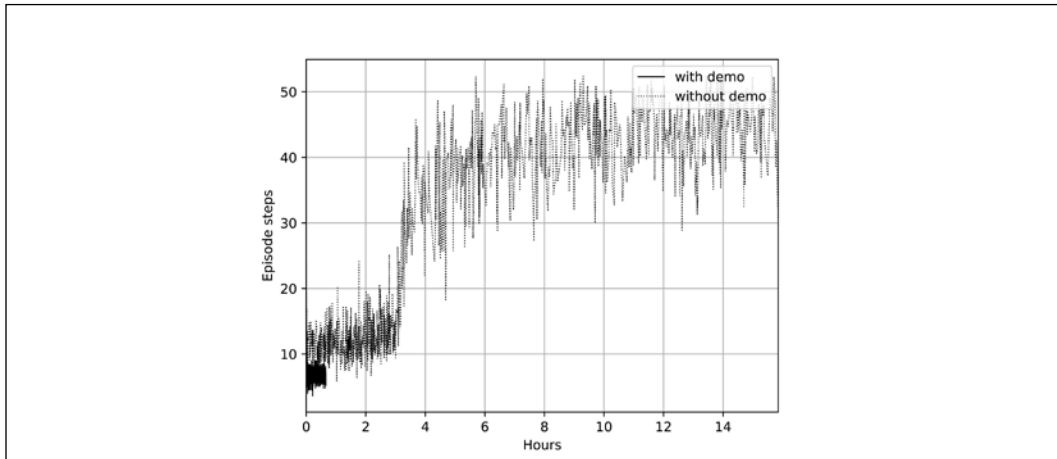
Figure 16.23: Episode steps with and without demonstrations

The tic-tac-toe problem

To check the effect of demonstrations on training, I took a more complex problem from MiniWoB: the tic-tac-toe game. I recorded some demonstrations (available in `Chapter16/demos/demo-TicTacToe.tgz`) and in total almost 200 actions, and some examples of them are as follows:
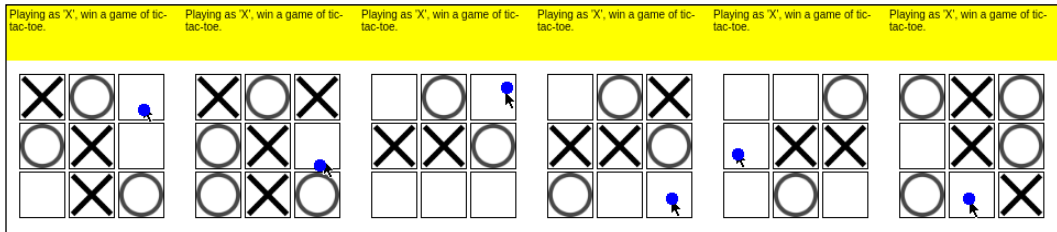
Figure 16.24: The tic-tac-toe environment with human actions

After one hour of training, the agent was able to reach the mean reward of 0.05, which means that it can win slightly more frequently than it loses. The training dynamics are shown as follows. To improve the exploration, demo training probability was decreased from 0.5 to 0.01 after 25k frames seen. Of course, the noisiness of the reward chart suggests that the policy is far from being consistently better than MiniWoB artificial intelligence.
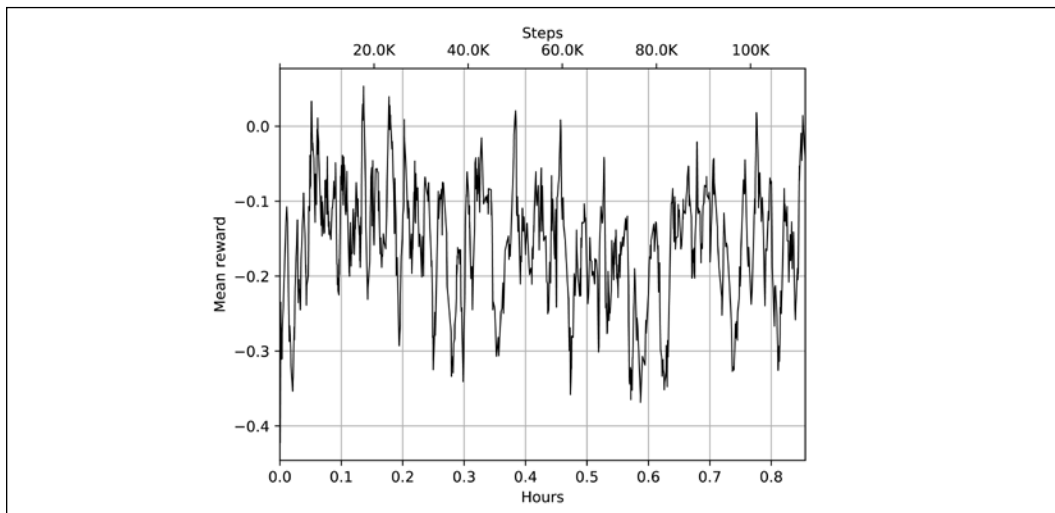
Figure 16.25: The tic-tac-toe mean reward dynamics
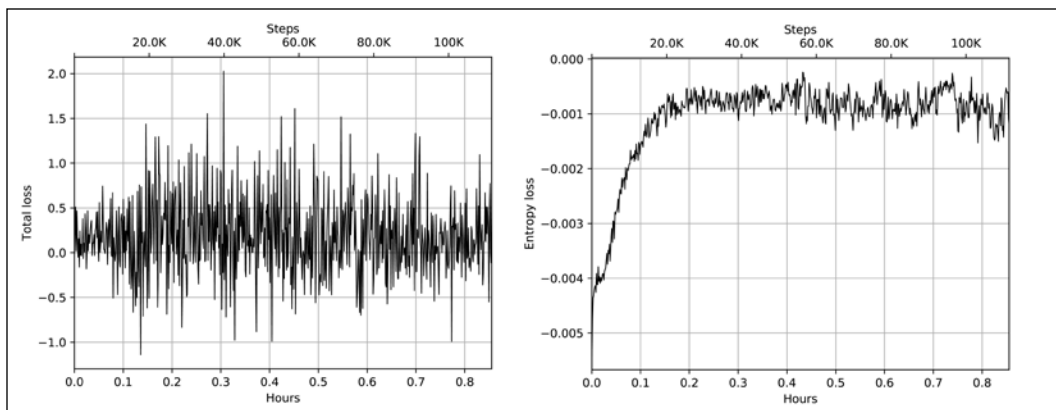
Figure 16.26: The total loss (left) and entropy loss (right)
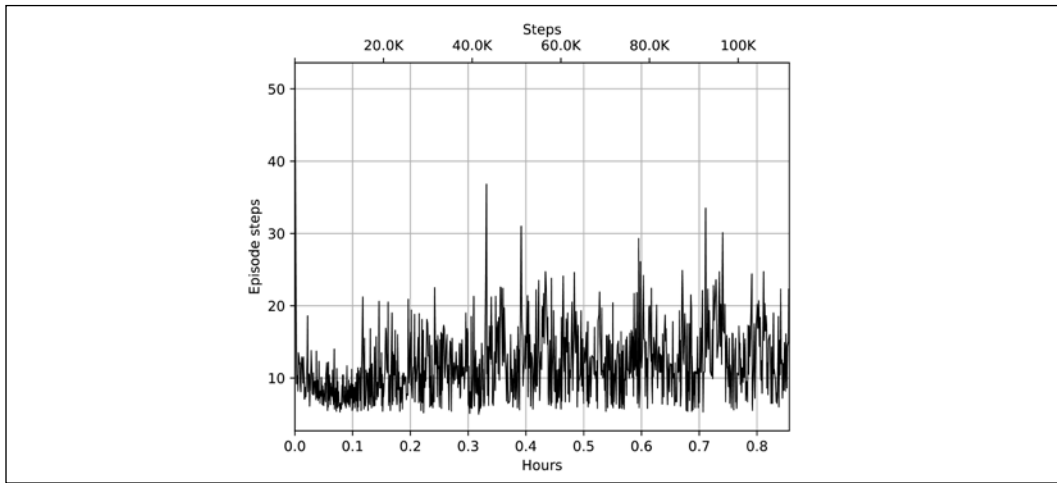
Figure 16.27: Lengths of training episodes

Using `wob_click_play.py`, we can check the agent's actions step by step. For example, the following are some games played by the best model with the mean reward of 0.187:
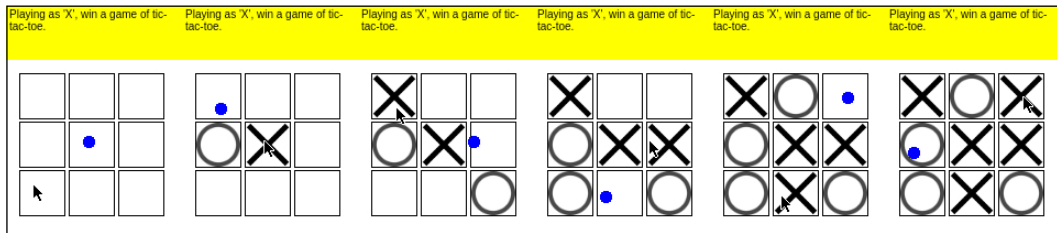
Figure 16.28: A game played by the agent
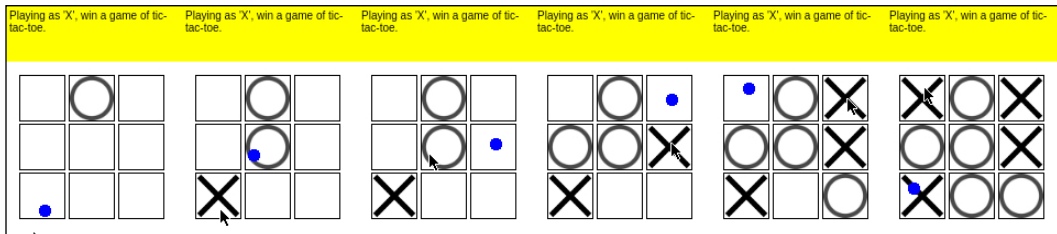
Figure 16.29: A second game played by the agent
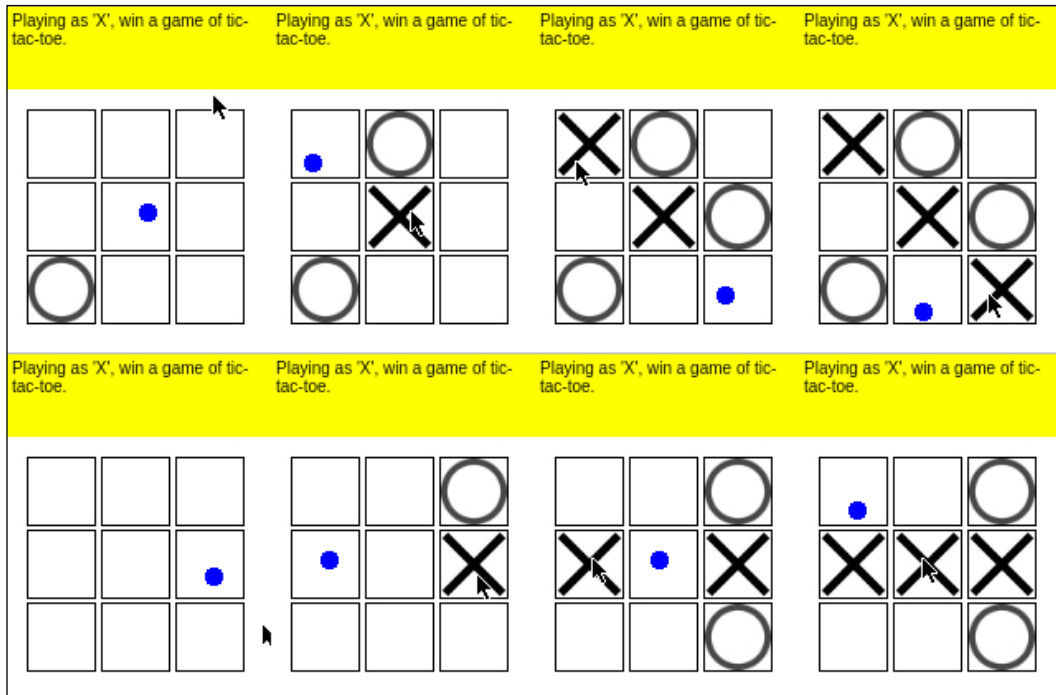
Figure 16.30: More games played by the agent

Adding text descriptions

As the last example of this chapter, we will add text descriptions of the problem into observations of our model. I have already mentioned that some problems contain vital information given in a text description, like the index of tabs needed to be clicked or the list of entries that the agent needs to check. The same information is shown on top of the image observation, but pixels are not always the best representation of simple text.

To take this text into account, we need to extend our model's input from an image only to an image and text data. We worked with text in the previous chapter, so a recurrent neural network (RNN) is quite an obvious choice (maybe not the best for such a toy problem, but it is flexible and scalable).

Implementation

I'm not going to cover this example in detail but will just focus on the most important points of the implementation. (The whole code is in `Chapter16/wob_click_mm_train.py`.) In comparison to our clicker model, text extension doesn't add too much.

First of all, we should ask the wrapper `MiniWoBCropper` to keep the text obtained from the observation. The complete source code of this class has already been shown earlier in this chapter. To keep the text, we should pass `keep_text=True` to the wrapper constructor, which makes this class return a tuple with a NumPy array and text string, instead of just a NumPy array with the image.

Then, we need to prepare our model to be able to process such tuples, instead of a batch of NumPy arrays. This needs to be done in two places: in our agent (when we use the model to choose the action) and in the training code. To adapt the observation in a model-friendly way, we can use a special functionality of the PTAN library, called `preprocessor`. The core idea is very simple: `preprocessor` is a callable function that needs to convert the list of observations in a form that is ready to be passed to the model. By default, `preprocessor` converts the list of NumPy arrays into a PyTorch tensor and, optionally, copies it in GPU memory. However, sometimes more sophisticated transformations are required, like in our case, when we need to pack the images into the tensor, but text strings require special handling. In that case, you can redefine the default `preprocessor` and pass it into the `ptan.Agent` class.

In theory, the `preprocessor` functionality could be moved into the model itself, thanks to PyTorch flexibility, but the default `preprocessor` simplifies our lives in cases when observations are just NumPy arrays. The following is the `preprocessor` class source code taken from the `Chapter16/lib/model_vnc.py` module:

```
MM_EMBEDDINGS_DIM = 50
MM_HIDDEN_SIZE = 128
MM_MAX_DICT_SIZE = 100

TOKEN_UNK = "#unk"

class MultimodalPreprocessor:
    log = logging.getLogger("MulitmodalPreprocessor")

    def __init__(self, max_dict_size=MM_MAX_DICT_SIZE,
                 device="cpu"):
        self.max_dict_size = max_dict_size
        self.token_to_id = {TOKEN_UNK: 0}
        self.next_id = 1
        self.tokenizer = TweetTokenizer(preserve_case=True)
        self.device = device
```

In the constructor, we create a mapping from the token to the identifier (which will be dynamically extended) and create the tokenizer from the `nltk` package.

```
def __len__(self):
    return len(self.token_to_id)

def __call__(self, batch):
    tokens_batch = []
    for img_obs, txt_obs in batch:
        tokens = self.tokenizer.tokenize(txt_obs)
        idx_obs = self.tokens_to_idx(tokens)
        tokens_batch.append((img_obs, idx_obs))
    # sort batch decreasing to seq len
    tokens_batch.sort(key=lambda p: len(p[1]), reverse=True)
    img_batch, seq_batch = zip(*tokens_batch)
    lens = list(map(len, seq_batch))
```

The goal of our preprocessor is to convert a batch of (image, text) tuples into two objects: the first has to be a tensor with the image data of shape (batch_size, 3, 210, 160), and the second has to contain the batch of tokens from text descriptions in the form of a packed sequence. The packed sequence is a PyTorch data structure suitable for efficient processing with an RNN. We discussed this in *Chapter 14*, *Training Chatbots with RL*.

As the first step of our transformation, we tokenize text strings into tokens and convert every token into the list of integer IDs. Then, we sort our batch by decreasing token length, which is a requirement of the underlying cuDNN library for efficient RNN processing.

```
img_v = torch.FloatTensor(img_batch).to(self.device)
```

In the preceding line, we convert observation images into the single tensor.

```
seq_arr = np.zeros(shape=(len(seq_batch),
                          max(len(seq_batch[0]), 1)),
                   dtype=np.int64)
for idx, seq in enumerate(seq_batch):
    seq_arr[idx, :len(seq)] = seq
    # Map empty sequences into single #UNK token
    if len(seq) == 0:
        lens[idx] = 1
```

To create the packed sequence class, we first need to create a *padded sequence* tensor, which is a matrix of (`batch_size, len_of_longest_seq`). We copy IDs of our sequences into this matrix.

```
seq_v = torch.LongTensor(seq_arr).to(self.device)
seq_p = rnn_utils.pack_padded_sequence(
    seq_v, lens, batch_first=True)
return img_v, seq_p
```

As a final step, we create the tensors from the NumPy matrix and convert them into *packed* form by using the PyTorch utility function. The result of the transformation is two objects: a tensor with images and a packed sequence with tokenized texts.

```
def tokens_to_idx(self, tokens):
    res = []
    for token in tokens:
        idx = self.token_to_id.get(token)
        if idx is None:
            if self.next_id == self.max_dict_size:
                self.log.warning(
                    "Maximum size of dict reached, token "
                    "'%s' converted to #UNK token", token)
                idx = 0
            else:
                idx = self.next_id
                self.next_id += 1
                self.token_to_id[token] = idx
        res.append(idx)
    return res
```

The preceding utility function has to convert the list of tokens into a list of IDs. The tricky thing is that we don't know in advance the size of the dictionary from the text descriptions. One approach would be to work on the character level and feed individual characters into the RNN, but it would result in sequences that are too long to process. The alternative solution is to hard-code some reasonable dictionary size, say 100 tokens, and dynamically assign token IDs to tokens that we have never seen before. In this implementation, the latter approach is used, but it might not be applicable to MiniWoB problems that contain randomly generated strings in the text description.

```
def save(self, file_name):
    with open(file_name, 'wb') as fd:
        pickle.dump(self.token_to_id, fd)
        pickle.dump(self.max_dict_size, fd)
        pickle.dump(self.next_id, fd)
```

```
    @classmethod
    def load(cls, file_name):
        with open(file_name, "rb") as fd:
            token_to_id = pickle.load(fd)
            max_dict_size = pickle.load(fd)
            next_id = pickle.load(fd)

            res = MultimodalPreprocessor(max_dict_size)
            res.token_to_id = token_to_id
            res.next_id = next_id
            return res
```

As our token-to-ID mapping is dynamically generated, our `preprocessor` has to provide a way to save and load this state in a file. The preceding two functions do exactly that. The next piece of the puzzle is the model class itself, which is an extension of our model used:

```
class ModelMultimodal(nn.Module):
    def __init__(self, input_shape, n_actions,
                 max_dict_size=MM_MAX_DICT_SIZE):
        super(ModelMultimodal, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 64, 5, stride=5),
            nn.ReLU(),
            nn.Conv2d(64, 64, 3, stride=2),
            nn.ReLU(),
        )

        conv_out_size = self._get_conv_out(input_shape)

        self.emb = nn.Embedding(max_dict_size, MM_EMBEDDINGS_DIM)
        self.rnn = nn.LSTM(MM_EMBEDDINGS_DIM, MM_HIDDEN_SIZE,
                           batch_first=True)

        self.policy = nn.Linear(
            conv_out_size + MM_HIDDEN_SIZE*2, n_actions)
        self.value = nn.Linear(
            conv_out_size + MM_HIDDEN_SIZE*2, 1)
```

The difference is in a new embedding layer, which converts integer token IDs into dense token vectors and a long short-term memory (LSTM) RNN.

The outputs from the convolution and RNN layers are concatenated and fed into the policy and value heads, so the dimensionality of their input is the image and text features combined.

```python
def _get_conv_out(self, shape):
    o = self.conv(torch.zeros(1, *shape))
    return int(np.prod(o.size()))

def _concat_features(self, img_out, rnn_hidden):
    batch_size = img_out.size()[0]
    if isinstance(rnn_hidden, tuple):
        flat_h = list(map(lambda t: t.view(batch_size, -1),
                          rnn_hidden))
        rnn_h = torch.cat(flat_h, dim=1)
    else:
        rnn_h = rnn_hidden.view(batch_size, -1)
    return torch.cat((img_out, rnn_h), dim=1)
```

The preceding function performs the concatenation of the image and RNN features into one single square tensor.

```python
def forward(self, x):
    x_img, x_text = x
    assert isinstance(x_text, rnn_utils.PackedSequence)

    # deal with text data
    emb_out = self.emb(x_text.data)
    emb_out_seq = rnn_utils.PackedSequence(
        emb_out, x_text.batch_sizes)
    rnn_out, rnn_h = self.rnn(emb_out_seq)

    # extract image features
    fx = x_img.float() / 256
    conv_out = self.conv(fx).view(fx.size()[0], -1)

    feats = self._concat_features(conv_out, rnn_h)
    return self.policy(feats), self.value(feats)
```

In the `forward` function, we expect two objects prepared by the `preprocessor`: a tensor with input images and packed sequences of the batch. Images are processed with convolutions, and text data is fed through the RNN; then, both results are concatenated, and the policy and value results are calculated.

That's most of the new code. The training Python script, `wob_click_mm_train.py`, is mostly a copy of `wob_click_train.py`, with just the tiny difference of the `preprocessor` created. `keep_text=True` was passed to the `MiniWoBCropper()` constructor and other small modifications.

Results

I ran several experiments on the environment `ClickButton-v0`, which has the goal to make a selection between several random buttons. Some of the recorded demonstrations are shown in the following figure:
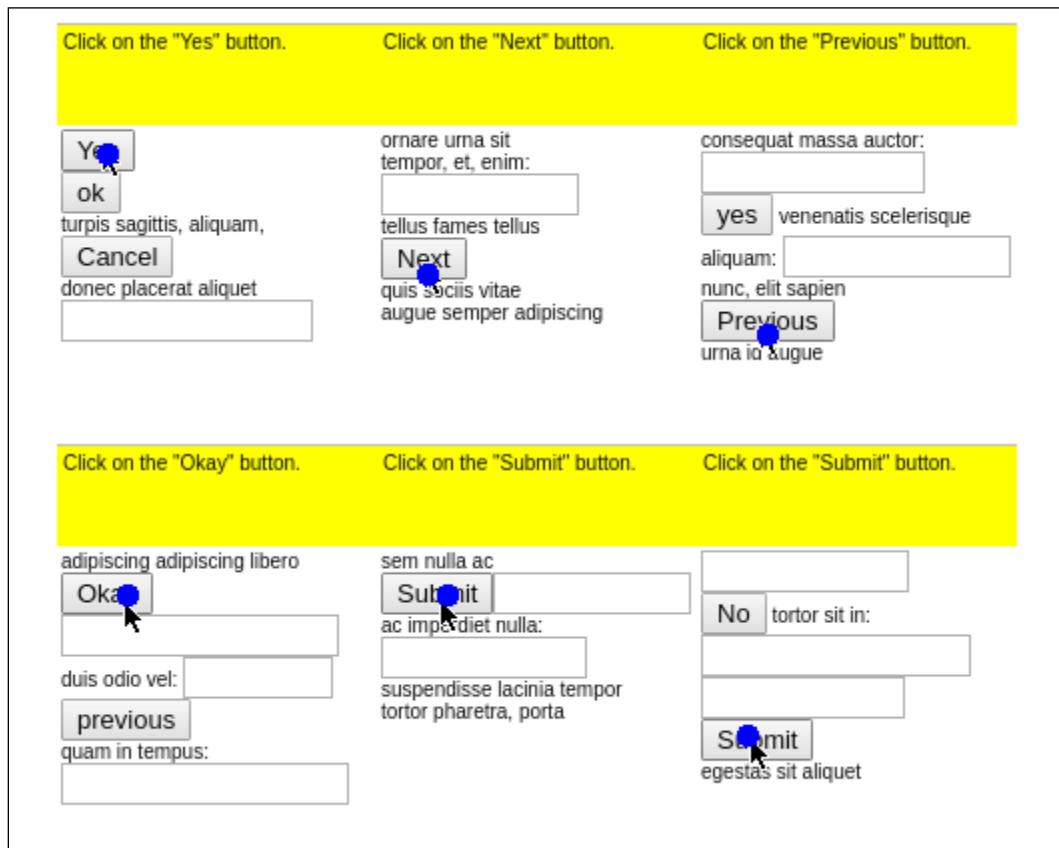
Figure 16.31: Screenshots of the ClickButton environment demonstrations

Even with demonstrations, a model without text descriptions was able to reach the mean reward of 0.5, which is still better than the random clicking policy, but is far from the optimal policy.

What follows are figures that show training dynamics.
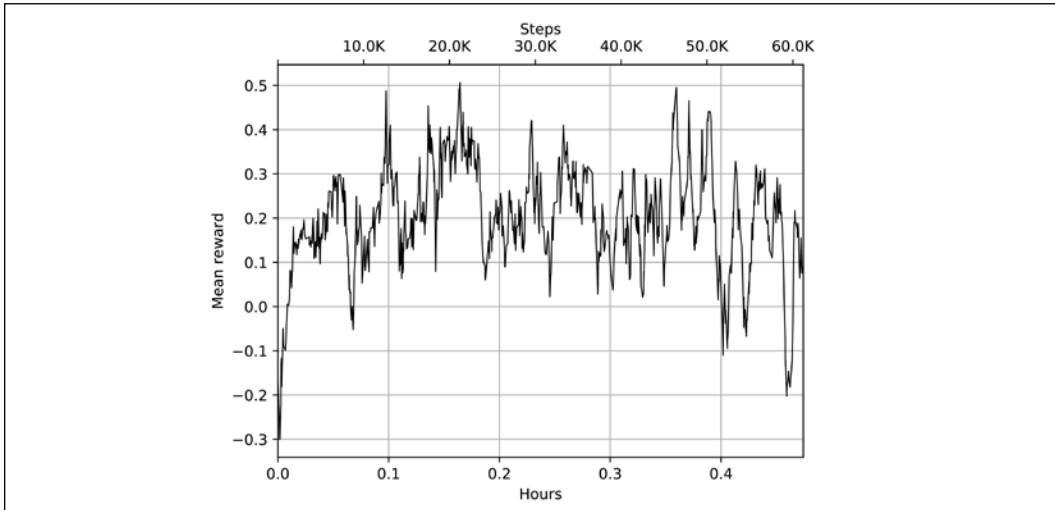
Figure 16.32: The reward for ClickButton without text descriptions
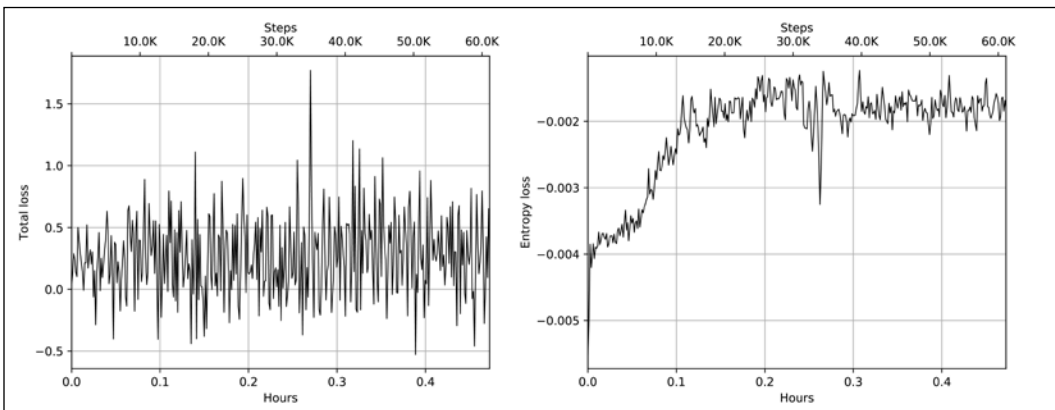
Figure 16.33: Loss dynamics for ClickButton without text descriptions
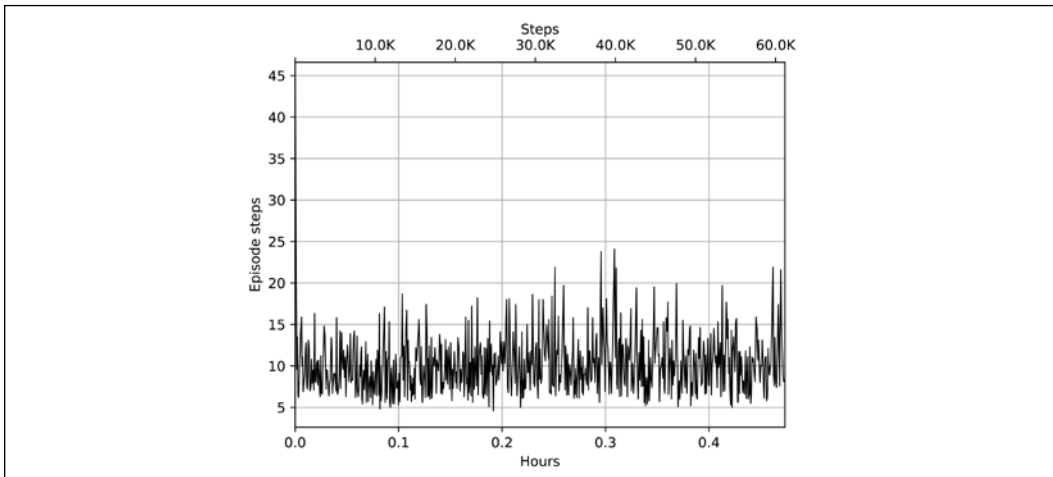
Figure 16.34: Episode steps

In addition, the agent enriched with text descriptions behaved worse (the best mean reward for 100 episodes was 0.45).
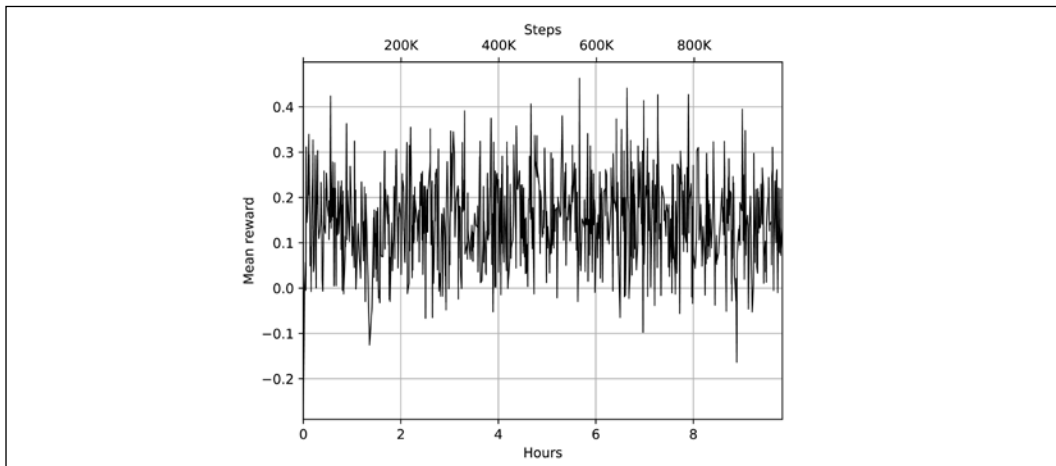
Figure 16.35: The reward of the ClickButton environment training with text descriptions
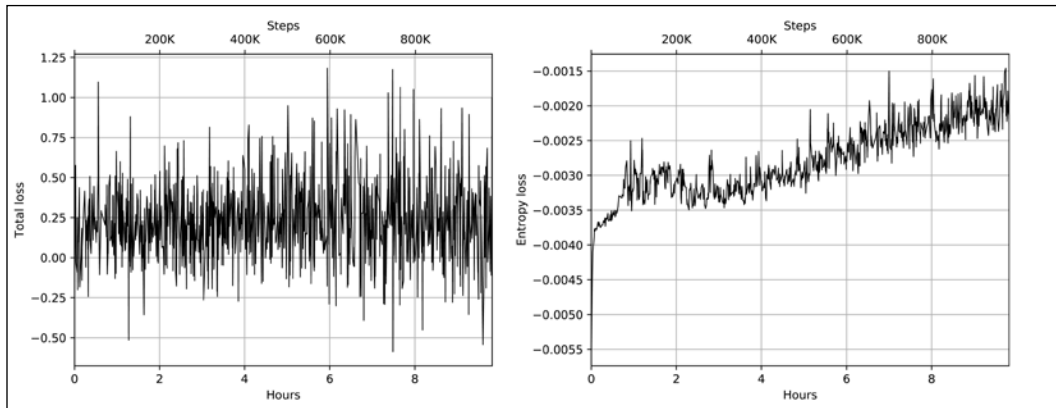
Figure 16.36: The loss dynamics of training with text descriptions
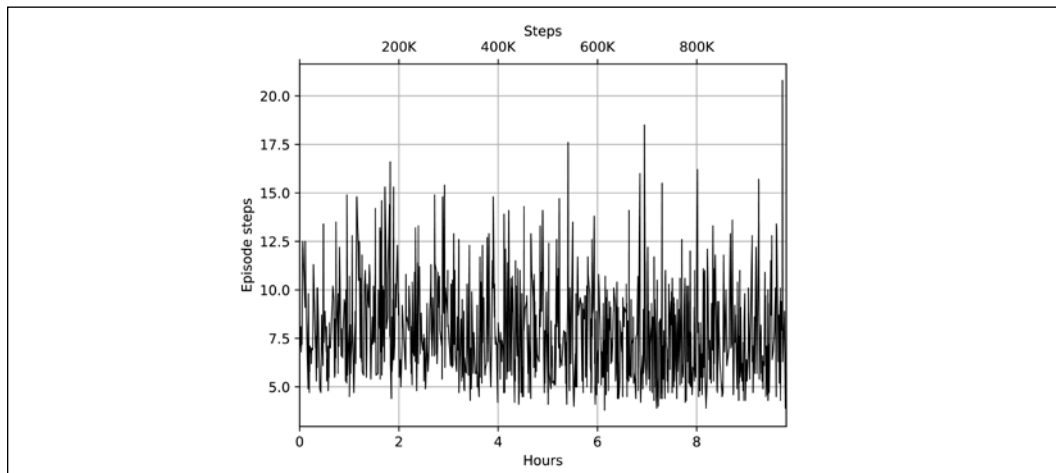
Figure 16.37: Episode steps

The reward dynamics of both models are quite noisy, which may signal that tuning hyperparameters and/or increasing the number of parallel environments could help.

Things to try

In this chapter, we only started playing with MiniWoB by touching upon the six easiest environments from the full set of 80 problems, so there is plenty of uncharted territory ahead. If you want to practice, there are several items you can experiment with:

- Testing the robustness of demonstrations to noisy clicks.

- Implementing training of the value head of A3C based on demonstration data.

- Implementing more sophisticated mouse control, like *move mouse N pixels left/ right/top/bottom*.

- Using some pretrained **optical character recognition** (**OCR**) network (or training your own!) to extract text information from the observations.

- Taking other problems and trying to solve them. There are some quite tricky and fun problems, like *sort items using drag-n-drop* or *repeat the pattern using checkboxes*.

- Checking MiniWoB++ (`https://stanfordnlp.github.io/miniwob-plusplus/`) from the Stanford NLP Group. It will require learning and writing new wrappers; as mentioned earlier, Selenium is used to work with browsers, rather than using a Universe approach with VNC and containers.

- Another challenging problem is to make Q-learning work with MiniWoB tasks. In RLSS 2019 (`https://rlss.inria.fr/`), I tried to use MiniWoB as one of the practice sessions. Despite all my efforts, DQN failed to converge even on simple problems like CloseDialog. It might be due to some software bug (but I failed to find one despite all my efforts), or an indication of something more fundamental. You can find the RLSS 2019 practice session materials in this GitHub repository: `https://github.com/yfletberliac/rlss-2019`. MiniWoB notebooks are available in the `labs/final_project/MiniWoB` (`https://github.com/yfletberliac/rlss-2019/tree/master/labs/final_project/MiniWoB`) subdirectory.

Summary

In this chapter, you saw the practical application of RL methods for browser automation and used the MiniWoB benchmark from OpenAI. This chapter concludes part three of the book. The next part will be devoted to more complicated and recent methods related to continuous action spaces, non-gradient methods, and other more advanced methods of RL.

In the next chapter, we will take a look at continuous control problems, which are an important subfield of RL, both theoretically and practically.

17

Continuous Action Space

This chapter kicks off the advanced reinforcement learning (RL) part of the book by taking a look at a problem that has only been briefly mentioned: working with environments when our action space is not discrete. In this chapter, you will become familiar with the challenges that arise in such cases and learn how to solve them.

Continuous action space problems are an important subfield of RL, both theoretically and practically, because they have essential applications in robotics (which will be the subject of the next chapter), control problems, and other fields in which we communicate with physical objects.

In this chapter, we will:

- Cover the continuous action space, why it is important, how it differs from the already familiar discrete action space, and the way it is implemented in the Gym API
- Discuss the domain of continuous control using RL methods
- Check three different algorithms on the problem of a four-legged robot

Why a continuous space?

All the examples that we have seen so far in the book had a discrete action space, so you might have the wrong impression that discrete actions dominate the field. This is a very biased view, of course, and just reflects the selection of domains that we picked our test problems from. Besides Atari games and simple, classic RL problems, there are many tasks that require more than just making a selection from a small and discrete set of things to do.

To give you an example, just imagine a simple robot with only one controllable joint that can be rotated in some range of degrees. Usually, to control a physical joint, you have to specify either the desired position or the force applied.

In both cases, you need to make a decision about a continuous value. This value is fundamentally different from a discrete action space, as the set of values on which you can make a decision is potentially infinite. For instance, you could ask the joint to move to a 13.5° angle or 13.512° angle, and the results could be different. Of course, there are always some physical limitations of the system. You can't specify the action with infinite precision, but the size of the potential values could be very large.

In fact, when you need to communicate with a physical world, a **continuous action space** is much more likely than having a discrete set of actions. As an example, different kinds of robots control systems (such as a heating/cooling controller). The methods of RL could be applied to this domain, but there are some details that you need to take into consideration before using the advantage actor-critic (A2C) or deep Q-network (DQN) methods.

In this chapter, we will explore how to deal with this family of problems. This will act as a good starting point for learning about this very interesting and important domain of RL.

The action space

The fundamental and obvious difference with a continuous action space is its continuity. In contrast to a discrete action space, when the action is defined as a discrete, mutually exclusive set of options to choose from (for example {left, right}, which contains only two elements), the continuous action has a value from some range (for instance, [0…1], which includes infinite elements, like 0.5, $\frac{\sqrt{3}}{2}$, and $\frac{\pi^3}{e^5}$). On every time step, the agent needs to select the concrete value for the action and pass it to the environment.

In Gym, a continuous action space is represented as the gym.spaces.Box class, which was described in *Chapter 2, OpenAI Gym*, when we talked about the observation space. You may remember that Box includes a set of values with a shape and bounds. For example, every observation from the Atari emulator was represented as Box(low=0, high=255, shape=(210, 160, 3)), which means 100,800 values organized as a 3D tensor, with values from the 0...255 range.

For the action space, it's unlikely that you'll work with such large numbers of actions. For example, the four-legged robot that we will use as a testing environment has eight continuous actions, which correspond to eight motors, two in every leg. For this environment, the action space will be defined as Box(low=-1, high=1, shape=(8,)), which means eight values from the range –1…1 have to be selected at every timestamp to control the robot.

In this case, the action passed to the `env.step()` at every step won't be an integer anymore; it will be a NumPy vector of some shape with individual action values. Of course, there could be more complicated cases when the action space is a combination of discrete and continuous actions, which may be represented with the `gym.spaces.Tuple` class.

Environments

Most of the environments that include continuous action spaces are related to the physical world, so physics simulations are normally used. There are lots of software packages that can simulate physical processes, from very simple open source tools to complex commercial packages that can simulate multiphysics processes (such as fluid, burning, and strength simulations).

In the case of robotics, one of the most popular packages is MuJoCo, which stands for **Multi-Joint dynamics with Contact** (http://www.mujoco.org). This is a physics engine in which you can define the components of the system and their interaction and properties. Then the simulator is responsible for *solving the system* by taking into account your intervention and finding the parameters (usually the location, velocities, and accelerations) of the components. This makes it ideal as a playground for RL environments, as you can define fairly complicated systems (such as multipede robots or robotic arms or humanoids) and then feed the observation into the RL agent, getting actions back.

Unfortunately, MuJoCo isn't free and requires a license. There is a one-month trial license available on websites, but after the trial, a full license will be required. For students, the MuJoCo developers provide a free license, but they impose different limitations. For post-university RL enthusiasts, buying a license could be overkill because there is an open source alternative, called **PyBullet**, that provides similar functionality (maybe at a cost of lower speed or accuracy) for free.

PyBullet is available at https://github.com/bulletphysics/bullet3 and can be installed by running `pip install pybullet` inside your virtual environment. The following code (which is available in `Chapter17/01_check_env.py`) allows you to check that PyBullet works. It looks at the action space and renders an image of the environment that we will use as a guinea pig in this chapter.

```
import gym
import pybullet_envs

ENV_ID = "MinitaurBulletEnv-v0"
RENDER = True

if __name__ == "__main__":
```

```
spec = gym.envs.registry.spec(ENV_ID)
spec._kwargs['render'] = RENDER
env = gym.make(ENV_ID)

print("Observation space:", env.observation_space)
print("Action space:", env.action_space)
print(env)
print(env.reset())
input("Press any key to exit\n")
env.close()
```

After you start the utility, it should open the graphical user interface (GUI) window with our four-legged robot, shown in the following figure, that we will train how to move.
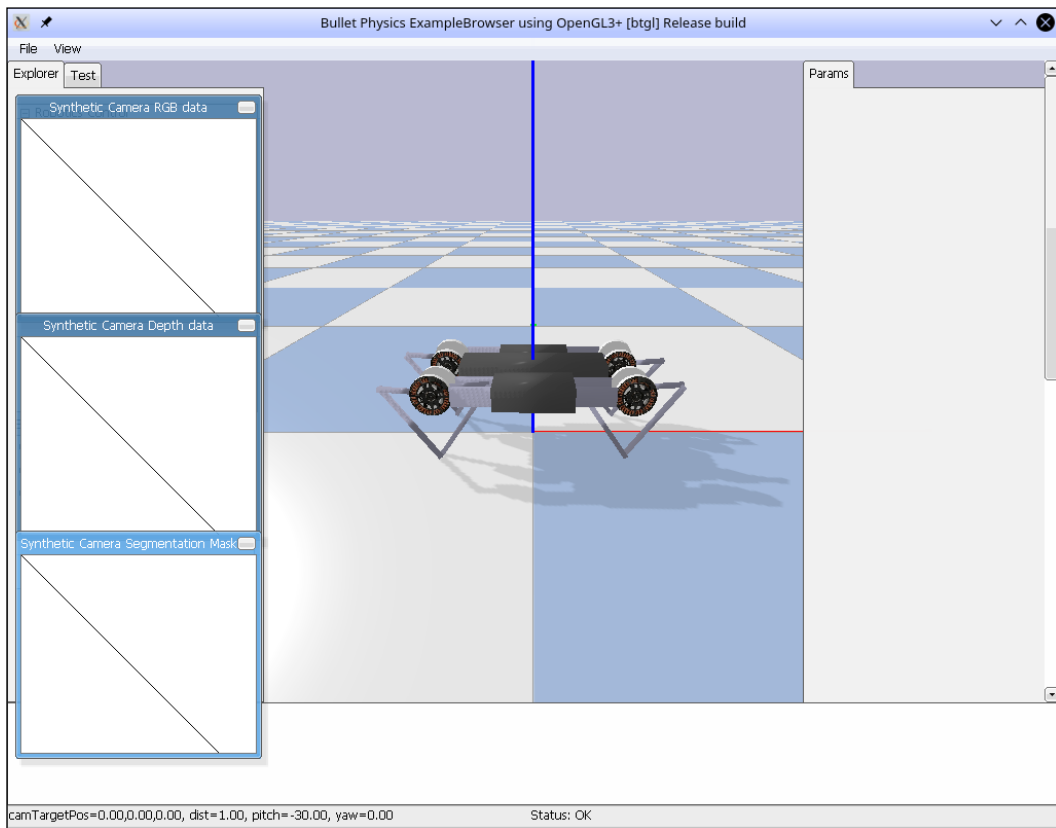
Figure 17.1: The Minitaur environment in the PyBullet GUI

This environment provides you with 28 numbers as the observation. They correspond to different physical parameters of the robot: velocity, position, and acceleration. (You can check the source code of `MinitaurBulletEnv-v0` for details.) The action space is eight numbers that define the parameters of the motors. There are two in every leg (one in every knee). The reward of this environment is the distance traveled by the robot minus the energy spent.

```
rl_book_samples/Chapter17$ ./01_check_env.py
[2019-11-15 15:02:14,305] Making new env: MinitaurBulletEnv-v0
pybullet build time: Nov 12 2019 14:02:55
...
Observation space: Box(28,) Action space: Box(8,)
<TimeLimit<MinitaurBulletEnv<MinitaurBulletEnv-v0>>>
[ 1.47892781e+00 1.47092442e+00 1.47486159e+00 1.46795948e+00
  1.48735227e+00 1.49067837e+00 1.48767487e+00 1.48856073e+00
  1.22760518e+00 1.23364264e+00 1.23980635e+00 1.23808274e+00
  1.23863620e+00 1.20957165e+00 1.22914063e+00 1.21966631e+00
  5.27463590e-01 5.87924378e-01 5.56949063e-01 6.10125678e-01
  4.58817873e-01 4.37388898e-01 4.57652322e-01 4.52128593e-01
 -3.00935339e-03 1.04264007e-03 -2.26649036e-04 9.99994903e-01]
Press any key to exit
```

The A2C method

The first method that we will apply to our walking robot problem is A2C, which we experimented with in part three of the book. This choice of method is quite obvious, as A2C is very easy to adapt to the continuous action domain. As a quick refresher, A2C's idea is to estimate the gradient of our policy as $\nabla J = \nabla_\theta \log \pi_\theta(a|s)\left(R - V_\theta(s)\right)$. The π_θ policy is supposed to provide the probability distribution of actions given the observed state. The quantity $V_\theta(s)$ is called a critic, equals to the value of the state, and is trained using the mean squared error (MSE) loss between the critic's return and the value estimated by the Bellman equation. To improve exploration, the entropy bonus $L_H = \pi_\theta(s) \log \pi_\theta(s)$ is usually added to the loss.

Obviously, the value head of the actor-critic will be unchanged for continuous actions. The only thing that is affected is the representation of the policy. In the discrete cases that you have seen, we had only one action with several mutually exclusive discrete values. For such a case, the obvious representation of the policy was the probability distribution over all actions.

In a continuous case, we usually have several actions, each of which can take a value from some range. With that in mind, the simplest policy representation will be just those values returned for every action. These values should not be confused with the value of the state, $V(s)$, which indicates how many rewards we can get from the state. To illustrate the difference, let's imagine a simple car steering case in which we can only turn the wheel. The action at every moment will be the wheel angle (action value), but the value of every state will be the potential discounted reward from the state, which is a totally different thing.

Returning to our action representation options, if you remember from *Chapter 11, Policy Gradients – an Alternative*, the representation of an action as a concrete value has different disadvantages, mostly related to the exploration of the environment. A much better choice will be something stochastic. The simplest alternative will be the network returning parameters of the Gaussian distribution. For *N* actions, this will be two vectors of size *N*. The first will be the mean values, μ, and the second vector will contain variances, σ^2. In that case, our policy will be represented as a random *N*-dimensional vector of uncorrelated, normally distributed random variables, and our network can make a selection about the mean and the variance of every variable.

By definition, the probability density function of the Gaussian distribution is

$$f(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

We could directly use this formula to get the probabilities, but to improve numerical stability, it is worth doing some math and simplifying the expression for $\log \pi_\theta(a|s)$.

The final result will be this: $\log \pi_\theta(a|s) = -\frac{(x-\mu)^2}{2\sigma^2} - \log \sqrt{2\pi\sigma^2}$. The entropy of the Gaussian distribution could be obtained using the differential entropy definition and will be $\ln \sqrt{2\pi e \sigma^2}$. Now we have everything we need to implement the A2C method. Let's do it.

Implementation

The complete source code is in `Chapter17/02_train_a2c.py`, `Chapter17/lib/model.py`, and `Chapter17/lib/common.py`. You will be familiar with most of the code, so the following includes only the parts that differ. Let's start with the model class defined in `Chapter17/lib/model.py`:

```
HID_SIZE = 128

class ModelA2C(nn.Module):
    def __init__(self, obs_size, act_size):
        super(ModelA2C, self).__init__()
```

```
        self.base = nn.Sequential(
            nn.Linear(obs_size, HID_SIZE),
            nn.ReLU(),
        )
        self.mu = nn.Sequential(
            nn.Linear(HID_SIZE, act_size),
            nn.Tanh(),
        )
        self.var = nn.Sequential(
            nn.Linear(HID_SIZE, act_size),
            nn.Softplus(),
        )
        self.value = nn.Linear(HID_SIZE, 1)
```

As you can see, our network has three heads, instead of the normal two for a discrete variant of A2C. The first two heads return the mean value and the variance of the actions, while the last is the critic head returning the value of the state. The mean value returned has an activation function of a hyperbolic tangent, which is the squashed output to the range of –1…1. The variance is transformed with the softplus activation function, which is $\log(1 + e^x)$ and has the shape of a smoothed rectified linear unit (ReLU) function. This activation helps to make our variance positive. The value head, as usual, has no activation function applied.

```
    def forward(self, x):
        base_out = self.base(x)
        return self.mu(base_out), self.var(base_out), \
               self.value(base_out)
```

The transformation is obvious: we apply the common layer first, and then we calculate individual heads.

```
class AgentA2C(ptan.agent.BaseAgent):
    def __init__(self, net, device="cpu"):
        self.net = net
        self.device = device

    def __call__(self, states, agent_states):
        states_v = ptan.agent.float32_preprocessor(states)
        states_v = states_v.to(self.device)

        mu_v, var_v, _ = self.net(states_v)
        mu = mu_v.data.cpu().numpy()
        sigma = torch.sqrt(var_v).data.cpu().numpy()
        actions = np.random.normal(mu, sigma)
        actions = np.clip(actions, -1, 1)
        return actions, agent_states
```

The next step is to implement the PTAN `Agent` class, which is used to convert the observation into actions. In the discrete case, we used the `ptan.agent.DQNAgent` and `ptan.agent.PolicyAgent` classes, but for our problem, we need to write our own, which is not complicated: you just need to write a class, derived from `ptan.agent.BaseAgent`, and override the `__call__` method, which needs to convert observations into actions.

In the preceding class, we get the mean and the variance from the network and sample the normal distribution using NumPy functions. To prevent the actions from going outside of the environment's –1…1 bounds, we use `np.clip`, which replaces all values less than –1 with –1, and values more than 1 with 1. The `agent_states` argument is not used, but it needs to be returned with the chosen actions, as our `BaseAgent` supports keeping the state of the agent. (It will become handy in the next section, when we will need to implement a random exploration using the **Ornstein-Uhlenbeck** (**OU**) process.)

With the model and the agent at hand, we can now go to the training process, defined in `Chapter17/02_train_a2c.py`. It consists of the training loop and two functions. The first is used to perform periodical tests of our model on the separate testing environment. During the testing, we don't need to do any exploration; we will just use the mean value returned by the model directly, without any random sampling. The testing function is as follows:

```python
def test_net(net, env, count=10, device="cpu"):
    rewards = 0.0
    steps = 0
    for _ in range(count):
        obs = env.reset()
        while True:
            obs_v = ptan.agent.float32_preprocessor([obs])
            obs_v = obs_v.to(device)
            mu_v = net(obs_v)[0]
            action = mu_v.squeeze(dim=0).data.cpu().numpy()
            action = np.clip(action, -1, 1)
            obs, reward, done, _ = env.step(action)
            rewards += reward
            steps += 1
            if done:
                break
    return rewards / count, steps / count
```

The second function defined in the training module implements the calculation of the logarithm of the taken actions given the policy. The formula for this was given earlier, and the function is a straightforward implementation of it.

The only tiny difference is in using the `torch.clamp()` function to prevent the division on zero when the returned variance is too small.

```
def calc_logprob(mu_v, var_v, actions_v):
    p1 = -((mu_v - actions_v) ** 2) / (2*var_v.clamp(min=1e-3))
    p2 = -torch.log(torch.sqrt(2 * math.pi * var_v))
    return p1 + p2
```

The training loop, as usual, creates the network and the agent, and then instantiates the two-step experience source and optimizer. The hyperparameters used are given as follows. They weren't tweaked much, so there is plenty of room for optimization.

```
ENV_ID = "MinitaurBulletEnv-v0"
GAMMA = 0.99
REWARD_STEPS = 2
BATCH_SIZE = 32
LEARNING_RATE = 5e-5
ENTROPY_BETA = 1e-4

TEST_ITERS = 1000
```

The code used to perform the optimization step on the collected batch is very similar to the A2C training that we implemented in *Chapter 12, The Actor-Critic Method*, and *Chapter 13, Asynchronous Advantage Actor-Critic*. The difference is only in using our `calc_logprob()` function and a different expression for the entropy bonus, which is shown next:

```
states_v, actions_v, vals_ref_v = \
    common.unpack_batch_a2c(
        batch, net, device=device,
        last_val_gamma=GAMMA ** REWARD_STEPS)
batch.clear()

optimizer.zero_grad()
mu_v, var_v, value_v = net(states_v)

loss_value_v = F.mse_loss(
    value_v.squeeze(-1), vals_ref_v)

adv_v = vals_ref_v.unsqueeze(dim=-1) - \
        value_v.detach()
log_prob_v = adv_v * calc_logprob(
    mu_v, var_v, actions_v)
loss_policy_v = -log_prob_v.mean()
ent_v = -(torch.log(2*math.pi*var_v) + 1)/2
```

```
                        entropy_loss_v = ENTROPY_BETA * ent_v.mean()

                        loss_v = loss_policy_v + entropy_loss_v + \
                                 loss_value_v
                        loss_v.backward()
                        optimizer.step()
```

Every `TEST_ITERS` frames, the model is tested, and in the case of the best reward obtained, the model weights are saved.

Results

In comparison to other methods that we will look at in this chapter, A2C shows the worst results, both in terms of the best reward and convergence speed. That's likely because of the single environment used to gather experience, which is a weak point of the policy gradient (PG) methods. So, you may want to check the effect of several (eight or more) parallel environments on A2C.

To start the training, we pass the `-n` argument with the run name, which will be used in TensorBoard and a new directory to save the models. The `--cuda` option enables GPU usage, but due to the small dimensionality of the input and the tiny network size, it gives only a marginal increase in speed. The sample output from the training is shown as follows:

```
Chapter17$ ./02_train_a2c.py -n test
pybullet build time: Nov 12 2019 14:02:55
ModelA2C(
  (base): Sequential(
    (0): Linear(in_features=28, out_features=128, bias=True)
    (1): ReLU()
  )
  (mu): Sequential(
    (0): Linear(in_features=128, out_features=8, bias=True)
    (1): Tanh()
  )
  (var): Sequential(
    (0): Linear(in_features=128, out_features=8, bias=True)
    (1): Softplus(beta=1, threshold=20)
  )
  (value): Linear(in_features=128, out_features=1, bias=True)
)
```

```
Test done is 20.32 sec, reward -0.786, steps 443
122: done 1 episodes, mean reward -0.473, speed 5.69 f/s
1123: done 2 episodes, mean reward -2.560, speed 27.54 f/s
1209: done 3 episodes, mean reward -1.838, speed 176.22 f/s
1388: done 4 episodes, mean reward -1.549, speed 137.63 f/s
```

After 6M frames, which took 18 hours of optimization, the training process reached the best score of 1.3 during the testing, which is not very impressive. The reward and episode steps during the training and testing are shown in the following charts:
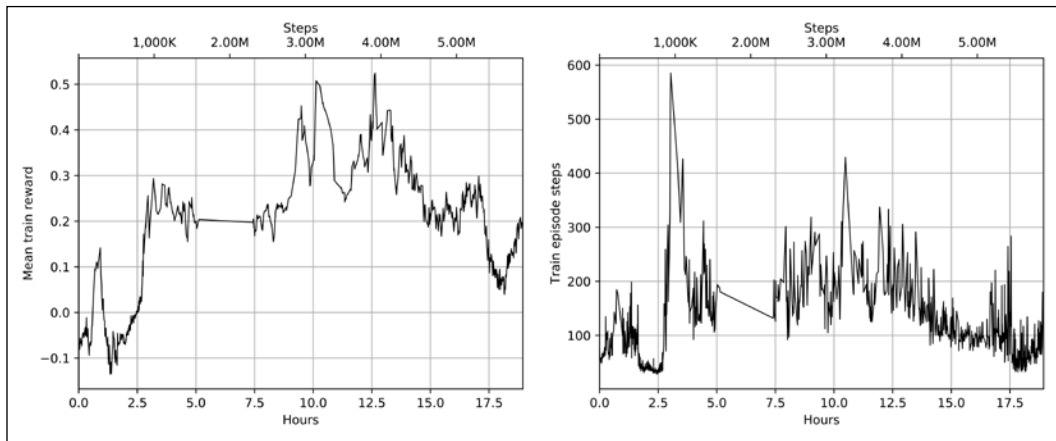
Figure 17.2: The reward (left) and steps (right) for training episodes
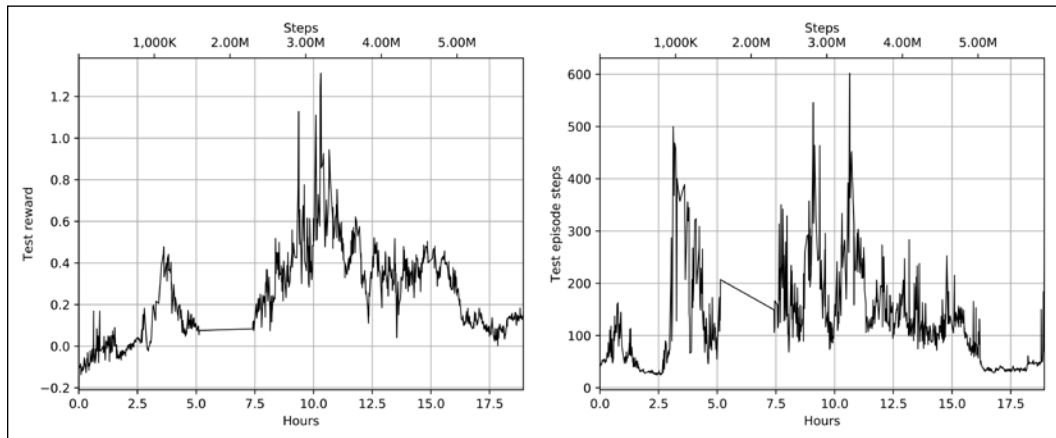
Figure 17.3: The reward (left) and steps (right) for test episodes

The *episode steps* chart (*Figure 17.2, right*) shows the average count of steps performed in the episode before the end. The time limit of the environment is 1,000 steps, so everything lower than 1,000 indicates that the episode was stopped due to environment checks. (For most of the environments, they are checking for self-damage, which stops the simulation.) The *Figure 17.3* charts show the mean reward and number of steps obtained during the testing.

Using models and recording videos

As you have seen before, the physical simulator can render the state of the environment, which makes it possible to see how our trained model behaves. To do that for our A2C models, there is a utility, `Chapter17/03_play_a2c.py`. Its logic is the same as in the `test_net()` function, so its code is not shown here. To start it, you need to pass the `-m` option with the model file and `-r` with a directory name, which will be created to save the video. To render the image, PyBullet requires OpenGL, so you need to use Xvfb to record the video on a headless server:

```
xvfb-run -s "-screen 0 640x480x24 +extension GLX" ./03_play_a2c.py -m
model.dat -r dest-dir
```

There is a script that does this in `Chapter17/adhoc/record_a2c.sh`. For example, the best model that I got from A2C training produced this:

```
Chapter17$ ./adhoc/record_a2c.sh res/a2c-t1-long/a2c-t1/
best_+1.188_203000.dat a2c-res/
pybullet build time: Nov 12 2019 14:02:55
In 738 steps we got 1.261 reward
```

In the directory specified, there will be a recorded movie with the agent's activity.

Deterministic policy gradients

The next method that we will take a look at is called deterministic policy gradients, which is an actor-critic method but has a very nice property of being off-policy. The following is my very relaxed interpretation of the strict proofs. If you are interested in understanding the core of this method deeply, you can always refer to the article by David Silver and others called *Deterministic Policy Gradient Algorithms*, published in 2014 (`http://proceedings.mlr.press/v32/silver14.pdf`), and the paper by Timothy P. Lillicrap and others called *Continuous Control with Deep Reinforcement Learning*, published in 2015 (`https://arxiv.org/abs/1509.02971`).

The simplest way to illustrate the method is through comparison with the already familiar A2C method. In this method, the actor estimates the stochastic policy, which returns the probability distribution over discrete actions or, as we have just covered in the previous section, the parameters of normal distribution. In both cases, our policy is *stochastic*, so, in other words, our action taken is sampled from this distribution.

Deterministic policy gradients also belong to the A2C family, but the policy is *deterministic*, which means that it directly provides us with the action to take from the state. This makes it possible to apply the chain rule to the Q-value, and by maximizing the *Q*, the policy will be improved as well. To understand this, let's look at how the actor and critic are connected in a continuous action domain.

Let's start with the actor, as it is the simpler of the two. What we want from it is the action to take for every given state. In a continuous action domain, every action is a number, so the actor network will take the state as an input and return N values, one for every action. This mapping will be deterministic, as the same network always returns the same output if the input is the same. (We're not going to use dropout or something similar; we're just going to use an ordinary feed-forward network.)

Now let's look at the critic. The role of the critic is to estimate the Q-value, which is a discounted reward of the action taken in some state. However, our action is a vector of numbers, so our critic network now accepts two inputs: the state and the action. The output from the critic will be the single number, which corresponds to the Q-value. This architecture is different from the DQN, when our action space was discrete and, for efficiency, we returned values for all actions in one pass. This mapping is also deterministic.

So, we have two functions: the actor, let's call it $\mu(s)$, which converts the state into the action, and the critic, which through the state and the action gives us the Q-value: $Q(s, a)$. We can substitute the actor function into the critic and get the expression with only one input parameter of our state: $Q(s, \mu(s))$. In the end, neural networks are just functions.

Now, the output of the critic gives us the approximation of the entity that we're interested in maximizing in the first place: the discounted total reward. This value depends not only on the input state, but also on the parameters of the θ_μ actor and the θ_Q critic networks. At every step of our optimization, we want to change the actor's weights to improve the total reward that we get. In mathematical terms, we want the gradient of our policy.

In his deterministic policy gradient theorem, David Silver proved that the stochastic policy gradient is equivalent to the deterministic policy gradient. In other words, to improve the policy, we just need to calculate the gradient of the $Q(s, \mu(s))$ function. By applying the chain rule, we get the gradient: $\nabla_a Q(s, a) \nabla_{\theta_\mu} \mu(s)$.

Note that, despite both the A2C and deep deterministic policy gradients (DDPG) methods belonging to the A2C family, the way that the critic is used is different. In A2C, we use the critic as a baseline for a reward from the experienced trajectories, so the critic is an optional piece (without it, we will get the REINFORCE method) and is used to improve the stability. This happens as the policy in A2C is stochastic, which builds a barrier in our backpropagation capabilities (we have no way of differentiating the random sampling step).

In DDPG, the critic is used in a different way. As our policy is deterministic, we can now calculate the gradients from Q, which is obtained from the critic network, which uses actions produced by the actor (check *Figure 17.4*), so the whole system is differentiable and could be optimized end to end with stochastic gradient descent (SGD). To update the critic network, we can use the Bellman equation to find the approximation of $Q(s, a)$ and minimize the MSE objective.

All this may look a bit cryptic, but behind it stands a quite simple idea: the critic is updated as we did in A2C, and the actor is updated in a way to maximize the critic's output. The beauty of this method is that it is off-policy, which means that we can now have a huge replay buffer and other tricks that we used in DQN training. Nice, right?

Exploration

The price we have to pay for all this goodness is that our policy is now deterministic, so we have to explore the environment somehow. We can do this by adding noise to the actions returned by the actor before we pass them to the environment. There are several options here. The simplest method is just to add the random noise to the $\mu(s) + \varepsilon \mathcal{N}$ actions. We will use this in the next method that we will consider in this chapter.

A fancier approach to the exploration is to use the previously mentioned stochastic model, which is very popular in the financial world and other domains dealing with stochastic processes (OU processes). This process models the velocity of a massive Brownian particle under the influence of friction and is defined by this stochastic differential equation: $\partial x_t = \theta(\mu - x_t)\partial t + \sigma \partial W$, where θ, μ, and σ are parameters of the process and W is the Wiener process. In a discrete-time case, the OU process could be written as $x_{t+1} = x_t + \theta(\mu - x_t) + \sigma N$. This equation expresses the next value generated by the process via the previous value of the noise, adding normal noise, $\mathcal{N}$. In our exploration, we will add the value of the OU process to the action returned by the actor.

Implementation

This example consists of three source files:

- `Chapter17/lib/model.py`: contains the model and the PTAN agent
- `Chapter17/lib/common.py`: has a function used to unpack the batch
- `Chapter17/04_train_ddpg.py`: has the startup code and the training loop

Here, I will show only the significant pieces of the code. The model consists of two separate networks for the actor and critic, and it follows the architecture from the *Continuous Control with Deep Reinforcement Learning* paper mentioned earlier. The actor is extremely simple and is feed-forward with two hidden layers. The input is an observation vector, whereas the output is a vector with N values, one for each action. The output actions are transformed with hyperbolic tangent nonlinearity to squeeze the values to the –1…1 range.

The critic is a bit unusual, as it includes two separate paths for the observation and the actions, and those paths are concatenated together to be transformed into the critic output of one number. The following is a diagram with the structures of both networks:
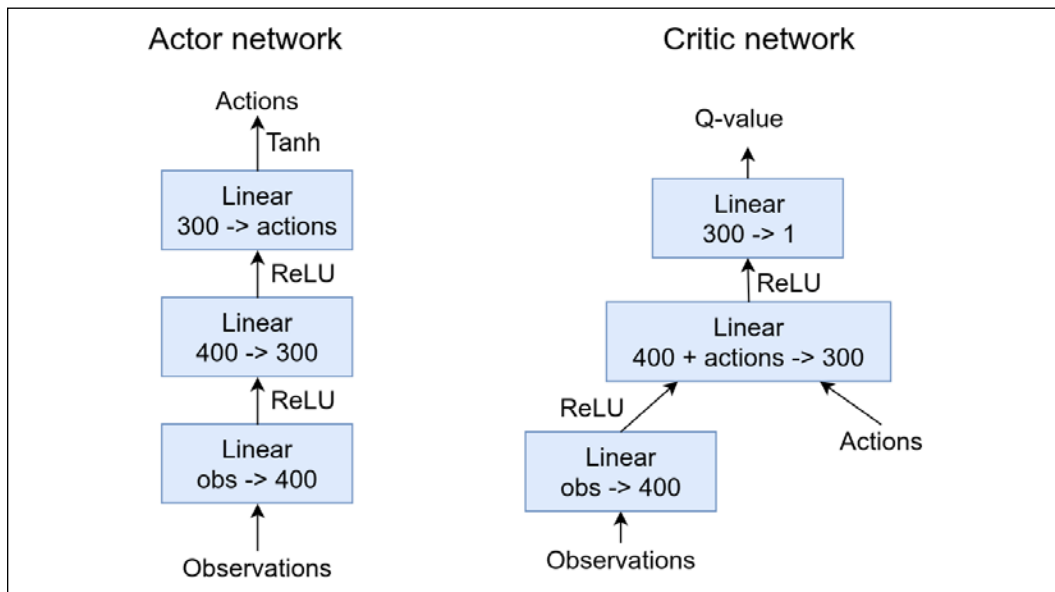
Figure 17.4: The DDPG actor and critic networks

The code for both classes is simple and straightforward:

```
class DDPGActor(nn.Module):
    def __init__(self, obs_size, act_size):
        super(DDPGActor, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(obs_size, 400),
            nn.ReLU(),
            nn.Linear(400, 300),
            nn.ReLU(),
            nn.Linear(300, act_size),
            nn.Tanh()
        )

    def forward(self, x):
        return self.net(x)


class DDPGCritic(nn.Module):
    def __init__(self, obs_size, act_size):
        super(DDPGCritic, self).__init__()

        self.obs_net = nn.Sequential(
            nn.Linear(obs_size, 400),
            nn.ReLU(),
        )

        self.out_net = nn.Sequential(
            nn.Linear(400 + act_size, 300),
            nn.ReLU(),
            nn.Linear(300, 1)
        )

    def forward(self, x, a):
        obs = self.obs_net(x)
        return self.out_net(torch.cat([obs, a], dim=1))
```

The `forward()` function of the critic first transforms the observations with its small network and then concatenates the output and given actions to transform them into one single value of *Q*. To use the actor network with the PTAN experience source, we need to define the agent class that has to transform the observations into actions. This class is the most convenient place to put our OU exploration process, but to do this properly, we should use the functionality of the PTAN agents that we haven't used so far: *optional statefulness*.

The idea is simple: our agent transforms the observations into actions. But what if it needs to remember something between the observations? All our examples have been stateless so far, but sometimes this is not enough. The issue with OU is that we have to track the OU values between the observations.

Another very useful use case for stateful agents is a partially observable Markov decision process (POMDP), which was briefly mentioned in *Chapter 16*, *Web Navigation*. The POMDP is a Markov decision process when the state observed by the agent doesn't comply with the Markov property and doesn't include the full information to distinguish one state from another. In that case, our agent needs to track the state along the trajectory to be able to take the action.

So, the code for the agent that implements the OU for exploration is as follows:

```
class AgentDDPG(ptan.agent.BaseAgent):
    def __init__(self, net, device="cpu", ou_enabled=True,
                 ou_mu=0.0, ou_teta=0.15, ou_sigma=0.2,
                 ou_epsilon=1.0):
        self.net = net
        self.device = device
        self.ou_enabled = ou_enabled
        self.ou_mu = ou_mu
        self.ou_teta = ou_teta
        self.ou_sigma = ou_sigma
        self.ou_epsilon = ou_epsilon
```

The constructor accepts a lot of parameters, most of which are the default values of OU taken from the previously mentioned *Continuous Control with Deep Reinforcement Learning* paper.

```
    def initial_state(self):
        return None
```

This method is derived from the `BaseAgent` class and has to return the initial state of the agent when a new episode is started. As our initial state has to have the same dimension as the actions (we want to have individual exploration trajectories for every action of the environment), we postpone the initialization of the state until the `__call__` method, as follows:

```
    def __call__(self, states, agent_states):
        states_v = ptan.agent.float32_preprocessor(states)
        states_v = states_v.to(self.device)
        mu_v = self.net(states_v)
        actions = mu_v.data.cpu().numpy()
```

This method is the core of the agent and the purpose of it is to convert the observed state and internal agent state into the action. As the first step, we convert the observations into the appropriate form and ask the actor network to convert them into deterministic actions. The rest of the method is for adding the exploration noise by applying the OU process.

```
if self.ou_enabled and self.ou_epsilon > 0:
    new_a_states = []
    for a_state, action in zip(agent_states, actions):
        if a_state is None:
            a_state = np.zeros(
                shape=action.shape, dtype=np.float32)
        a_state += self.ou_teta * (self.ou_mu - a_state)
        a_state += self.ou_sigma * np.random.normal(
            size=action.shape)
```

In this loop, we iterate over the batch of observations and the list of the agent states from the previous call, and we update the OU process value, which is a straightforward implementation of the preceding formula.

```
        action += self.ou_epsilon * a_state
        new_a_states.append(a_state)
```

To finalize the loop, we add the noise from the OU process to our actions and save the noise value for the next step.

```
else:
    new_a_states = agent_states

actions = np.clip(actions, -1, 1)
return actions, new_a_states
```

Finally, we clip the actions to enforce them to fall into the –1...1 range, otherwise PyBullet will throw an exception.

The final piece of the DDPG implementation is the training loop in the `Chapter17/04_train_ddpg.py` file. To improve the stability, we use the replay buffer with 100,000 transitions and target networks for both the actor and the critic. We discussed both in *Chapter 6, Deep Q-Networks*.

```
act_net = model.DDPGActor(
    env.observation_space.shape[0],
    env.action_space.shape[0]).to(device)
crt_net = model.DDPGCritic(
    env.observation_space.shape[0],
    env.action_space.shape[0]).to(device)
```

```
print(act_net)
print(crt_net)
tgt_act_net = ptan.agent.TargetNet(act_net)
tgt_crt_net = ptan.agent.TargetNet(crt_net)

writer = SummaryWriter(comment="-ddpg_" + args.name)
agent = model.AgentDDPG(act_net, device=device)
exp_source = ptan.experience.ExperienceSourceFirstLast(
    env, agent, gamma=GAMMA, steps_count=1)
buffer = ptan.experience.ExperienceReplayBuffer(
    exp_source, buffer_size=REPLAY_SIZE)
act_opt = optim.Adam(act_net.parameters(), lr=LEARNING_RATE)
crt_opt = optim.Adam(crt_net.parameters(), lr=LEARNING_RATE)
```

We also use two different optimizers to simplify the way that we handle gradients for the actor and critic training steps. The most interesting code is inside the training loop. On every iteration, we store the experience into the replay buffer and sample the training batch:

```
batch = buffer.sample(BATCH_SIZE)
states_v, actions_v, rewards_v, \
dones_mask, last_states_v = \
    common.unpack_batch_ddqn(batch, device)
```

Then, two separate training steps are performed. To train the critic, we need to calculate the target Q-value using the one-step Bellman equation, with the target critic network as the approximation of the next state.

```
# train critic
crt_opt.zero_grad()
q_v = crt_net(states_v, actions_v)
last_act_v = tgt_act_net.target_model(
    last_states_v)
q_last_v = tgt_crt_net.target_model(
    last_states_v, last_act_v)
q_last_v[dones_mask] = 0.0
q_ref_v = rewards_v.unsqueeze(dim=-1) + \
          q_last_v * GAMMA
```

When we have got the reference, we can calculate the MSE loss and ask the critic's optimizer to tweak the critic's weights. The whole process is similar to the training for the DQN, so nothing is really new here.

```
critic_loss_v = F.mse_loss(q_v, q_ref_v.detach())
critic_loss_v.backward()
crt_opt.step()
```

```
            tb_tracker.track("loss_critic",
                               critic_loss_v, frame_idx)
            tb_tracker.track("critic_ref",
                               q_ref_v.mean(), frame_idx)
```

On the actor's training step, we need to update the actor's weights in a direction that will increase the critic's output. As both the actor and critic are represented as differentiable functions, what we need to do is just pass the actor's output to the critic and then minimize the negated value returned by the critic.

```
            # train actor
            act_opt.zero_grad()
            cur_actions_v = act_net(states_v)
            actor_loss_v = -crt_net(states_v, cur_actions_v)
            actor_loss_v = actor_loss_v.mean()
```

This negated output of the critic could be used as a loss to backpropagate it to the critic network and, finally, the actor. We don't want to touch the critic's weights, so it's important to ask only the actor's optimizer to do the optimization step. The weights of the critic will still keep the gradients from this call, but they will be discarded on the next optimization step.

```
            actor_loss_v.backward()
            act_opt.step()
            tb_tracker.track("loss_actor",
                               actor_loss_v, frame_idx)
```

As the last step of the training loop, we perform the update of the target networks in an unusual way. Previously, we synced the weights from the optimized network into the target every *n* steps. In continuous action problems, such syncing works worse than so-called *soft sync*. The soft sync is carried out on every step, but only a small ratio of the optimized network's weights are added to the target network. This makes a smooth and slow transition from the old weight to the new ones.

```
            tgt_act_net.alpha_sync(alpha=1 - 1e-3)
            tgt_crt_net.alpha_sync(alpha=1 - 1e-3)
```

Results

The code can be started in the same way as the A2C example: you need to pass the run name and optional `--cuda` flag. My experiments have shown ~30% speed increase from a GPU, so if you're in a hurry, using CUDA may be a good idea, but the increase is not that dramatic, as you have seen in the case of Atari games.

After 5M observations, which took about a day, the DDPG algorithm was able to reach the mean reward of 6.5 on 10 test episodes, which is an improvement over the A2C result. The training dynamics are as follows:
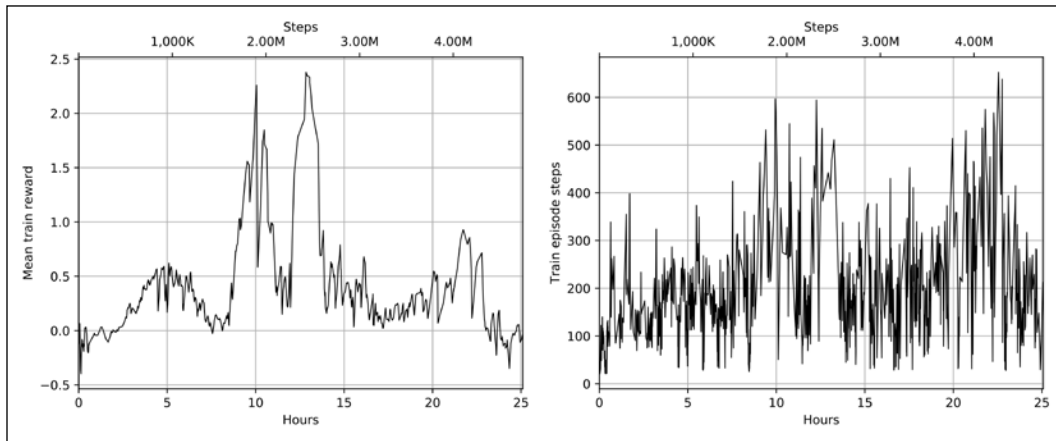
Figure 17.5: The DDPG training reward (left) and episode steps (right)
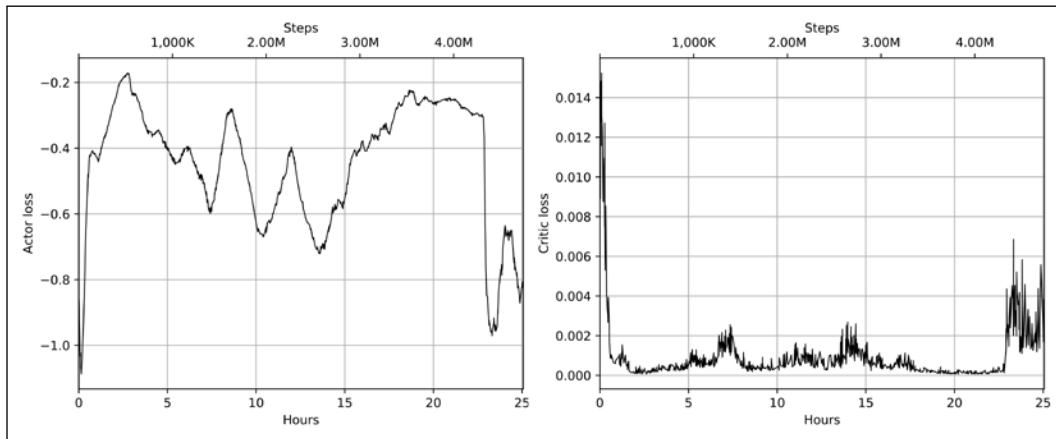
Figure 17.6: The actor loss (left) and critic loss (right) during the DDPG training

The `episode_steps` value shows the mean length of the episodes that we used for training. The critic loss is an MSE loss and should be low, but the actor loss, as you will remember, is the negated critic's output, so the smaller it is, the better reward that the actor can (potentially) achieve.

From the preceding charts, the training is not very stable and is noisy.
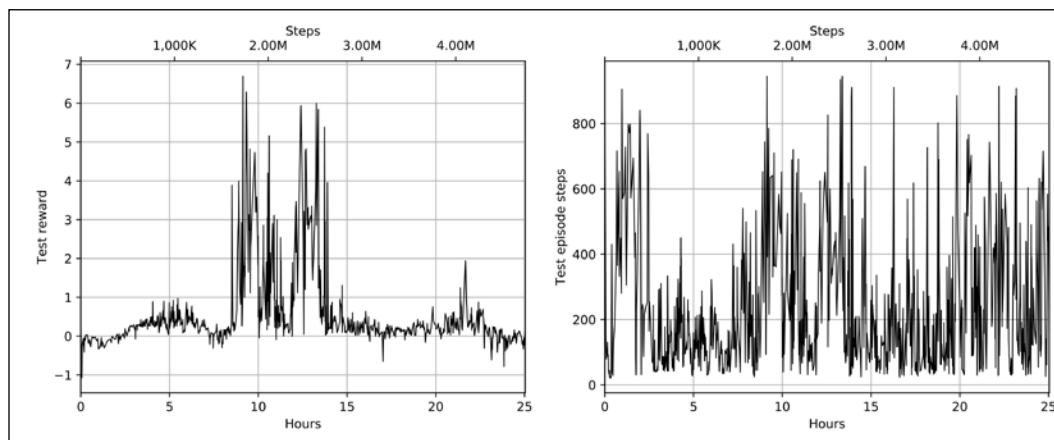
Figure 17.7: The test reward (left) and test steps (right)

The preceding charts are also quite noisy.

Recording videos

To check the trained agent in action, we can record a video in the same way as we recorded it for the A2C agent. For DDPG, there is a separate tool, `Chapter17/05_play_ddpg.py`, that is almost the same as for the A2C method, but just uses different classes for the actor. The result from my model is as follows:

```
rl_book_samples/Chapter17$ adhoc/record_ddpg.sh saves/ddpg-t5-simpler-
critic/best_+3.933_2484000.dat res/play-ddpg
pybullet build time: Nov 12 2019 14:02:55
In 1000 steps we got 5.346 reward
```

Distributional policy gradients

As the last method of this chapter, we will take a look at the very recent paper by Gabriel Barth-Maron, Matthew W. Hoffman, and others, called *Distributed Distributional Deterministic Policy Gradients*, published in 2018 (https://arxiv.org/abs/1804.08617).

The full name of the method is **distributed distributional deep deterministic policy gradients** or **D4PG** for short. The authors proposed several improvements to the DDPG method to improve stability, convergence, and sample efficiency.

First of all, they adapted the distributional representation of the Q-value proposed in the paper by Marc G. Bellemare and others called *A Distributional Perspective on Reinforcement Learning*, published in 2017 (`https://arxiv.org/abs/1707.06887`). We discussed this approach in *Chapter 8, DQN Extensions*, when we talked about DQN improvements, so refer to it or to the original Bellemare paper for details. The core idea is to replace a single Q-value from the critic with a probability distribution. The Bellman equation is replaced with the Bellman operator, which transforms this distributional representation in a similar way.

The second improvement was the usage of the n-step Bellman equation, unrolled to speed up the convergence. We also discussed this in detail in *Chapter 8*.

Another improvement versus the original DDPG method was the usage of the prioritized replay buffer instead of the uniformly sampled buffer. So, strictly speaking, the authors took relevant improvements from the paper by Matteo Hassel and others, called *Rainbow: Combining Improvements in Deep Reinforcement Learning*, which was published in 2017 (`https://arxiv.org/abs/1710.02298`), and adapted them to the DDPG method. The result was impressive: this combination showed state-of-the-art results on the set of continuous control problems. Let's try to reimplement the method and check it ourselves.

Architecture

The most notable change is the critic's output. Instead of returning the single Q-value for the given state and the action, it now returns `N_ATOMS` values, corresponding to the probabilities of values from the predefined range. In my code, I used `N_ATOMS=51` and the distribution range of `Vmin=-10` and `Vmax=10`, so the critic returned 51 numbers, representing the probabilities of the discounted reward falling into bins with bounds in [–10, –9.6, –9.2, …, 9.6, 10].

Another difference between D4PG and DDPG is the exploration. DDPG used the OU process for exploration, but according to the D4PG authors, they tried both OU and adding simple random noise to the actions, and the result was the same. So, they used a simpler approach for exploration in the paper.

The last significant difference in the code is related to the training, as D4PG uses cross-entropy loss to calculate the difference between two probability distributions (returned by the critic and obtained as a result of the Bellman operator). To make both distributions aligned to the same supporting atoms, distribution projection is used in the same way as in the original paper by Bellemare.

Implementation

The complete source code is in `Chapter17/06_train_d4pg.py`, `Chapter17/lib/model.py`, and `Chapter17/lib/common.py`. As before, we start with the model class. The actor class has exactly the same architecture, so during the training class, `DDPGActor` is used. The critic has the same size and count of hidden layers; however, the output is not a single number, but `N_ATOMS`.

```python
class D4PGCritic(nn.Module):
    def __init__(self, obs_size, act_size,
                 n_atoms, v_min, v_max):
        super(D4PGCritic, self).__init__()

        self.obs_net = nn.Sequential(
            nn.Linear(obs_size, 400),
            nn.ReLU(),
        )

        self.out_net = nn.Sequential(
            nn.Linear(400 + act_size, 300),
            nn.ReLU(),
            nn.Linear(300, n_atoms)
        )

        delta = (v_max - v_min) / (n_atoms - 1)
        self.register_buffer("supports", torch.arange(
            v_min, v_max + delta, delta))
```

We also create a helper PyTorch buffer with reward supports, which will be used to get from the probability distribution to the single mean Q-value.

```python
    def forward(self, x, a):
        obs = self.obs_net(x)
        return self.out_net(torch.cat([obs, a], dim=1))

    def distr_to_q(self, distr):
        weights = F.softmax(distr, dim=1) * self.supports
        res = weights.sum(dim=1)
        return res.unsqueeze(dim=-1)
```

As you can see, softmax is not part of the network, as we're going to use the more stable `log_softmax()` function during the training. Due to this, `softmax()` needs to be applied when we want to get actual probabilities. The agent class is much simpler for D4PG and has no state to track.

```python
class AgentD4PG(ptan.agent.BaseAgent):
```

```
        def __init__(self, net, device="cpu", epsilon=0.3):
            self.net = net
            self.device = device
            self.epsilon = epsilon

        def __call__(self, states, agent_states):
            states_v = ptan.agent.float32_preprocessor(states)
            states_v = states_v.to(self.device)
            mu_v = self.net(states_v)
            actions = mu_v.data.cpu().numpy()
            actions += self.epsilon * np.random.normal(
                size=actions.shape)
            actions = np.clip(actions, -1, 1)
            return actions, agent_states
```

For every state to be converted to actions, the agent applies the actor network and adds Gaussian noise to the actions, scaled by the epsilon value. In the training code, we have the hyperparameters shown as follows. I used a smaller replay buffer of 100,000, and it worked fine. (In the D4PG paper, the authors used 1M transitions in the buffer.) The buffer is prepopulated with 10,000 samples from the environment, and then the training starts.

```
ENV_ID = "MinitaurBulletEnv-v0"
GAMMA = 0.99
BATCH_SIZE = 64
LEARNING_RATE = 1e-4
REPLAY_SIZE = 100000
REPLAY_INITIAL = 10000
REWARD_STEPS = 5

TEST_ITERS = 1000

Vmax = 10
Vmin = -10
N_ATOMS = 51
DELTA_Z = (Vmax - Vmin) / (N_ATOMS - 1)
```

For every training loop, we perform the same two steps as before: we train the critic and the actor. The difference is in the way that the loss for the critic is calculated.

```
            batch = buffer.sample(BATCH_SIZE)
            states_v, actions_v, rewards_v, \
            dones_mask, last_states_v = \
                common.unpack_batch_ddqn(batch, device)

            # train critic
```

```
crt_opt.zero_grad()
crt_distr_v = crt_net(states_v, actions_v)
last_act_v = tgt_act_net.target_model(
    last_states_v)
last_distr_v = F.softmax(
    tgt_crt_net.target_model(
        last_states_v, last_act_v), dim=1)
```

As the first step in the critic's training, we ask it to return the probability distribution for the states and actions taken. This probability distribution will be used as an input in the cross-entropy loss calculation. To get the target probability distribution, we need to calculate the distribution from the last states in the batch and then perform the Bellman projection of the distribution.

```
proj_distr_v = distr_projection(
    last_distr_v, rewards_v, dones_mask,
    gamma=GAMMA**REWARD_STEPS, device=device)
```

This projection function is a bit complicated and will be explained after the training loop code. For now, it calculates the transformation of the `last_states` probability distribution, which is shifted according to the immediate reward and scaled to respect the discount factor. The result is the target probability distribution that we want our network to return. As there are no general cross-entropy loss functions in PyTorch, we calculate it manually by multiplying the logarithm of the input probability by the target probabilities.

```
prob_dist_v = -F.log_softmax(
    crt_distr_v, dim=1) * proj_distr_v
critic_loss_v = prob_dist_v.sum(dim=1).mean()
critic_loss_v.backward()
crt_opt.step()
```

The actor's training is much simpler, and the only difference from the DDPG method is the use of the `distr_to_q()` function to convert from the probability distribution to the single mean Q-value using support atoms.

```
# train actor
act_opt.zero_grad()
cur_actions_v = act_net(states_v)
crt_distr_v = crt_net(states_v, cur_actions_v)
actor_loss_v = -crt_net.distr_to_q(crt_distr_v)
actor_loss_v = actor_loss_v.mean()
actor_loss_v.backward()
act_opt.step()
tb_tracker.track("loss_actor", actor_loss_v,
                frame_idx)
```

Now comes the most complicated piece of code in D4PG implementation: the projection of the probability using the Bellman operator. It was already explained in *Chapter 8, DQN Extensions*, but the function is tricky, so let's do it again. The overall goal of the function is to calculate the result of the Bellman operator and project the resulting probability distribution to the same support atoms as the original distribution. The Bellman operator has the form of $Z(x, a) \overset{D}{=} R(x, a) + \gamma Z(x', a')$ and it is supposed to transform the probability distribution.

```
def distr_projection(next_distr_v, rewards_v, dones_mask_t,
                     gamma, device="cpu"):
    next_distr = next_distr_v.data.cpu().numpy()
    rewards = rewards_v.data.cpu().numpy()
    dones_mask = dones_mask_t.cpu().numpy().astype(np.bool)
    batch_size = len(rewards)
    proj_distr = np.zeros((batch_size, N_ATOMS), dtype=np.float32)
```

In the beginning, we convert the provided tensors to NumPy arrays and create an empty array for the resulting projected distribution.

```
    for atom in range(N_ATOMS):
        tz_j = np.minimum(Vmax, np.maximum(
            Vmin, rewards + (Vmin + atom * DELTA_Z) * gamma))
```

In the loop, we iterate over our atoms and as the first step, calculate the place that our atom will be projected to by the Bellman operator, taking into account the value range `Vmin … Vmax`.

```
        b_j = (tz_j - Vmin) / DELTA_Z
```

The preceding line calculates the index of the atom that this projected value belongs to. Of course, the value may fall between the atoms, so in that case, we project the value proportionally to both atoms.

```
        l = np.floor(b_j).astype(np.int64)
        u = np.ceil(b_j).astype(np.int64)
        eq_mask = u == l
        proj_distr[eq_mask, l[eq_mask]] += \
            next_distr[eq_mask, atom]
```

The preceding code handles the rare case when the project value lands exactly on the atom. In that case, we just add the value to the atom. Of course, we work with a batch, so some of the samples might comply to this case, but some might not. That's why we need to calculate the mask and filter with it.

```
        ne_mask = u != l
        proj_distr[ne_mask, l[ne_mask]] += \
```

```
                next_distr[ne_mask, atom] * (u - b_j)[ne_mask]
        proj_distr[ne_mask, u[ne_mask]] += \
                next_distr[ne_mask, atom] * (b_j - l)[ne_mask]
```

As the final step of the loop, we need to process the case when the projected value is somewhere between two atoms. We calculate the proportion and distribute the projected value among two atoms.

```
if dones_mask.any():
    proj_distr[dones_mask] = 0.0
    tz_j = np.minimum(Vmax, np.maximum(
        Vmin, rewards[dones_mask]))
    b_j = (tz_j - Vmin) / DELTA_Z
```

In this branch, we handle the situation when the episode is over and our projected distribution will contain only one stripe corresponding to the atom of the reward that we have obtained. Here we take the same actions as before, but our source distribution is just reward.

```
l = np.floor(b_j).astype(np.int64)
u = np.ceil(b_j).astype(np.int64)
eq_mask = u == l
eq_dones = dones_mask.copy()
eq_dones[dones_mask] = eq_mask
if eq_dones.any():
    proj_distr[eq_dones, l[eq_mask]] = 1.0
ne_mask = u != l
ne_dones = dones_mask.copy()
ne_dones[dones_mask] = ne_mask
if ne_dones.any():
    proj_distr[ne_dones, l[ne_mask]] = (u - b_j)[ne_mask]
    proj_distr[ne_dones, u[ne_mask]] = (b_j - l)[ne_mask]
```

At the end of the function, we pack the distribution into the PyTorch tensor and return it:

```
return torch.FloatTensor(proj_distr).to(device)
```

Results

The D4PG method showed the best results in both convergence speed and the reward obtained. Following seven hours of training, after about 1M observations, it was able to reach the mean test reward of 7.2. The training process was much more stable than in other methods in this chapter, but the outcomes from different runs varied significantly. For example, the same code in experiments I did for the first edition of the book was able to reach a reward of 12.9 after 2M observations.

What follows are the same set of charts with the training dynamics as for the previous methods.
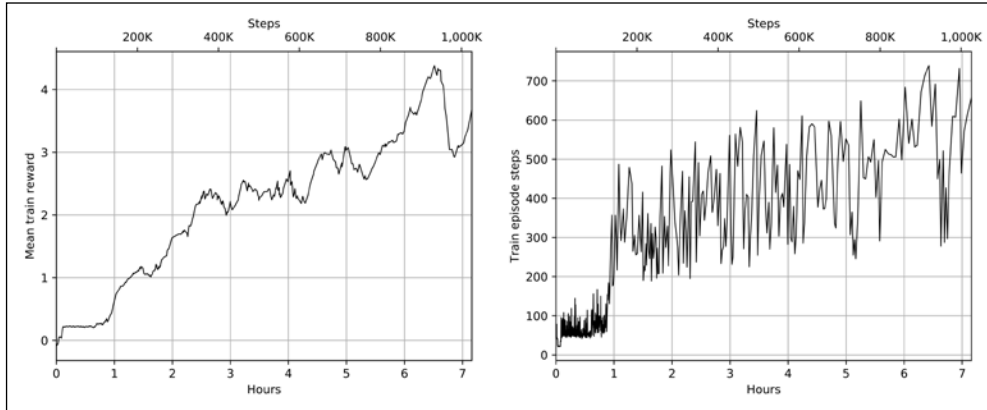
Figure 17.8: The D4PG training episode rewards (left) and steps (right)
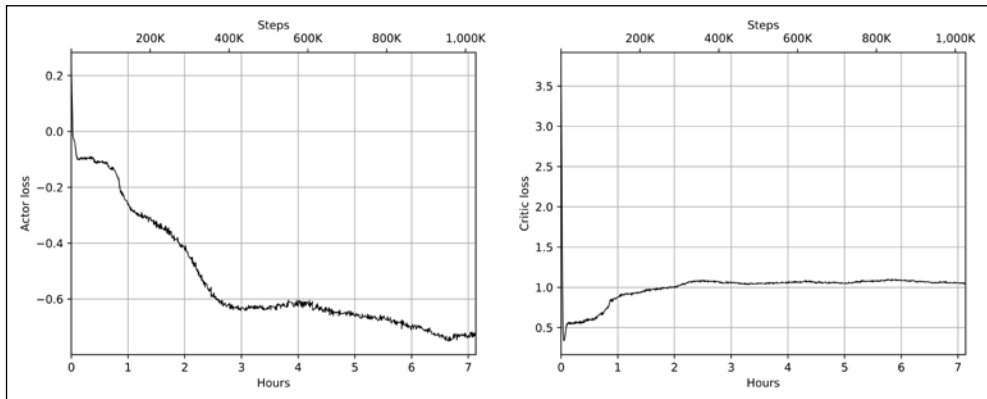
Figure 17.9: The D4PG actor (left) and critic (right) losses
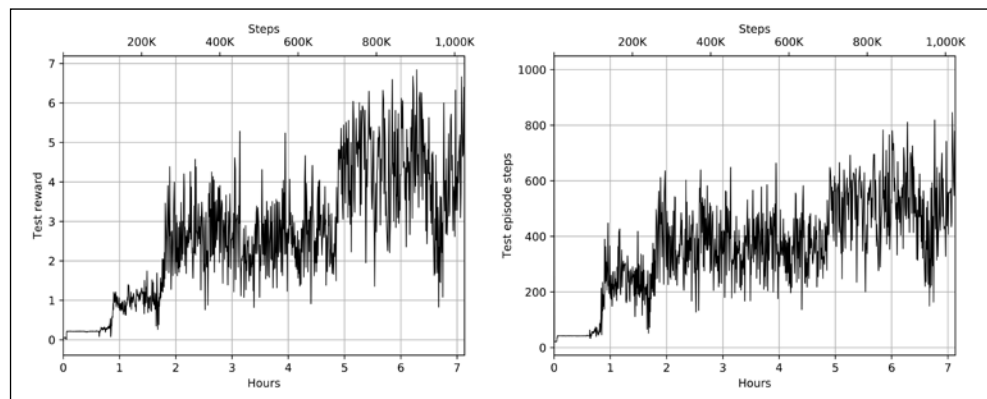
Figure 17.10: The test reward (left) and number of steps (right)

Video recordings

To record the video of the model activity, you can use the same `Chapter17/05_play_ddpg.py` utility, as the actor's architecture is exactly the same. The best video I got from the trained model with the 12.9 score can be found here: `https://youtu.be/BMt40odLfyk`.

Things to try

Here is a list of things you can do to improve your understanding of the topic:

1. In the D4PG code, I used a simple replay buffer, which was enough to get good improvement over DDPG. You can try to switch the example to the prioritized replay buffer in the same way as we did in *Chapter 8*, *DQN Extensions*, and check the effect.

2. There are lots of interesting and challenging environments around. For example, you can start with other PyBullet environments, but there is also the **DeepMind Control Suite** (*Tassa, Yuval*, et al., *DeepMind Control Suite*, arXiv abs/1801.00690 (2018)), MuJoCo-based environments in Gym, and many others.

3. You can request the trial license of MuJoCo and compare its stability, performance, and resulting policy with PyBullet.

4. You can play with the very challenging *Learning to Run* competition from NIPS-2017 (which also took place in 2018 and 2019 with more challenging problems), where you are given a simulator of the human body and your agent needs to figure out how to move it around.

Summary

In this chapter, we quickly skimmed through the very interesting domain of continuous control using RL methods, and we checked three different algorithms on one problem of a four-legged robot. In our training, we used an emulator, but there are real models of this robot made by the Ghost Robotics company. (You can check out the cool video on YouTube: `https://youtu.be/bnKOeMoibLg`.) We applied three training methods to this environment: A2C, DDPG, and D4PG (which showed the best results).

In the next chapter, we will continue exploring the continuous action domain and check a different set of improvements: *trust region extension*.

18

RL in Robotics

This chapter is a bit unusual in comparison to the other chapters in this book for the following reasons:

- It took me almost four months to gather all the materials, do the experiments, write the examples, and so on

- This is the only chapter in which we will try to step beyond emulated environments into the physical world

- In this chapter, we will build a small robot from accessible and cheap components to be controlled using reinforcement learning (RL) methods

This topic is an amazing and fascinating field for many reasons that couldn't be covered in a whole book, much less in a short chapter. So, this chapter doesn't pretend to offer anywhere close to complete coverage of the robotics field. It is just a short introduction that shows what can be done with commodity components, and outlines future directions for your own experiments and research. In addition, I have to admit that I'm not an expert in robotics and have never worked in the field professionally, so feedback is welcome.

Robots and robotics

I'm sure you know what the word "robot" means and have seen them both in real life and in science fiction movies. Putting aside fictitious ones, there are many robots in industry (check "Tesla assembly line" on YouTube), the military (if you haven't seen the Boston Dynamics videos, you should stop reading and check them out), agriculture, medicine, and our homes. Automatic vacuum cleaners, modern coffee machines, 3D printers, and many other applications are examples of specialized mechanisms with complicated logic that are driven by some kind of software. At a high level, all those robots have common features, which we're going to discuss. Of course, this kind of classification is not perfect. As is often the case, there are lots of outliers that might fulfill the given criteria, but could still hardly be considered as robots.

Firstly, robots are connected to the world around them with some kind of sensors or other communication channels. Without this connection, a robot is blind and not very useful, as it can't adequately react to the events outside of it. There is a wide variety of different kinds of sensors, from the very simple to the extremely complicated and expensive; the following is just a small subset with examples:

- *A simple push button with two states, on and off*: This is a very popular solution for detecting physical contact with objects in simple cases. For example, a 3D printer normally has so-called endstops, which are just buttons triggered when the moving parts reach some boundary limits. When this happens, the internal software (also called *firmware*) reacts to this event, for instance, by stopping the motors.

- *Range sensors*: These measure the distance to the objects in front of the sensor by using sound waves or laser beams. For example, robotic vacuum cleaners normally have a "cliff detector" to prevent them from falling down staircases. This is just a laser range sensor that measures the distance to the floor underneath the robot. If this distance suddenly becomes large, the robot knows that it's dangerous to move forward and reacts appropriately.

- *Light detection and ranging (LiDAR) sensors*: These are more complicated and expensive versions of range sensors, with the difference being that the sensors rotate and thus constantly scan objects on the horizontal plane. The output from the sensors is the stream of points (the so-called "point cloud") showing the range to obstacles around the robot. LiDARs are very popular in self-driving cars, as they provide a reliable stream of information about obstacles, but they are quite expensive due to complicated mechanics and optics.

- *Cameras*: Such sensors stream video in the same way as any modern smartphone does, but the stream normally has to be processed by the robot's software to detect objects, like corners or cats. Video processing is quite heavy in terms of computations, so specialized hardware, like embedded graphics processing units (GPUs) or other neural network (NN) accelerators, has to be used. The good thing is that cameras can be quite cheap and provide a rich stream of information about the outside world.

- *Physical sensors*: This category includes accelerometers, digital gyroscopes, magnetometers, pressure sensors, thermometers, and so on. They measure the physical parameters of the world around the robot and usually are quite compact, cheap, and easy to use. All the details and complications are normally hidden inside the small chip, providing measures in digital form using some kind of communication protocol. This kind of sensor is very popular among hobbyists and can be found in a wide range of toys, consumer electronics, and DIY projects. For example, a drone can measure altitude using a pressure sensor, determine horizontal orientation with a magnetometer, and localize itself using a GPS receiver.

The list could be continued, but I think you've got the idea!

The second common property of robots is that they normally have a way to influence the outside world. It might be something obvious like turning on a motor or blinking a light, but it also could be something really sophisticated like rotating an antenna according to the tracked position and sending a radio signal to Earth over a distance of more than 20B kilometers, as NASA's Voyager 1 routinely does. In that group, we have a large variety of ways that a robot can influence the outside world. A small set of examples follows:

- Motors and actuators of various kinds, including very cheap small servo motors, stepper motors used in 3D printers, and the very complicated and precise actuators used in industrial robots.

- Producing sound and light, like flashing an LED or turning on a buzzer. These are normally used to attract a human's attention to some condition. As this is one of the simplest external devices to work with, blinking an LED is commonly used as a "Hello, World!" application in hobbyist electronics projects.

- Heaters, water pumps, water sprinklers, or printers.

The third group of features involves the presence of some kind of software that connects sensors and actuators together to solve a practical task at hand. The idea behind robots is to free humans up from doing those tasks themselves. Robots might be used for something trivial, like pulling up blinds according to a schedule and timer, or extremely complicated, like NASA's Mars Curiosity Rover. According to a very nice book by Rob Manning and William L. Simon, *Mars Rover Curiosity*, the Rover's actions are not controlled fully from Earth due to signal transmission latency; rather, every day, a very sophisticated semi-automated program is created at the mission control center and uploaded to the robot. Then, during the execution, the robot has some freedom in its actions as it takes into account the world around it.

Before automatic control, humans pulled levers and turned knobs according to their knowledge and objectives. This activity was replaced by automating software and offloading or decreasing humans' direct influence on the process. This understanding might disagree with the common understanding of the term "robot" (thanks again to Hollywood movies), but in fact, an automatic watering machine, and a hobbyist project of a camera attached to a printer that detects, photographs, and prints photos of cats passing nearby, for example, can also be considered robots.

Robot complexities

Now let's think how all this relates to the topic of this book. The connection lies in the controlling software's complexity. To illustrate this, let's consider a simple example of an automatic blind opener, which might be part of a "smart house" solution and solve the task of opening blinds in the morning and shutting them in the evening. It should have a motor that pulls the blind open and closed and a light sensor to detect when the sun is shining.

Even in such a simple operation, the logic and corner cases might be overwhelming. For example, you need to take into account season changes; for example, during the summer, the sun might rise too early to open the blinds at that time, whereas during the winter, it might rise too late for getting up for work.

Additional complexity adds physical world connections: sensors and actuators. Values returned by sensors normally include the noise added to the underlying physical value being measured. This noise is a source of *variance* in the readings. Different instances of sensors might also have *bias* added to the readings, caused by variability in the manufacturing process. Both this variance and bias need to be addressed in software by performing *calibration* of the sensors.

Complications might also come from unexpected directions; for example, the light sensor might become dirty over time, so its readings might change; the motor could get stuck; there might be mechanical differences from one particular motor unit to another; and so on. We also haven't thought yet about solar eclipses, which are not that frequent, but still happen and also need to be taken into account. All those details might lead to the situation where our system, despite working perfectly in the lab, does weird things after being installed. This is only in our simple example of an automatic blind; imagine how many corner cases exist in making a robot that can walk as humans do!

An additional source of headaches in the development of such systems is that computing resources are usually heavily constrained, due to power, space, and weight limitations. For instance, even powerful 32-bit microcontrollers might have less than 1MB of memory, so controlling software is normally written in low-level languages such as C, or even in machine code, which makes the implementation of complicated logic even more tricky.

On the other hand, the machine learning approach to software development provides (or at least promises to) better ways of handling such complicated, noisy, and incompletely defined problems: instead of writing complicated code or manually handling all the corner cases and conditions, let's *learn* the function from some kind of input and output! Given enough time and data, we can capture all the details without writing tons of manual rules.

This sounds very appealing, especially with our familiar RL problem setup: we have observations from our sensors, we can execute actions using our actuators, and we need to define some kind of high-level reward that is related to the problem that we're solving. Plug all this into some RL method and voilà! Of course, I'm simplifying this a lot; there are tons of details, but the overall idea and connection should be obvious.

Let's now start checking our setup and pursuing the goal of this chapter: applying RL methods to simple robotics problems.

The hardware overview

The goal of this book is to make RL methods accessible to a broad audience of hobbyists, enthusiasts, and researchers. This was the driving force behind the selection of examples, their complexity, the software requirements (for instance, using open source PyBullet instead of commercial MuJoCo), and other factors influencing the book's contents. Despite the GPU requirement for most of the examples, they still don't need an extreme amount of horsepower. Just one GPU rented on a cloud will be enough. Unfortunately, in the robotics domain, you can't rent hardware in the same way that you can allocate a GPU-powered instance. You still need to buy the board, sensors, and actuators to be able to play with them.

The good news is that modern electronics and hobbyist platforms, like Arduino and Raspberry Pi, are usually cheap and powerful enough for our purposes. Online stores like AliExpress contain thousands of components from which you can assemble your small working robot for the price of $100 or less with minimal or no soldering (this depends on the board variant). The project in this chapter will be a small four-legged robot, inspired by the Minitaur platform from the previous chapter.

You may remember that Minitaur is a real robot, developed by Ghost Robotics, a Philadelphia-based company that develops four-legged robots for the military, industry, and other applications. If you're curious, you can check the YouTube channel (`https://www.youtube.com/channel/UCG4Xp4nghgyWK4ud5Xbo-4g`). Recently, Google researchers used this robot to experiment with the transferability of policies learned in emulators to real hardware (*Jie Tan*, et.al., *Sim-to-Real: Learning Agile Locomotion For Quadruped Robots*, May 2018, arXiv:1804.10332).

Unfortunately, a Minitaur robot is still too large and costly for enthusiasts to play with. So, in my work, I decided to sacrifice the practical usefulness of the machine to make it as inexpensive as possible.

At a high level, the robot consists of four components:

- *Board*: An STM32F405-based board that is compatible with MicroPython (a reduced version of Python for embedded applications: `http://micropython.org`). The price for the board I used is about $25, but I've seen variants that cost only $15.

- *Sensors*: An integrated board with four sensor chips that provide an accelerometer, gyroscope, magnetometer, and barometer. All those sensors are available on one Inter-Integrated Circuit (I²C) bus. The price is about $15.

- *Servo motors*: Four really cheap pulse width modulation (PWM)-driven servo motors that can be bought on AliExpress or other discount retailers. They are not very reliable, but they are more than suitable for our purposes. The price is $2 per servo.

- *Frame*: A 3D-printed support to mount all the components together. The price for 3D printing services may vary, but it should be about $5, as the frame is small and requires tiny amounts of plastic.

So, the final price barely exceeds $70, which is much more affordable than several thousand for Minitaur or even $500 for Lego Mindstorms (which is also a bit limited for RL applications).

The platform

As mentioned, for the brain of the robot, I used a MicroPython-compatible board with an STM32F405 ARM processor. There are several variants of this board available with different sizes and prices, but as size is not the major issue, you can use any compatible variant you want with an adjustment to the frame.

The board variant I used is available on AliExpress as "The Latest PyBoard V1.1 MicroPython Development Board STM32F405 OpenMV3 Cam M7" (shown in the following photo). The benefits of this board family are that the processor is quite powerful, with an operating frequency of 168MHz and 512KB of memory, but the board is still small and lightweight enough for small servos. In addition, this board (as with most pyboards) has a MicroSD slot that allows us to significantly increase the amount of storage available for the program and other data. This is nice, as we might want to record observations on the robot itself, and our NN models are quite heavy by embedded standards. A pyboard normally comes with a portion of internal flash memory exposed as a filesystem, but it is tiny, so an SD card is really needed.
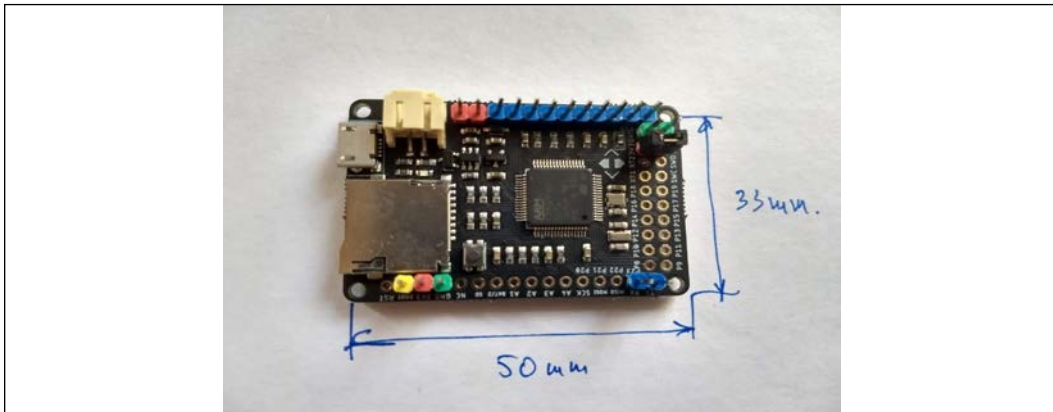
Figure 18.1: The main board

There is one drawback to this board: as it is not the original pyboard distributed by the MicroPython authors, it has different pin names and the SD card wiring is slightly different. The original firmware this board comes with works fine, but if you want to upgrade the version or use your custom firmware, you need to make modifications to the board definition file. For this particular board, I put the changes required, along with a detailed description of how to use them, in the `Chapter18/micropython` directory. But if your board is different, you might need to figure out the changes on your own.

As an alternative, you might consider using the original MicroPython pyboard, which is more expensive, but has out-of-the-box support from the MicroPython source code. From my perspective, this is not critical, but it might be useful if you haven't done anything hardware related before (like microcontrollers programming). There are also other MicroPython-compatible boards available; for example, at the beginning of this project, I used HydraBus, which has exactly the same ARM processor, but a larger form factor. If you're brave and experienced enough, you might even consider using ESP8266-based hardware, which is also supported by MicroPython and provides Wi-Fi connectivity. However, it uses a different processor and has fewer IO ports, so some code modifications might be needed.

The sensors

Robots are supposed to sense the outside world, and as we have discussed, there are plenty of ways to do that. In our small project, we will use a standard set of sensors that is also called an **inertial measurement unit** (**IMU**). Historically, IMUs were built as tricky mechanical devices, with the goal of sensing forces, angular velocities, and the orientation of the object in Earth's magnetic field.

Nowadays, those three sensors are implemented as compact chips that use specific electronic effects in materials to do the measurements and provide their results in a digital or analog form. This makes them much more compact, robust, and convenient to use in a wide variety of projects. A modern IMU board is shown in *Figure 18.2* and includes the following sensor chips (all STMicroelectronics semiconductors, which are the four small squares on the top of the board):

- The *accelerometer* (chip LIS331DLH) measures accelerations up to ±8g on three axes, has 16-bit resolution of measurements, and can survive up to a 10,000g shock. The same accelerometer is installed in the iPhone 4S, for example.

- The *gyroscope* (chip L3G4200D) measures angular velocities on all three axes and can measure up to 2,000 degrees per second, which is a pretty fast rotation (more than 300 rpm).

- The *magnetometer* (chip LIS3MDL) measures the magnetic field direction according to its axes, so it basically shows the direction for north.

- The *barometer* (chip LPS331AP) measures the atmospheric pressure, which could be used to get the current altitude. This chip is useless for our purposes, as the accuracy of measurements doesn't exceed 30 cm. As our robot is not supposed to jump from the roof, this sensor won't be used, but in other cases, like for drones or plane models, knowledge of the altitude might be critical.

In my particular case, I used a board produced in Russia for Amperka kits (shown in the following photo). This exact board might be hard to get if you live outside of Russia, but there are plenty of alternatives you can get everywhere. All those chips provide the digital interface and share the same I²C bus, so to attach all four sensors, only four wires are required (two for the power and two for the communication). It's not very important to use the integrated IMU board; you can get the same results using separate boards with individual sensors or an accelerometer installed on the original pyboard. What are important are the interface and capabilities of the sensor chip. Normally, I²C, or Serial Peripheral Interface (SPI), is the most popular and convenient digital interface.
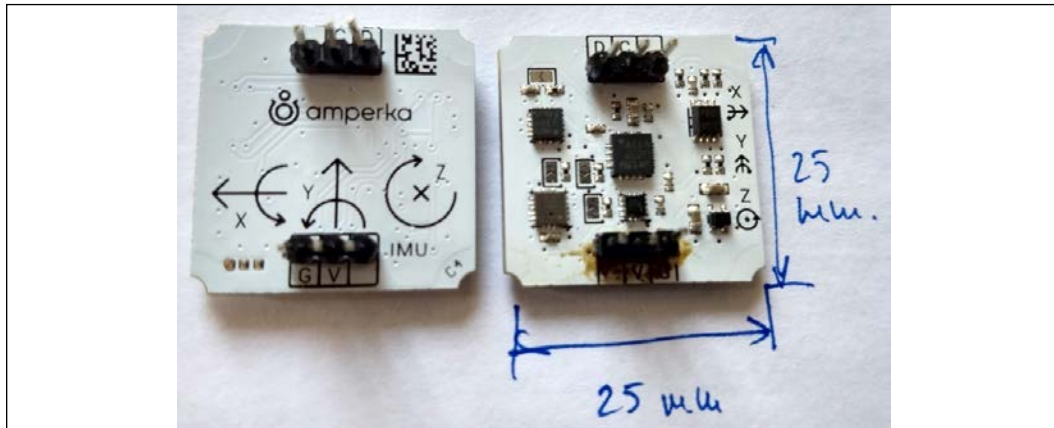
Figure 18.2: The IMU sensor board

There are a lot of other sensors that you might want to attach to your design, including from our earlier list, but processing power requirements will increase.

The actuators

The original idea for our robot is movement, optimizing some high-level reward that we need to formulate. To interact with the physical world, we need some kind of actuators, and there are plenty of options to choose from. The cheapest and most popular options for DIY projects are the aforementioned servo motors (or *servos* for short). Servos might have different interfaces, internal mechanics, and characteristics, but the most common are PWM-controlled analog servos. We will discuss later what this means, but for now, it will be enough to say that we can tell the servo the angle it needs to rotate or maintain, and it will do it.

In my design, I used small, very inexpensive servos that are marked GH-S37A (shown in *Figure 18.3*). They are not very reliable, so you will probably need to replace one or two of them during your experiments, but the price is quite cheap. You might consider other options, like larger servos with metal gears inside, which are more reliable and powerful, but they cost more and drain more current.

Another affordable option is to use stepper motors (with which 3D printers are normally built). They consume much more power and require special driver boards, but they are more precise than analog servos. As an example, Minitaur from Ghost Robotics is built using stepper motors.
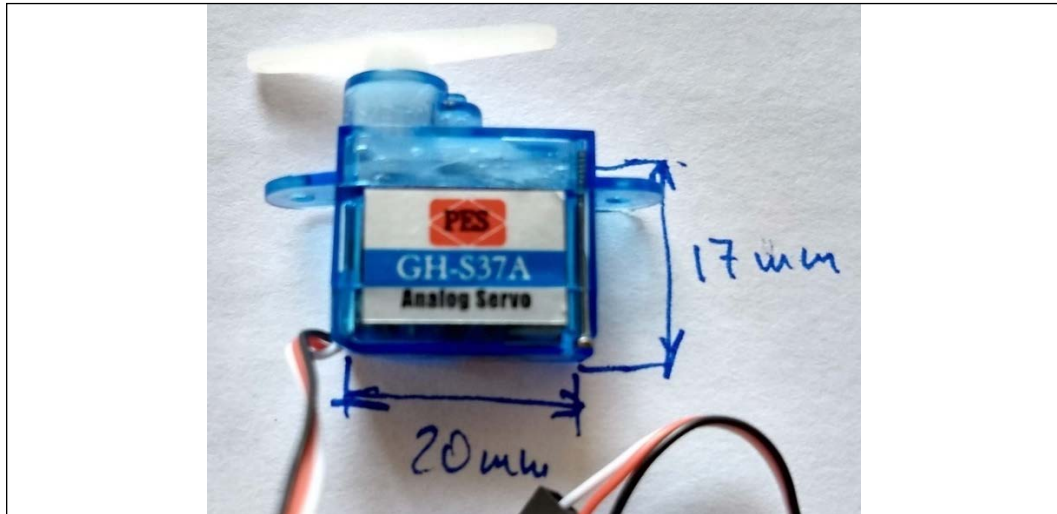
Figure 18.3: The analog servo motor

The frame

The last component in our list is the frame or base, which is an important part in any hardware design, as all the components need to be mounted on it. This is essential for the robot, which will move, so all the actuators, controlling boards, and sensors need to be reliably attached together. At the same time, the base needs to be light; otherwise, the weight will limit the robot's movement capabilities. Some time ago, people used wood, metal, or even cardboard to build the frame, and those options are still there. However, nowadays we have a very nice alternative: 3D printing, which has many benefits over the old-style solutions.

First of all, plastic is lightweight, and in your design, you can choose the level of infill (the ratio at which the internals will be filled by plastic) and make it even lighter. At the same time, when properly designed, the result is quite strong. Plastic still can't compete with metal when you need something reliable and strong, but for most cases, this is not needed.

Another very good benefit is the major simplification of the design and manufacturing process. You can prototype your design using open source tools, send it to the printer, and in a couple of minutes or hours (depending on the size and complexity), you can hold the result in your hands. Also, if needed, you can easily refine the design. This offers a significant speed benefit for design iterations in comparison to wood processing, not to mention metal.

Finally, but very importantly, it's a lot of fun to draw something on your computer and then watch your design materialize from nowhere. You still need to learn basic 3D design software, how to prepare your model for printing, and what the key decisions are during the design, but, still, this is a very interesting activity. On sites like `https://www.thingiverse.com/`, you can find a lot of 3D-printed designs that people are creating, sharing, and improving. I started to experiment with 3D printing during this chapter's preparation, but it quickly became an important hobby of its own. If you're interested, you can find my personal designs here: `https://www.thingiverse.com/shmuma/designs`.

To make a frame for your robot, it needs to be designed using one of the available software packages. There are several alternatives, starting from very simple browser-based design apps for kids that allow you to combine simple shapes together and ending with full-featured commercial software packages used to draw things as complex as modern cars or aircraft engines. For our purposes, we need something in between, and there are many choices here.

My personal favorite for interactive design is Fusion 360 from Autodesk, which is free and relatively simple, but powerful and oriented for engineering design purposes (in contrast to 3D modeling tools like Blender). Another option that might be even simpler for software developers to master is OpenSCAD, which doesn't provide rich interactive tools for design, but rather provides a specialized programming language for describing 3D shapes. Then this program is "compiled" into the 3D object that you can print. The concept is really cool and extremely flexible, but it requires some time to master. If you're curious, you can check `https://www.openscad.org/` or `https://www.thingiverse.com/groups/openscad/things` for examples of such designs.

The frame of the robot for the components that I used was designed using OpenSCAD, and the full source code of the frame is in `Chapter18/stl/frame_short_legs_pyboard.scad`. In the following screenshots, the final version of the frame and the design process are shown. In addition to the source code, I've provided the STL format of the frame in `Chapter18/stl/frame_short_legs_pyboard.stl`.

This is a standard format for exchanging 3D models that are ready to be printed, but it is quite hard to modify them. So, if you want to adjust the model for your board dimensions, for example, you will need to install OpenSCAD, tweak the source code, and then export the model as an STL file.
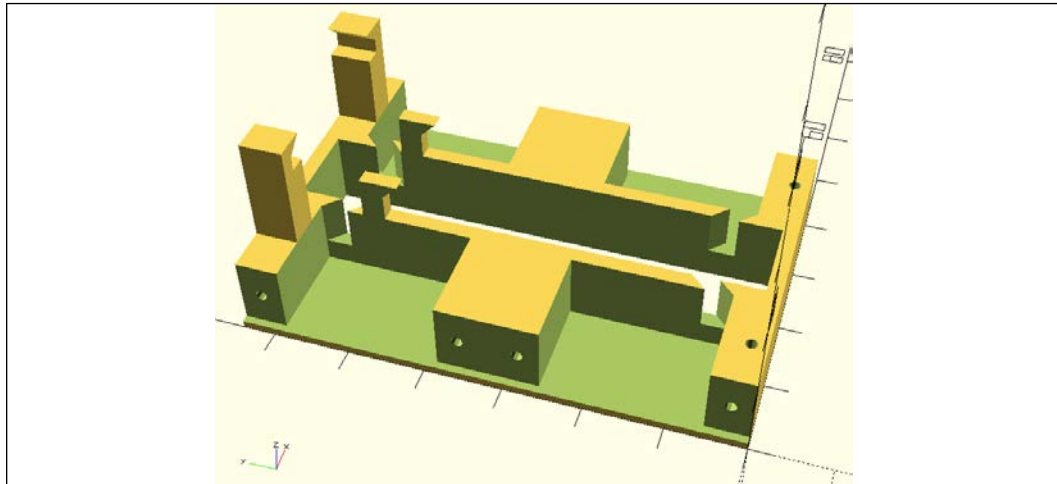
Figure 18.4: The render of the robot frame
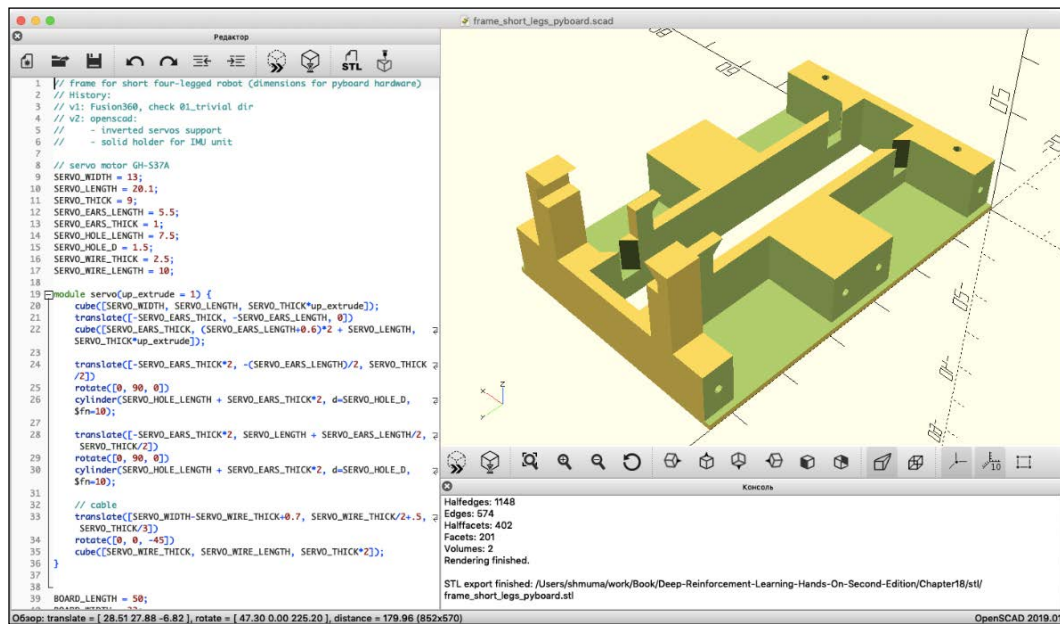
Figure 18.5: The OpenSCAD design process

Once the STL file is ready, it can be printed, which can be done following different paths. One option is to use the services of a company for the 3D printing of your model. It might be some local company or a global web-based manufacturing service. In that case, you normally need to upload the STL file, fill out a form with printing options (like the color, material, level of infill, and so forth), pay, and wait for the result to be mailed to you.

Another option is to use one of the 3D printers that might be available around you. Nowadays, schools, university labs, or hackerspaces have 3D printers. In that case, you will need to prepare the input for the printer firmware from the STL file. This input is called G-code and it contains lots of low-level instructions for the specific printer, like move at this coordinate or extrude that amount of plastic. The process of converting STL files into G-code is called *slicing* because fused deposition modeling (FDM) printers produce objects layer by layer, so a 3D model needs to be sliced to those layers. To do this, specialized software packages called *slicers* are used. If you've never done 3D printing before, you might want to ask somebody familiar with the particular printer settings to assist you, as the slicing process has tons of options and parameters that influence the result, like layer thickness and specific plastic parameters. On the other hand, you definitely don't need a PhD for this. There is a lot of information available online, and the whole process is quite logical; it just might take some time to master.

The frame design is very simple to print, requires no supports (extra plastic supporting the overhanging parts, as 3D printers can't print on air), and normally can be printed in an hour to an hour and 30 minutes, depending on the printer setting. The photo of the resulting frame is shown in *Figure 18.6*.
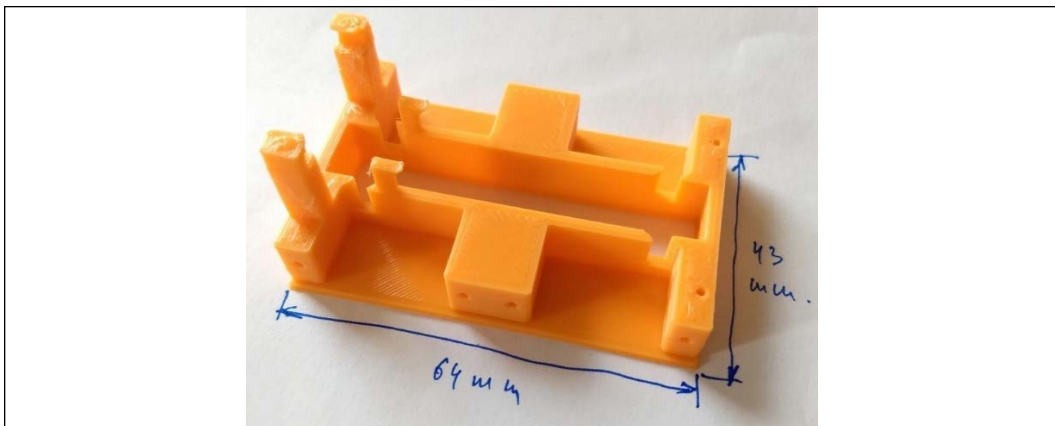
Figure 18.6: The printed robot frame

To give you an idea of what the assembled thing looks like, I've included the following photo. The details of the connection process will be covered in the subsequent section, when we will start to program the hardware.
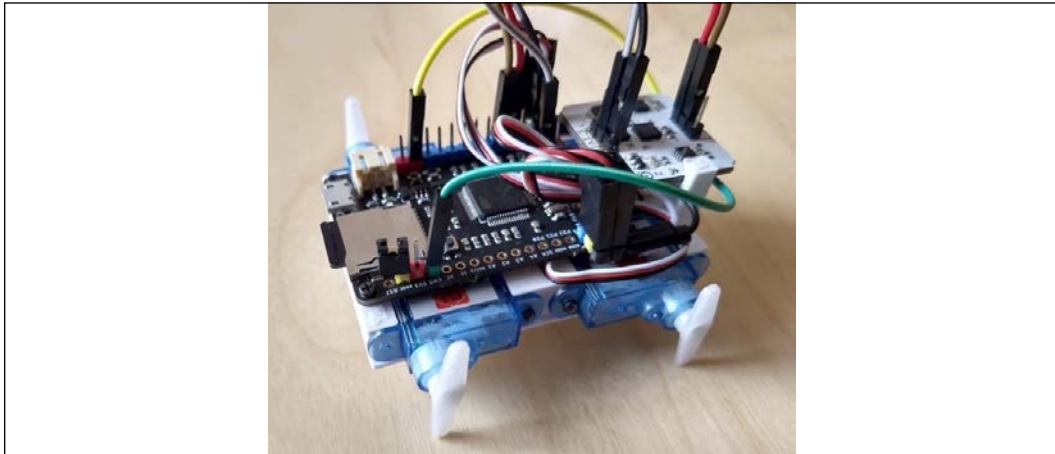
Figure 18.7: The assembled robot (almost) ready for world domination (it still needs the power plug)

The first training objective

Let's now discuss what we want our robot to do and how we're going to get there. It's not very hard to notice that the potential capabilities of the hardware described are quite limited:

- *We have only four servos with a constrained angle of rotation*: This makes our robot's movements highly dependent on friction with the surface, as it can't bring its individual legs up, which is also the case with the Minitaur robot, which has two motors attached to every leg.

- *Our hardware capacity is small*: The memory is limited, the central processing unit (CPU) is not very fast, and no hardware accelerators are present. In the subsequent sections, we will take a look at how to deal with those limitations to some extent.

- *We have no external connectivity besides a micro-USB port*: Some boards might have Wi-Fi hardware, which could be used to offload the NN inference to a larger machine, but in this chapter's example, I'm leaving this path unexplored. It might be an interesting project to add an ESP8266 board (which is an extremely popular microcontroller board with embedded Wi-Fi connectivity) to the robot (or you could even use an ESP8266 board as the main board) and make it wirelessly connected.

My goal for this chapter is not to describe how to build your own T800 in your basement, but rather to show how a very simple robot can be programmed using RL methods from the very beginning. The result for this particular robot might not be very impressive, but when you have become familiar with the process, it will be much easier to extend the capabilities in all directions, like using extra servos, building much more complicated mechanics, adding sensors, using external GPUs, or even using a field-programmable gate array (FPGA) for NN acceleration, and so on.

Before that, we need to solve the essential problems, like getting the data from the sensors, controlling the servos, putting our policy on the hardware, determining what our inference loop will look like, and, of course, training the policy. Those problems are completely non-trivial for beginners, so we will take a very simple objective and solve it, helping you to gain experience and self-confidence along the path. The first objective will be simple: standing up from a lying position. On the one hand, I'd even say that this is trivial, as to stand up, the robot needs only to move the servos to the fixed location and keep them there. On the other hand, this objective is very simple to express in RL terms, and the trained policy could be easily checked on the hardware.

Besides the objective, we need to make a decision about the training process, for which we basically have two options. The first option is to train on live hardware, gathering the observations from the attached sensors, passing the actual commands to the servos, and measuring the objective (in our case, the distance of the robot's frame to the surface). Obviously, our hardware is too weak for deep learning, but even if we upgraded the hardware, there would be no way to make PyTorch work on this specialized microcontroller.

If you want to go to the extreme, you can implement your own backpropagation version on raw Python or low-level firmware, but that's way much more complicated than our goal here. If you're curious about doing this, one of the homework assignments in Stanford University's *CS231n* course on deep learning was to implement NN training using only the NumPy library, which is hard to do properly, but an extremely useful exercise to get an understanding of what's going on under the hood.

As a workaround, we could transfer observations using Wi-Fi or another communication channel to some "big computer" to train the policy and then send the weights back to our robot for execution, but as I've mentioned, our hardware lacks Wi-Fi connectivity.

In addition to computation complexity, training on live hardware has extra issues, like hardware safety (we might want to prevent our robot from breaking during the training), sample efficiency, and the necessity of getting reliable reward information. The last point might be quite hard to solve, as the robot itself has no way to accurately measure the altitude, so some external hardware to estimate the reward will be required.

Another option for training is to use an emulator, in the same way as we did in the previous chapter when we used PyBullet. In that case, the training was performed on the emulated physical world, about which we had complete information, so measuring the reward was trivial. The only tricky thing here is how to get the accurate model of the real world. As you will quickly see in the next section, it might be quite complicated even for the simple mechanics of our tiny robot. Nevertheless, this path looks simpler than doing everything on real hardware, although you could try to follow the alternative path in your own experiments. Ultimately, this chapter is not a complete solution to the robotics problem, but rather it is the outline of a possible path.

The emulator and the model

In this section, we will cover the process of obtaining the policy that we will deploy on the hardware. As mentioned, we will use a physics emulator (PyBullet in our case) to simulate our robot. I won't describe in detail how to set up PyBullet, as it was covered in the previous chapter. Let's jump into the code and the model definition.

In the previous chapter, we used robot models already prepared for us, like Minitaur and HalfCheetah, which exposed the familiar and simple Gym interface with the reward, observations, and actions. Now we have custom hardware and have formulated our own reward objective, so we need to make everything ourselves. From my personal experiments, it turned out to be surprisingly complex to implement a low-level robot model and wrap it in a Gym environment. There were several reasons for that:

- PyBullet classes are quite complicated and poorly designed from a software engineer point of view. They contain a lot of side effects, like fields influencing the behavior of methods, non-obvious execution flow, and just too complicated and obfuscated logic. The documentation is almost missing, so the only sources of information are the existing models given in the source code, which is not the greatest way to learn. All this led to a lot of mistakes that I needed to solve before I got something workable.

- Internally, PyBullet uses the Bullet library written on C++ and provides a generic physics emulator. Bullet has documentation about the methods and the overall structure, but this is not the easiest to read and is far from being complete.

- To define your model, you need to provide the geometry of your robot and environment, and you need to define all the actuators and properties of the model that you're simulating, like frictions, actuator types, parameters, masses, inertia moments, and so on. To describe the model structure, several formats are supported: URDF (Unified Robot Description Format), MJCF (the MuJoCo XML format), and others. I didn't find any of them very easy to write, as I was defining all the properties at a very low level. The MuJoCo format turned out to be slightly better documented, so I used it for my robot's definition, but it was still painful to understand the requirements of the file format and express the geometry and actuators even for our simple design. Maybe I've missed some super cool tool that simplifies the process, but I checked several alternatives and found nothing better than writing a MuJoCo XML file in a text editor. It was complicated and long, so obviously better solutions are needed for more complex designs.

- I encountered bugs in Bullet and PyBullet. As Bullet imports the MuJoCo file format into its own representation, it doesn't support all the features of MuJoCo. Even for the supported parts, it might suddenly fail to load or behave weirdly, which doesn't add simplicity to the design process.

Better alternatives likely exist. Some people use game engines, like Unreal Engine for physics simulation (which means that the game industry evolved to simulate the real world with quality comparable to professional simulators, which is fun), but I haven't checked this alternative.

The complete PyBullet environment is contained in two files: `Chapter18/lib/microtaur.py` and `Chapter18/models/four_short_legs.xml`. The Python module follows the structure of other PyBullet robot environments and includes two classes:

- `FourShortLegsRobot`, which inherits from the `pybullet_envs.robot_bases.MJCFBasedRobot` class. It loads the XML MuJoCo model and is responsible for low-level control of the robot, like setting the servos positions, getting the joint orientations, and gathering the observation vector from the internal PyBullet state.

- `FourShortLegsEnv`, which inherits `from pybullet_envs.env_bases.MJCFBaseBulletEnv` and provides the Gym-compatible interface for the robot. It uses the `FourShortLegsRobot` class to obtain the observation, apply actions, and get the reward value.

In addition to those two classes, there is the third class
`FourShortLegsRobotParametrized`, which subclasses `FourShortLegsRobot`
and allows the user to redefine internal parameters of the XML model file.
It won't be used in this chapter's example because when I tried it during my
experiments to make PyBullet simulation more realistic, it didn't turn out to be
useful. I've left this class with the accompanying template file `Chapter18/models/`
`four_short_legs.xml.tmpl` if you would like to experiment with this optimization.

The model definition file

The logic of `FourShortLegsRobot` is heavily dependent on the underlying MuJoCo
model file, so let's start with it. The full specification of all the tags supported by
MuJoCo is given here: `http://mujoco.org/book/XMLreference.html`. This
is a long, detailed description of the file format used by MuJoCo to describe the
environment, the properties of objects, and the way they can interact. I'm not going
to give you a full description as I'm not an expert in MuJoCo. The following is the
definition of our robot geometry with my comments:

```
<mujoco model="four_short_legs">
    <compiler inertiafromgeom="true" settotalmass="0.05"/>
```

In the outer tag, the file specifies the name of the model. Then the second line
contains options for the parser and compiler responsible for assembling the
internal objects from file statements. This tag might contain several options
influencing the model as a whole. In our file, we give two parameters: with
`inertiafromgeom="true"` we say that the compiler needs to deduct inertial
properties from the underlying geometry, and with option `settotalmass="0.05"`
we define the total mass of our robot in kilograms. The weight of the assembled
robot is 50 grams according to my kitchen scales. Inertial properties might be very
important to get realistic results from simulation, so the first step in improving the
model probably will be to disable automatic inertia deduction and give it explicitly.

```
    <default>
        <joint armature="0" damping="1" limited="true"
frictionloss="0"/>
        <geom friction="0.5 0.1 0.1"/>
        <position kp="10"/>
    </default>
```

The `default` section includes parameters that will be used as fallback values for all
objects of the applicable kind. Here we define the parameters of joints in our object.
In our robot, we have five joints: four are rotating legs and we have one fixed joint
that connects the IMU sensor to the robot frame.

The parameter `geom` sets the friction parameters for all of our geometries. It has three values for different kinds of friction that the emulator deals with: sliding friction, torsional friction, and rolling friction. I haven't spent much time optimizing this, so it definitely could be improved.

The last parameter given in the `default` section applies to our servos, which are positional servos. MuJoCo supports several kinds of rotating actuators (they are called "hinge joints") for which you can set various control parameters. Those parameters might be the position (when you set the angle the actuator needs to get), velocity, or force.

Different types of actuators are controlled in a different way. For example, a simple motor has a speed of rotation proportional to the voltage applied to the motor, so the velocity parameter is the most appropriate. In our case, we have a positional servo, to which we just tell the angle it needs to keep, so our actuator will be controlled by the position. There are many internal parameters that define intricate details of actuator dynamics, so to get an accurate model, they need to be measured or taken from the actuator specifications and applied. In our laypeople approach, we just use the defaults, and the only parameter specified is `kp="10"`, which sets the so-called *position feedback gain* that specifies the force coefficient the motor is applying to get to the desired position.

```
<option gravity="0 0 -9.8" integrator="RK4" timestep="0.02"/>
<size nstack="3000"/>
```

The last two parameters in our file header are control options specifying the gravity direction, the timestep, and the internal integrator used. As we're going to load this file into PyBullet, probably only the gravity will be taken into account; however, I copied other options from other files, so it won't do any harm to have them. The `nstack` parameter gives the size of the pre-allocated stack for the model interpreter.

```
<worldbody>
    <!--<geom name="ground" type="plane" pos="0 0 0"/>-->
    <body name="base" pos="0 0 0.01">
        <geom name="base" pos="0 0 0.0" type="box"
              size="0.032 0.032 0.01" euler="0 0 0"/>
```

The `worldbody` tag starts the main part of the file and describes the parts of your model, their geometry, and their interaction. The first (commented) geometry object defines the ground plane. I commented it out because the ground plane is added automatically by PyBullet classes, but this plane will be needed if you want to load it into another system; otherwise, the robot will fall into the void.

The next tag specifies the frame body, which lies flat on the ground plane. The geometry of the frame is set as `type="box"`. The `size` parameter defines half of the box dimensions, which is not very convenient, but that's a requirement of the MuJoCo file format (though I don't know why). Our frame (including the mounted servos) is 64 mm x 64 mm and about 20 mm high, so by dividing those dimensions by two, we get the geometry size. The `euler` parameter specifies the orientation of the geometry as a rotation around the XYZ axes. Our frame is not rotated, so we have zeros there. If you have trouble imagining our model from the coordinates, you can peek at *Figure 18.8*, which shows the initial position of the robot, or you can try the tool `Chapter18/show_model.py` (described later).

```
<body name="leg_rb" pos="0.035 -0.022 -0.005">
    <joint axis="1 0 0" name="servo_rb" pos="0 0 0"
            range="0 180" type="hinge"/>
    <geom fromto="0 -0.013 0 0 0.013 0" name="leg"
            size="0.003 0.002" type="capsule"/>
</body>
<body name="leg_rf" pos="0.035 0.022 -0.005">
    <joint axis="1 0 0" name="servo_rf" pos="0 0 0"
            range="-180 0" type="hinge"/>
    <geom fromto="0 -0.013 0 0 0.013 0" name="leg"
            size="0.003 0.002" type="capsule"/>
</body>
<body name="leg_lb" pos="-0.035 -0.022 -0.005">
    <joint axis="1 0 0" name="servo_lb" pos="0 0 0"
            range="0 180" type="hinge"/>
    <geom fromto="0 -0.013 0 0 0.013 0" name="leg"
            size="0.003 0.002" type="capsule"/>
</body>
<body name="leg_lf" pos="-0.035 0.022 -0.005">
    <joint axis="1 0 0" name="servo_lf" pos="0 0 0"
            range="-180 0" type="hinge"/>
    <geom fromto="0 -0.013 0 0 0.013 0" name="leg"
            size="0.003 0.002" type="capsule"/>
</body>
```

Then, inside the base body object, we have four sub-bodies defining our robot's legs. All of them have the same parameters, except the position attribute specifying where each leg is located. To make sense of the coordinates, you need to know that the child body's position is defined relative to the parent body's geometry. So, the Z coordinate of every leg is set to –0.005, instead of 0.005.

Every leg is given by the `body` section with the `name` attribute. Using this name, we will be able to check its attributes during the simulation. Inside every leg's `body` tag, we have two parameters. The `joint` parameter sets the type, location, and range of the corresponding servo actuator. All our joints are aligned along the X axis, so we have `axis="1 0 0"`. In addition, we need to specify the geometry of the leg using the `geom` parameter. In our case, the simplest approximation of our robot's legs will be `type="capsule"`, which is a rounded cylinder with the specified dimensions.

```
            <!-- IMU sensor box -->
            <body name="IMU" pos="0.0025 0.03 0.02" euler="0 0 0">
                <geom name="IMU" pos="0 0 0" type="box"
                        size="0.02 0.02 0.003" euler="0 0 0"/>
                <joint armature="0" damping="0" limited="false"
                        axis="0 0 0" name="IMUroot" pos="0 0 0"
                        stiffness="0" type="fixed"/>
            </body>
        </body>
    </worldbody>
```

The final sub-body of our robot is a sensor board, which is represented by a separate box shifted towards the front of the frame and elevated a bit. This box is attached to the base frame by a special joint with `type="fixed"`, which specifies the rigid attachment of two bodies. The sensor board is modeled by a separate body because it will be used to obtain the accelerations and orientation for our observations.

```
    <actuator>
        <position joint="servo_rb" name="servo_rb"/>
        <position joint="servo_rf" name="servo_rf"/>
        <position joint="servo_lb" name="servo_lb"/>
        <position joint="servo_lf" name="servo_lf"/>
    </actuator>
</mujoco>
```

At the tail of the file, outside the `worldbody` tag, we define the joints and the type of actuators used in our model. As I mentioned, we're using position servos that require the angle they need to be rotated by; thus, our actuators are positional. In the following figure, you can see the rendered model specified by the file that we have just described. The model might look silly, but it contains all the important parts of our robot needed for RL training.
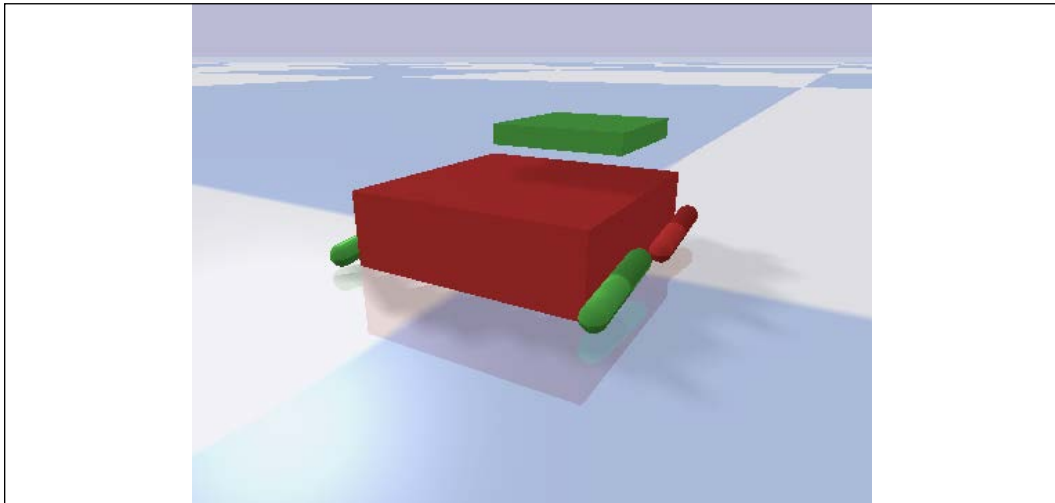
Figure 18.8: The initial position of the robot model in the PyBullet world

The robot class

The low-level behavior of the robot model is defined by the class `FourShortLegsRobot` in `Chapter18/lib/microtaur.py`. This class inherits from the PyBullet class `MJCFBasedRobot`, which provides functionality for loading the MuJoCo file and building connections between the model's components. Unfortunately, the code quality of `MJCFBasedRobot` and its parent class `XmlBasedRobot` is quite low, which makes it hard to implement working custom wrappers. It took me a while to make this class work properly. Hopefully, my modifications will save you some time in your experiments.

So, let's consider the class. At a high level, its responsibility is to load our XML file into PyBullet and allow us to apply the action vector to the actuators, and request the observations that we need. This is achieved by redefining certain methods from the parent class and defining our own methods dealing with PyBullet low-level details, like joint positions.

```
class FourShortLegsRobot(robot_bases.MJCFBasedRobot):
    IMU_JOINT_NAME = "IMUroot"
```

```
SERVO_JOINT_NAMES = ('servo_rb', 'servo_rf',
                     'servo_lb', 'servo_lf')

def __init__(self, time_step: float, zero_yaw: bool):
    action_dim = 4
    # current servo positions + pitch, roll and yaw angles
    obs_dim = 4 + 3

    super(FourShortLegsRobot, self).__init__(
        "", "four_short_legs", action_dim, obs_dim)
    self.time_step = time_step
    self.imu_link_idx = None
    self.scene = None
    self.state_id = None
    self.pose = None
    self.zero_yaw = zero_yaw
```

In the constructor, we pass a lot of arguments to the parent constructor, like the name of our robot, the action, and the observation dimensionality. Despite the low-level nature of those classes in the PyBullet hierarchy, they create the Gym action and observation spaces. They are not used, which is the first obvious example of the bad design of those classes. Unfortunately, we will see much more of this later.

In our constructor, we need the timestep in the simulated environment and a `zero_yaw` flag, which controls the observation calculation (which will be explained later). The action space is four numbers specifying the angle of rotation of our servos. To simplify the task of our robot, the servos located on the back rotate in the counterclockwise direction (relative to the direction of the Y axis), but the servos on the front of the robot rotate in the opposite direction. This allows the robot to rise even with small rotations of the servos; owerwise, it will just slip forward or backward. The observation vector is composed of four current servos positions plus the orientation of the robot's frame, which gives us seven numbers for every observation.

```
def get_model_dir(self):
    return "models"

def get_model_file(self):
    return "four_short_legs.xml"

# override reset to load model from our path
def reset(self, bullet_client):
    self._p = bullet_client
    if not self.doneLoading:
        self._p.setAdditionalSearchPath(self.get_model_dir())
```

```
        self.objects = self._p.loadMJCF(self.get_model_file())
        assert len(self.objects) == 1
        self.parts, self.jdict, \
        self.ordered_joints, self.robot_body = \
            self.addToScene(self._p, self.objects)

        self.imu_link_idx = self._get_imu_link_index(
            self.IMU_JOINT_NAME)
        self.doneLoading = 1
        self.state_id = self._p.saveState()
    else:
        self._p.restoreState(self.state_id)
    self.robot_specific_reset(self._p)
    self.pose = self.robot_body.pose()
    return self.calc_state()
```

The `reset()` method is overridden from the parent and is responsible for loading the MJCF file and registering loaded robot parts in the class internals. We load the file only once. If it has already been loaded, we restore the PyBullet emulator state from a previously saved checkpoint.

```
def _get_imu_link_index(self, joint_name):
    for j_idx in range(self._p.getNumJoints(self.objects[0])):
        info = self._p.getJointInfo(self.objects[0], j_idx)
        name = str(info[1], encoding='utf-8')
        if name == joint_name:
            return j_idx
    raise RuntimeError
```

The preceding code shows the internal method that is supposed to find the IMU link index by the link name. We will use this index in other helper methods to get the IMU position and orientation.

```
def _joint_name_direction(self, j_name):
    # forward legs are rotating in inverse direction
    if j_name[-1] == 'f':
        return -1
    else:
        return 1
```

This method returns the direction multiplier applied to the servo joint. As I already mentioned, the forward actuators rotate in the opposite direction to the servos on the back.

```
def calc_state(self):
    res = []
```

```
        for idx, j_name in enumerate(self.SERVO_JOINT_NAMES):
            j = self.jdict[j_name]
            dir = self._joint_name_direction(j_name)
            res.append(j.get_position() * dir / np.pi)
        rpy = self.pose.rpy()
        if self.zero_yaw:
            res.extend(rpy[:2])
            res.append(0.0)
        else:
            res.extend(rpy)
        return np.array(res, copy=False)
```

The `calc_state()` method returns the observation vector from the current model's state. The first four values are our servos' angles (the positions are normalized to the range 0…1). The last three values in the observation vector are angles (in radians) of the base orientation in the space: roll, pitch, and yaw values. If the parameter `zero_yaw` (passed to the constructor) is `True`, the yaw component will be zeros. The reason behind this will be explained in the *Controlling the hardware* section, when we will start to deal with observations from the real hardware and the transferability of the policy.

```
    def robot_specific_reset(self, client):
        for j in self.ordered_joints:
            j.reset_current_position(0, 0)


    def apply_action(self, action):
        for j_name, act in zip(self.SERVO_JOINT_NAMES, action):
            pos_mul = self._joint_name_direction(j_name)
            j = self.jdict[j_name]
            res_act = pos_mul * act * np.pi
            self._p.setJointMotorControl2(
                j.bodies[j.bodyIndex], j.jointIndex,
                controlMode=p.POSITION_CONTROL,
                targetPosition=res_act, targetVelocity=50,
                positionGain=1, velocityGain=1, force=2,
                maxVelocity=100)
```

The last two methods perform the reset of the robot (setting the actuators to the initial position) and apply the actions. The action value is expected to be in the range of 0…1 and it defines the ratio of the servo range to position the servo.

That's it for the `FourShortLegsRobot` class. As you might guess, it's not very usable for our training, but rather provides the underlying functionality for the higher-level Gym environment.

This environment is defined in the second class in the file: `FourShortLegsEnv`.

```
class RewardScheme(enum.Enum):
    MoveForward = 0
    Height = 1
    HeightOrient = 2


class FourShortLegsEnv(env_bases.MJCFBaseBulletEnv):
    HEIGHT_BOUNDARY = 0.035
    ORIENT_TOLERANCE = 1e-2

    def __init__(self, render=False, target_dir=(0, 1),
                 timestep: float = 0.01, frameskip: int = 4,
                 reward_scheme: RewardScheme =
                     RewardScheme.Height,
                 zero_yaw: bool = False):
        self.frameskip = frameskip
        self.timestep = timestep / self.frameskip
        self.reward_scheme = reward_scheme
        robot = FourShortLegsRobot(self.timestep,
            zero_yaw=zero_yaw)
        super(FourShortLegsEnv, self).__init__(robot,
            render=render)
        self.target_dir = target_dir
        self.stadium_scene = None
        self._prev_pos = None
        self._cam_dist = 1
```

This class supports several reward schemes, which will be described later.
In addition to that, the constructor enables rendering and setting the timestamp and
the number of emulator steps that we will perform on every action (`frameskip`).

```
    def create_single_player_scene(self, bullet_client):
        self.stadium_scene =
                scene_stadium.SinglePlayerStadiumScene(
            bullet_client, gravity=9.8, timestep=self.timestep,
            frame_skip=self.frameskip)
        return self.stadium_scene
```

The preceding method is called by the parent's `reset()` method (another example
of ugly design!) to create the scene of our robot: the ground plane and simulator
instance.

```
    def _reward_check_height(self):
        return self.robot.get_link_pos()[-1] >
            self.HEIGHT_BOUNDARY
```

```
def _reward_check_orient(self):
    orient = self.robot.get_link_orient()
    orient = p.getEulerFromQuaternion(orient)
    return (abs(orient[0]) < self.ORIENT_TOLERANCE) and \
           (abs(orient[1]) < self.ORIENT_TOLERANCE)
```

Two internal methods are used to calculate the reward in different reward schemes. The first checks that the robot's base height is above the specified boundary height. As height is measured from the IMU unit, which is lifted above the robot frame, the boundary is 35 mm above the ground. If you're curious about the height in the model, you can experiment with the Chapter18/show_model.py tool, which outputs the height. My tests show that in the lying flat position, the IMU height is 30 mm (0.03), and in the standing position, it is 41 mm (0.0409). We start rewarding the agent when it is halfway to the full standing position.

The second method checks that the roll and pitch angles are smaller than the given tolerance, which effectively means that our base is almost parallel to the ground.

```
def _reward(self):
    result = 0
    if self.reward_scheme == RewardScheme.MoveForward:
        pos = self.robot.get_link_pos()
        if self._prev_pos is None:
            self._prev_pos = pos
            return 0.0
        dx = pos[0] - self._prev_pos[0]
        dy = pos[1] - self._prev_pos[1]
        self._prev_pos = pos
        result = dx * self.target_dir[0] + \
                 dy * self.target_dir[1]
    elif self.reward_scheme == RewardScheme.Height:
        result = int(self._reward_check_height())
    elif self.reward_scheme == RewardScheme.HeightOrient:
        cond = self._reward_check_height() and \
               self._reward_check_orient()
        result = int(cond)
    return result
```

This function is also internal and is responsible for calculating the reward value according to the given reward scheme. If the MoveForward scheme is selected, the robot is rewarded for moving as far as possible in the given direction (the default is the Y axis). The reward scheme Height gives a reward of 1 if the base is above the threshold.

The third reward scheme, `HeightOrient`, is almost the same, but in addition to the height boundary, it checks that the base is parallel to the ground.

```
def step(self, action):
    self.robot.apply_action(action)
    self.scene.global_step()
    return self.robot.calc_state(), self._reward(), False, {}

def reset(self):
    r = super(FourShortLegsEnv, self).reset()
    if self.isRender:
        distance, yaw = 0.2, 30
        self._p.resetDebugVisualizerCamera(
            distance, yaw, -20, [0, 0, 0])
    return r

def close(self):
    self.robot.close()
    super(FourShortLegsEnv, self).close()
```

The rest of the class is almost trivial: we just combine the functionality of the functions that we have already discussed.

DDPG training and results

To train the policy using our model, we will use **deep deterministic policy gradients** (**DDPGs**), which we covered in detail in *Chapter 17, Continuous Action Space*. I won't spend time here showing the code, which is in `Chapter18/train_ddpg.py` and `Chapter18/lib/ddpg.py`. For exploration, the Ornstein-Uhlenbeck process was used in the same way as for the Minitaur model.

The only thing I'd like to emphasize is the size of the model, in which the actor part was intentionally reduced to meet our hardware limitations. The actor has one hidden layer with 20 neurons, giving just two matrices (not counting the bias) of 28×20 and 20×4. The input dimensionality is 28, due to observation stacking, where four past observations are passed to the model. This dimensionality reduction leads to very fast training, which can be done without a GPU involved.

To train the model, you should run the `train_ddpg.py` program, which accepts the following arguments:

- `-n` (`--name`): The name of the run, which is used in the TensorBoard metrics and save directory

- `--cuda`: This enables the GPU, which is almost useless in this case

- `--reward`: The scheme of the reward to be used during the training, which could be one of the following: `MoveForward`, `Height`, or `HeightOrient`

- `--zero-yaw`: This enables putting in a zero value instead of the yaw observation angle

The motivation of the `--zero-yaw` option and `HeightOrient` reward scheme will be explained a little later, when we get to the actual hardware running our policy. For now, just the `Height` reward objective will be enough.

To get our first "stand up" policy, you should use `./train_ddpg.py -n some_name –reward Height`, which converges quite quickly and in 20-30 minutes produces a model file that is able to reach 996-998 reward during the test. Given that the environment limits the number of timesteps to 1,000 steps, this means that the robot's IMU is above the threshold during ~99.7% of timesteps, which is almost perfect.

The following are the charts of convergence that I got from my experiments.
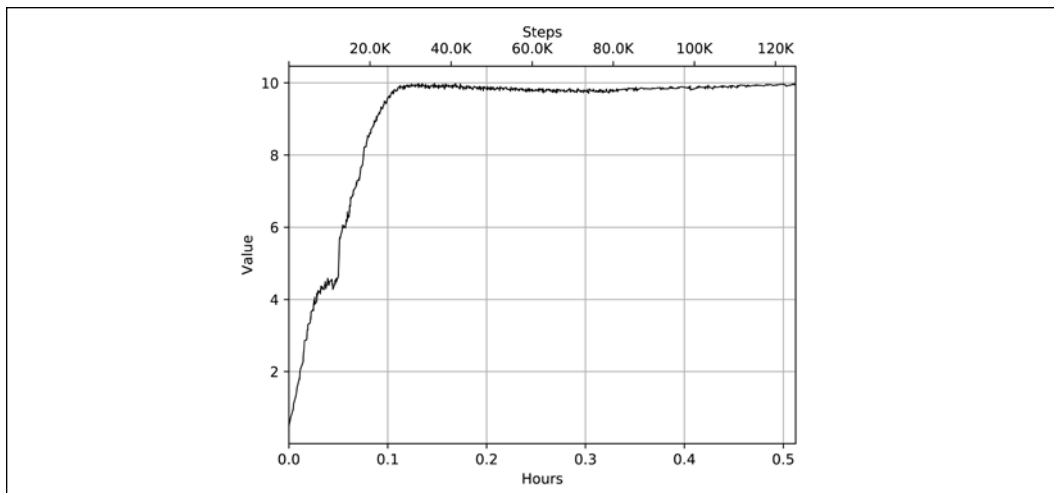
Figure 18.9: The reference value predicted by the critic network
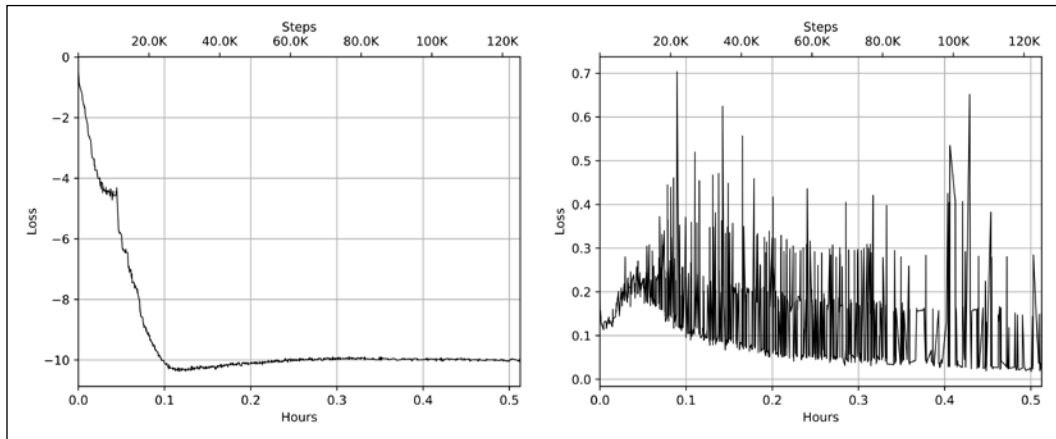
Figure 18.10: The loss of the actor (left) and critic (right)
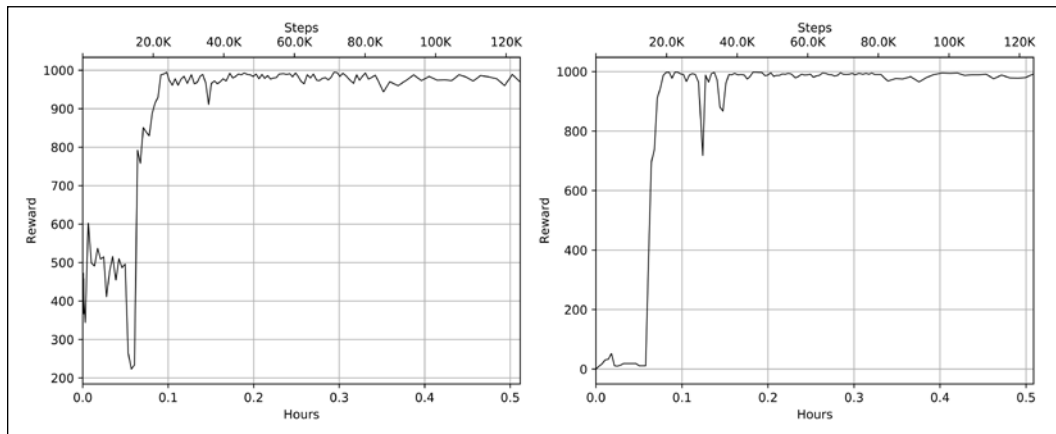
Figure 18.11: The rewards from training episodes (left) and test episodes (right)

Every time a new best reward is obtained from the testing, the actor model is saved into the file. To check the model of the robot in action, you can use the Chapter18/show_model.py utility. It accepts the following command-line arguments:

- -m (--model): The file name to be loaded and used to infer actions from observations. This parameter is optional; if it is not given, the constant action (0.0 by default) is executed on the model.

- -v (--value): The constant value to be executed as an action. By default, it equals 0.0. This argument allows us to experiment with different leg rotations.

- -r (--rotate): If given, this sets the leg index 0…3 to be rotated back and forth. This allows us to check the dynamic behavior of the simulated model.

- --zero-yaw: This passes zero_yaw=True to the environment.

This utility can be used for different purposes. First of all, it allows us to experiment with our robot model (for instance, to check the geometry of the robot), validate the reward scheme used, and see in the dynamics what happens when one of the legs is rotated. For this use case, you shouldn't pass the `-m` option with a trained model file. For instance, the following are screenshots of the model with different values passed to `-v`.
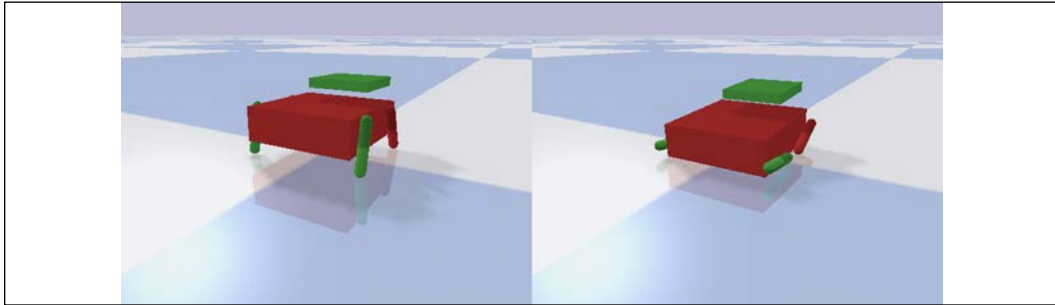
Figure 18.12: The model of the robot with different actions applied: 0.4 (left) and 0.9 (right)

Another use case of the `show_model.py` utility is to check the trained policy in action. To do that, you should pass the saved actor model file in the `-m` command-line parameter. In this case, the model will be loaded and used to get actions from observations on every step.

By checking one of the models saved during the training, you might notice that the dynamics of our robot resemble reality, but they are sometimes weird. For instance, the model might show springy jumps when the legs are rotated suddenly. This is caused by imperfections in the MJCF file parameters; very likely, our actuator parameters are far from realistic, and the inertia properties may also not be accurate. Nevertheless, we have something, which is a great start. Now the question is: how do we move the policy to the real hardware? Let's try it out!

Controlling the hardware

In this section, I will describe how we can use the trained model on the real hardware.

MicroPython

For a very long time, the only option in embedded software development was using low-level languages like C or assembly. There are good reasons behind this: limited hardware capabilities, power-efficiency constraints, and the necessity of dealing with real-world events predictably. Using a low-level language, you normally have full control over the program execution and can optimize every tiny detail of your algorithm, which is great.

The downside of this is complexity in the development process, which becomes tricky, error-prone, and lengthy. Even for hobbyist projects that don't have very high efficiency standards, platforms like Arduino offer a quite limited set of languages, which normally includes C and C++.

MicroPython (`http://micropython.org`) provides an alternative to this low-level development by bringing the Python interpreter to microcontrollers. There are some limitations of course: the set of supported boards is not very large, the Python standard library is not fully ported, and there is still memory pressure. But for noncritical applications and quick prototyping, MicroPython provides a really nice alternative to low-level languages. If you're interested, you can find the full documentation about the project at `http://micropython.org`. Here, I'll just give a quick overview of MicroPython's capabilities and limitations to get you started.

Normally, MicroPython comes as a firmware for a particular board. Usually, when you buy the board, it already has a preflashed image, so you just need to attach the power to it. But knowing how to build and flash the latest firmware might be useful if you want to install the latest version. This process is described in the documentation, so I'm not going to repeat it here.

Most pyboards are powered via a micro USB port, which also provides the communication channel to the board. Some boards, like the one I used in this project, might have an additional socket for a battery that enables standalone usage. (In *Figure 18.1*, that socket is in the top-left corner of the board, close to the micro USB socket.)

Once the board is attached to the computer, it exposes a serial port interface that could be connected using any serial communication client. In my case of using macOS, I normally use the `screen` utility installed from the `brew` ports. To connect, a USB port device file needs to be passed to the tool.

```
$ screen /dev/tty.usbmodem314D375A30372

MicroPython v1.11 on 2019-08-19; PYBv1.1 with STM32F405RG

Type "help()" for more information.

>>>
```

As you can see, it provides the normal Python REPL, which you can start using immediately for board communication. Besides the serial port, the board exposes a mass storage device that can be used to put programs and libraries onto the board. If a microSD card is attached, it is exposed; otherwise, the internal board flash (normally very tiny) is made available for the operating system.

The following shows making the disk visible to the operating system in two different situations: when the board is attached to the microSD card and without it attached.

```
$ df -h
Filesystem    Size  Used  Avail Capacity   iused ifree %iused  Mounted on
/dev/disk2s1 3.7Gi 3.6Mi 3.7Gi       1%       0     0   100%   /Volumes/NO
NAME
$ df -h
Filesystem    Size  Used  Avail Capacity   iused ifree %iused  Mounted on
/dev/disk2s1 95Ki  6.0Ki 89Ki        7%     512     0   100%   /Volumes/
PYBFLASH
```

As you can see, the internal flash is quite small, which might be okay for simple projects, but for larger storage, a microSD card should be used. The Python program also has access to this SD card under the /sd mount point, so your program might use the card for reading and writing. The following is an example of an interactive session showing the filesystem structure from the MicroPython interpreter:

```
MicroPython v1.11 on 2019-08-19; PYBv1.1 with STM32F405RG
Type "help()" for more information.
>>> import os
>>> os.listdir("/")
['flash', 'sd']
>>> os.listdir("/flash")
['boot.py', 'main.py', 'pybcdc.inf', 'README.txt']
>>> os.listdir("/sd")
['.Spotlight-V100', 'zero.py', 'run.py', 'libhw', 'bench.py', 'obs.py']
>>> os.listdir("/sd/libhw")
['sensors.py', '__init__.py', 't1.py', 'nn.py', 'hw_sensors', 'sensor_
buffer.py', 'postproc.py', 'servo.py', 't1zyh.py', 't1zyho.py']
>>> import sys
>>> sys.path
['', '/sd', '/sd/lib', '/flash', '/flash/lib']
>>>
MPY: sync filesystems
MPY: soft reboot
MicroPython v1.11 on 2019-08-19; PYBv1.1 with STM32F405RG
Type "help()" for more information.
>>>
```

On the last line, *Ctrl+D* is pressed, which restarts the interpreter and remounts the filesystem. As you can see from the preceding, the Python path includes both the flash and SD card root, which provides a simple way to check your programs on the hardware: you attach the board, put your program file and libraries on the SD card (or flash), unmount the disk from the operating system (this is needed to flush the buffers), and then after pressing *Ctrl+D* in the interpreter, all your new files become available for execution. If you name your file `boot.py` or `main.py`, it will be executed automatically after the reset or power on.

MicroPython supports most Python 3 dialect with some portion of the standard library available, but still, some modules might be missing or limited. Normally, that's not a big problem, given the limited amount of memory in the board. During my experiments, I noticed only one difference from the standard Python library; it was related to the `collection.deque` class having a different constructor and limited functionality. The rest of the language worked just the same, but of course, you might be unlucky and face some limitations. In that case, you can probably implement a workaround. Finally, MicroPython is an open source project, so everybody can contribute and make it better.

Besides the standard library and the core language compatibility, MicroPython provides a set of extensions to the language and extra libraries to make it convenient to deal with the underlying hardware, as we still work with microcontrollers. All those extensions are covered in the documentation, and we will use some of them later when working with sensors and low-level things like timers. The following is a list of extra functionalities in no particular order:

- GPIO (general-purpose input/output) access to all pins on the board, with access to alternative functions exposed by the hardware. The main class is `pyb.Pin`.
- Access to millisecond and microsecond timers exposed by the hardware.
- The class `pyb.Timer`, which allows you to fire callbacks on timer events.
- Working with hardware PWM channels, which offloads your program from controlling timings. We will use this function to drive the servos.
- Support of common communication protocols like UART (universal asynchronous receiver-transmitter), I²C, SPI, and CAN (Controller Area Network), which use hardware support when it is available.
- Writing interrupt handlers in Python, which has some limitations but is still a very useful way to react to external events.
- MicroPython decorators, `@micropython.native` and `@micropython.viper`, for switching functions from Python bytecode to native instructions, which sometimes might speed up the execution of the code.

- The option to work with the hardware at a very low level (memory registers), which sometimes is necessary to minimize the reaction time.

With all this being said, let's now check how we can implement our robot on MicroPython, starting with the observations provided by the sensor board.

Dealing with sensors

As I mentioned, the sensors that I used in my experiments are mounted on a single board, exposing four chips on the same I²C bus. In the world of electronics, documentation about the components comes as datasheets, in the form of documents (normally PDFs) with detailed specifications of the components, starting with dimensionality, electrical characteristics, applicability limitations, and, of course, a detailed specification of chip functionality. Simple components like LEDs or transistors might have 10-20 pages in the datasheet, but complex electronic components like microcontrollers could have documentation covering thousands of pages. So, it's quite easy to get lost in that amount of information. But datasheets are a very valuable source of knowledge about the details of electronic devices.

In my case, all four sensors mounted on the board were produced by the same company, STMicroelectronics, so they provided a uniform interface and similar semantics for internal registers, which was great, as it simplified my experimentations and dealing with the internals. As I explained earlier in the *The sensors* section, my concrete set of sensors is the following:

- Accelerometer LIS331DLH
- Gyroscope L3G4200D
- Magnetometer LIS3MDL
- Barometer LPS331AP

On average, the datasheet for each sensor is 40 pages long, which is not that much to read. Even better, most of those documents are not relevant to us because besides the basic functionality we really need, they include information about physical and electrical characteristics, and specifications of regimes that we are not really interested in, like interrupts about special conditions. In our case, all the sensor communication will be simple. We will do the following:

- Initialize the sensor by setting the measurement regime (all of them support different scales of measurements)
- Periodically read the values from the sensor

The I²C bus

To get started with our sensors, we need to understand the bus to which all the sensors are connected. In my case, it is I²C, which is a very popular way to establish low-speed communication between digital electronic components. The bus itself is very simple and uses two wires for communication. One wire is the so-called *clock*, which is often abbreviated to SCL and is used to synchronize the slave devices on the bus with the master device. The other wire is used for serial data transfer and is usually called SDA. Given that power is transferred on the other two wires (+3.3V and ground), the sensor board is attached with only four connectors.

A pyboard normally has plenty of +3.3v and ground pins. In terms of SDA and SCL lines, almost any GPIO pins can be used. In my case, I have the following connections:

- The SCL is attached to pin Y11, which is marked on the board as P0, but MicroPython firmware knows it as Y11 or B0.

- The SDA is connected to pin X12. On my board, it is named P1, but the firmware can work with it as X12 or C5.

It might take some time to figure out the connections between the pins on the board and their firmware names, especially if your board is not an official MicroPython one (as in my case). To do that, I used an LED connected to the pin of interest and then turned pins on and off to get the name. You can obtain the list of all pins on your board from REPL, as the `pyb.Pin` class contains the list of all GPIO pins known by the firmware. REPL also supports tab completion, which is handy. The following shows an example of how to get the pin names and change the state of a particular pin. This detection might be automated by reading the GPIO pin state, instead of lighting the LED.

```
MicroPython v1.11 on 2019-08-19; PYBv1.1 with STM32F405RG

Type "help()" for more information.

>>> import pyb

>>> pyb.Pin.board.
__class__          __name__          LED_BLUE          LED_GREEN
LED_RED            LED_YELLOW        MMA_AVDD          MMA_INT
SD                 SD_CK             SD_CMD            SD_D0
SD_D1              SD_D2             SD_D3             SD_SW
SW                 USB_DM            USB_DP            USB_ID
USB_VBUS           X1                X10               X11
X12                X17               X18               X19
X2                 X20               X21               X22
```

X3	X4	X5	X6
X7	X8	X9	Y1
Y10	Y11	Y12	Y2
Y3	Y4	Y5	Y6
Y7	Y8	Y9	

Here, after the *Tab* key is pressed, the interpreter shows the list of GPIO pins that it knows about. To set the particular pin on and off, you just need to create the instance of the class and change the pin state by calling the method `on()` or `off()`.

```
>>> p = pyb.Pin('X11', pyb.Pin.OUT)
>>> p
Pin(Pin.cpu.C4, mode=Pin.OUT)
>>> p.on()
>>> p.off()
>>>
```

The only limitation in pin selection that you'll use for sensor communications is avoiding using pins that are used for microSD card communication. The SD card is attached to the microcontroller with seven pins, so if you occasionally use the same pins for sensors, the SD operations might fail. To figure out which pins are used by the SD card, you can check the documentation for the pyboard. Official pyboards have very nice and clear illustrations about which pins attach to which peripherals. For instance, here is the illustration for the latest PYBv1.1: `http://micropython.org/resources/pybv11-pinout.jpg`.

If you're using an unofficial board, it's not a huge problem. You can figure out pins occupied by the SD card using the firmware. The `pyb.Pin` class in the namespace `board` lists not only the raw GPIO pin names, but also their aliases. All names starting with an `SD` prefix are pins used by SD cards. The following is the list for my board:

```
>>> pyb.Pin.board.SD
Pin(Pin.cpu.A15, mode=Pin.IN, pull=Pin.PULL_UP)
>>> pyb.Pin.board.SD_CK
Pin(Pin.cpu.C12, mode=Pin.ALT, pull=Pin.PULL_UP, af=12)
>>> pyb.Pin.board.SD_CMD
Pin(Pin.cpu.D2, mode=Pin.ALT, pull=Pin.PULL_UP, af=12)
>>> pyb.Pin.board.SD_D0
Pin(Pin.cpu.C8, mode=Pin.ALT, pull=Pin.PULL_UP, af=12)
>>> pyb.Pin.board.SD_D1
```

```
Pin(Pin.cpu.C9, mode=Pin.ALT, pull=Pin.PULL_UP, af=12)
>>> pyb.Pin.board.SD_D2
Pin(Pin.cpu.C10, mode=Pin.ALT, pull=Pin.PULL_UP, af=12)
>>> pyb.Pin.board.SD_D3
Pin(Pin.cpu.C11, mode=Pin.ALT, pull=Pin.PULL_UP, af=12)
```

The pin names with the `cpu` prefix, like `cpu.A15` or `cpu.C12`, are the names of pins you shouldn't use. Of course, if an SD card is not used, you can employ those pins if you need to.

The situation with unofficial MicroPython-compatible boards might be even more interesting if you want to upgrade the firmware, but the board definition file (which defines all the aliases for low-level microcontroller pins) is wrong. It might end up being a non-working SD card after reflash. But that's normally quite easy to fix by changing a couple of lines in the board definition file in the MicroPython source code tree. For my board, I put the fixes to make the SD card work in the `Chapter18/micropython` directory.

Anyway, I hope you've now chosen the pins to use for SDA and SCL. On the sensor board side, the connections are much easier. My board has clear marking of its four pins as V = 3.3V, G = Ground, D = Data, and C = Clock.

Once the sensors are connected, it's very simple to communicate with the board thanks to the existing MicroPython classes. For I²C communication, there is a handy class that can be used to scan the attached sensors and receive and send the data from/to them. If your sensors support a different bus type (it might be SPI, for example), MicroPython might have support for it as well.

This is an example of a device scan of what's attached to the sensor board:

```
>>> from machine import I2C
>>> i2c = I2C(freq=400000, scl='Y11', sda='X12')
>>> i2c.scan()
[24, 28, 92, 104]
```

We have got four numbers, which correspond to the I²C bus ID of every sensor attached to the board. If you get no result or too many numbers, check your wiring. The datasheets for my sensors include default values for those bus IDs and instructions for how to change them, which might be handy if your microcontroller needs to communicate with several identical sensors. In our case, however, this is not needed. If you're interested, there is a lot of information about I²C bus internals available on the Internet. Just to show that there are many things happening under the hood, the next figure shows the oscillogram of SDA and SCL lines during the `i2c.scan()` command.
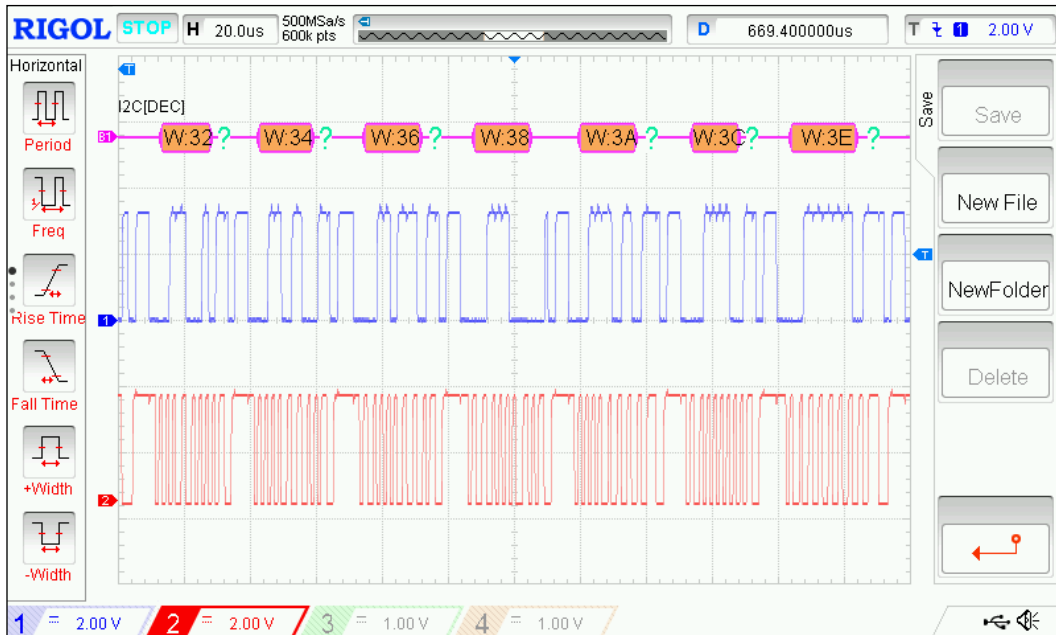
Figure 18.13: I²C communication

Sensor initialization and reading

We've found our sensors' bus IDs, so now let's do something useful with them. In the simplest mode of operation, all those sensors represent some kind of memory device that exposes a set of registers you can read or write using I²C commands. Detailed descriptions of commands and addresses of registers are device-dependent and documented in the datasheet.

I'm not going to give the full information about the registers of all the sensors here, especially given that your sensors might be different. Instead, we will take a look at a couple of simple examples of how to read and write some registers from one sensor. Then, we will look at classes that I've written to simplify communication with those sensors. The classes are available in the `Chapter18/hw/libhw/hw_sensors` package and provide the unified interface to all our sensors.

Before checking the classes, let's check how communication is performed in general. The class `I2C` provides three groups of methods to deal with the I²C bus:

- Primitive methods: `start()`, `stop()`, `readinto()`, and `write()` operate with a bus on a very low level. A detailed explanation of them is well beyond the scope of the book, but overall, those methods allow the bus controller to perform data transfers with attached devices.

- Standard operations: `readfrom()`, `writeto()`, and variations; these methods exchange the data with the specified slave device.

- Memory operations: `readfrom_mem()` and `writeto_mem()` work with slaves that provide a *memory device* interface: a set of registers that you can read or write. This is the case for us, so we will use those methods for communication.

Datasheets describe in detail all the registers exposed by the chip, their addresses, and their functionality. For illustration, let's take the simplest register, WHO_AM_I, which has the address 0x0F on all my sensors. (There is some sort of unification, as all the sensors were produced by the same company.) According to the datasheet, this is a read-only 8-bit register that holds the device identification value. For LIS331DLH, it should have the value 0x32. Let's check.

```
MicroPython v1.11 on 2019-08-19; PYBv1.1 with STM32F405RG
Type "help()" for more information.
>>> from machine import I2C
>>> i2c = I2C(freq=400000, scl='Y11', sda='X12')
>>> i2c.scan()
[24, 28, 92, 104]
>>> i2c.readfrom_mem(24, 0x0F, 1)
b'2'
>>> ord(i2c.readfrom_mem(24, 0x0F, 1))
50
>>> hex(50)
'0x32'
```

The function `readfrom_mem()` accepts three arguments: the device bus ID, the register to read, and the length in bytes. The result is a bytes string, which we convert with the `ord()` function. We are lucky because the first device checked turned out to be the accelerometer with the expected WHO_AM_I=0x32. Let's check other sensors:

```
>>> hex(ord(i2c.readfrom_mem(24, 0x0F, 1)))
'0x32'
>>> hex(ord(i2c.readfrom_mem(28, 0x0F, 1)))
'0x3d'
>>> hex(ord(i2c.readfrom_mem(92, 0x0F, 1)))
'0xbb'
>>> hex(ord(i2c.readfrom_mem(104, 0x0F, 1)))
'0xd3'
>>>
```

As you can see, the values in this register are not related to the I²C bus ID. Writing to the device is performed in a similar way using the `writeto_mem()` method, but instead of the amount of bytes to receive, you need to pass a bytes string with data to be written into the register.

To practice, let's initialize our accelerometer. According to the datasheet, to enable its operation, we need to set up CTRL_REG1 (with address 0x20) to the proper value. This register has a bit-field structure, when different bits in the 8-bit register are responsible for different functions:

- Bits 8-6: Power mode selection; the normal operation corresponds to a bit value of 001.
- Bits 5-4: Data rate selection, which defines the frequency measurements that will be updated in the internal registers. Four different data rates are available: 50Hz, 100Hz, 400Hz, and 1,000Hz. We don't need very frequent updates, so 100Hz will be more than enough. According to the datasheet tables, we need to write the value 01 to switch into 100Hz mode.
- Bit 3: Enables or disables Z axis measurements.
- Bit 2: Enables Y axis measurements.
- Bit 1: Enables X axis measurements.

For our use case, we need all three axes, the data rate is 100Hz, and our power mode is normal. By combining all the bits together, we get a bit value of 0b00101111, which equals to the decimal value 47. This register is marked as read-write, so we can check that our value was indeed updated.

```
>>> ord(i2c.readfrom_mem(24, 0x20, 1))
7
>>> bin(7)
'0b111'
>>> i2c.writeto_mem(24, 0x20, bytes([47]))
>>> ord(i2c.readfrom_mem(24, 0x20, 1))
47
>>> bin(47)
'0b101111'
```

Yes, it works. Initially, it was in the power-down state with 50Hz frequency and all axes enabled. After our initialization, the device starts to measure the acceleration 100 times per second and keeps the measurements in its internal registers. The only thing we need to do is read them periodically. To do that, a bit of math is required, as measurements are stored in 16-bit registers in two's complement form, which is a way to store signed numbers in an unsigned format.

In addition, raw measurements are integers, but the acceleration is normally a float number. So, we need to divide the measurement by the scale before using it. The actual scale depends on the measurement range, which can be selected by writing into another control register, CTRL_REG4. Our accelerometer supports three scales of operation: ±2g, ±4g, and ±8g. Of course, a wider range means a lower precision of measurement, as the number of bits is fixed. As our robot is not going to crash into something, we will use the ±2g mode. This register is also responsible for other settings of the device, but I'm not going to cover them; you can check the datasheet.

The following is an example of how to read the acceleration value from the Z axis. We should measure 9.8 after all the transformations.

```
>>> from machine import I2C
>>> i2c = I2C(freq=400000, scl='Y11', sda='X12')
>>> i2c.writeto_mem(24, 0x20, bytes([47]))
>>> i2c.writeto_mem(24, 0x23, bytes([128]))
>>> v = ord(i2c.readfrom_mem(24, 0x2C, 1)) + 256 * ord(i2c.readfrom_
mem(24, 0x2D, 1))
>>> v
16528
>>> v = -(0x010000-v) if 0x8000 & v else v
>>> v
16528
>>> v /= 16380
>>> v
1.009035
```

For an extra check, I rotated the robot upside down and then onto its side.

```
>>> v = ord(i2c.readfrom_mem(24, 0x2C, 1)) + 256 * ord(i2c.readfrom_
mem(24, 0x2D, 1))
>>> v = -(0x010000-v) if 0x8000 & v else v
>>> v /= 16380
>>> v
-0.809768
>>> v = ord(i2c.readfrom_mem(24, 0x2C, 1)) + 256 * ord(i2c.readfrom_
mem(24, 0x2D, 1))
>>> v = -(0x010000-v) if 0x8000 & v else v
>>> v /= 16380
>>> v
```

```
0.01855922
```

```
>>>
```

Looks good.

Sensor classes and timer reading

Reading values one by one is not very efficient, especially if we want to do it frequently and apply some transformations to them, for example, smoothing values, which might make sense, as raw sensor readings might be noisy. In addition, it would be nice to have a uniform sensor API, so we can work with a sensor without getting too much into the low-level details. In this case, if you have a different sensor, you just need to write a small class to deal with it and plug it into the system.

I implemented such classes, but I'm not going to describe them in detail. I'll just skim through the API and basic design assumptions, and show how they should be used. The complete source code is in `Chapter18/hw`. (This is a directory with the hardware-related part of the project, and its content is supposed to be copied on the microSD card's root.)

First of all, I need to describe the way that sensors are used. Instead of constantly querying sensors in the main loop of the program (also known as the *polling mode*), we will use timer interrupts. We will set up a function that will be called with the specified frequency, say 100 times per second. The job of the function will be to query all the sensors that we need and store the values in a special buffer. This method has the benefit of data being gathered at predictable intervals, rather than on every loop of our program (which might become delayed by IO, NN inference, or other things). But this method has some limitations that you need to account for in your code.

First of all, the interrupt handlers need to be as short as possible and do the minimal amount of work required, and exit. This is motivated by the nature of the interrupt handlers, which *interrupt* the flow of the main code on some event (timer expiration, in our case). If your interrupt handler works longer than the timer interval (1/100th of a second), another interrupt call might be triggered during your routine, which could create a mess.

Another limitation of interrupt handlers is MicroPython-specific: inside the interrupt routine, you can't allocate new memory; you can use only existing preallocated buffers. Sometimes this limitation might lead to surprising results, as Python allocates memory automatically and that might happen during innocent operations, like floating-point calculations. This is caused by the way floating points are implemented in Python and just means that you can't do floating-point math inside interrupt handlers. But with careful implementation, it's not very complicated to meet this requirement and use interrupt handlers.

The following is an example of how we use timer interrupts in MicroPython and an illustration of such unintended memory allocation due to a floating-point operation:

```
>>> f = lambda t: print("Hello!")
>>> t = pyb.Timer(1, freq=2, callback=f)
>>> Hello!
Hello!
Hello!
Hello!
>>> t.deinit()
>>>
>>> f = lambda t: print(2.0/1.0)
>>> t = pyb.Timer(1, freq=2)
>>> t.callback(f)
>>> uncaught exception in Timer(1) interrupt handler
MemoryError:
```

The class that implements the sensor query using timer interrupts is in the module `Chapter18/hw/libhw/sensor_buffer.py`. Its logic might be tricky due to the timer callback limitations, but the interface is quite straightforward. The class `SensorsBuffer` is supposed to be created with a list of sensor object instances (those classes will be covered later), the index of the timer to be used, the polling frequency, the size of one batch, and the count of batches in the circular buffer. Every batch is a list of subsequent measures from all the sensors that we want to query. The buffer is a list of batches, so the capacity of the whole buffer is defined on buffer creation. Of course, our memory is limited, so we need to be careful with allocating too much memory for our sensor reading. For example, we have an accelerometer that we want to query 100 times every second. We also know that we will process the buffer at least once per second. In that case, we can create the buffer with a batch size that equals 10 and the total number of batches as 10. With those settings, the buffer will keep one second of data readings.

Here is an example of how this buffer could be created. For this example to work, you need to copy the `Chapter18/hw` directory into the microSD card root and have the same type of sensor.

```
>>> from machine import I2C
>>> i2c = I2C(freq=400000, scl='Y11', sda='X12')
>>> i2c.scan()
[24, 28, 92, 104]
>>> from libhw.hw_sensors.lis331dlh import Lis331DLH
```

```
>>> accel = Lis331DLH(i2c)
>>> accel.query(Lis331DLH.decode)
[0.01660562, -0.03028083, 1.029548]
```

That's our accelerometer sensor. Let's attach it to the buffer:

```
>>> from libhw.sensor_buffer import SensorsBuffer
>>> buf = SensorsBuffer([accel], 1, freq=100, batch_size=10, buffer_
size=10)
>>> buf.start()
>>> for b in buf:
...     print(b)
...
[bytearray(b''\x00\x10\xfe0B'), bytearray(b'0\x00@\xfe\xd0A'),
bytearray(b'\xb0\x00\xf0\xfd\xb0A'), bytearray(b'\x80\x00\x00\xfe\
xc0A'), bytearray(b'0\x00 \xfe\xc0A'), bytearray(b'0\x00p\xfe\x90A'),
bytearray(b'\xa0\x000\xfe\xa0A'), bytearray(b'P\x00 \xfe\x10B'),
bytearray(b'\xa0\x000\xfe\xe0A'), bytearray(b'0\x00 \xfe\x10B')]
```

The buffer provides the iterator interface, generating all complete batches stored in the buffer. Every entry in the batch is a raw sensor reading, as we got it from the sensor's register. (Because we can't use floating-point operations in the timer callback, we can't do a transformation of raw bytes into floats.) This transformation needs to be done during the reading.

```
>>> for b in buf:
...     for v in b:
...         data = Lis331DLH.decode(v)
...         print(data)
...
[0.06544567, -0.4610501, 0.9094017]
[0.05567766, -0.4620269, 0.9084249]
[0.05763126, -0.4551893, 0.9045177]
[0.06446887, -0.4630037, 0.9054945]
[0.05665446, -0.4590965, 0.9064713]
[0.06056166, -0.4561661, 0.9025641]
[0.05958486, -0.4581197, 0.9064713]
[0.06056166, -0.4561661, 0.9113553]
[0.05958486, -0.4581197, 0.9074481]
[0.05958486, -0.4649573, 0.9094017]
```

The main program needs to fetch the data from the buffer periodically; otherwise, new values from the sensors might overwrite the existing data (as the buffer is circular).

Let's now discuss the common sensor interface, which is fairly simple. On the top is the abstract class `Sensor`, defined in `libhw/sensors.py`. The base class is responsible for allocating the buffer, which will be used to store the sensor's readings, and defines the following abstract methods, which have to be redefined by subclasses:

- `__len__()` has to return the length of raw sensor readings in bytes. Different sensors provide different amounts of data; for example, my accelerometer returns three 16-bit values for each axis, but my magnetometer returns four values because it contains an internal temperature register, whose readings are also exposed.
- `refresh()` has to fetch fresh data from underlying sensors and store it inside the internal buffer.
- `decode(buf)`, which is a class method, needs to convert byte representation into the list of floats.

Observations

We are slowly but surely getting to our primary goal: obtaining the observations from the sensors. As the first, very simple example, I'll use only accelerometer readings, which will be used to estimate the roll and pitch angles of our robot. "Roll" means the rotation around the axis along the robot (in my sensor board location, this is a Y axis), and "pitch" is the rotation around the axis horizontally perpendicular to the roll axis, which defines the incline of the robot (which is the X axis in my case).

The third angle, the so-called yaw angle or rotation around the Z axis, is not possible to reliably estimate from the accelerometer alone. However, there are methods to find all three rotation angles using the combined readings from the accelerometer and gyroscope, but this implementation is left as an exercise for the advanced reader. You might find this article useful: `https://www.monocilindro.com/2016/06/04/how-to-calculate-tait-bryan-angles-acceleration-and-gyroscope-sensors-signal-fusion/`.

In my code, I used the simple method that gives only the roll and pitch angles by comparing accelerometer readings on different angles. The postprocessing code is in `Chapter18/hw/libhw/postproc.py`.

```
def pitch_roll_simple(gx, gy, gz):
    g_xz = math.sqrt(gx*gx + gz*gz)
    pitch = math.atan2(gy, g_xz)
```

```
        roll = math.atan2(-gx, gz)
        return pitch, roll
```

This function calculates the pitch and roll from float acceleration readings.

```
class PostPitchRoll:
    SMOOTH_WINDOW = 50

    def __init__(self, buffer, pad_yaw):
        assert isinstance(buffer, SensorsBuffer)
        assert len(buffer.sensors) == 1
        assert isinstance(buffer.sensors[0],
                          hw_sensors.lis331dlh.Lis331DLH)
        self.buffer = buffer
        self.smoother = Smoother(self.SMOOTH_WINDOW, components=3)
        self.pad_yaw = pad_yaw

    def __iter__(self):
        for b_list in self.buffer:
            for b in b_list:
                data = hw_sensors.lis331dlh.Lis331DLH.decode(b)
                self.smoother.push(data)
                pitch, roll = \
                    pitch_roll_simple(*self.smoother.values())
                res = [pitch, roll]
                if self.pad_yaw:
                    res.append(0.0)
                yield res
```

This class takes the buffer, which should have only the accelerometer sensor attached, and converts the data from the buffer into pitch and roll values. Before calculating those angle components, the readings are smoothed by the simple moving average method, which is implemented by the `Smoother` class in the same `postproc.py` file. The method can't estimate the yaw angle, but our model expects to have it, so we put zero as the third component.

For testing this code, I have a small utility in `Chapter18/hw/obs.py`, which takes all the classes, but just shows the pitch and roll angles on the console. Its code is very simple and is shown here:

```
import pyb
from machine import I2C
from libhw.hw_sensors import lis331dlh as lis
from libhw.sensor_buffer import SensorsBuffer
from libhw.postproc import PostPitchRoll
```

```
    SDA = 'X12'
    SCL = 'Y11'

    def run():
        i2c = I2C(freq=400000, scl=SCL, sda=SDA)
        acc = lis.Lis331DLH(i2c)
        buf = SensorsBuffer([acc], timer_index=1, freq=100,
                            batch_size=10, buffer_size=100)
        post = PostPitchRoll(buf, pad_yaw=True)
        buf.start()
        try:
            while True:
                for v in post:
                    print("pitch=%s, roll=%s, yaw=%s" % tuple(v))
        finally:
            buf.stop()
```

Let's test it on our hardware.

```
>>> import obs
>>> obs.run()
pitch=-0.4448922, roll=-0.2352999, yaw=0.0
pitch=-0.4403271, roll=-0.2364806, yaw=0.0
pitch=-0.440678, roll=-0.237506, yaw=0.0
pitch=-0.4407347, roll=-0.2369987, yaw=0.0
pitch=-0.4420413, roll=-0.2360859, yaw=0.0
pitch=-0.4424292, roll=-0.2354495, yaw=0.0
pitch=-0.4430317, roll=-0.2366674, yaw=0.0
```

The shown angles are in radians and should change as you incline the sensor board in different directions.

Driving servos

Our servos are simple and cheap, so they are being controlled by PWM, the most popular method in this class of actuators. The idea of this method is very simple: the digital signal controlling the servo angle has the form of pulses with a fixed period, which most often is 50Hz. As our signal is digital, it could be in only the 0 or 1 state, but the ratio between the time when the signal is in state 1 and the time it is in state 0 may vary. This ratio, also called the *duty cycle*, controls the angle of the servo.

As this method is very popular in toy models, aviation, robotics, and other applications, there is some degree of compatibility here, so you can replace the servo in your radio-controlled car model and be more or less sure it will be compatible with the rest of the car's electronics. As usual, it might have some variations, but normally, it just works.

The standard parameters of position-controlled servos can be easily found on the internet and usually are the following: the pulse interval is 50Hz (which is 20 ms), the zero angle corresponds to ~3% duty cycle (which is approximately 600 us), and the maximum angle (normally 180°) is ~12% duty cycle (which is 2.4 ms). The concrete timings may slightly vary from one servo model to another. In my case, I found that the minimum angle is reached at 2.3% of the duty cycle, and the maximum at about 12.6%.

To illustrate this, *Figure 18.14* and *Figure 18.15* show oscillograms of three servos set in different positions. The top one is at the minimum angle, the middle one is at the 50% position, and the bottom one is at the maximum angle.
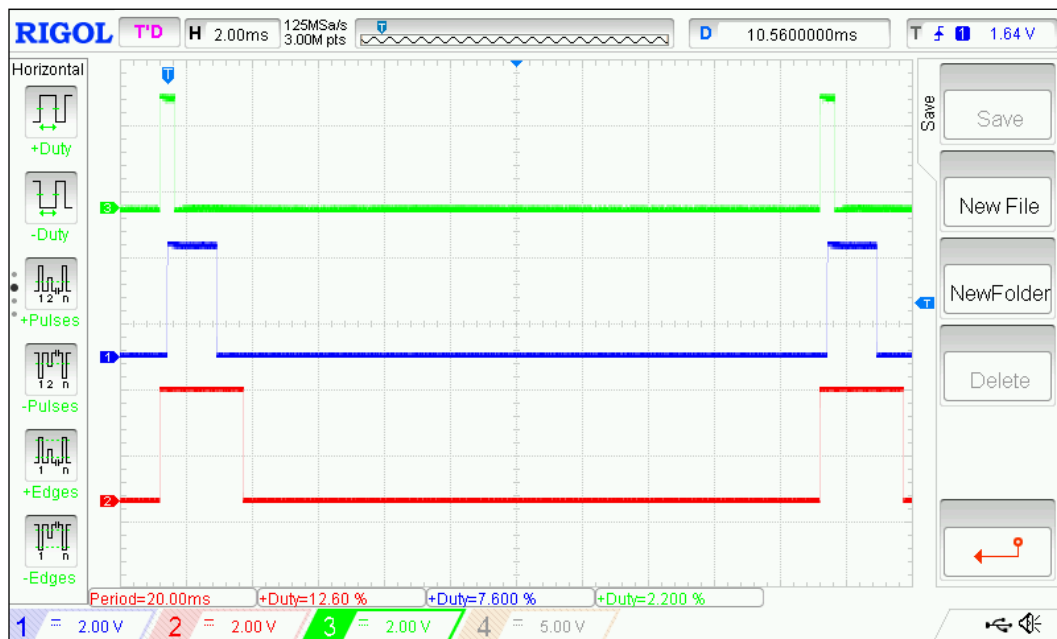
Figure 18.14: Three servos at different positions

Figure 18.15: The same three servos, but with the signal zoomed in and the bottom legend switched to timings

Before we get into the software part, something needs to be said about the hardware. Our servos have three connectors: power (normally a red wire), ground (black), and control (white). In terms of voltage, the servos work fine with +3.3V or +5V. The only possible issue is that the servos include small motors, which might drain the power from the USB port.

My measurement shows that the peak current from four servos doesn't exceed 0.7A, which is fine for most USB ports, but as this drain happens in short spikes (when a servo needs to rotate suddenly), this might introduce noise to the USB signal, and communication between the computer and the board might be lost.

To overcome this, several simple solutions exist. First of all, the servos could be powered from a different USB port. This will increase the number of wires to the robot, but it will help. Another option that I tried, which worked very well, was to add an electrolytic capacitor between the servo's power and ground wires. I used a capacitor of 100uF, which works as a filter and smooths power spikes. A capacitor in combination with a short USB cable solved my problem, and the connection worked reliably even during sudden moves of all four servos.

Maybe an even better solution would be to switch to a wireless connection to the robot and add the battery to give it full wireless freedom. That could be included in my next version.

Now the question is: how do we generate such pulses to drive our four servos? The most obvious, but not very reliable, way is to drive pins attached to the servos up and down in the main loop. The issue with this approach is that the timings need to be precise: if our pulses are delayed for some reason, the servos might suddenly jump to a wrong position. As Python is not a very fast language, it might be tricky to generate stable pulses of a specific width.

Servo control is not our only problem. Besides controlling the servos, we will need to query the sensors, process their readings, and do inferences using our NN. Juggling all those tasks and ensuring precise timing in four PWM signals might be too much complexity.

A better approach will be to use timer interrupts, which can be set up to trigger periodically, check the desired servo pin signal, and change it. I implemented this approach at the beginning of my experiments, and it worked fairly well. If you're curious, the source of the class that can drive several servos is in module `Chapter18/hw/libhw/servo.py` and is called `ServoBrainOld`. The code uses multiple timers: one timer is for driving 50Hz pulses, and several timers (one per servo being controlled) are responsible for measuring the time the signal should be in the high state. This method worked, but when checking it in conjunction with sensor readings, servo control wasn't very reliable. The servos sometimes jumped to a wrong position. Probably, this was caused by timer interrupts overlapping or some bug in my calculations.

To overcome this, I found a third, and from my point of view ideal, solution to servo control using the hardware PWM support implemented in the microcontroller. This also requires timers to be set up, but in this case, we will use timers not to trigger our function in a specific interval, but to employ the extra capabilities of timers to produce PWM signals directly on their own. The only thing that is needed from our side is to properly configure the timer and set the duty cycle that we want it to generate. From that moment, all the pulses will be generated by the hardware without any further control from our side. If we want the servos to jump to a different position, we just need to set a different duty cycle of a timer.

To understand how this works, we need to step a bit deeper into the microcontroller hardware that we're using. STM32F4xx microcontrollers can have up to 14 different timers, and some of them support up to four *channels*, each of which could be configured in a specific way. For instance, one channel of the timer might generate PWM pulses of a specific interval on some pin, but another channel of the same timer might produce pulses of a different length, which potentially allows us to drive dozens of servos, with zero CPUs involved, just using hardware. Controlling timer channels is supported by MicroPython, although it doesn't have very detailed documentation.

The following example shows how to drive one servo attached to the pin B6 (which on my board is marked as P26):

```
>>> import pyb
>>> t = pyb.Timer(4, freq=50)
>>> p = pyb.Pin("B6", pyb.Pin.OUT)
>>> ch = t.channel(1, pyb.Timer.PWM, pin=p)
>>> ch
TimerChannel(timer=4, channel=1, mode=PWM)
>>> p
Pin(Pin.cpu.B6, mode=Pin.ALT, af=Pin.AF2_TIM4)
>>> t
Timer(4, freq=50, prescaler=124, period=13439, mode=UP, div=1)
>>> ch.pulse_width_percent(3)
>>> ch.pulse_width_percent(12)
>>> ch.pulse_width_percent((12+3)/2)
```

After the line `ch.pulse_width_percent(3)`, the servo should move to the minimal angle; the second command will jump it into the maximum position; and the final line will set it to the middle. This is extremely simple and efficient.

To find which pins have a specific timer and channel index, you can use the datasheet on the specific microcontroller, which includes concrete hardware details about the concrete chip. In contrast to the family reference manual (which, in the case of STM32F405 has 1,700 pages), the datasheet (`https://www.st.com/resource/en/datasheet/stm32f405rg.pdf`) has only 200 pages and in *Table 9* states all the timer and channel indices associated with the microcontroller pin (I want to thank Ernest Gungl, who pointed me to the proper way to find timer and channel indices).

My set of pins is saved as a constant in the `servo.py` file and maps the pin name to the pair (timer, channel).

```
_PINS_TO_TIMER = {
    "B6": (4, 1),
    "B7": (4, 2),
    "B8": (4, 3),
    "B9": (4, 4),
    "B10": (2, 3),
    "B11": (2, 4),
}
```

I have also found that putting four servos on one timer is not the best idea, as they do not always work. So, in my hardware, I have two servos attached to pins B6 and B7, which are driven by timer 4, and two servos controlled by timer 2 on pins B10 and B11.

The class that implements this method is called `ServoBrain` and it sits in the `servo.py` module. As it is quite simple, I'm not going to put it here; I will just show how it is supposed to be used.

```
>>> from libhw import servo
>>> PINS = ["B6", "B7", "B10", "B11"]
>>> ch = servo.pins_to_timer_channels(PINS)
>>> ch
[('B6', (4, 1)), ('B7', (4, 2)), ('B10', (2, 3)), ('B11', (2, 4))]
>>> brain = servo.ServoBrain()
>>> brain.init(ch)
>>> brain.positions
[0.0, 0.0, 0.0, 0.0]
>>> brain.positions = [0.5, 1, 0.2, 0]
>>> brain.deinit()
>>>
```

At first, we build the mapping of our pin names to timers/channels. Then, the servo controller is created and initialized. After that, we read the last positions and change them using the `positions` property. The position of each servo is controlled by a float from 0 to 1.

In addition, the file `Chapter18/hw/zero.py` contains several helper methods that can be used to experiment with servos, like slowly rotating a specific leg (function `cycle()`), jumping the robot from a lying to a standing position (function `stand()`), or setting the servos to specific positions and keeping them.

A final point about the servos in our robot is that in my design, the front servos are flipped upside down, which means that their direction needs to be reversed. The `ServoBrain` class supports this by accepting an optional Boolean list in the `init` method.

It looks like you know how to query sensors and change the position of our servos. The missing part is how to apply our model, which is supposed to glue observations to actions. Let's do it next.

Moving the model to hardware

The model, trained using the DDPG method, includes two parts: the actor and critic networks. The actor takes the observations vector and returns the actions that we need to take. The critic takes the observations and the actions returned by the actor and predicts the value that an action in a state might bring us. When the training is done, we don't need the critic network anymore. The final outcome is the actor model alone.

Under the hood, our actor model is just a bunch of matrices that does the transformation of the input vector with stacked observations to produce a four-component action vector, every value of which is the rotation angle of the corresponding servo motor. This is the structure of our actor network shown by PyTorch:

```
DDPGActor(
  (net): Sequential(
    (0): Linear(in_features=28, out_features=20, bias=True)
    (1): ReLU()
    (2): Linear(in_features=20, out_features=4, bias=True)
    (3): Tanh()
  )
)
```

When we worked with an emulator on a large computer, we imported PyTorch, loaded the weights, and then used the model class to do the inference. Unfortunately, in MicroPython, we can't do the same because we don't have PyTorch ported for microcontrollers, and even if it was ported, our memory limits are too low to use PyTorch. To overcome this, we need to go to a lower level in our abstractions and work with our model as matrices. At that level, to do the inference, we need to multiply the observation vector (having 7×4=28 numbers) to the weight matrix of the first layer (28×20), add its bias vector, and apply rectified linear unit (ReLU) nonlinearity. This will be the vector of 20 numbers, which should be passed to the second layer, again multiplying it to a weight matrix of size 20×4. We will then add bias and after tanh nonlinearity, it will give us the output.

There are libraries like `numpy` that have very efficient and highly optimized matrix multiplication routines, which is not a surprise, as those routines have a very long history (decades!) of the brightest computer scientists optimizing them for the best performance. We're also not very lucky here, as NumPy is not available for MicroPython either. During my experiments, I did some quick research and found the only library that implements matrix operations on raw Python, `https://github.com/jalawson/ulinalg`, which is quite old and hasn't been maintained for a while.

After some quick benchmarks, it turned out that my very naïve matrix multiplication function is about 10 times faster than the version provided by `ulinalg` (mostly by avoiding copying data into temporary lists).

The function used during the model inference is in module `Chapter18/hw/libhw/nn.py`. It's just one screen of code, so let's take a look.

```python
import math as m

def matmul(a, b):
    res = []
    for r_idx, a_r in enumerate(a):
        res_row = [
            sum([a_v * b_r[b_idx] for a_v, b_r in zip(a_r, b)])
            for b_idx in range(len(b[0]))
        ]
        res.append(res_row)
    return res
```

The function `matmul()` produces a very simple and straightforward implementation of matrix multiplication. To better support our model's shapes, the first argument is expected to have a shape of (n, m), and the second matrix needs to be of the shape (m, k); then, the produced matrix will have a shape of (n, k). The following is a very simple example of its use. The module `t1` contains the exported model, which will be covered later.

```
>>> from libhw import nn, t1
>>> len(t1.WEIGHTS[0][0])
20
>>> len(t1.WEIGHTS[0][0][0])
28
>>> b = [[0.0]]*28
>>> nn.matmul(t1.WEIGHTS[0][0], b)
[[0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0],
[0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0]]
>>> r = nn.matmul(t1.WEIGHTS[0][0], b)
>>> len(r)
20
>>>
```

The next function in the `nn.py` module implements the in-place sum of two matrices. The second matrix is transposed during the operation. This is not very efficient, as the only use of this function is to add the bias vectors, which might be transposed during the model export, but I haven't done much optimization of the code. If you wish, you can experiment with it.

```python
def matadd_t(a, b):
    for a_idx, r_a in enumerate(a):
        for idx in range(len(r_a)):
            r_a[idx] += b[idx][a_idx]
    return a
```

The following is an illustration of how this function is used:

```
>>> len(t1.WEIGHTS[0][1])
1
>>> len(t1.WEIGHTS[0][1][0])
20
>>> nn.matadd_t(r, t1.WEIGHTS[0][1])
[[0.7033747], [0.9434632], [-0.4041394], [0.6279096], [0.1745763],
[0.9096519], [0.375916], [0.6591344], [0.1707696], [0.5205367],
[0.2283022], [-0.1557583], [0.1096208], [-0.295357], [0.636062],
[0.2571596], [0.7055011], [0.7361195], [0.3907547], [0.6808118]]
>>> r
[[0.7033747], [0.9434632], [-0.4041394], [0.6279096], [0.1745763],
[0.9096519], [0.375916], [0.6591344], [0.1707696], [0.5205367],
[0.2283022], [-0.1557583], [0.1096208], [-0.295357], [0.636062],
[0.2571596], [0.7055011], [0.7361195], [0.3907547], [0.6808118]]
```

The operation is in-place to avoid allocation of the extra memory. The next group of methods is used for nonlinearities that we have in the model.

```python
def apply(m, f):
    for r in m:
        for idx, v in enumerate(r):
            r[idx] = f(v)
    return m
```

This method does an in-place element-wise application of the function to the matrix.

```python
def relu(x):
    return apply(x, lambda v: 0.0 if v < 0.0 else v)


def tanh(x):
    return apply(x, m.tanh)
```

The two preceding functions use the `apply()` method to implement ReLU and tanh nonlinearities, and they should be obvious.

```
def linear(x, w_pair):
    w, b = w_pair
    return matadd_t(matmul(w, x), b)
```

The last method combines the `matmul` and `matadd_t` functions to apply linear layer transformation to the input matrix x. The second argument, `w_pair`, is expected to contain the tuple with the weight matrix and the bias.

The model export

To convert the model obtained from the training, I wrote a special utility: `Chapter18/export_model.py`. Its task is to load the model file and generate the Python module that contains the weights of the model and several functions to transform the observation vector into the actions to be executed. This is a classic example of metaprogramming, when one program generates another. The logic is not very complicated and is shown here:

```
import pathlib
import argparse
import torch
import torch.nn as nn
from lib import ddpg

DEFAULT_INPUT_DIM = 28
ACTIONS_DIM = 4
```

Initially, we import the required modules and define the constants. The utility is not very generic, and the output dimensions are hardcoded. (The input size could be redefined from the command line.)

```
def write_prefix(fd):
    fd.write("""from . import nn


""")
```

The function `write_prefix` is supposed to write the header of the module and just writes the import of the only module that we will use. To fulfill the PEP8 standards, we write the extra empty line, although it might be redundant.

```
def write_weights(fd, weights):
    fd.write("WEIGHTS = [\n")
    for w, b in weights:
        fd.write("(%s, [%s]),\n" % (
```

```
            w.tolist(),   b.tolist()
        ))
    fd.write("]\n")
```

The next function, `write_weights`, is supposed to write the list declaration of our NN parameters. The input argument `weights` should be a list of tuples, where each tuple contains parameters for a specific layer. The tuple should hold two NumPy arrays, the first of which is the matrix with the linear layer weights, and the second of which is the bias vector.

```
def write_forward_pass(fd, forward_pass):
    fd.write("""

def forward(x):
""")

    for f in forward_pass:
        fd.write("    %s\n" % f)

    fd.write("    return x\n")
```

The function `write_forward_pass` generates the `forward()` function in the module, which is supposed to transform the input vector into the NN's output. In arguments, it accepts the file descriptor to feed the generated text and the list of steps that our network consists of. The way this list is generated will be described later, but for now, the important point is that it has to contain valid Python lines to be executed one by one, transforming the input, x, according to the NN's structure.

```
def write_suffix(fd, input_dim):
    fd.write(f"""

def test():
    x = [[0.0]] * {input_dim}
    y = forward(x)
    print(y)


def show():
    for idx, (w, b) in enumerate(WEIGHTS):
        print("Layer %d:" % (idx+1))
        print("W: (%d, %d), B: (%d, %d)" % (len(w), len(w[0]),
            len(b), len(b[0])))

""")
```

The final function in the tool generates two functions of the module: `test()`, which applies the network to the zero vector, and `show()`, which prints the structure of the weights.

Finally, let's take a look at the main body of the utility:

```python
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-m", "--model", required=True,
        help="Model data file to be exported")
    parser.add_argument("-o", "--output", required=True,
        help="Name of output python file to be created")
    parser.add_argument("--input-dim", type=int,
                        default=DEFAULT_INPUT_DIM,
                        help="Dimension of the input,
                        default=%s" % DEFAULT_INPUT_DIM)
    args = parser.parse_args()
    output_path = pathlib.Path(args.output)
```

In the beginning, we parse arguments of the tool, which allow us to specify the file name with the model to be converted and the output Python module to be produced. In addition, we redefine the input dimensionality.

```python
    act_net = ddpg.DDPGActor(args.input_dim, ACTIONS_DIM)
    act_net.load_state_dict(torch.load(
        args.model, map_location=lambda storage, loc: storage))
```

The next step is to construct our actor module and load weights from the file given in the arguments.

```python
    weights_data = []
    forward_pass = []

    for m in act_net.net:
        if isinstance(m, nn.Linear):
            w = [m.weight.detach().numpy(),
                m.bias.detach().numpy()]
            forward_pass.append(
                f"x = nn.linear(x, WEIGHTS[{len(weights_data)}])")
            weights_data.append(w)
        elif isinstance(m, nn.ReLU):
            forward_pass.append("x = nn.relu(x)")
        elif isinstance(m, nn.Tanh):
            forward_pass.append("x = nn.tanh(x)")
        else:
            print('Unsupported layer! %s' % m)
```

Now we're going to the core of the utility logic. We iterate the actor network and inspect its structure. According to the classes of every layer, we populate two lists: `weights_data` and `forward_pass`. The first list contains tuples with the `Linear` layer's parameters: `weight` and `bias`. At the moment, only the `Linear` layer is supported, so if you want the convolution layer to be exported, you will need to extend the utility. Another list, `forward_pass`, contains strings with Python transformations to be applied. This list will be passed to the `write_forward_pass` function that we have just seen. In those transformations, we're using methods from the `nn.py` module.

```
with output_path.open("wt", encoding='utf-8') as fd_out:
    write_prefix(fd_out)
    write_weights(fd_out, weights_data)
    write_forward_pass(fd_out, forward_pass)
    write_suffix(fd_out, args.input_dim)
```

Finally, we open the output file and write its sections one by one, using the functions that we have just defined. The module produced could be copied to the microSD card of our robot and used to convert observations to actions.

Benchmarks

Initially, I was quite doubtful about the ability of a microcontroller to do an NN inference in a timely manner, so I wrote a simple tool to check the speed of the inference. The tool is in the file `Chapter18/hw/bench.py` and it just performs the inference in a loop, measuring the time it takes on every loop. The work of this program on my hardware is shown next:

```
>>> import bench
>>> bench.run()
17096
19092
16911
19100
16902
19105
16962
```

The numbers shown in the loop are microseconds, so the inference takes 17-19 milliseconds, which corresponds to 53-59 inferences per second. This is pleasantly surprising, as it is way more than we need. Of course, our model is very simple and contains just 664 parameters. In the case of larger models, some optimizations might be needed. Now, I want to show possible ways to make your model faster and use less memory.

First of all, the performance might be improved by using MicroPython decorators to produce the native code of the microcontroller instead of Python bytecode. Those decorators are `@micropython.native` and `@micropython.viper`, and they are covered in the documentation: `http://docs.micropython.org/en/latest/reference/speed_python.html`.

Another possible way to significantly speed up the code would be to move the inference from Python to firmware. In that case, the matrix multiplications would need to be implemented in C, and the weight could be built into the firmware. This method has the additional benefit of using the native floating point operations of the underlying hardware, rather than using emulated Python floating point. This method might be a bit complicated, but it promises the largest improvements in both performance and memory usage. Information about the use of native floating-point using assembler is given in this documentation section: `http://docs.micropython.org/en/latest/reference/isr_rules.html#overcoming-the-float-limitation`. (Yes, you can write MicroPython functions in assembly!)

As a simpler alternative, you might consider using fixed-point operations rather than floating point. In this case, your transformation might be done in integers and only on the final step converted into the floating point. But you need to be careful with possible overflow in the network.

A different problem you might encounter is the amount of memory available for the weights. In my code, I stored weights in the module text, so during the import, this text needed to be parsed by the interpreter to get the Python representation of the weights. With a larger model, you might quickly reach the limits of the interpreter and available memory. The MicroPython documentation outlines possible ways to decrease the amount of memory required by the parser and intermediate representation: `http://docs.micropython.org/en/latest/reference/packages.html` (see the section *Cross-installing packages with freezing*). This covers the way to precompile your module and bundle it into the firmware so that it stops occupying the RAM and sits in the flash ROM. This makes total sense, as our weights are not supposed to be modified.

Combining everything

Wrapping up all the preparations, we want to combine all three pieces into one: observations from the sensors are fed into the network, which produces the actions to be executed by the servos. The code that does exactly that is in `Chapter18/hw/run.py`, and it needs the `libhw` package and the actor model converted into Python code. This Python source could have any name you wish, but it needs to be placed into the `libhw` directory on the microSD card.

For my demonstration, I've included my three models, named `t1.py`, `t1zyh.py`, and `t1zyho.py`. If you want, you can use them. The program `run.py` defines the only function, `run()`, which accepts one argument: the name of the module to be used for inference. It should be the name of the module and not the file name, in other words, without the `.py` suffix.

The source code of `run.py` is quite short, so let's take a look at it.

```python
import pyb
import utime
from machine import I2C
from libhw.hw_sensors import lis331dlh as lis
from libhw.sensor_buffer import SensorsBuffer
from libhw.postproc import PostPitchRoll
from libhw import servo


SDA = 'X12'
SCL = 'Y11'
PINS = ["B6", "B7", "B10", "B11"]
INV = [True, False, True, False]
STACK_OBS = 4
```

At the beginning, it imports everything we need and defines constants with the pin names of the sensor board and servos, the list of legs to be inverted in a particular direction, and the depth of our observation stack. You might need to tweak those parameters to meet your hardware.

```python
def do_import(module_name):
    res = __import__("libhw.%s" % module_name,
                     globals(), locals(), [module_name])
    return res
```

Then, a short, but a bit tricky, helper function is defined. We don't want to constantly change the name of the module to be used for inference, so this name will be passed as a variable. Python provides a way to import the module, the name of which is given in a variable; this way is called the `importlib` module. However, MicroPython lacks this module, so another approach has to be used. This approach uses the `__import__` built-in function, which can do the same thing. So, this `do_import` helper calls this built-in function and returns the module instance as a result.

```python
def run(model_name):
    model = do_import(model_name)

    i2c = I2C(freq=400000, scl=SCL, sda=SDA)
    acc = lis.Lis331DLH(i2c)
    buf = SensorsBuffer([acc], timer_index=1, freq=100,
```

```
                    batch_size=10, buffer_size=100)
    post = PostPitchRoll(buf, pad_yaw=True)
    buf.start()
    ch = servo.pins_to_timer_channels(PINS)
    brain = servo.ServoBrain()
    brain.init(ch, inversions=INV)
```

In the beginning of the main function, we import the inference module and construct our hardware classes: we connect to the I²C bus, create a sensor wrapped into the sensor buffer, and create a servo controller.

```
    obs = []
    obs_len = STACK_OBS*(3+4)
    frames = 0
    frame_time = 0
    ts = utime.ticks_ms()

    try:
        while True:
            for v in post:
                for n in brain.positions:
                    obs.append([n])
                for n in v:
                    obs.append([n])
                obs = obs[-obs_len:]
```

In the loop, we construct the observation vector, which consists of the four current positions of the servos and the three values of roll, pitch, and yaw generated by the postprocessor. Those seven values are stacked to form 28 values in observations. To get them, we organize a sliding window to keep the four most recent observations.

```
                if len(obs) == obs_len:
                    frames += 1
                    frame_time += utime.ticks_diff(
                                utime.ticks_ms(), ts)
                    ts = utime.ticks_ms()
                    res = model.forward(obs)
                    pos = [v[0] for v in res]
                    print("%s, FPS: %.3f" % (pos,
                            frames*1000/frame_time))
                    brain.positions = pos
```

When the observations are ready, we do the inference using the `model.forward()` method. Then, the result is transformed a little and assigned to the servo controller's `positions` property to drive the servos.

The performance is also monitored, just to satisfy our curiosity.

```
finally:
    buf.stop()
    brain.deinit()
```

Finally, when *Ctrl+C* is pressed, we stop sensor buffer population and the timers driving the servos.

Here is the part of the session on my hardware:

```
>>> import run
>>> run.run("t1")
[0.02104034, 0.5123497, 0.6770669, 0.2932276], FPS: 22.774
[-0.5350959, -0.8016349, -0.3589837, 0.454527], FPS: 22.945
[0.1015142, 0.9643133, 0.5971705, 0.5618412], FPS: 23.132
[-0.6485986, -0.7048597, -0.4380577, 0.5697376], FPS: 23.256
[-0.2681193, 0.9673722, 0.5179501, 0.4708525], FPS: 23.401
```

As you can see, the full code is able to do everything about 23 times per second, which is not that bad for such limited hardware.

Policy experiments

The first model that I trained was with the `Height` objective and without zeroing the yaw component. A video of the robot executing the policy is available here: `https://www.youtube.com/watch?v=u5rDogVYs9E`. The movements are not very natural. In particular, the front-right leg is not moving at all. This model is available in the source tree as `Chapter18/hw/libhw/t1.py`.

As this might be related to the yaw observation component, which is different during the training and inference, the model was retrained with the `--zero-yaw` command-line option. The result is a bit better: all legs are now moving, but the robot's actions are still not very stable. The video is here: `https://www.youtube.com/watch?v=1JVVnWNRi9k`. The model used is in `Chapter18/hw/libhw/t1zyh.py`.

The third experiment was done with a different training objective, `HeightOrient`, which not only takes into account the height of the model, but also checks that the body of the robot is parallel to the ground. The model file is available in `Chapter18/hw/libhw/t1zyho.py`. This model finally produced the outcome I expected: the robot is able to stand and can be kept in the standing position. The video is here: `https://www.youtube.com/watch?v=_eA4Dq8FMmo`.

I also experimented a bit with the `MoveForward` objective, but even in simulations, the resulting policy looks weird. Very likely, this is caused by bad model parameters that need to be improved to give a more realistic simulation of the robot. But during the experiments, a quite amusing result was achieved. The initial version of the code contained a mistake in the reward objective, so the robot was rewarded for moving sideways, instead of for forward movements. The video of the best policy is available here: `https://www.youtube.com/watch?v=JFGFB8pDY0Y`. Obviously, the optimization was able to find the policy to maximize the wrong reward objective, which gives us another illustration of how important the reward is for RL problems.

Summary

Thanks for reaching the end! I hope you enjoyed reading this chapter as much as I enjoyed writing it. This field is very interesting; we have just touched on it a little, but I hope that this chapter will show you a direction for your own experiments and projects. The goal of the chapter wasn't building a robot that will stand, as this could be done in a much easier and more efficient way; the true goal was to show how the RL way of thinking can be applied to robotics problems, and how you can do your own experiments with real hardware without having access to expensive robotic arms, complex robots, and so on.

At the same time, I see some potential for the RL approach to be applied to complex robots, and who knows, maybe you will build the next version of iRobot Corporation to bring more robots into our lives. If you are interested in buying the kits for the robot platform described in this chapter, it would be really helpful if you could fill out this form: `https://forms.gle/Ug1p9C7F1PszzfYz9`.

In the next chapter, we will continue our exploration of continuous control problems by checking a different set of improvements: *trust regions*.

19

Trust Regions – PPO, TRPO, ACKTR, and SAC

Next, we will take a look at the approaches used to improve the stability of the stochastic policy gradient method. Some attempts have been made to make the policy improvement more stable, and in this chapter, we will focus on three methods:

- **Proximal policy optimization** (**PPO**)
- **Trust region policy optimization** (**TRPO**)
- **Advantage actor-critic (A2C)** using **Kronecker-factored trust region** (**ACKTR**).

In addition, we will compare those methods to a relatively new off-policy method called **soft actor-critic** (**SAC**), which is a development of the deep deterministic policy gradients (DDPG) method described in *Chapter 17*, *Continuous Action Space*. To compare them to the A2C baseline, we will use several environments from the **Roboschool library** created by OpenAI.

The overall motivation of the methods that we will look at is to improve the stability of the policy update during the training. There is a dilemma: on the one hand, we'd like to train as fast as we can, making large steps during the stochastic gradient descent (SGD) update. On the other hand, a large update of the policy is usually a bad idea. The policy is a very nonlinear thing, so a large update can ruin the policy we've just learned.

Things can become even worse in the reinforcement learning (RL) landscape, as making a bad update of the policy once won't be recovered from by subsequent updates. Instead, the bad policy will bring us bad experience samples that we will use on subsequent training steps, which could break our policy completely. Thus, we want to avoid making large updates by all means possible.

One of the naïve solutions would be to use a small learning rate to take baby steps during the SGD, but this would significantly slow down the convergence.

To break this vicious cycle, several attempts have been made by researchers to estimate the effect that our policy update is going to have in terms of the future outcome. One of the popular approaches is *trust region* optimization extension, which constrains the step taken during the optimization to limit its effect on the policy. The main idea is to prevent a dramatic policy update during the loss optimization by checking the Kullback-Leibler (KL) divergence between the old and the new policy. Of course, this is a very handwavy explanation, but it can help with understanding the idea, especially as those methods are quite math-heavy (especially TRPO).

Roboschool

To experiment with the methods in this chapter, we will use Roboschool, which uses PyBullet as a physics engine and has 13 environments of various complexity. PyBullet has similar environments, but at the time of writing, it isn't possible to create several instances of the same environment due to an internal OpenGL issue.

In this chapter, we will explore two problems: RoboschoolHalfCheetah-v1, which models a two-legged creature, and RoboschoolAnt-v1, which has four legs. Their state and action spaces are very similar to the Minitaur environment that we saw in *Chapter 17*, *Continuous Action Space*: the state includes characteristics from joints, and the actions are activations of those joints. The goal for both is to move as far as possible, minimizing the energy spent. *Figure 19.1* shows screenshots of the two environments.
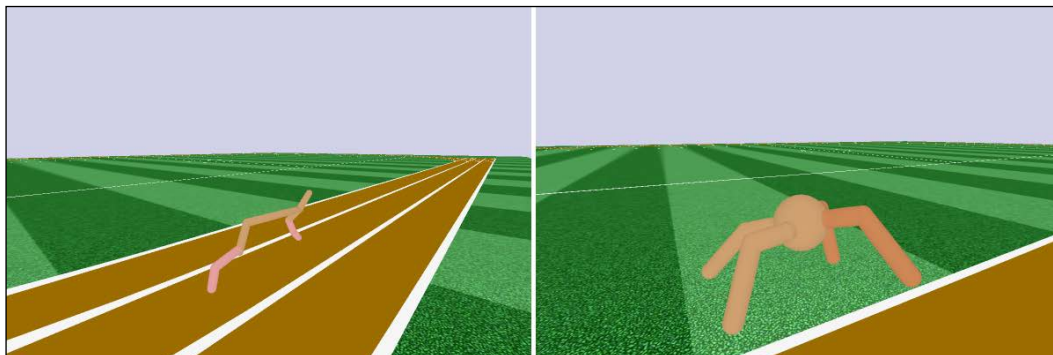
Figure 19.1: Screenshots of two Roboschool environments: RoboschoolHalfCheetah and RoboschoolAnt

To install Roboschool, you need to follow the instructions at `https://github.com/openai/roboschool`. This requires extra components to be installed in the system and a modified PyBullet to be built and used.

After the Roboschool installation, you should be able to use `import roboschool` in your code to get access to the new environments.

Installation may not be very easy and smooth; for instance, on my system, it required building the package from sources, because the precompiled Python package had a more recent version of libc as a dependency. In any case, the instructions in GitHub are helpful.

The A2C baseline

To establish the baseline results, we will use the A2C method in a very similar way to the code in the previous chapter.

Implementation

The complete source is in files `Chapter19/01_train_a2c.py` and `Chapter19/lib/model.py`. There are a few differences between this baseline and the version we used in the previous chapter. First of all, there are 16 parallel environments used to gather experience during the training. The second difference is the model structure and the way that we perform exploration. To illustrate them, let's look at the model and the agent classes.

Both the actor and critic are placed in the separate networks without sharing weights. They follow the approach used in the previous chapter, with our critic estimating the mean and the variance for the actions. However, now variance is not a separate head of the base network; it is just a single parameter of the model. This parameter will be adjusted during the training by SGD, but it doesn't depend on the observation.

```python
HID_SIZE = 64

class ModelActor(nn.Module):
    def __init__(self, obs_size, act_size):
        super(ModelActor, self).__init__()

        self.mu = nn.Sequential(
            nn.Linear(obs_size, HID_SIZE),
            nn.Tanh(),
            nn.Linear(HID_SIZE, HID_SIZE),
            nn.Tanh(),
            nn.Linear(HID_SIZE, act_size),
            nn.Tanh(),
        )
        self.logstd = nn.Parameter(torch.zeros(act_size))
```

```
        def forward(self, x):
            return self.mu(x)
```

The actor network has two hidden layers of 64 neurons each with tanh nonlinearity. The variance is modeled as a separate network parameter and interpreted as a logarithm of the standard deviation.

```
    class ModelCritic(nn.Module):
        def __init__(self, obs_size):
            super(ModelCritic, self).__init__()

            self.value = nn.Sequential(
                nn.Linear(obs_size, HID_SIZE),
                nn.ReLU(),
                nn.Linear(HID_SIZE, HID_SIZE),
                nn.ReLU(),
                nn.Linear(HID_SIZE, 1),
            )

        def forward(self, x):
            return self.value(x)
```

The critic network also has two hidden layers of the same size, with one single output value, which is the estimation of $V(s)$, which is a discounted value of the state.

```
    class AgentA2C(ptan.agent.BaseAgent):
        def __init__(self, net, device="cpu"):
            self.net = net
            self.device = device

        def __call__(self, states, agent_states):
            states_v = ptan.agent.float32_preprocessor(states)
            states_v = states_v.to(self.device)

            mu_v = self.net(states_v)
            mu = mu_v.data.cpu().numpy()
            logstd = self.net.logstd.data.cpu().numpy()
            rnd = np.random.normal(size=logstd.shape)
            actions = mu + np.exp(logstd) * rnd
            actions = np.clip(actions, -1, 1)
            return actions, agent_states
```

The agent that converts the state into the action also works by simply obtaining the predicted mean from the state and applying the noise with variance, dictated by the current value of the `logstd` parameter.

Results

By default, the RoboschoolHalfCheetah-v1 environment is used, but to change it, you can pass the -e argument with the desired environment ID. So, to start HalfCheetah optimization, you should use the following command line:

```
$ ./01_train_a2c.py --cuda -n t1
ModelActor(
  (mu): Sequential(
    (0): Linear(in_features=26, out_features=64, bias=True)
    (1): Tanh()
    (2): Linear(in_features=64, out_features=64, bias=True)
    (3): Tanh()
    (4): Linear(in_features=64, out_features=6, bias=True)
    (5): Tanh()
  )
)
ModelCritic(
  (value): Sequential(
    (0): Linear(in_features=26, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=64, bias=True)
    (3): ReLU()
    (4): Linear(in_features=64, out_features=1, bias=True)
  )
)
Test done in 0.14 sec, reward 3.199, steps 18
854: done 44 episodes, mean reward -9.703, speed 853.93 f/s
2371: done 124 episodes, mean reward -6.723, speed 1509.12 f/s
3961: done 204 episodes, mean reward -4.228, speed 1501.50 f/s
5440: done 289 episodes, mean reward -8.149, speed 1471.60 f/s
6969: done 376 episodes, mean reward -6.250, speed 1501.39 f/s
```

The following are convergence charts that I got after 100M observations, which took slightly less than one day of training. The dynamics suggest that the policy could be further improved with more time given to optimization, but for our purpose of method comparison, it should be enough. Of course, if you're curious and have plenty of time, you can run this for longer and find the point when the policy stops improving.

According to research papers, HalfCheetah has a maximum score around 4,000-5,000.
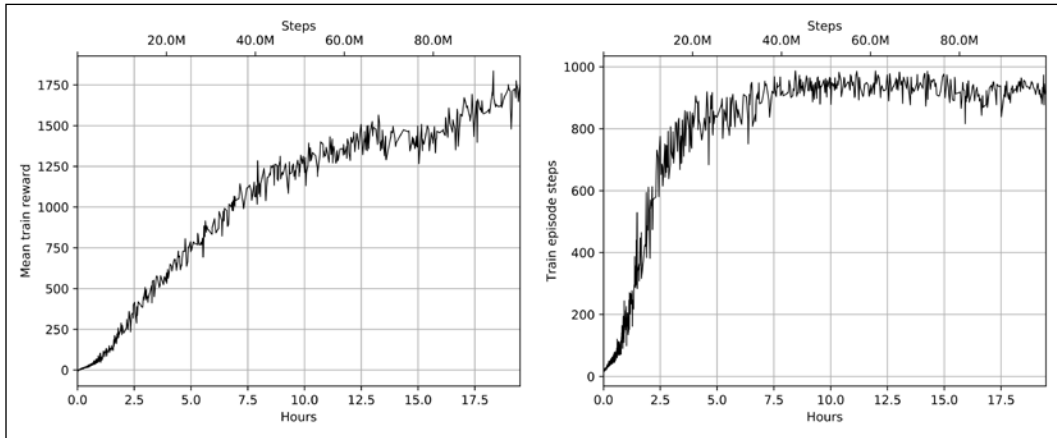
Figure 19.2: The training reward (left) and episode length (right) of A2C trained on HalfCheetah
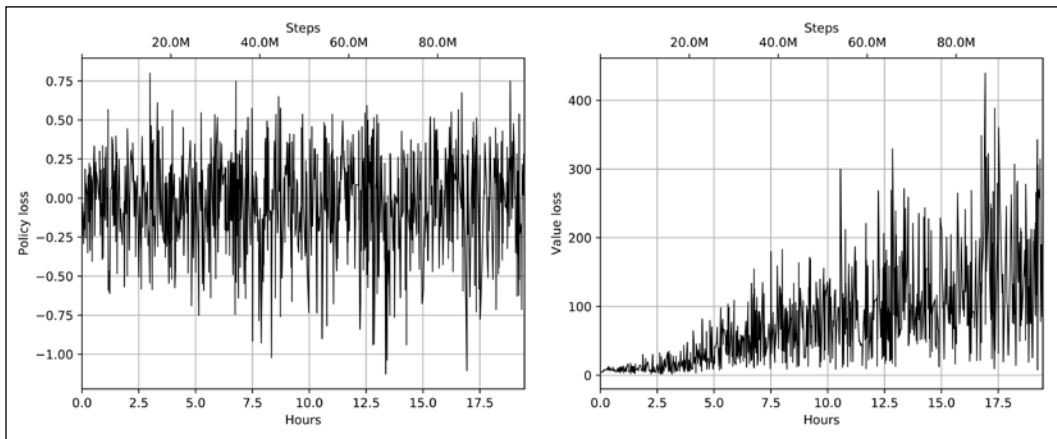
Figure 19.3: Policy loss (left) and value loss (right)
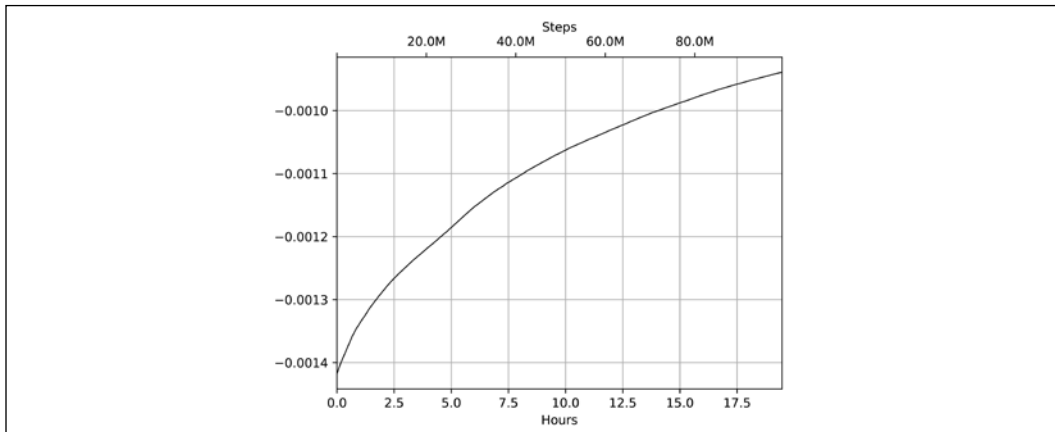
Figure 19.4: Entropy loss during the training
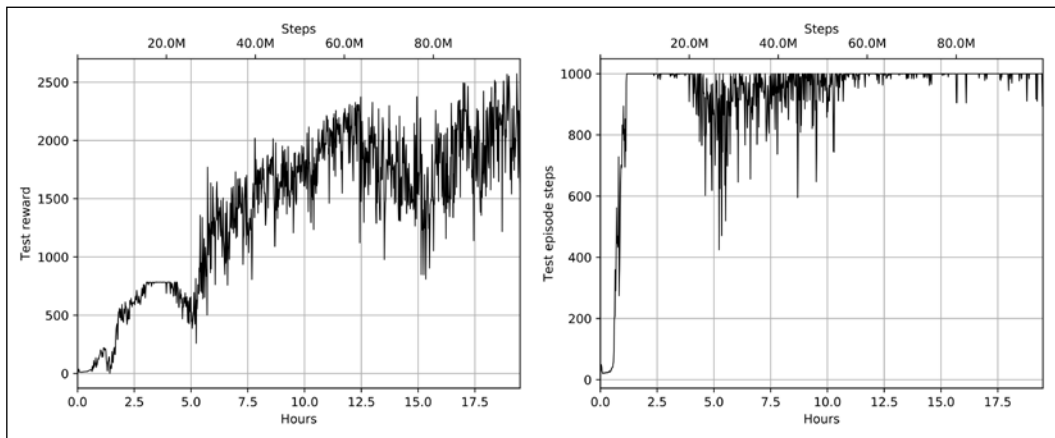
Figure 19.5: The test reward (left) and test episode length (right)

The charts shown in *Figure 19.5* demonstrate a better reward due to lack of noise injected into actions during the training episodes, but, overall, there are similar dynamics on the charts for training episodes.

Another environment used for comparison is RoboschoolAnt-v1, which should be started with the explicit environment name in the command line, as shown here:

```
$ ./01_train_a2c.py --cuda -n t1 -e RoboschoolAnt-v1
```

This environment is slower, so in 19 hours, optimization resulted in 85M observations. The following figures show the results:
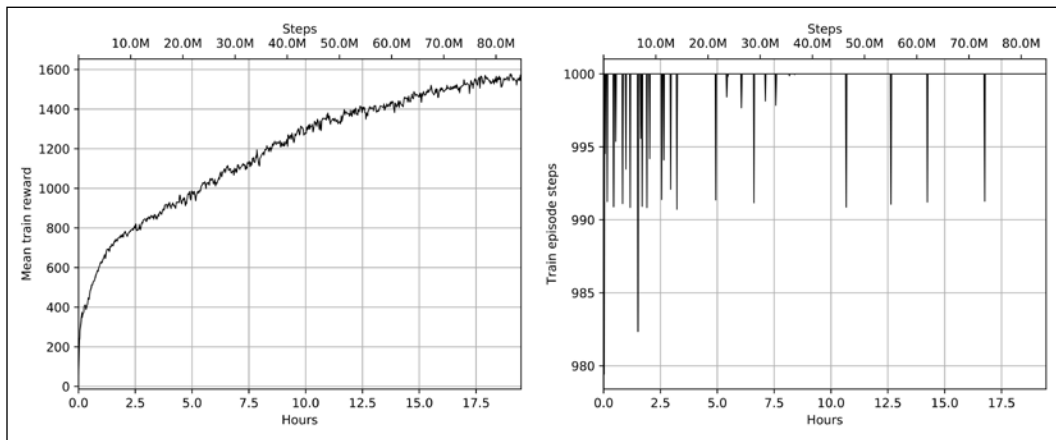
Figure 19.6: Training reward (left) and episode steps (right) for the Ant environment
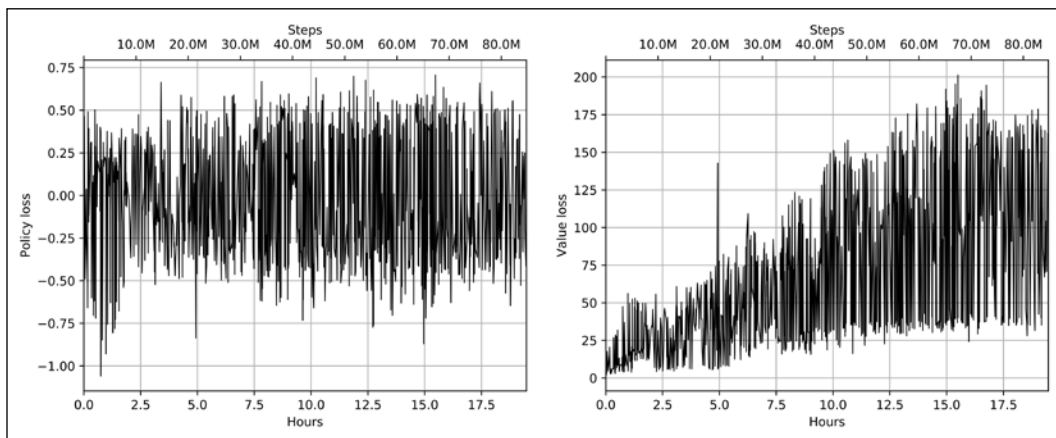
Figure 19.7: Policy loss (left) and value loss (right) for the Ant environment
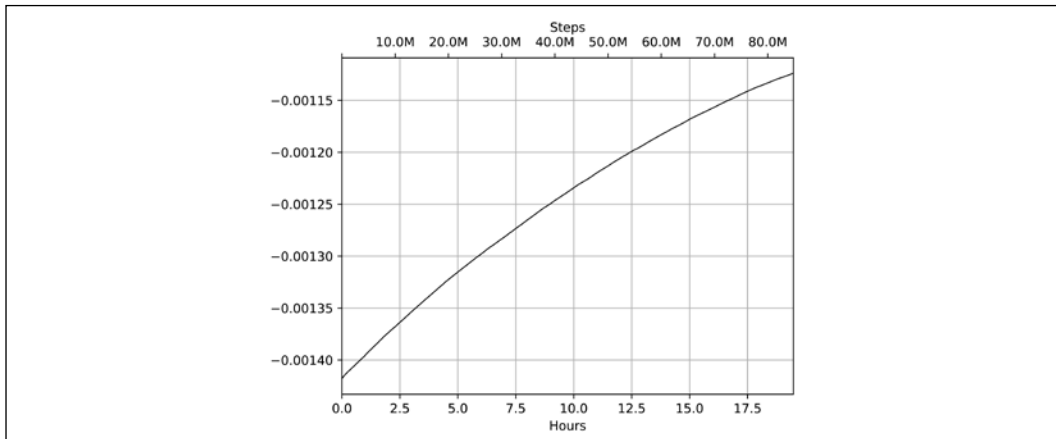
Figure 19.8: Entropy loss during the training
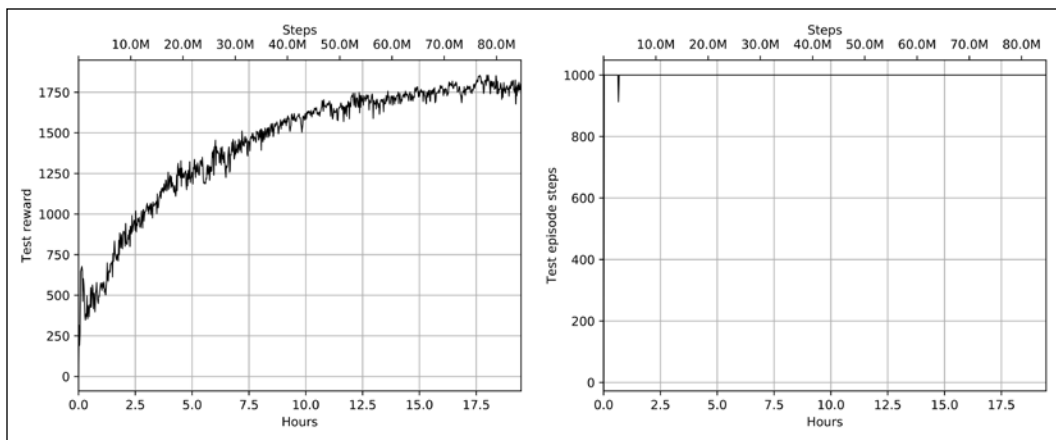
Figure 19.9: Test reward (left) and episode steps (right) for Ant

Video recording

As usual, there is a utility that can benchmark the trained model and record a video with the agent in action. It is in file `Chapter19/02_play.py` and can accept any model from the methods in this chapter (as the actor network is the same in all methods). You can also change the environment name using the `-e` command option.

PPO

Historically, the PPO method came from the OpenAI team and it was proposed long after TRPO, which is from 2015. However, PPO is much simpler than TRPO, so we will start with it. The 2017 paper in which it was proposed is by John Schulman et. al., and it is called *Proximal Policy Optimization Algorithms* (arXiv:1707.06347).

The core improvement over the classic A2C method is changing the formula used to estimate the policy gradients. Instead of using the gradient of logarithm probability of the action taken, the PPO method uses a different objective: the ratio between the new and the old policy scaled by the advantages.

In math form, the old A2C objective could be written as $J_\theta = E_t[\nabla_\theta \log \pi_\theta(a_t|s_t)A_t]$.

The new objective proposed by PPO is $J_\theta = E_t\left[\dfrac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A_t\right]$.

The reason behind changing the objective is the same as with the cross-entropy method covered in *Chapter 4, The Cross-Entropy Method*: importance sampling. However, if we just start to blindly maximize this value, it may lead to a very large update to the policy weights. To limit the update, the clipped objective is used. If we write the ratio between the new and the old policy as $r_t(\theta) = \dfrac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, the clipped objective could be written as $J_\theta^{clip} = \mathbb{E}_t[\min(r_t(\theta)A_t, clip(r_t(\theta), 1 - \varepsilon, 1 + \varepsilon)A_t)]$.

This objective limits the ratio between the old and the new policy to be in the interval $[1 - \varepsilon, 1 + \varepsilon]$, so by varying ε, we can limit the size of the update.

Another difference from the A2C method is the way that we estimate the advantage. In the A2C paper, the advantage obtained from the finite-horizon estimation of T steps is in the form $A_t = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t+1}r_{T-1} + \gamma^{T-t}V(s_t)$.
In the PPO paper, the authors used a more general estimation in the form
$A_t = \sigma_t + (\gamma\lambda)\sigma_{t+1} + (\gamma\lambda)^2\sigma_{t+2} + \cdots + (\gamma\lambda)^{T-t+1}\sigma_{t-1}$, where $\sigma_t = r_t + \gamma V(s_{t+1}) - V(s_t)$.
The original A2C estimation is a special case of the proposed method with $\lambda = 1$.
The PPO method also uses a slightly different training procedure: a long sequence of samples is obtained from the environment and then the advantage is estimated for the whole sequence before several epochs of training are performed.

Implementation

The code of the sample is placed in two source code files: `Chapter19/04_train_ppo.py` and `Chapter19/lib/model.py`. The actor, the critic, and the agent classes are exactly the same as we had in the A2C baseline.

The differences are in the training procedure and the way that we calculate advantages, but let's start with the hyperparameters.

```
ENV_ID = "RoboschoolHalfCheetah-v1"
GAMMA = 0.99
GAE_LAMBDA = 0.95
```

The value of `GAMMA` is already familiar, but `GAE_LAMBDA` is the new constant that specifies the lambda factor in the advantage estimator. The value of `0.95` was used in the PPO paper.

```
TRAJECTORY_SIZE = 2049
LEARNING_RATE_ACTOR = 1e-5
LEARNING_RATE_CRITIC = 1e-4
```

The method assumes that a large amount of transitions will be obtained from the environment for every subiteration. (As mentioned previously in this section while describing PPO, during training, it does several epochs over the sampled training batch.) We also use two different optimizers for the actor and the critic (as they have no shared weights).

```
PPO_EPS = 0.2
PPO_EPOCHES = 10
PPO_BATCH_SIZE = 64
```

For every batch of `TRAJECTORY_SIZE` samples, we perform `PPO_EPOCHES` iterations of the PPO objective, with mini-batches of 64 samples. The value `PPO_EPS` specifies the clipping value for the ratio of the new and the old policy.

```
TEST_ITERS = 1000
```

For every thousand observations obtained from the environment, we perform a test of 10 episodes to obtain the total reward and the count of steps for the current policy. The following function takes the trajectory with steps and calculates advantages for the actor and reference values for the critic training. Our trajectory is not a single episode, but can be several episodes concatenated together.

```
def calc_adv_ref(trajectory, net_crt, states_v, device="cpu"):
    values_v = net_crt(states_v)
    values = values_v.squeeze().data.cpu().numpy()
```

As the first step, we ask the critic to convert states into values.

```
last_gae = 0.0
result_adv = []
result_ref = []
for val, next_val, (exp,) in zip(reversed(values[:-1]),
```

```
                        reversed(values[1:]),
                        reversed(trajectory[:-1])):
```

This loop joins the values obtained and experience points. For every trajectory step, we need the current value (obtained from the current state) and the value for the next subsequent step (to perform the estimation using the Bellman equation). We also traverse the trajectory in reverse order, to be able to calculate more recent values of the advantage in one step.

```
        if exp.done:
            delta = exp.reward - val
            last_gae = delta
        else:
            delta = exp.reward + GAMMA * next_val - val
            last_gae = delta + GAMMA * GAE_LAMBDA * last_gae
```

On every step, our action depends on the `done` flag for this step. If this is the terminal step of the episode, we have no prior reward to take into account. (Remember, we're processing the trajectory in reverse order.) So, our value of `delta` in this step is just the immediate reward minus the value predicted for the step. If the current step is not terminal, the delta will be equal to the immediate reward plus the discounted value from the subsequent step, minus the value for the current step. In the classic A2C method, this delta was used as an advantage estimation, but here, the smoothed version is used, so the advantage estimation (tracked in the `last_gae` variable) is calculated as the sum of deltas with discount factor $\gamma\lambda$.

```
        result_adv.append(last_gae)
        result_ref.append(last_gae + val)
```

The goal of the function is to calculate advantages and reference values for the critic, so we save them in lists.

```
    adv_v = torch.FloatTensor(list(reversed(result_adv)))
    ref_v = torch.FloatTensor(list(reversed(result_ref)))
    return adv_v.to(device), ref_v.to(device)
```

In the training loop, we gather a trajectory of the desired size using the `ExperienceSource(steps_count=1)` class from the PTAN library. With such configuration, it provides us with individual steps from the environment in tuples `(state, action, reward, done)`. The following is the relevant training part of the loop:

```
        trajectory.append(exp)
        if len(trajectory) < TRAJECTORY_SIZE:
            continue

        traj_states = [t[0].state for t in trajectory]
```

```
traj_actions = [t[0].action for t in trajectory]
traj_states_v = torch.FloatTensor(traj_states)
traj_states_v = traj_states_v.to(device)
traj_actions_v = torch.FloatTensor(traj_actions)
traj_actions_v = traj_actions_v.to(device)
traj_adv_v, traj_ref_v = calc_adv_ref(
    trajectory, net_crt, traj_states_v, device=device)
```

When we've got a trajectory large enough for training (which is given by the TRAJECTORY_SIZE hyperparameter), we convert states and taken actions into tensors and use the already-described function to obtain advantages and reference values. Although our trajectory is quite long, the observations of our test environments are quite short, so it's fine to process our batch in one step. In the case of Atari frames, such a batch could cause a graphics processing unit (GPU) memory error.

In the next step, we calculate the logarithm of probability of the actions taken. This value will be used as $\pi_{\theta_{old}}$ in the objective of PPO. Additionally, we normalize the advantage's mean and variance to improve the training stability.

```
mu_v = net_act(traj_states_v)
old_logprob_v = calc_logprob(
    mu_v, net_act.logstd, traj_actions_v)
traj_adv_v = traj_adv_v - torch.mean(traj_adv_v)
traj_adv_v /= torch.std(traj_adv_v)
```

The two subsequent lines drop the last entry from the trajectory to reflect the fact that our advantages and reference values are one step shorter than the trajectory length (as we shifted values in the loop inside the calc_adv_ref function).

```
trajectory = trajectory[:-1]
old_logprob_v = old_logprob_v[:-1].detach()
```

When all the preparations have been done, we perform several epochs of training on our trajectory. For every batch, we extract the portions from the corresponding arrays and do the critic and the actor training separately.

```
for epoch in range(PPO_EPOCHES):
    for batch_ofs in range(0, len(trajectory),
                           PPO_BATCH_SIZE):
        batch_l = batch_ofs + PPO_BATCH_SIZE
        states_v = traj_states_v[batch_ofs:batch_l]
        actions_v = traj_actions_v[batch_ofs:batch_l]
        batch_adv_v = traj_adv_v[batch_ofs:batch_l]
        batch_adv_v = batch_adv_v.unsqueeze(-1)
        batch_ref_v = traj_ref_v[batch_ofs:batch_l]
        batch_old_logprob_v = \
            old_logprob_v[batch_ofs:batch_l]
```

To train the critic, all we need to do is calculate the mean squared error (MSE) loss with the reference values calculated beforehand.

```
opt_crt.zero_grad()
value_v = net_crt(states_v)
loss_value_v = F.mse_loss(
    value_v.squeeze(-1), batch_ref_v)
loss_value_v.backward()
opt_crt.step()
```

In the actor training, we minimize the negated clipped objective:

$$\mathbb{E}_t[\min(r_t(\theta)A_t, clip(r_t(\theta), 1 - \varepsilon, 1 + \varepsilon)A_t)], \text{ where } r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}.$$

```
opt_act.zero_grad()
mu_v = net_act(states_v)
logprob_pi_v = calc_logprob(
    mu_v, net_act.logstd, actions_v)
ratio_v = torch.exp(
    logprob_pi_v - batch_old_logprob_v)
surr_obj_v = batch_adv_v * ratio_v
c_ratio_v = torch.clamp(ratio_v,
                        1.0 - PPO_EPS,
                        1.0 + PPO_EPS)
clipped_surr_v = batch_adv_v * c_ratio_v
loss_policy_v = -torch.min(
    surr_obj_v, clipped_surr_v).mean()
loss_policy_v.backward()
opt_act.step()
```

Results

After being trained on both our test environments, the PPO method has shown a major improvement over the A2C method. The following charts show the training progress on the RoboschoolHalfCheetah-v1 environment, when the methods were able to reach 2,500 reward on the test episode after two hours of training and less than 5M observations, which is much better than A2C (with 100M observations and 19 hours to get the same result).

Unfortunately, after reaching the record, the reward increase stopped, which suggests that hyperparameter tweaking is required. The current hyperparameters were only slightly tuned.
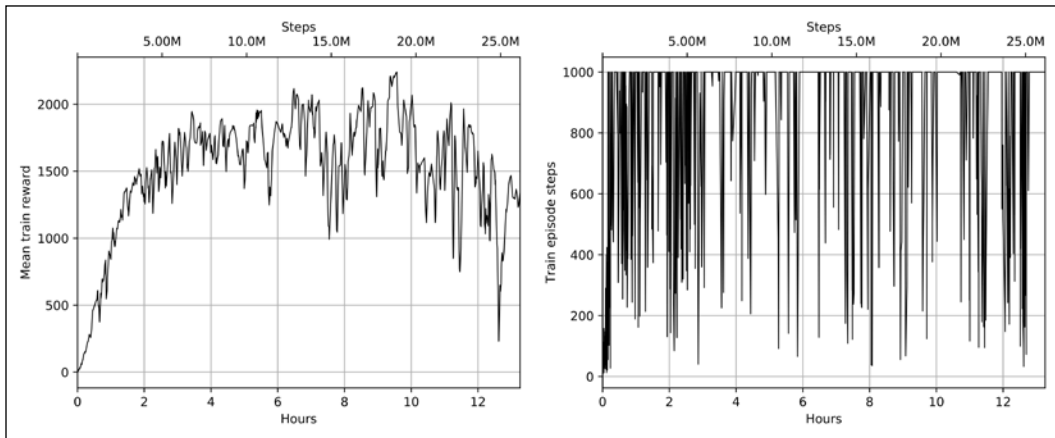
Figure 19.10: The training reward (left) and steps (right) of PPO training on HalfCheetah
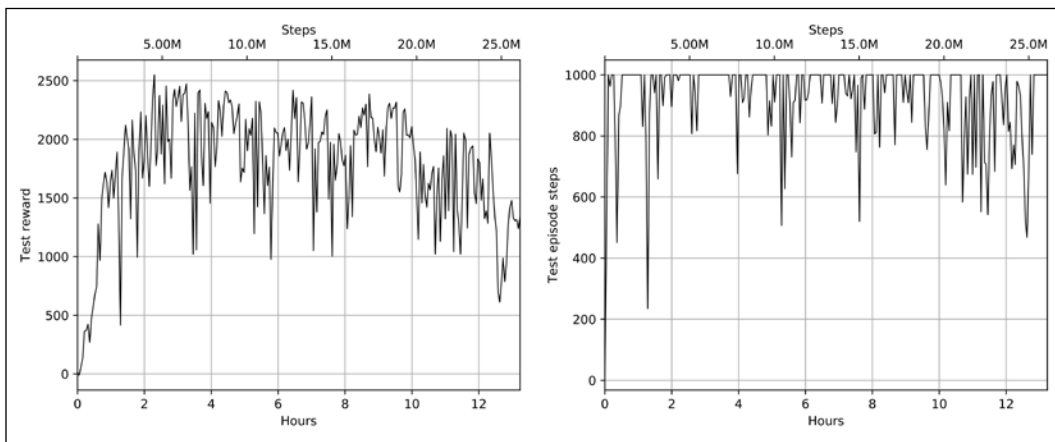
Figure 19.11: The test reward (left) and steps (right) of PPO on HalfCheetah

On RobochoolAnt-v1, the reward increase was much more stable and was able to reach a better policy than A2C.

Figure 19.12 shows the training and test rewards of PPO, while *Figure 19.13* compares PPO and A2C.
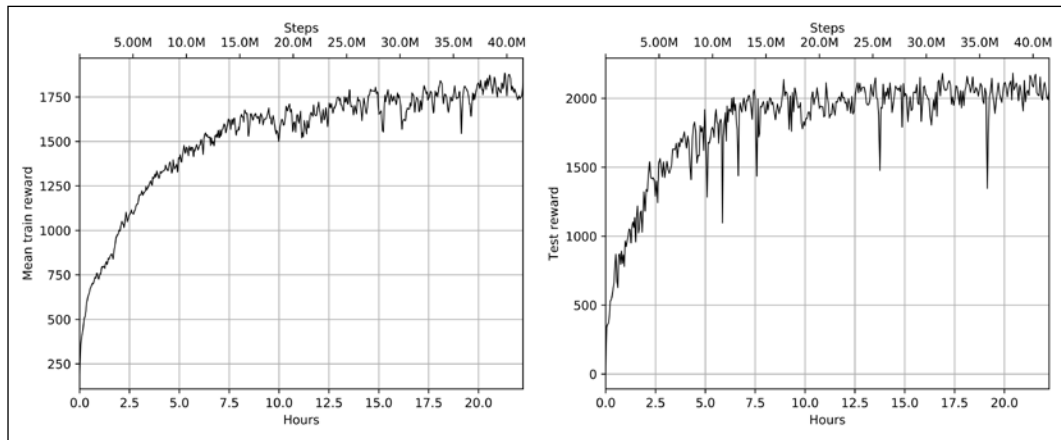
Figure 19.12: The training reward (left) and test reward (right) of PPO on the Ant environment
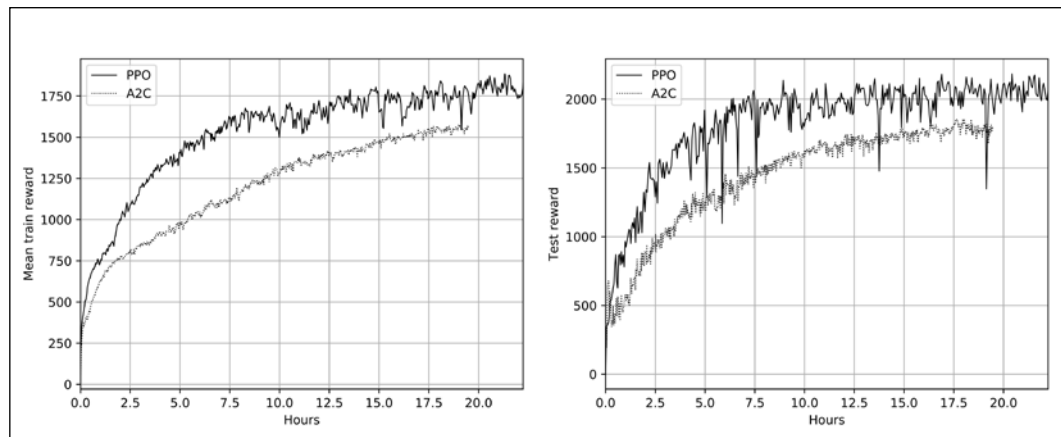
Figure 19.13: A comparison of the training (left) and test (right) rewards of PPO and A2C on Ant

TRPO

TRPO was proposed in 2015 by Berkeley researchers in a paper by John Schulman et. al., called *Trust Region Policy Optimization* (arXiv:1502.05477). This paper was a step towards improving the stability and consistency of stochastic policy gradient optimization and has shown good results on various control tasks.

Unfortunately, the paper and the method are quite math-heavy, so it can be hard to understand the details of the method. The same could be said about the implementation, which uses the conjugate gradients method to efficiently solve the constrained optimization problem.

As the first step, the TRPO method defines the discounted visitation frequencies of the state: $\rho_\pi(s) = P(s_0 = s) + \gamma P(s_1 = s) + \gamma^2 P(s_2 = s) + \cdots$. In this equation, $P(s_i = s)$ equals the sampled probability of state s, to be met at position i of the sampled trajectories. Then, TRPO defines the optimization objective as

$$L_\pi(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_\pi(s) \sum_a \tilde{\pi}(a|s) A_\pi(s, a) \text{ where } \eta(\pi) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t)\right] \text{ is the}$$

expected discounted reward of the policy and $\tilde{\pi} = \arg\max_a A_\pi(s, a)$ defines the deterministic policy.

To address the issue of large policy updates, TRPO defines the additional constraint on the policy update, which is expressed as a maximum KL divergence between the old and the new policies, which could be written as $\overline{D}_{KL}^{\rho_{\theta_{old}}}(\theta_{old}, \theta) \leq \delta$.

Implementation

Most of the TRPO implementations available on GitHub, or other open source repositories, are very similar to each other, probably because all of them grew from the original John Schulman TRPO implementation here: `https://github.com/joschu/modular_rl`. My version of TRPO is also not very different and uses the core functions that implement the conjugate gradient method (used by TRPO to solve the constrained optimization problem) from this repository: `https://github.com/ikostrikov/pytorch-trpo`.

The complete example is in `Chapter19/03_train_trpo.py` and `Chapter19/lib/trpo.py`, and the training loop is very similar to the PPO example: we sample the trajectory of transitions of the predefined length and calculate the advantage estimation using the smoothed formula given in the PPO section. (Historically, this estimator was proposed first in the TRPO paper.) Next, we do one training step of the critic using MSE loss with the calculated reference value, and one step of the TRPO update, which consists of finding the direction we should go in by using the conjugate gradients method and doing a linear search along this direction to find a step that preserves the desired KL divergence.

The following is the piece of the training loop that carries out both those steps:

```
opt_crt.zero_grad()
value_v = net_crt(traj_states_v)
loss_value_v = F.mse_loss(
    value_v.squeeze(-1), traj_ref_v)
loss_value_v.backward()
opt_crt.step()
```

To perform the TRPO step, we need to provide two functions: the first will calculate the loss of the current actor policy, which uses the same ratio as in PPO of the new and the old policies multiplied by the advantage estimation. The second function has to calculate KL divergence between the old and the current policy.

```
def get_loss():
    mu_v = net_act(traj_states_v)
    logprob_v = calc_logprob(
        mu_v, net_act.logstd, traj_actions_v)
    dp_v = torch.exp(logprob_v - old_logprob_v)
    action_loss_v = -traj_adv_v.unsqueeze(dim=-1)*dp_v
    return action_loss_v.mean()

def get_kl():
    mu_v = net_act(traj_states_v)
    logstd_v = net_act.logstd
    mu0_v = mu_v.detach()
    logstd0_v = logstd_v.detach()
    std_v = torch.exp(logstd_v)
    std0_v = std_v.detach()
    v = (std0_v ** 2 + (mu0_v - mu_v) ** 2) / \
        (2.0 * std_v ** 2)
    kl = logstd_v - logstd0_v + v - 0.5
    return kl.sum(1, keepdim=True)

trpo.trpo_step(net_act, get_loss, get_kl, args.maxkl,
               TRPO_DAMPING, device=device)
```

In other words, the PPO method is TRPO that uses the simple clipping of the policy ratio to limit the policy update, instead of the complicated conjugate gradients and line search.

Results

TRPO showed a slower reward increase than PPO, but it finally was able to find a better policy than both PPO and A2C. After 20 hours and 70M observations, TRPO showed 2,804 mean reward on 10 test episodes.

Figure 19.14 shows the TRPO training and test reward dynamics, and *Figure 19.15* compares them with PPO optimization.
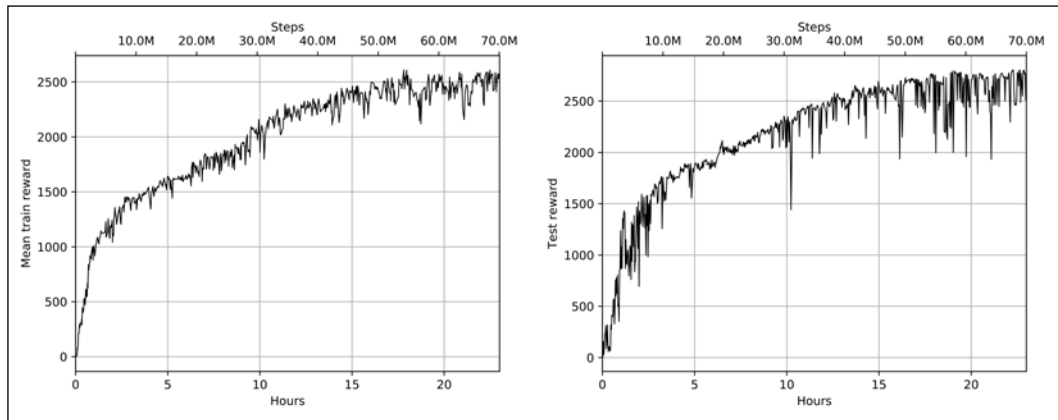
Figure 19.14: TRPO on HalfCheetah—training reward (left) and test reward (right)
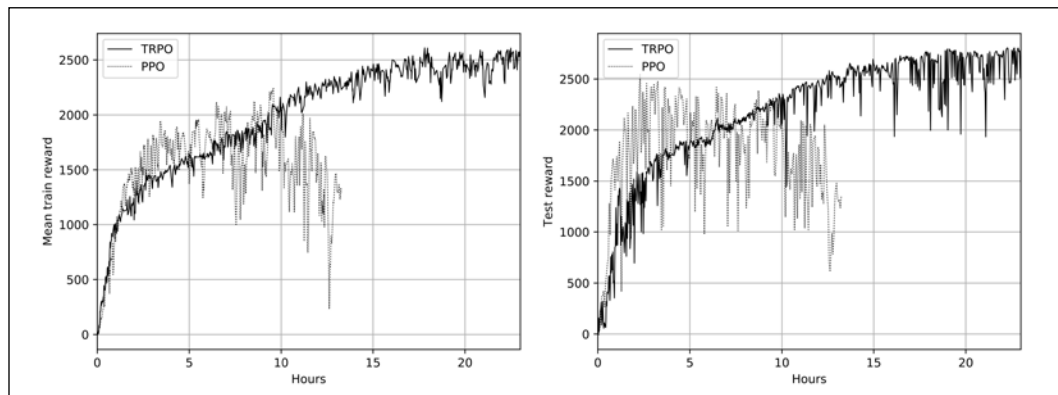
Figure 19.15: A comparison of TRPO and PPO on HalfCheetah—train reward (left) and test reward (right)

Training on RoboschoolAnt-v1 was much less successful. Right after the start, the training process followed the PPO dynamics, but after 30 minutes, the training process diverged, with the best reward being 600. The following charts are for TRPO training alone, with accompanying PPO plots.
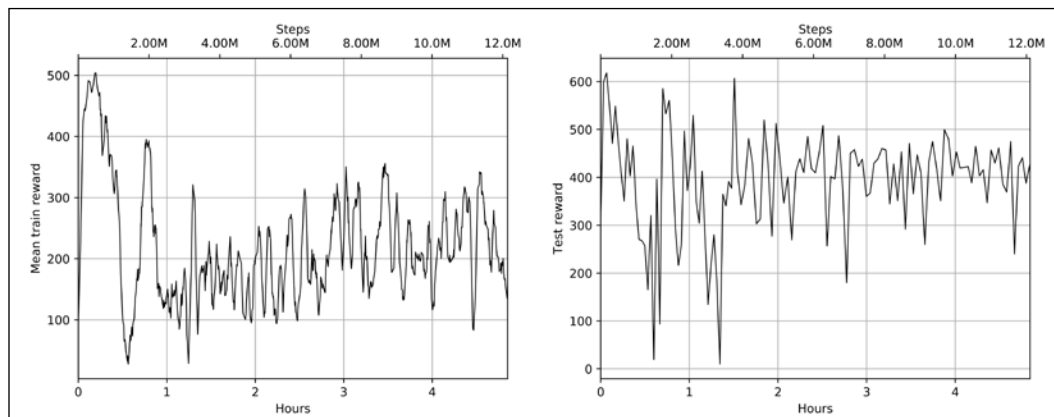
Figure 19.16: TRPO on the Ant environment — training reward (left) and test reward (right)
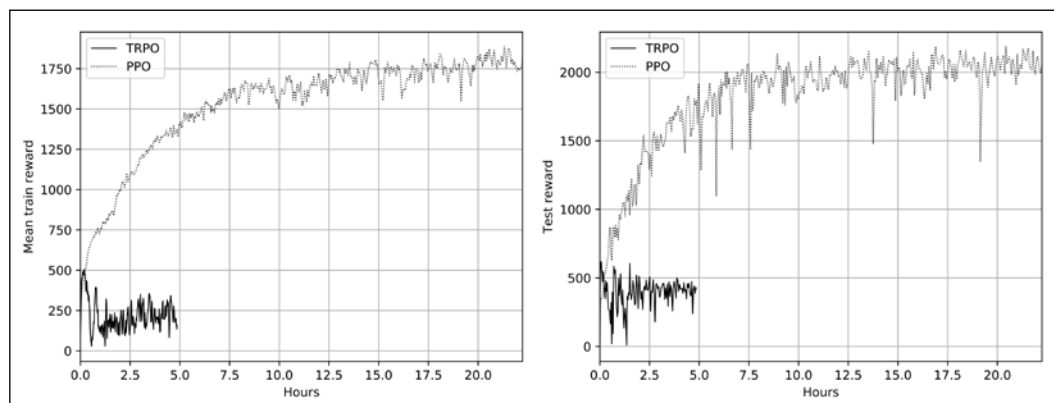
Figure 19.17: A comparison of TRPO and PPO on the Ant environment

ACKTR

The third method that we will compare, ACKTR, uses a different approach to address SGD stability. In the paper by Yuhuai Wu and others called *Scalable Trust-Region Method for Deep Reinforcement Learning Using Kronecker-Factored Approximation*, published in 2017 (arXiv:1708.05144), the authors combined the second-order optimization methods and trust region approach.

The idea of the second-order methods is to improve the traditional SGD by taking the second-order derivatives of the optimized function (in other words, its curvature) to improve the convergence of the optimization process. To make things more complicated, working with the second derivatives usually requires you to build and invert a Hessian matrix, which can be prohibitively large, so the practical methods typically approximate it in some way. This area is currently very active in research, as developing robust, scalable optimization methods is very important for the whole machine learning domain.

One of the second-order methods is called **Kronecker-factored approximate curvature** (**K-FAC**), which was proposed by James Martens and Roger Grosse in their paper *Optimizing Neural Networks with Kronecker-Factored Approximate Curvature*, published in 2015. However, a detailed description of this method is well beyond the scope of this book.

Implementation

As the K-FAC method is quite recent, there is no optimizer for implementing this method included in PyTorch. The only available PyTorch prototype is from Ilya Kostrikov and is available here: `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr`. There is another version of K-FAC for TensorFlow, which comes with OpenAI Baselines, but porting and testing it on PyTorch can be a hard thing to do.

For my experiments, I've taken K-FAC from the Kostrikov link and adopted it to the existing code, which required replacing the optimizer and doing an extra `backward()` call to gather Fisher information. The critic was trained in the same way as in A2C.

The complete example is in `Chapter19/05_train_acktr.py` and is not shown here, as it's basically the same as A2C. The only difference is that a different optimizer was used.

Results

On the RoboschoolAnt-v1 environment, the ACKTR method run was very unstable and wasn't able to reach any competitive policy in 10M observations.

Obviously, accurate fine-tuning of the method is needed.
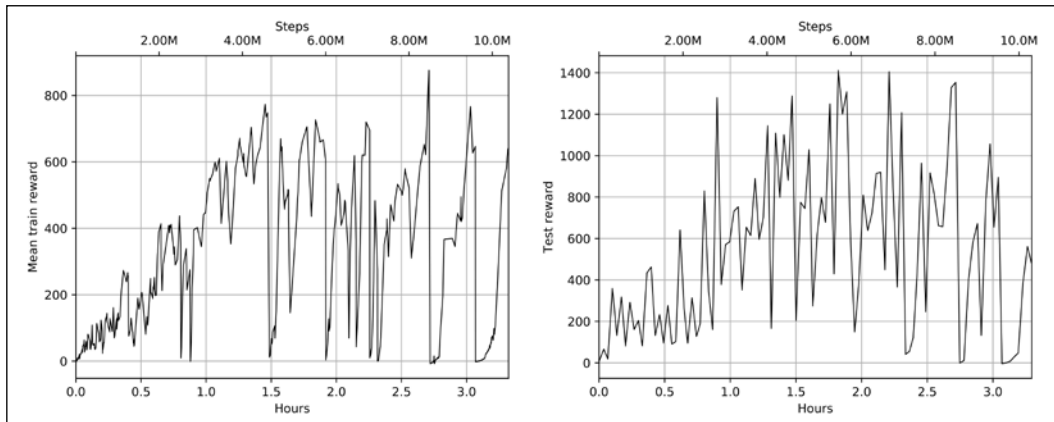
Figure 19.18: ACKTR on HalfCheetah

On the Ant environment, ACKTR was slightly more stable, but still, the default parameters didn't show much improvement over the A2C method.
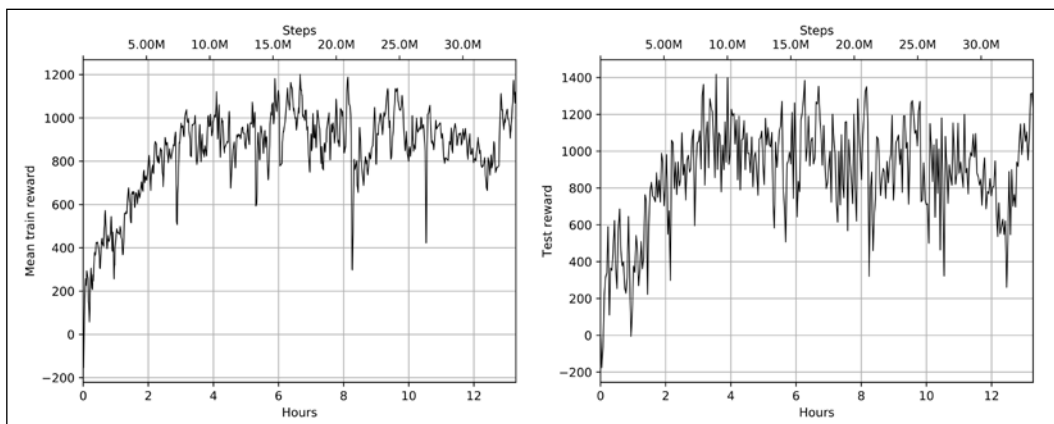
Figure 19.19: ACKTR on Ant – training reward (left) and test reward (right)

SAC

In the final section, we will check our environments on the latest state-of-the-art method, called SAC, which was proposed by a group of Berkeley researchers and introduced in the paper *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning*, by *Tuomas Taarnoja* et. al. arXiv 1801.01290, published in 2018.

At the moment, it's considered to be one of the best methods for continuous control problems. The core idea of the method is closer to the DDPG method than to A2C policy gradients. The SAC method might have been more logically described in *Chapter 17, Continuous Action Space*. However, in this chapter, we have the chance to compare it directly with PPO's performance, which was considered to be the de facto standard in continuous control problems for a long time.

The central idea in the SAC method is **entropy regularization**, which adds a bonus reward at each timestamp that is proportional to the entropy of the policy at this timestamp. In mathematical notation, the policy we're looking for is the following:

$$\pi^* = \arg \max_{a} E_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \alpha H\big(\pi(\cdot \,|s_t)\big) \right) \right]$$

$H(P) = \mathbb{E}_{x \sim P}[-\log P(x)]$ is an entropy of distribution P. In other words, we give the agent a bonus for getting into situations where the entropy is at maximum, which is very similar to the advanced exploration methods covered in *Chapter 21, Advanced Exploration*.

In addition, the SAC method incorporates the clipped double-Q trick, when, in addition to the value function, we learn two networks predicting Q-values, and choose the minimum of them for Bellman approximation. According to researchers, this helps with dealing with Q-value overestimation during the training.

So, in total, we train four networks: the policy, $\pi(s)$, value, $V(s, a)$ and two Q-networks, $Q_{1,2}(s, a)$. For the value network, $V(s, a)$, the target network is used. So, in total, SAC training looks like the following:

- Q-networks are trained using the MSE objective by doing Bellman approximation using the target value network: $y_q(r, s') = r + \gamma V_{tgt}(s')$ (for non-terminating steps)

- The V-network is trained using the MSE objective with the following target: $y_v(s) = \min_{i=1,2} Q_i(s, \tilde{a}) - \alpha \log \pi_\theta(\tilde{a}|s)$, where $\tilde{a}$ is sampled from policy $\pi_\theta(\cdot \,|s)$.

- The policy network, π_θ, is trained in DDPG style by maximizing the following objective: $Q_1\big(s, \tilde{a}_\theta(s)\big) - \alpha \log \pi_\theta(\tilde{a}_\theta(s)|s)$, where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot \,|s)$

Implementation

The implementation of the SAC method is in `Chapter19/06_train_sac.py`. The model consists of the following networks, defined in `Chapter19/lib/model.py`:

- `ModelActor`: This is the same policy that we used in the previous examples. As the policy variance is not parametrized by the state (the field `logstd` is not a network, but just a tensor), the training objective does not 100% comply with SAC. On the one hand, it might influence the convergence and performance, as the core idea in the SAC method is entropy regularization, which can't be implemented without parametrized variance. On the other hand, this decreases the number of parameters in the model. If you're curious, you can extend the example with parametrized variance of the policy and implement a proper SAC method.

- `ModelCritic`: This is the same value network as in the previous examples.

- `ModelSACTwinQ`: These two networks take the state and action as the input and predict Q-values.

The first function implementing the method is `unpack_batch_sac()`, and it is defined in `Chapter19/lib/common.py`. Its goal is to take the batch of trajectory steps and calculate target values for V-networks and twin Q-networks.

```python
@torch.no_grad()
def unpack_batch_sac(batch, val_net, twinq_net, policy_net,
                     gamma: float, ent_alpha: float,
                     device="cpu"):
    states_v, actions_v, ref_q_v = \
        unpack_batch_a2c(batch, val_net, gamma, device)

    mu_v = policy_net(states_v)
    act_dist = distr.Normal(mu_v, torch.exp(policy_net.logstd))
    acts_v = act_dist.sample()
    q1_v, q2_v = twinq_net(states_v, acts_v)
    ref_vals_v = torch.min(q1_v, q2_v).squeeze() - \
                 ent_alpha * act_dist.log_prob(acts_v).sum(dim=1)
    return states_v, actions_v, ref_vals_v, ref_q_v
```

The first step of the function uses the already defined `unpack_batch_a2c()` method, which unpacks the batch, converts states and actions into tensors, and calculates the reference for Q-networks using Bellman approximation. Once this is done, we need to calculate the reference for the V-network from the minimum of the twin Q-values minus the scaled entropy coefficient. The entropy is calculated from our current policy network. As was already mentioned, our policy has the parametrized mean value, but the variance is global and doesn't depend on the state.

In the main training loop, we use the function defined previously and do three different optimization steps: for *V*, for *Q*, and for the policy. The following is the relevant part of the training loop defined in `Chapter19/06_train_sac.py`:

```
batch = buffer.sample(BATCH_SIZE)
states_v, actions_v, ref_vals_v, ref_q_v = \
    common.unpack_batch_sac(
        batch, tgt_crt_net.target_model,
        twinq_net, act_net, GAMMA,
        SAC_ENTROPY_ALPHA, device)
```

In the beginning, we unpack the batch to get the tensors and targets for the Q- and V-networks.

```
twinq_opt.zero_grad()
q1_v, q2_v = twinq_net(states_v, actions_v)
q1_loss_v = F.mse_loss(q1_v.squeeze(),
                       ref_q_v.detach())
q2_loss_v = F.mse_loss(q2_v.squeeze(),
                       ref_q_v.detach())
q_loss_v = q1_loss_v + q2_loss_v
q_loss_v.backward()
twinq_opt.step()
```

The twin Q-networks are optimized by the same target value.

```
crt_opt.zero_grad()
val_v = crt_net(states_v)
v_loss_v = F.mse_loss(val_v.squeeze(),
                      ref_vals_v.detach())
v_loss_v.backward()
crt_opt.step()
```

The critic network is also optimized with the trivial MSE objective, using the already calculated target value.

```
act_opt.zero_grad()
acts_v = act_net(states_v)
q_out_v, _ = twinq_net(states_v, acts_v)
act_loss = -q_out_v.mean()
act_loss.backward()
act_opt.step()
```

In comparison to the formulas given previously, the code is missing the entropy regularization term and corresponds to DDPG training. As our variance doesn't depend on the state, it can be omitted from the optimization objective.

This modification makes the code different from the SAC paper, but from my experiments, it leads to more stable training. If you're curious, in the repository, I have a branch named `sac-experiment` with properly parametrized variance and optimization with the entropy regularization. But this modification makes it hard to compare the results with the previous methods tried in the chapter.

Results

I ran SAC training on both the HalfCheetah and Ant environments for 13 hours, with 5M observations. The results are a bit contradictory. On the one hand, the sample efficiency and reward growing dynamics of SAC were better than the PPO method. For example, SAC was able to reach a reward of 900 after just 0.5M observations on HalfCheetah. PPO required more than 1M observations to reach the same policy. On the other hand, due to the off-policy nature of SAC, the training speed was much slower, as we did more calculations than with on-policy methods. To compare this, it took SAC an hour and 30 minutes to reach the reward 900, whereas PPO was able to reach the same reward after just 30 minutes of training.

This demonstrates the trade-offs between on-policy and off-policy methods, as you have seen many times in the book: if your environment is fast and observations are "cheap" to obtain, an on-policy method like PPO might be the best choice. But if your observations are hard to obtain, off-policy methods will do the better job, but require more calculations to be performed.

The following charts are SAC training alone, followed by a comparison with the PPO method.
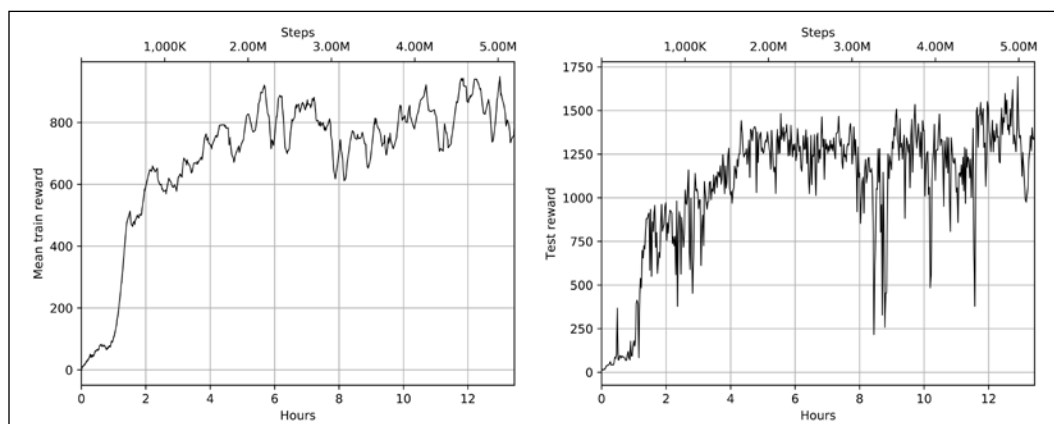
Figure 19.20: SAC training on HalfCheetah — training reward (left) and test reward (right)
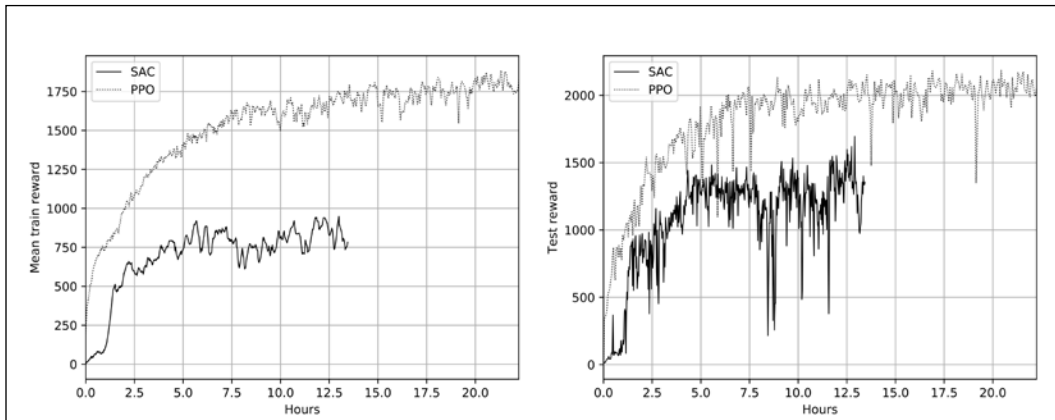
Figure 19.21: A comparison of SAC with PPO on HalfCheetah

Training on the Ant environment showed slightly better dynamics and stability than HalfCheetah, but it was still comparable to PPO in terms of sample efficiency and slower in terms of wall clock time.
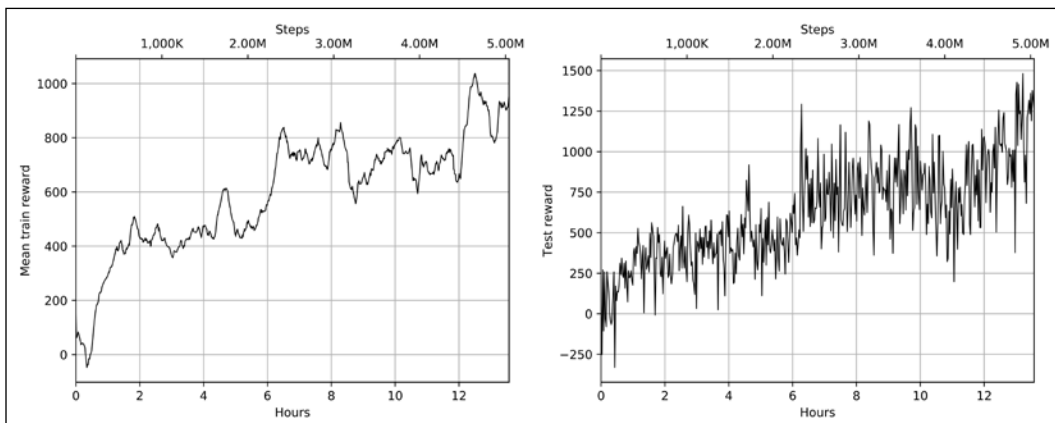
Figure 19.22: The SAC method on the Ant environment
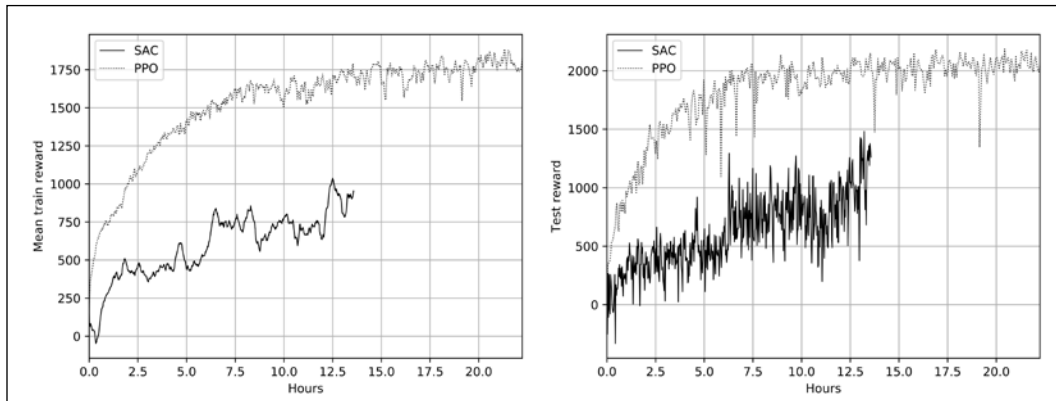
Figure 19.23: A comparison of SAC with PPO on Ant

Summary

In this chapter, we checked three different methods with the aim of improving the stability of the stochastic policy gradient and compared them to the A2C implementation on two continuous control problems. With methods from the previous chapter (DDPG and D4PG), they create the basic tools to work with a continuous control domain. Finally, we checked a relatively new off-policy method that is an extension of DDPG: SAC

In the next chapter, we will switch to a different set of RL methods that have been becoming popular recently: *black-box or gradient-free methods*.

20
Black-Box Optimization in RL

In this chapter, we will change our perspective on reinforcement learning (RL) training again and switch to the so-called **black-box optimizations**. In particular, this chapter will cover two examples of black-box optimization methods:

- **Evolution strategies**
- **Genetic algorithms**

These methods are at least a decade old, but recently, several research studies were conducted that showed the applicability of the methods to large-scale RL problems, and their competitiveness with the value iteration and policy gradient methods.

Black-box methods

To begin with, let's discuss the whole family of black-box methods and how it differs from what we've covered so far. Black-box optimization methods are the general approach to the optimization problem, when you treat the objective that you're optimizing as a black box, without any assumptions about the differentiability, the value function, the smoothness of the objective, and so on. The only requirement that those methods expose is the ability to calculate the **fitness function**, which should give us the measure of suitability of a particular instance of the optimized entity at hand.

One of the simplest examples in this family is random search, which is when you randomly sample the thing you're looking for (in the case of RL, it's the policy, $\pi(a|s)$), check the fitness of this candidate, and if the result is good enough (according to some reward criteria), then you're done. Otherwise, you repeat the process again and again. Despite the simplicity and even naivety of this approach, especially when compared to the sophisticated methods that you've seen so far, this is a good example to illustrate the idea of black-box methods.

Furthermore, with some modifications, as you will see shortly, this simple approach can be compared in terms of efficiency and the quality of the resulting policies to the deep Q-network (DQN) and policy gradient methods.

Black-box methods have several very appealing properties:

-   They are at least two times faster than gradient-based methods, as we don't need to perform the backpropagation step to obtain the gradients.

-   There are very few assumptions about the optimized objective and the policy that are treated as a black box. Traditional methods struggle with situations when the reward function is non-smooth or the policy contains steps with random choice. All of this is not an issue for black-box methods, as they don't expect much from the black-box internals.

-   The methods can generally be parallelized very well. For example, the aforementioned random search can easily scale up to thousands of central processing units (CPUs) or graphics processing units (GPUs) working in parallel, without any dependency on each other. This is not the case for DQN or policy gradient methods, when you need to accumulate the gradients and propagate the current policy to all parallel workers, which decreases the parallelism.

The downside of the preceding is usually lower sample efficiency. In particular, the naïve random search of the policy, parameterized with the neural network (NN) with half a million parameters, has a very low probability of succeeding.

Evolution strategies

One subset of black-box optimization methods is called evolution strategies (ES), and it was inspired by the evolution process. With ES, the most successful individuals have the highest influence on the overall direction of the search. There are many different methods that fall into this class, and in this chapter, we will consider the approach taken by the OpenAI researchers *Tim Salimans*, *Jonathan Ho*, and others in their paper, *Evolution Strategies as a Scalable Alternative to Reinforcement Learning* [1], published in March 2017.

The underlying idea of ES methods is simple: on every iteration, we perform random perturbation of our current policy parameters and evaluate the resulting policy fitness function. Then, we adjust the policy weights proportionally to the relative fitness function value.

The concrete method used in the paper is called **covariance matrix adaptation evolution strategy** (**CMA-ES**), in which the perturbation performed is the random noise sampled from the normal distribution with the zero mean and identity variance.

Then, we calculate the fitness function of the policy with weights equal to the weights of the original policy plus the scaled noise. Next, according to the obtained value, we adjust the original policy weights by adding the noise multiplied by the fitness function value, which moves our policy toward weights with a higher value of the fitness function. To improve the stability, the update of the weights is performed by averaging the batch of such steps with different random noise.

More formally, the preceding method could be expressed as this sequence of steps:

1. Initialize the learning rate, α, the noise standard deviation, σ, and the initial policy parameters, θ_0

2. For $t = 0, 1, 2, \ldots$ perform:

    1. The sample batch of noise with the shape of the weights:
        $\varepsilon_1, \ldots, \varepsilon_n \sim \mathcal{N}(0,1)$

    2. Compute returns $F_i = F(\theta_t + \sigma \varepsilon_i)$ for $i = 1,\ldots, n$

    3. $\theta_{t+1} \leftarrow \theta_t + \alpha \dfrac{1}{n\sigma} \sum_{i=1}^{n} F_i \varepsilon_i$ to update weights

The preceding algorithm is the core of the method presented in the paper, but, as usual in the RL domain, the core method is not enough to obtain good results. So, the paper includes several tweaks to improve the method, although the core is the same. Let's implement and test it on our *fruit fly* environment: CartPole.

ES on CartPole

The complete example is in `Chapter20/01_cartpole_es.py`. In this example, we will use the single environment to check the fitness of the perturbed network weights. Our fitness function will be the undiscounted total reward for the episode.

```python
#!/usr/bin/env python3
import gym
import time
import numpy as np

import torch
import torch.nn as nn

from tensorboardX import SummaryWriter
```

From the `import` statements, you will notice how self-contained our example is. We're not using PyTorch optimizers, as we don't perform backpropagation at all.

In fact, we could avoid using PyTorch completely and work only with NumPy, as the only thing we use PyTorch for is to perform a forward pass and calculate the network's output.

```
MAX_BATCH_EPISODES = 100
MAX_BATCH_STEPS = 10000
NOISE_STD = 0.01
LEARNING_RATE = 0.001
```

The number of hyperparameters is also small and includes the following values:

- `MAX_BATCH_EPISODES` and `MAX_BATCH_STEPS`: The limit of episodes and steps we use for training

- `NOISE_STD`: The standard deviation, σ, of the noise used for weight perturbation

- `LEARNING_RATE`: The coefficient used to adjust the weights on the training step

Now let's check the network:

```
class Net(nn.Module):
    def __init__(self, obs_size, action_size):
        super(Net, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(obs_size, 32),
            nn.ReLU(),
            nn.Linear(32, action_size),
            nn.Softmax(dim=1)
        )

    def forward(self, x):
        return self.net(x)
```

The model we're using is a simple one-hidden-layer NN, which gives us the action to take from the observation. We're using PyTorch NN machinery here only for convenience, as we need only the forward pass, but it could be replaced by the multiplication of matrices and nonlinearities.

```
def evaluate(env, net):
    obs = env.reset()
    reward = 0.0
    steps = 0
    while True:
        obs_v = torch.FloatTensor([obs])
        act_prob = net(obs_v)
```

```
            acts = act_prob.max(dim=1)[1]
            obs, r, done, _ = env.step(acts.data.numpy()[0])
            reward += r
            steps += 1
            if done:
                break
        return reward, steps
```

The preceding function plays a full episode using the given policy and returns the total reward and the number of steps. The reward will be used as a fitness value, while the count of steps is needed to limit the amount of time we spend on forming the batch. The action selection is performed deterministically by calculating argmax from the network output. In principle, we could do the random sampling from the distribution, but we've already performed the exploration by adding noise to the network parameters, so the deterministic action selection is fine here.

```
    def sample_noise(net):
        pos = []
        neg = []
        for p in net.parameters():
            noise = np.random.normal(size=p.data.size())
            noise_t = torch.FloatTensor(noise)
            pos.append(noise_t)
            neg.append(-noise_t)
        return pos, neg
```

In the `sample_noise` function, we create random noise with zero mean and unit variance equal to the shape of our network parameters. The function returns two sets of noise tensors: one with positive noise and another with the same random values taken with a negative sign. These two samples are later used in a batch as independent samples. This technique is known as *mirrored sampling* and is used to improve the stability of the convergence. In fact, without the negative noise, the convergence becomes very unstable.

```
    def eval_with_noise(env, net, noise):
        old_params = net.state_dict()
        for p, p_n in zip(net.parameters(), noise):
            p.data += NOISE_STD * p_n
        r, s = evaluate(env, net)
        net.load_state_dict(old_params)
        return r, s
```

The preceding function takes the noise array created by the function we've just seen and evaluates the network with noise added.

To achieve this, we add the noise to the network's parameters and call the `evaluate` function to obtain the reward and number of steps taken. After this, we need to restore the network weights to their original state, which is completed by loading the state dictionary of the network.

The last and the central function of the method is `train_step`, which takes the batch with noise and respective rewards and calculates the update to the network parameters by applying the formula $\theta_{t+1} \leftarrow \theta_t + \alpha \dfrac{1}{n\sigma} \sum_{i=1}^{n} F_i \varepsilon_i$.

```python
def train_step(net, batch_noise, batch_reward, writer, step_idx):
    weighted_noise = None
    norm_reward = np.array(batch_reward)
    norm_reward -= np.mean(norm_reward)
    s = np.std(norm_reward)
    if abs(s) > 1e-6:
        norm_reward /= s
```

In the beginning, we normalize rewards to have zero mean and unit variance, which improves the stability of the method.

```python
for noise, reward in zip(batch_noise, norm_reward):
    if weighted_noise is None:
        weighted_noise = [reward * p_n for p_n in noise]
    else:
        for w_n, p_n in zip(weighted_noise, noise):
            w_n += reward * p_n
```

Then, we iterate every pair (noise, reward) in our batch and multiply the noise values with the normalized reward, summing together the respective noise for every parameter in our policy.

```python
m_updates = []
for p, p_update in zip(net.parameters(), weighted_noise):
    update = p_update / (len(batch_reward) * NOISE_STD)
    p.data += LEARNING_RATE * update
    m_updates.append(torch.norm(update))
writer.add_scalar("update_l2", np.mean(m_updates), step_idx)
```

As a final step, we use the accumulated scaled noise to adjust the network parameters. Technically, what we do here is a gradient ascent, although the gradient was not obtained from backpropagation but from the Monte Carlo sampling method. This fact was also demonstrated in the previously mentioned ES paper [1], where the authors showed that CMA-ES is very similar to the policy gradient methods, differing in just the way that we get the gradients' estimation.

```
if __name__ == "__main__":
    writer = SummaryWriter(comment="-cartpole-es")
    env = gym.make("CartPole-v0")

    net = Net(env.observation_space.shape[0], env.action_space.n)
    print(net)
```

The preparation before the training loop is simple: we create the environment and the network.

```
step_idx = 0
while True:
    t_start = time.time()
    batch_noise = []
    batch_reward = []
    batch_steps = 0
    for _ in range(MAX_BATCH_EPISODES):
        noise, neg_noise = sample_noise(net)
        batch_noise.append(noise)
        batch_noise.append(neg_noise)
        reward, steps = eval_with_noise(env, net, noise)
        batch_reward.append(reward)
        batch_steps += steps
        reward, steps = eval_with_noise(env, net, neg_noise)
        batch_reward.append(reward)
        batch_steps += steps
        if batch_steps > MAX_BATCH_STEPS:
            break
```

Every iteration of the training loop starts with batch creation, where we sample the noise and obtain rewards for both positive and negated noise. When we reach the limit of episodes in the batch, or the limit of the total steps, we stop gathering the data and do a training update.

```
step_idx += 1
m_reward = np.mean(batch_reward)
if m_reward > 199:
    print("Solved in %d steps" % step_idx)
    break

train_step(net, batch_noise, batch_reward,
           writer, step_idx)
```

To perform the update of the network, we call the `train_step()` function that we've already covered.

Its goal is to scale the noise according to the total reward and then adjust the policy weights in the direction of the averaged noise.

```
writer.add_scalar("reward_mean", m_reward, step_idx)
writer.add_scalar("reward_std", np.std(batch_reward),
                  step_idx)
writer.add_scalar("reward_max", np.max(batch_reward),
                  step_idx)
writer.add_scalar("batch_episodes", len(batch_reward),
                  step_idx)
writer.add_scalar("batch_steps", batch_steps, step_idx)
speed = batch_steps / (time.time() - t_start)
writer.add_scalar("speed", speed, step_idx)
print("%d: reward=%.2f, speed=%.2f f/s" % (
    step_idx, m_reward, speed))
```

The final steps in the training loop write metrics into TensorBoard and show the training progress on the console.

Results

Training can be started by just running the program without the arguments:

```
Chapter20$ ./01_cartpole_es.py
Net(
  (net): Sequential(
    (0): Linear(in_features=4, out_features=32, bias=True)
    (1): ReLU()
    (2): Linear(in_features=32, out_features=2, bias=True)
    (3): Softmax(dim=1)
  )
)
1: reward=9.54, speed=6471.63 f/s
2: reward=9.93, speed=7308.94 f/s
3: reward=11.12, speed=7362.68 f/s
4: reward=18.34, speed=7116.69 f/s
...
20: reward=141.51, speed=8285.36 f/s
21: reward=136.32, speed=8397.67 f/s
22: reward=197.98, speed=8570.06 f/s
23: reward=198.13, speed=8402.74 f/s
Solved in 24 steps
```

From my experiments, it usually takes ES about 40-60 batches to solve CartPole. The convergence dynamics for the preceding run are shown on the following charts, with quite a steady result:
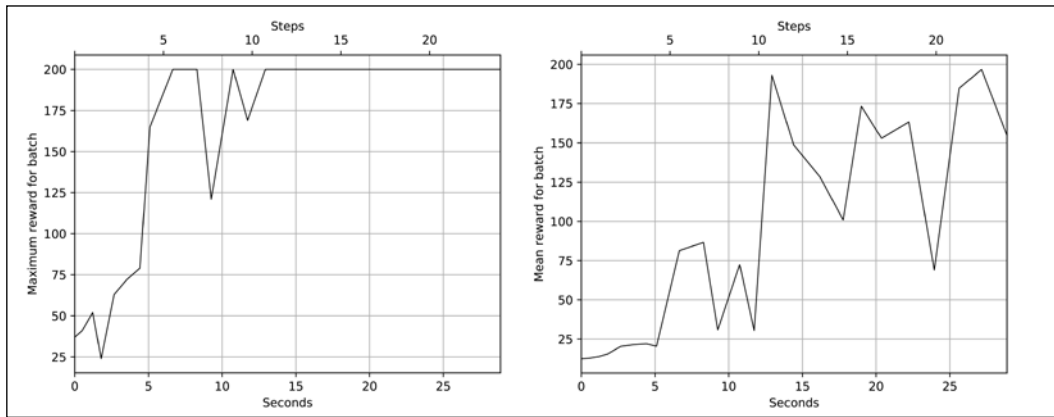
Figure 20.1: ES on CartPole: the maximum reward (left) and mean reward (right) in the training batch
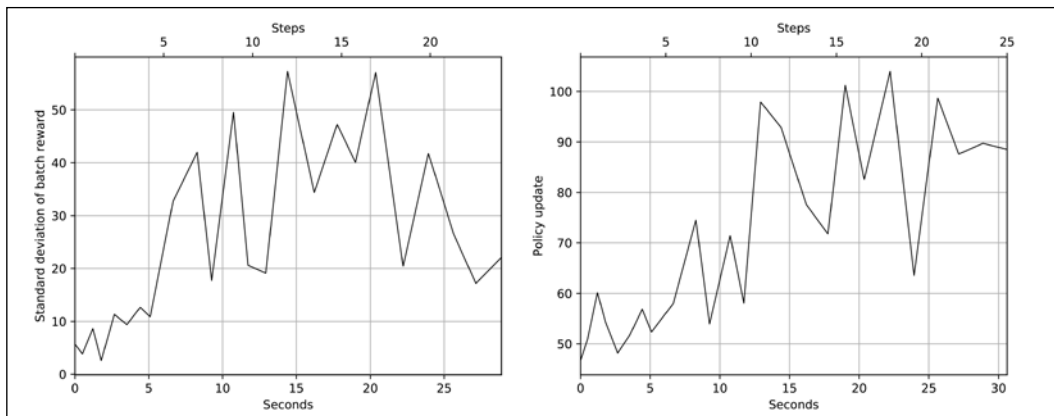
Figure 20.2: The standard deviation of the reward (left) and mean policy update (right)

ES on HalfCheetah

In the next example, we will go beyond the simplest ES implementation and look at how this method can be parallelized efficiently using the *shared seed* strategy proposed by the ES paper [1]. To show this approach, we will use the environment from the Roboschool library that we already experimented with in *Chapter 19, Trust Regions – PPO, TRPO, ACKTR, and SAC*, **HalfCheetah**, which is a continuous action problem where a weird two-legged creature gains reward by running forward without injuring itself.

First, let's discuss the idea of shared seeds. The performance of the ES algorithm is mostly determined by the speed at which we can gather our training batch, which consists of sampling the noise and checking the total reward of the perturbed noise. As our training batch items are independent, we can easily parallelize this step to a large number of workers sitting on remote machines. (That's a bit similar to the example from *Chapter 13*, *Asynchronous Advantage Actor-Critic*, when we gathered gradients from A3C workers.) However, naïve implementation of this parallelization will require a large amount of data to be transferred from the worker machine to the central *master*, which is supposed to combine the noise checked by the workers and perform the policy update. Most of this data is the noise vectors, the size of which is equal to the size of our policy parameters.

To avoid this overhead, a quite elegant solution was proposed by the paper's authors. As noise sampled on a worker is produced by a pseudo-random number generator, which allows us to set the random seed and reproduce the random sequence generated, the worker can transfer to the master only the seed that was used to generate the noise. Then, the master can generate the same noise vector again using the seed. Of course, the seed on every worker needs to be generated randomly to still have a random optimization process. This allows for dramatically decreasing the amount of data that needs to be transferred from workers to the master, improving the scalability of the method. For example, the paper's authors reported linear speed up in optimizations involving 1,440 CPUs in the cloud. In our example, we will check local parallelization using the same approach.

Implementation

The code is placed in `Chapter20/02_cheetah_es.py`. As the code significantly overlaps with the CartPole version, we will focus here only on the differences.

We will begin with the worker, which is started as a separate process using the PyTorch `multiprocessing` wrapper. The worker's responsibilities are simple: for every iteration, it obtains the network parameters from the master process, and then it performs the fixed number of iterations, where it samples the noise and evaluates the reward. The result with the random seed is sent to the master using the queue.

```
RewardsItem = collections.namedtuple(
    'RewardsItem', field_names=['seed', 'pos_reward',
                                'neg_reward', 'steps'])
```

The preceding `namedtuple` structure is used by the worker to send the results of the perturbed policy evaluation. It includes the random seed, the reward obtained with the noise, the reward obtained with the negated noise, and the total number of steps we performed in both tests.

```
def worker_func(worker_id, params_queue, rewards_queue,
                device, noise_std):
    env = make_env()
    net = Net(env.observation_space.shape[0],
              env.action_space.shape[0]).to(device)
    net.eval()

    while True:
        params = params_queue.get()
        if params is None:
            break
        net.load_state_dict(params)
```

On every training iteration, the worker waits for the network parameters to be broadcasted from the master. The value of None means that the master wants to stop the worker.

```
        for _ in range(ITERS_PER_UPDATE):
            seed = np.random.randint(low=0, high=65535)
            np.random.seed(seed)
            noise, neg_noise = sample_noise(net, device=device)
            pos_reward, pos_steps = eval_with_noise(
                env, net, noise, noise_std, device=device)
            neg_reward, neg_steps = eval_with_noise(
                env, net, neg_noise, noise_std, device=device)
            rewards_queue.put(RewardsItem(
                seed=seed, pos_reward=pos_reward,
                neg_reward=neg_reward, steps=pos_steps+neg_steps))
```

The rest is almost the same as the previous example, with the main difference being in the random seed generated and assigned before the noise generation. This allows the master to regenerate the same noise, only from the seed. Another difference lies in the function used by the master to perform the training step.

```
def train_step(optimizer, net, batch_noise, batch_reward,
               writer, step_idx, noise_std):
    weighted_noise = None
    norm_reward = compute_centered_ranks(np.array(batch_reward))
```

In the previous example, we normalized the batch of rewards by subtracting the mean and dividing by the standard deviation. According to the ES paper [1], better results could be obtained using ranks instead of actual rewards. As ES has no assumptions about the fitness function (which is a reward in our case), we can make any rearrangements in the reward that we want, which wasn't possible in the case of DQN, for example.

Here, **rank transformation** of the array means replacing the array with indices of the sorted array. For example, array `[0.1, 10, 0.5]` will have the rank array `[0, 2, 1]`. The `compute_centered_ranks` function takes the array with the total rewards of the batch, calculates the rank for every item in the array, and then normalizes those ranks. For example, an input array of `[21.0, 5.8, 7.0]` will have ranks `[2, 0, 1]`, and the final centered ranks will be `[0.5, -0.5, 0.0]`.

```
for noise, reward in zip(batch_noise, norm_reward):
    if weighted_noise is None:
        weighted_noise = [reward * p_n for p_n in noise]
    else:
        for w_n, p_n in zip(weighted_noise, noise):
            w_n += reward * p_n
m_updates = []
optimizer.zero_grad()
for p, p_update in zip(net.parameters(), weighted_noise):
    update = p_update / (len(batch_reward) * noise_std)
    p.grad = -update
    m_updates.append(torch.norm(update))
writer.add_scalar("update_l2", np.mean(m_updates), step_idx)
optimizer.step()
```

Another major difference in the training function is the use of PyTorch optimizers. To understand why they are used, and how this is possible without doing backpropagation, some explanations are required. First of all, in the ES paper, it was shown that the optimization method used by the ES algorithm is very similar to gradient ascent on the fitness function, with the difference being how the gradient is calculated. The way the stochastic gradient descent (SGD) method is usually applied is that the gradient is obtained from the loss function by calculating the derivative of the network parameters with respect to the loss value. This imposes the limitation on the network and loss function to be differentiable, which is not always the case; for example, the rank transformation performed by the ES method is not differentiable.

On the other hand, optimization performed by ES works differently. We randomly sample the neighborhood of our current parameters by adding the noise to them and calculating the fitness function. According to the fitness function change, we adjust the parameters, which pushes our parameters in the direction of a higher fitness function. The result of this is very similar to gradient-based methods, but the requirements imposed on our fitness function are much looser: the only requirement is our ability to calculate it.

However, if we're estimating some kind of gradient by randomly sampling the fitness function, we can use standard optimizers from PyTorch. Normally, optimizers adjust the parameters of the network using gradients accumulated in the parameters' `grad` fields.

Those gradients are accumulated after the backpropagation step, but due to PyTorch's flexibility, the optimizer doesn't care about the source of the gradients. So, the only thing we need to do is copy the estimated parameters' update in the `grad` fields and ask the optimizer to update them. Note that the update is copied with a negative sign, as optimizers normally perform gradient descent (as in a normal operation, we *minimize* the loss function), but in this case, we want to do gradient ascent. This is very similar to the **actor-critic method**, when the estimated policy gradient is taken with the negative sign, as it shows the direction to *improve* the policy.

The last chunk of different code is taken from the training loop performed by the master process. Its responsibility is to wait for data from worker processes, perform the training update of the parameters, and broadcast the result to the workers. The communication between the master and workers is performed by two sets of queues. The first queue is per-worker and is used by the master to send the current policy parameters to use. The second queue is shared by the workers and is used to send the already mentioned `RewardItem` structure with the random seed and rewards.

```python
params_queues = [
    mp.Queue(maxsize=1)
    for _ in range(PROCESSES_COUNT)
]
rewards_queue = mp.Queue(maxsize=ITERS_PER_UPDATE)
workers = []

for idx, params_queue in enumerate(params_queues):
    p_args = (idx, params_queue, rewards_queue,
            device, args.noise_std)
    proc = mp.Process(target=worker_func, args=p_args)
    proc.start()
    workers.append(proc)

print("All started!")
optimizer = optim.Adam(net.parameters(), lr=args.lr)
```

In the beginning of the master, we create all those queues, start the worker processes, and start the optimizer.

```python
for step_idx in range(args.iters):
    # broadcasting network params
    params = net.state_dict()
    for q in params_queues:
        q.put(params)
```

Every training iteration starts with the network parameters being broadcast to the workers.

```
t_start = time.time()
batch_noise = []
batch_reward = []
results = 0
batch_steps = 0
batch_steps_data = []
while True:
    while not rewards_queue.empty():
        reward = rewards_queue.get_nowait()
        np.random.seed(reward.seed)
        noise, neg_noise = sample_noise(net)
        batch_noise.append(noise)
        batch_reward.append(reward.pos_reward)
        batch_noise.append(neg_noise)
        batch_reward.append(reward.neg_reward)
        results += 1
        batch_steps += reward.steps
        batch_steps_data.append(reward.steps)

    if results == PROCESSES_COUNT * ITERS_PER_UPDATE:
        break
    time.sleep(0.01)
```

Then, in the loop, the master waits for enough data to be obtained from the workers. Every time a new result arrives, we reproduce the noise using the random seed.

```
train_step(optimizer, net, batch_noise, batch_reward,
           writer, step_idx, args.noise_std)
```

As the last step in the training loop, we call the `train_step()` function that you've already seen, which calculates the update from the noise and rewards, and calls the optimizer to adjust the weights.

Results

The code supports the optional `--cuda` flag, but from my experiments, I get zero speeding up from the GPU, as the network is too shallow and the batch size is only one for every parameter's evaluation. This also suggests potential speed improvements by increasing the batch size that we use during the evaluation, which can be done using multiple environments in every worker and carefully working with noise data inside the network. The values shown for every iteration are the mean reward obtained, the speed of training (in observations per second), two timing values (showing how long (in seconds) it took to gather data and perform the training step), and then three values about the episode lengths: the mean, minimum, and maximum number of steps during the episodes.

```
Chapter20$ ./02_cheetah_es.py
Net(
  (mu): Sequential(
    (0): Linear(in_features=26, out_features=64, bias=True)
    (1): Tanh()
    (2): Linear(in_features=64, out_features=64, bias=True)
    (3): Tanh()
    (4): Linear(in_features=64, out_features=6, bias=True)
    (5): Tanh()
  )
)
All started!
0: reward=10.86, speed=1486.01 f/s, data_gather=0.903, train=0.008,
steps_mean=45.10, min=32.00, max=133.00, steps_std=17.62
1: reward=11.39, speed=4648.11 f/s, data_gather=0.269, train=0.005,
steps_mean=42.53, min=33.00, max=65.00, steps_std=8.15
2: reward=14.25, speed=4662.10 f/s, data_gather=0.270, train=0.006,
steps_mean=42.90, min=36.00, max=59.00, steps_std=5.65
3: reward=14.33, speed=4901.02 f/s, data_gather=0.257, train=0.006,
steps_mean=43.00, min=35.00, max=56.00, steps_std=5.01
4: reward=14.95, speed=4566.68 f/s, data_gather=0.281, train=0.005,
steps_mean=43.60, min=37.00, max=54.00, steps_std=4.41
```

The dynamics of the training show very quick policy improvement in the beginning
(in just 100 updates, which is seven minutes of training, the agent was able to reach
the score of 700-800; however, afterward, it got stuck and wasn't able to switch from
keeping the balance (when Cheetah can reach up to 900-1,000 total reward) to the
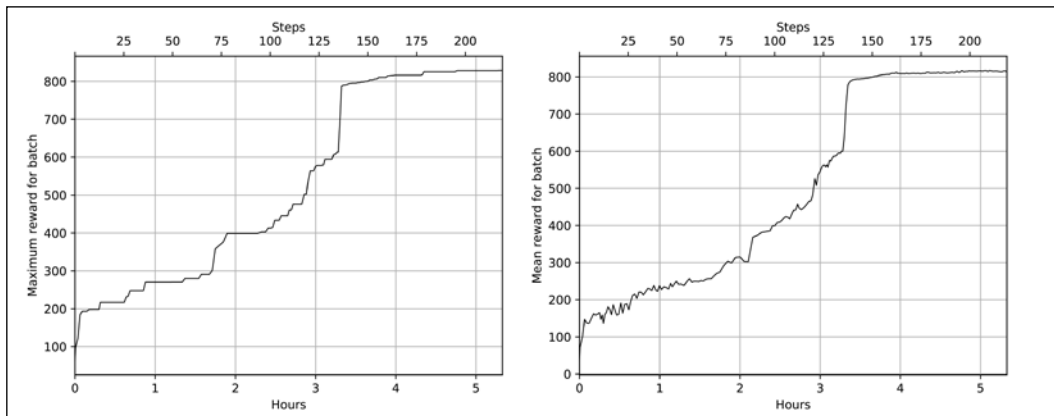running mode with a much higher reward of 2,500 or more:
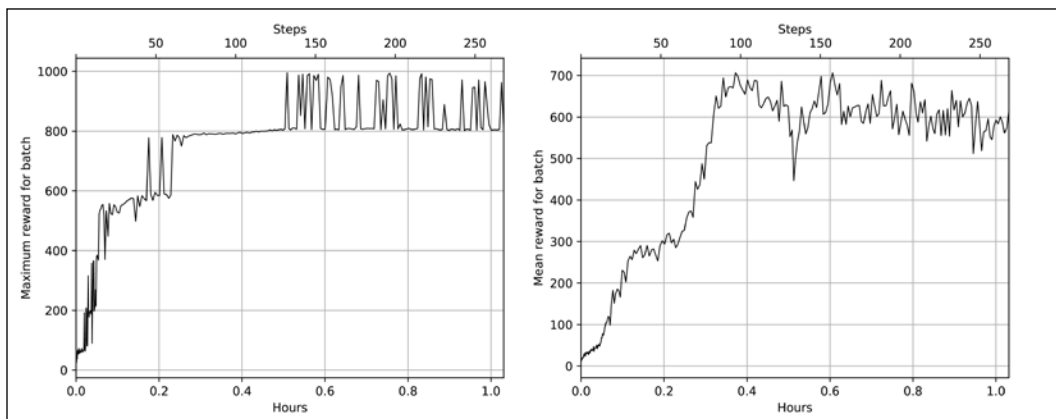
Figure 20.3: ES on HalfCheetah: the maximum reward (left) and mean reward (right)
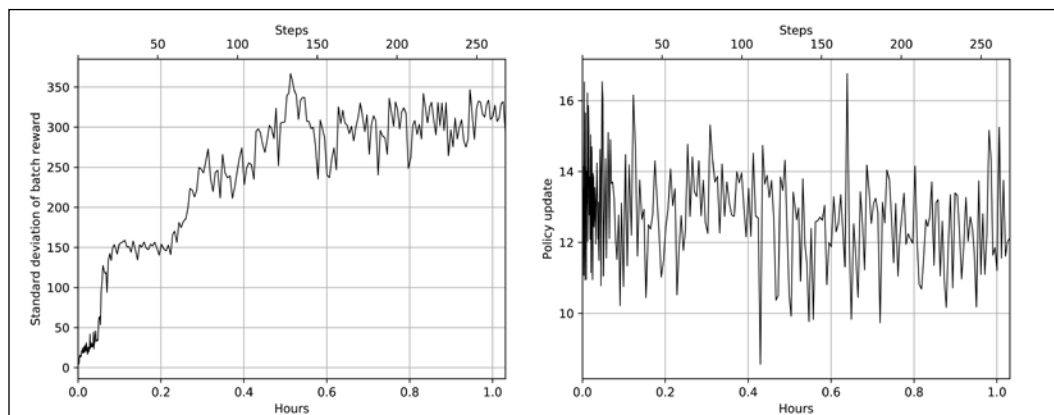
Figure 20.4: The standard deviation of reward (left) and policy update (right)

Genetic algorithms

Another class of black-box methods that has recently become a popular alternative to the value-based and policy gradient methods is genetic algorithms (GA). It is a large family of optimization methods with more than two decades of history behind it and a simple core idea of generating a population of *N* individuals, each of which is evaluated with the fitness function. Every individual means some combination of model parameters. Then, some subset of top performers is used to produce (called *mutation*) the next generation of the population. This process is repeated until we're satisfied with the performance of our population.

There are a lot of different methods in the GA family, for example, how to complete the mutation of the individuals for the next generation or how to rank the performers. Here, we will consider the simple GA method with some extensions, published in the paper by *Felipe Petroski Such*, *Vashisht Madhavan*, and others called *Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning* [2].

In this paper, the authors analyzed the simple GA method, which performs Gaussian noise perturbation of the parent's weights to perform mutation. On every iteration, the top performer was copied without modification. In an algorithm form, the steps of a simple GA method can be written as follows:

1. Initialize the mutation power, σ, the population size, *N*, the number of selected individuals, *T*, and the initial population, P^0, with *N* randomly initialized policies and their fitness: $F^0 = \{F(P_i^0) | i = 1 \dots N\}$

2. For generation $g = 1 \dots G$:

   1. Sort generation P^{g-1} in the descending order of the fitness function value F^{g-1}

   2. Copy elite $P_1^g = P_1^{g-1}, F_1^g = F_1^{g-1}$

   3. For individual $i = 2 \dots N$:

      ◦ k = the randomly select parent from $1 \dots T$

      ◦ Sample $\varepsilon \sim \mathcal{N}(0, I)$

      ◦ Mutate the parent: $P_i^g = P_i^{g-1} + \sigma\varepsilon$

      ◦ Get its fitness: $F_i^g = F(P_i^g)$

There have been several improvements to this basic method from the paper [2], which we will discuss later. For now, let's check the implementation of the core algorithm.

GA on CartPole

The source code is in `Chapter20/03_cartpole_ga.py`, and it has a lot in common with our ES example. The difference is in the lack of the gradient ascent code, which was replaced by the network mutation function as follows:

```
def mutate_parent(net):
    new_net = copy.deepcopy(net)
    for p in new_net.parameters():
        noise = np.random.normal(size=p.data.size())
        noise_t = torch.FloatTensor(noise)
        p.data += NOISE_STD * noise_t
    return new_net
```

The goal of the function is to create a mutated copy of the given policy by adding a random noise to all weights. The parent's weights are kept untouched, as a random selection of the parent is performed with replacement, so this network could be used again later.

```
NOISE_STD = 0.01
POPULATION_SIZE = 50
PARENTS_COUNT = 10
```

The count of hyperparameters is even smaller than with ES and includes the standard deviation of the noise added-on mutation, the population size, and the number of top performers used to produce the subsequent generation.

```
if __name__ == "__main__":
    writer = SummaryWriter(comment="-cartpole-ga")
```

```
env = gym.make("CartPole-v0")

gen_idx = 0
nets = [
    Net(env.observation_space.shape[0], env.action_space.n)
    for _ in range(POPULATION_SIZE)
]
population = [
    (net, evaluate(env, net))
    for net in nets
]
```

Before the training loop, we create the population of randomly initialized networks and obtain their fitness.

```
while True:
    population.sort(key=lambda p: p[1], reverse=True)
    rewards = [p[1] for p in population[:PARENTS_COUNT]]
    reward_mean = np.mean(rewards)
    reward_max = np.max(rewards)
    reward_std = np.std(rewards)

    writer.add_scalar("reward_mean", reward_mean, gen_idx)
    writer.add_scalar("reward_std", reward_std, gen_idx)
    writer.add_scalar("reward_max", reward_max, gen_idx)
    print("%d: reward_mean=%.2f, reward_max=%.2f, "
          "reward_std=%.2f" % (
        gen_idx, reward_mean, reward_max, reward_std))
    if reward_mean > 199:
        print("Solved in %d steps" % gen_idx)
        break
```

In the beginning of every generation, we sort the previous generation according to their fitness and record statistics about future parents.

```
    prev_population = population
    population = [population[0]]
    for _ in range(POPULATION_SIZE-1):
        parent_idx = np.random.randint(0, PARENTS_COUNT)
        parent = prev_population[parent_idx][0]
        net = mutate_parent(parent)
        fitness = evaluate(env, net)
        population.append((net, fitness))
    gen_idx += 1
```

In a separate loop over new individuals to be generated, we randomly sample a parent, mutate it, and evaluate its fitness score.

Results

Despite the simplicity of the method, it works even better than ES, solving the CartPole environment in just several generations. In my experiments with the preceding code, it takes five to 15 generations to solve the environment:

```
Chapter20$ ./03_cartpole_ga.py
0: reward_mean=39.90, reward_max=131.00, reward_std=32.97
1: reward_mean=82.00, reward_max=145.00, reward_std=33.14
2: reward_mean=131.40, reward_max=154.00, reward_std=15.94
3: reward_mean=151.90, reward_max=200.00, reward_std=19.45
4: reward_mean=170.20, reward_max=200.00, reward_std=32.46
5: reward_mean=184.00, reward_max=200.00, reward_std=14.09
6: reward_mean=187.50, reward_max=200.00, reward_std=16.87
7: reward_mean=186.60, reward_max=200.00, reward_std=15.41
8: reward_mean=191.50, reward_max=200.00, reward_std=11.67
9: reward_mean=199.50, reward_max=200.00, reward_std=1.20
Solved in 9 steps
```

The dynamics of the convergence are shown on the following charts.
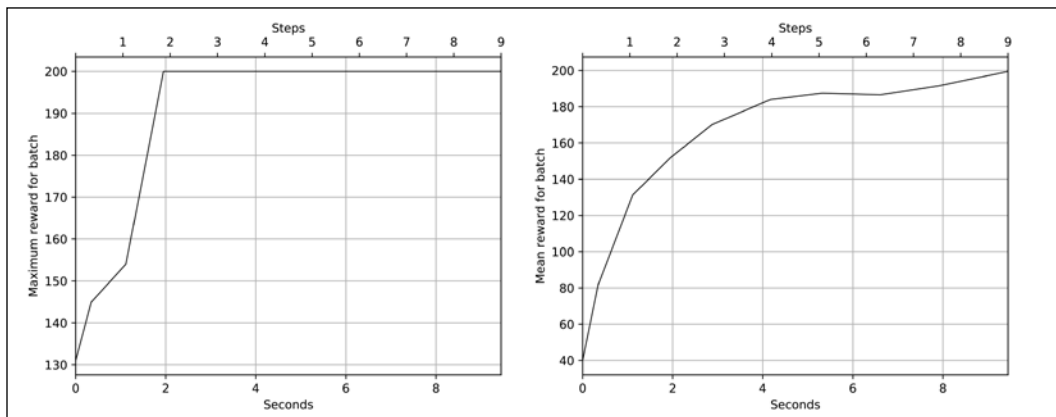
Figure 20.5: GA on CartPole: the maximum reward (left) and mean reward (right)
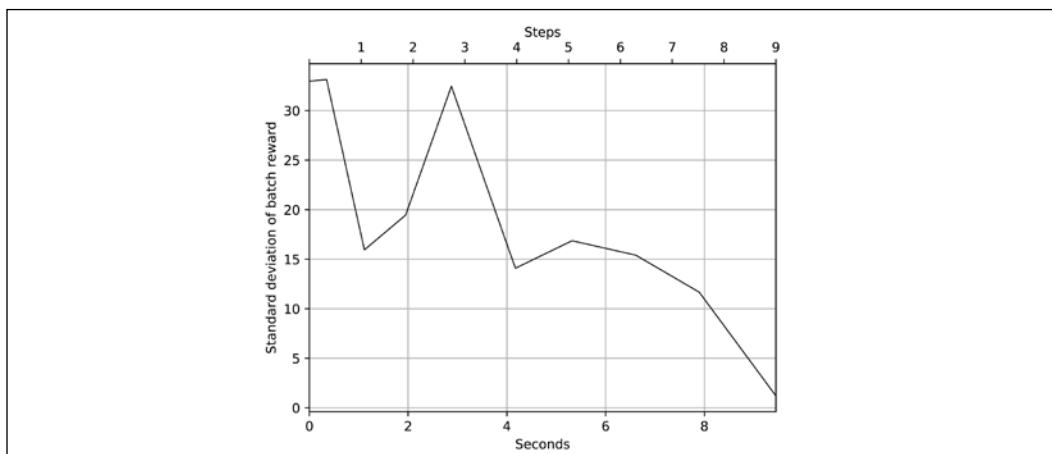
Figure 20.6: The standard deviation of the reward in the batch

GA tweaks

In the *Deep Neuroevolution* paper [2], the authors checked two tweaks to the basic GA algorithm. The first, with the name **deep GA**, aimed to increase the scalability of the implementation, and the second, called **novelty search**, was an attempt to replace the reward objective with a different metric of the episode. In the example used in the following *GA on HalfCheetah* section, we will implement the first improvement, whereas the second one is left as an optional exercise.

Deep GA

Being a gradient-free method, GA is potentially even more scalable than ES methods in terms of speed, with more CPUs involved in the optimization. However, the simple GA algorithm that you have seen has a similar bottleneck to ES methods: the policy parameters have to be exchanged between the workers. In the previously mentioned paper [2], the authors proposed a trick similar to the shared seed approach, but taken to an extreme. They called it deep GA, and at its core, the policy parameters are represented as a list of random seeds used to create this particular policy's weights.

In fact, the initial network's weights were generated randomly on the first population, so the first seed in the list defines this initialization. On every population, mutations are also fully specified by the random seed for every mutation. So, the only thing we need to reconstruct the weights is the seeds themselves. In this approach, we need to reconstruct the weights on every worker, but usually this overhead is much less than the overhead of transferring full weights over the network.

Novelty search

Another modification to the basic GA method is novelty search (NS), which was proposed by *Lehman and Stanley* in their paper, *Abandoning Objectives: Evolution through the Search for Novelty Alone*, which was published in 2011 [3].

The idea of NS is to change the objective in our optimization. We're no longer trying to increase our total reward from the environment, but rather reward the agent for exploring the behavior that it has never checked before (that is, *novel*). According to the authors' experiments on the maze navigation problem, with many traps for the agent, NS works much better than other reward-driven approaches.

To implement NS, we define the so-called **behavior characteristic** (**BC**) (π), which describes the behavior of the policy and a distance between two BCs. Then, the k-nearest neighbors approach is used to check the novelty of the new policy and drive the GA according to this distance. In the *Deep Neuroevolution* paper [2], sufficient exploration by the agent was needed. The approach of NS significantly outperformed the ES, GA, and other more traditional approaches to RL problems.

GA on HalfCheetah

In our final example in this chapter, we will implement the parallelized deep GA on the HalfCheetah environment. The complete code is in `Chapter20/04_cheetah_ga.py`. The architecture is very close to the parallel ES version, with one master process and several workers. The goal of every worker is to evaluate the batch of networks and return the result to the master, which merges partial results into the complete population, ranks the individuals according to the obtained reward, and generates the next population to be evaluated by the workers.

Every individual is encoded by a list of random seeds used to initialize the initial network weights and all subsequent mutations. This representation allows very compact encoding of the network, even when the number of parameters in the policy is not very large. For example, in our network with two hidden layers of 64 neurons, we have 6,278 float values (the input is 26 values and the action is six floats). Every float occupies 4 bytes, which is the same size used by the random seed. So, the deep GA representation proposed by the paper will be smaller up to 6,278 generations in the optimization.

In our example, we will perform parallelization on local CPUs, so the amount of data transferred back and forth doesn't matter much; however, if you have a couple of hundred cores to utilize, the representation might become a significant issue.

```
NOISE_STD = 0.01
POPULATION_SIZE = 2000
PARENTS_COUNT = 10
```

```
WORKERS_COUNT = 6
SEEDS_PER_WORKER = POPULATION_SIZE // WORKERS_COUNT
MAX_SEED = 2**32 - 1
```

The set of hyperparameters is the same as in the CartPole example, with the difference of a larger population size.

```
def mutate_net(net, seed, copy_net=True):
    new_net = copy.deepcopy(net) if copy_net else net
    np.random.seed(seed)
    for p in new_net.parameters():
        noise = np.random.normal(size=p.data.size())
        noise_t = torch.FloatTensor(noise)
        p.data += NOISE_STD * noise_t
    return new_net
```

There are two functions used to build the networks based on the seeds given. The first one performs one mutation on the already created policy network, and it can perform the mutation in place or by copying the target network based on arguments. (Copying is needed for the first generation.)

```
def build_net(env, seeds):
    torch.manual_seed(seeds[0])
    net = Net(env.observation_space.shape[0],
              env.action_space.shape[0])
    for seed in seeds[1:]:
        net = mutate_net(net, seed, copy_net=False)
    return net
```

The second function creates the network from scratch using the list of seeds. The first seed is passed to PyTorch to influence the network initialization, and subsequent seeds are used to apply network mutations.

The worker function obtains the list of seeds to evaluate and outputs individual `OutputItem` tuples for every result obtained. The function maintains the cache of networks to minimize the amount of time spent recreating the parameters from the list of seeds. This cache is cleared for every generation, as every new generation is created from the current generation winners, so there is only a tiny chance that old networks can be reused from the cache.

```
OutputItem = collections.namedtuple(
    'OutputItem', field_names=['seeds', 'reward', 'steps'])

def worker_func(input_queue, output_queue):
    env = gym.make("RoboschoolHalfCheetah-v1")
    cache = {}
```

```
while True:
    parents = input_queue.get()
    if parents is None:
        break
    new_cache = {}
    for net_seeds in parents:
        if len(net_seeds) > 1:
            net = cache.get(net_seeds[:-1])
            if net is not None:
                net = mutate_net(net, net_seeds[-1])
            else:
                net = build_net(env, net_seeds)
        else:
            net = build_net(env, net_seeds)
        new_cache[net_seeds] = net
        reward, steps = evaluate(env, net)
        output_queue.put(OutputItem(
            seeds=net_seeds, reward=reward, steps=steps))
    cache = new_cache
```

The code of the master process is also straightforward. For every generation, we send the current population's seeds to workers for evaluation and wait for the results. Then, we sort the results and generate the next population based on the top performers. On the master's side, the mutation is just a seed number generated randomly and appended to the list of seeds of the parent.

```
batch_steps = 0
population = []
while len(population) < SEEDS_PER_WORKER * WORKERS_COUNT:
    out_item = output_queue.get()
    population.append((out_item.seeds, out_item.reward))
    batch_steps += out_item.steps
if elite is not None:
    population.append(elite)
population.sort(key=lambda p: p[1], reverse=True)

elite = population[0]
for worker_queue in input_queues:
    seeds = []
    for _ in range(SEEDS_PER_WORKER):
        parent = np.random.randint(PARENTS_COUNT)
        next_seed = np.random.randint(MAX_SEED)
        s = list(population[parent][0]) + [next_seed]
        seeds.append(tuple(s))
    worker_queue.put(seeds)
```

Results

To start the training, just launch the source code file. For every generation, it shows the result on the console:

```
Chapter20$ ./04_cheetah_ga.py
0: reward_mean=31.28, reward_max=34.37, reward_std=1.46, speed=5495.65 f/s
1: reward_mean=45.41, reward_max=54.74, reward_std=3.86, speed=6748.35 f/s
2: reward_mean=60.74, reward_max=69.25, reward_std=5.33, speed=6749.70 f/s
3: reward_mean=67.70, reward_max=84.29, reward_std=8.21, speed=6070.31 f/s
4: reward_mean=69.85, reward_max=86.38, reward_std=9.37, speed=6612.48 f/s
5: reward_mean=65.59, reward_max=86.38, reward_std=7.95, speed=6542.46 f/s
6: reward_mean=77.29, reward_max=98.53, reward_std=11.13, speed=6949.59 f/s
```

The overall dynamics are similar to ES experiments on the same environment, with the same problems getting out of the local optima of 1,010 reward. After five hours of training and 250 generations, the agent was able to learn how to stand perfectly but wasn't able to figure out that running could bring more reward. Possibly, the NS method could overcome this issue.

The following charts show the optimization dynamics.

Figure 20.7: GA on HalfCheetah: the maximum reward (left) and mean reward (right)
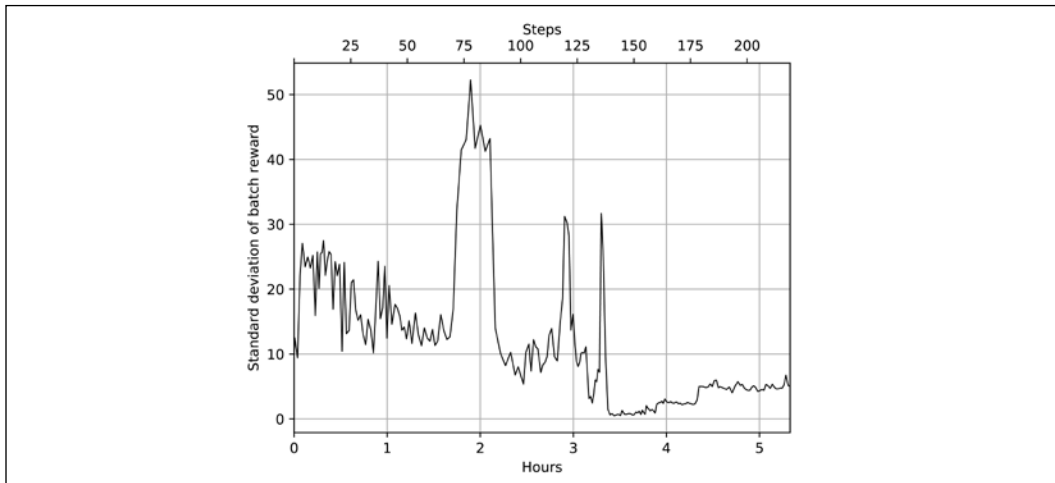
Figure 20.8: The standard deviation of the reward in the batch

Summary

In this chapter, you saw two examples of black-box optimization methods: ES and GA, which can provide competition for other analytical gradient methods. Their strength lies in good parallelization on a large number of resources and the smaller number of assumptions that they have on the reward function.

In the next chapter, we will take a look at a different sphere of modern RL development: *model-based methods*.

References

1. *Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, Ilya Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning*, arXiv:1703.03864

2. *Felipe Petroski Such, Vashisht Madhavan*, and others. *Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning*, arXiv:1712.06567

3. *Joel Lehman* and *Kenneth O. Stanley, Abandoning Objectives: Evolution through the Search for Novelty Alone, Evolutionary Computation*. Volume 19 Issue 2, Summer 2011, Pages 189-223.

21
Advanced Exploration

Next, we will talk about the topic of exploration in reinforcement learning (RL). It has been mentioned several times in the book that the exploration/exploitation dilemma is a fundamental thing in RL and very important for efficient learning. However, in the previous examples, we used quite a trivial approach to exploring the environment, which was, in most cases, ε-greedy action selection. Now it's time to go deeper into the exploration subfield of RL.

In this chapter, we will:

- Discuss why exploration is such a fundamental topic in RL
- Explore the effectiveness of the epsilon-greedy (ε-greedy) approach
- Take a look at alternatives and try them on different environments

Why exploration is important

In this book, lots of environments and methods have been discussed and in almost every chapter, exploration was mentioned. Very likely, you've already got ideas about why it's important to explore the environment effectively, so I'm just going to give a list of the main reasons.

Before that, it might be useful to agree on the term "effective exploration." In theoretical RL, a strict definition of this exists, but the high-level idea is simple and intuitive. Exploration is effective when we don't waste time in states of the environment that have already been seen by and are familiar to the agent. Rather than taking the same actions again and again, the agent needs to look for a new experience. As we've already discussed, exploration has to be balanced by exploitation, which is the opposite and means using our knowledge to get the best reward in the most efficient way. Let's now quickly discuss why we might be interested in effective exploration in the first place.

First of all, good exploration of the environment might have a fundamental influence on our ability to learn a good policy. If the reward is sparse and the agent obtains a good reward on some rare conditions, it might experience a positive reward only once in many episodes, so the ability of the learning process to explore the environment effectively and fully might bring more samples with a good reward that the method could learn from.

In some cases, which are very frequent in practical applications of RL, a lack of good exploration might mean that the agent will never experience a positive reward at all, which makes everything else useless. If you have no good samples to learn from, you can have the most efficient RL method, but the only thing it will learn is that there is no way to get a good reward. This is the case for lots of practically interesting problems around us. Later in the chapter, we will take a closer look at the MountainCar environment, which has trivial dynamics, but due to a sparse reward scheme is quite tricky to solve.

On the other hand, even if the reward is not sparse, effective exploration increases the training speed due to better convergence and training stability. This happens because our sample from the environment becomes more diverse and requires less communication with the environment. As a result, our RL method has the chance to learn a better policy in a shorter time.

What's wrong with ε-greedy?

Throughout the book, we have used the ε-greedy exploration strategy as a simple, but still acceptable, approach to exploring the environment. The underlying idea behind ε-greedy is to take a random action with the probability of ε; otherwise, (with $1 - \varepsilon$ probability) we act greedily. By varying the ε hyperparameter, we can change the exploration ratio. This approach was used in most of the value-based methods described in the book.

Quite a similar idea was used in policy-based methods, when our network returns the probability distribution over actions to take. To prevent the network from becoming too certain about actions (by returning a probability of 1 for a specific action and 0 for others), we added the entropy loss, which is just the entropy of the probability distribution multiplied by some hyperparameter. In the early stages of the training, this entropy loss pushes our network toward taking random actions (by regularizing the probability distribution), but in later stages, when we have explored the environment enough and our reward is relatively high, the policy gradient dominates over this entropy regularization. But this hyperparameter requires tuning to work properly.

At a high level, both approaches are doing the same thing: to explore the environment, we introduce randomness into our actions. However, recent research shows that this approach is very far from being ideal:

- In the case of value iteration methods, random actions taken in some pieces of our trajectory introduce bias into our Q-value estimation. The Bellman equation assumes that the Q-value for the next state is obtained from the action with the largest Q. In other words, the rest of the trajectory is supposed to be from our optimal behavior. But with ε-greedy, we might take not the optimal action, but just a random action, and this piece of trajectory will be stored in the replay buffer for a long time, until our ε is decayed and old samples are pushed from the buffer. Before that happens, we will learn wrong Q-values.

- With random actions injected into our trajectory, our policy changes with every step. With the frequency defined by the value of ε or the entropy loss coefficient, our trajectory constantly switches from a random policy to our current policy. This might lead to poor state space coverage in situations when multiple steps are needed to reach some isolated areas in the environment's state space.

To illustrate the last issue, let's consider a simple example taken from the paper by Strehl and Littman called *An analysis of model-based Interval Estimation for Markov Decision Processes*, which was published in 2008 [1]. The example is called "River Swim" and it models a river that the agent needs to cross. The environment contains six states and two actions: *left* and *right*. The following is the transition diagram for the first two states: 1 and 2.
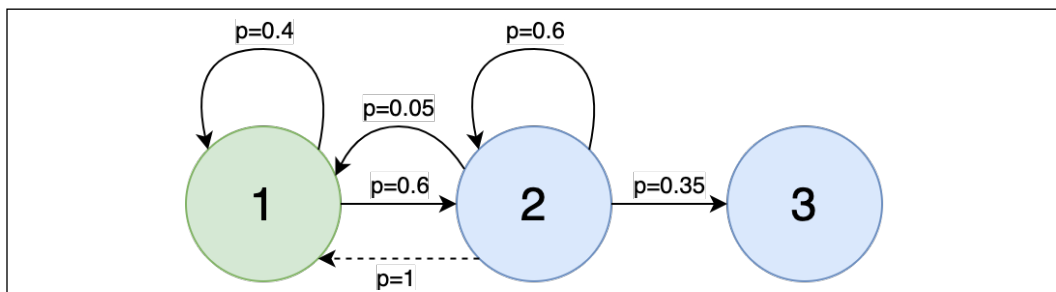
Figure 21.1: Transitions for the first two states of the River Swim environment

In the first state (shown as the first/green circle) the agent stands on the ground of the riverbank. The only action is *right*, which means entering the river and swimming against the current to state 2. But the current is strong, and our *right* action from state 1 succeeds only with a probability of 60% (the solid line from state 1 to state 2).

With a probability of 40%, the current keeps us in state 1 (the solid line connecting state 1 to itself).

In the second state (the second/blue circle) we have two actions: *left*, which is shown by the dashed line connecting states 2 and 1 (this action always succeeds), and *right*, which means swimming against the current to state 3. As before, swimming against the current is hard, so the probability of getting from state 2 to state 3 is just 35% (the solid line connecting states 2 and 3). With a probability of 60%, our *left* action ends up in the same state (the curved solid line connecting state 2 to itself). But sometimes, despite our efforts, our *left* action ends up in state 1, which happens with a 5% probability (the curved line connecting states 2 and 1).

As I've said, there are six states in River Swim, but the transitions for states 3, 4, and 5 are identical to those for state 2. The full state diagram is shown in the following figure. The last state, 6, is similar to state 1, so there is only one action available there: *left*, meaning to swim back.
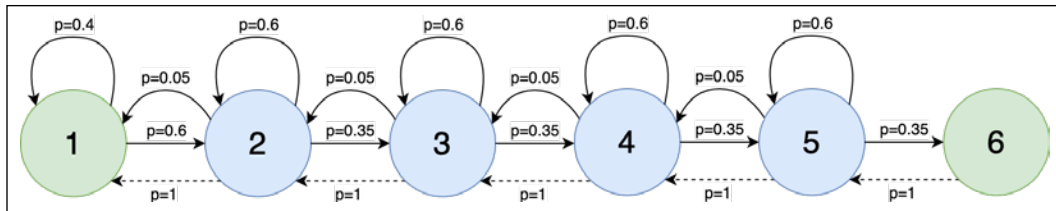
Figure 21.2: The full transition diagram for the River Swim environment

In terms of the reward, the agent gets a small reward of 1 for the transition between states 1 to 5, but it gets a very high reward of 1,000 for getting into state 6, which acts as compensation for all the efforts of swimming against the current.

Despite the simplicity of the environment, its structure creates a problem for the ε-greedy strategy being able to fully explore the state space. To check this, I implemented a very simple simulation of this environment, which is in `Chapter21/riverswim.py`. The simulated agent always acts randomly ($\varepsilon = 1$) and the result of the simulation is the frequency of various state visits. The number of steps the agent can take in one episode is limited to 10, but this can be changed using the command line. The code is very simple, so let's just check the results of the experiments:

```
$ ./riverswim.py
1: 40
2: 39
3: 17
4: 3
```

```
5: 1
6: 0
```

With default command-line options, the simulation of 100 steps (10 episodes) was performed. As you can see, the agent never reached state 6 and was only in state 5 once. By increasing the number of episodes, the situation became a bit better, but not much:

```
$ ./riverswim.py -n 1000
1: 441
2: 452
3: 93
4: 12
5: 2
6: 0
```

With 10 times more episodes simulated, we still didn't visit state 6, so the agent had no idea about the large reward there.

```
$ ./riverswim.py -n 10000
1: 4056
2: 4506
3: 1095
4: 281
5: 57
6: 5
```

Only with 1,000 episodes simulated were we able to get to state 6, but only five times, which is 0.05% of all the steps. It's not very likely that the training will be efficient, even with the best RL method. Also, that's just six states; imagine how inefficient it will be with 20 or 50 states, which is not that unlikely; for example, in Atari games, there might be hundreds of decisions to be made before something interesting happens.

If you want to, you can experiment with the riverswim.py tool, which allows you to change the random seed, the amount of steps in the episode, the total amount of steps, and even the amount of states in the environment.

This simple example illustrates the issue with random actions in exploration. By acting randomly, our agent does not try to actively explore the environment; it just hopes that random actions will bring something new to its experience, which is not always the best thing to do.

Let's now discuss more efficient approaches to the exploration problem.

Alternative ways of exploration

In this section, we will cover an overview of a set of alternative approaches to the exploration problem. This won't be an exhaustive list of approaches that exist, but rather will provide an outline of the landscape.

We're going to check three different approaches to exploration:

- **Randomness in the policy**, when stochasticity is added to the policy that we use to get samples. The method in this family is noisy networks, which we have already covered.

- **Count-based methods**, which keep track of the count of times the agent has seen the particular state. We will check two methods: the direct counting of states and the pseudo-count method.

- **Prediction-based methods**, which try to predict something from the state and from the quality of the prediction. We can make judgements about the familiarity of the agent with this state. To illustrate this approach, we will take a look at the policy distillation method, which has shown state-of-the-art results on hard-exploration Atari games like Montezuma's Revenge.

In the next section, we will implement the methods described to solve a toy, but still challenging, problem called MountainCar. This will allow us to better understand the methods, the way they could be implemented, and their behavior. After that, we will try to tackle a harder problem from the Atari suite.

Noisy networks

Let's start with an approach that is already familiar to us. We covered the method called noisy networks [2] in *Chapter 8, DQN Extensions*, when we discussed deep Q-network (DQN) extensions. The idea is to add Gaussian noise to the network's weights and learn the noise parameters (mean and variance) using backpropagation, in the same way that we learn the model's weights. In *Chapter 8*, this simple approach gave a significant boost in Pong training.

At the high level, this might look very similar to the ε-greedy approach, but the authors of the paper [2] claimed a difference. The difference lies in the way we apply stochasticity to the network. In ε-greedy, randomness is added to the actions. In noisy networks, randomness is injected into part of the network itself (several fully connected layers close to the output), which means adding stochasticity to our current policy. In addition, parameters of the noise might be learned during the training, so the training process might increase or decrease this policy randomness if needed.

According to the paper, the noise in noisy layers needs to be sampled from time to time, which means that our training samples are not produced by our current policy, but by the ensemble of policies. With this, our exploration becomes directed, as random values added to the weights produce a different policy.

Count-based methods

This family of methods is based on the intuition to visit states that were not explored before. In simple cases, when the state space is not very large and different states are easily distinguishable from each other, we just count the number of times we have seen the state or state + action and prefer to get to the states for which this count is low.

This could be implemented as an *intrinsic reward* that is added to the reward obtained from the environment (which is called *extrinsic reward* in this context). One of the options to formulate such a reward is to use the **bandits exploration** approach (which is an important variation of the RL problem): $r_i = c \dfrac{1}{\sqrt{\tilde{N}(s)}}$. Here, $\tilde{N}(s)$ is a count or pseudo-count of times we have seen the state, s, and value c defines the weight of the intrinsic reward.

If the amount of states is small, like in the tabular learning case, we can just count them. In more difficult cases, when there are too many states, some transformation of the states needs to be introduced, like the hashing function or some embeddings of the states.

For pseudo-count methods, $\tilde{N}(s)$ is factorized into the density function and the total amount of states visited. There are several different methods for how to do this, but they might be tricky to implement, so we won't deal with complex cases in this chapter. If you're curious, you can check the paper from Georg Ostrovski et al. called *Count-Based Exploration with Neural Density Models*, 2017, arxiv:1703.01310v2 [3].

A special case of introducing the intrinsic reward is called *curiosity-driver exploration*, when we don't take the reward from the environment into account at all. In that case, the training and exploration is driven 100% by the novelty of the agent's experience. Surprisingly, this approach might be very efficient not only in discovering new states in the environment, but also in learning quite good policies.

Prediction-based methods

The third family of exploration methods is based on another idea of predicting something from the environment data. If the agent can make accurate predictions, it means the agent has been in this situation enough and it isn't worth exploring it.

But if something unusual happens and our prediction is significantly off, it might mean that we need to pay attention to the state that we're currently in. There are many different approaches to doing this, but in this chapter, we will talk about one recent paper that implemented this approach and was published by Yuri Burda et al. It is called *Exploration by random network distillation* and was published in 2018, arxiv:1810.12894 [4]. The authors were able to reach state-of-the-art results in so-called hard-exploration games in Atari.

The approach used in the paper is quite simple: we add the intrinsic reward, which is calculated from the ability of one neural network (NN) (which is being trained) to predict the output from another randomly initialized (untrained) NN. The input to both NNs is the current observation, and the intrinsic reward is proportional to the mean squared error (MSE) of the prediction.

MountainCar experiments

In this section, we will try to implement and compare the effectiveness of different exploration approaches on a simple, but still challenging, environment, which could be classified as a "classical RL" problem that is very similar to the familiar CartPole. But in contrast to CartPole, the MountainCar problem is quite challenging from an exploration point of view.

The problem's illustration is shown in the following figure and it consists of a small car starting from the bottom of the valley. The car can move left and right, and the goal is to reach the top of the mountain on the right.
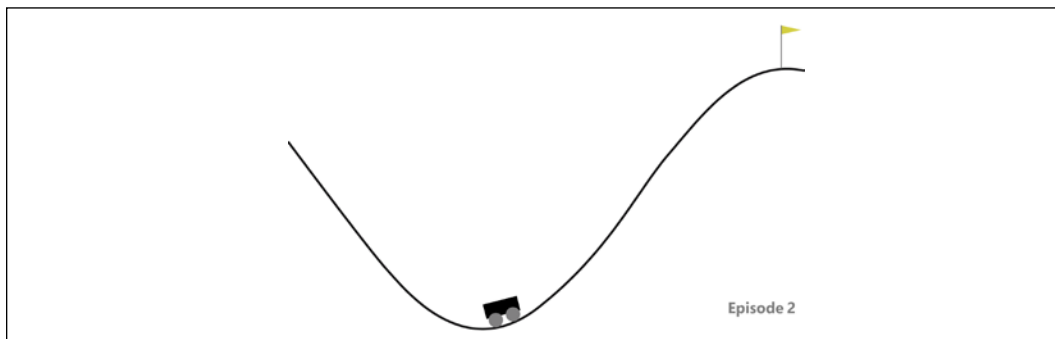
Figure 21.3: The MountainCar environment

The trick here is in the environment's dynamics and the action space. To reach the top, the actions need to be applied in a particular way to swing the car back and forth to speed it up. In other words, the agent needs to apply the actions for several time steps to make the car go faster and eventually reach the top.

Obviously, this coordination of actions is not something that is easy to achieve with just random actions, so the problem is hard from the exploration point of view and very similar to our River Swim example.

In Gym, this environment has the name `MountainCar-v0` and it has a very simple observation and action space. Observations are just two numbers: the first one gives the horizontal position of the car and the second value is the car's velocity. The action could be 0, 1, or 2, where 0 means pushing the car to the left, 1 applies no force, and 2 pushes the car to the right. The following is a very simple illustration of this in Python REPL.

```
>>> import gym
>>> e = gym.make("MountainCar-v0")
>>> e.reset()
array([-0.53143793,  0.        ])
>>> e.observation_space
Box(2,)
>>> e.action_space
Discrete(3)
>>> e.step(0)
(array([-0.51981535, -0.00103615]), -1.0, False, {})
>>> e.step(0)
(array([-0.52187987, -0.00206452]), -1.0, False, {})
>>> e.step(0)
(array([-0.52495728, -0.00307741]), -1.0, False, {})
>>> e.step(0)
(array([-0.52902451, -0.00406722]), -1.0, False, {})
>>> e.step(1)
(array([-0.53305104, -0.00402653]), -1.0, False, {})
>>> e.step(1)
(array([-0.53700669, -0.00395565]), -1.0, False, {})
>>> e.step(1)
(array([-0.54086181, -0.00385512]), -1.0, False, {})
>>> e.step(1)
(array([-0.54458751, -0.0037257 ]), -1.0, False, {})
```

As you can see, on every step we get the reward of -1, so the agent needs to learn how to get to the goal as soon as possible to get as little total negative reward as possible. By default, the amount of steps is limited to 200, so if we haven't got to the goal (which happens most of the time), our total is –200.

The DQN method with ε-greedy

The first method that we will check will be our traditional ε-greedy approach to exploration. It is implemented in the source file `Chapter21/mcar_dqn.py`. I won't include the source code here, as it is already familiar to you. The source file implements various exploration strategies on top of the DQN method, to make a selection between them. Option `-p` allows you to specify the hyperparameters set. To launch the normal ε-greedy method, the option `-p egreedy` needs to be passed. The hyperparameters are decreasing ε from 1.0 to 0.02 for the first 10^5 training steps.

The training is quite fast; it takes just two to three minutes to do 10^5 training steps. But from the charts that follow, it is obvious that during those 10^5 steps, which was 500 episodes, we didn't reach the goal state even once. That's really bad news, as our ε has decayed, so we will do no more exploration in the future.

The 2% of random actions that we still perform are just not enough, because it requires dozens of coordinated steps to reach the top of the mountain (the best policy on MountainCar has a total reward of around –80). We can now continue our training for millions of steps, but the only data we will get from the environment will be episodes, which will take 200 steps with –200 total reward. This illustrates once more how important exploration is. Regardless of the training method we have, without proper exploration, we might just fail to train.
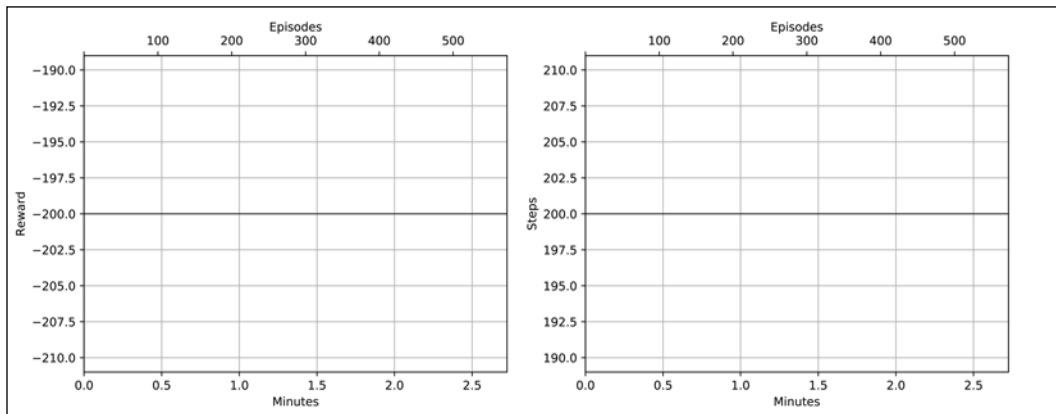
Figure 21.4: The reward and steps during the DQN training with the ε-greedy strategy
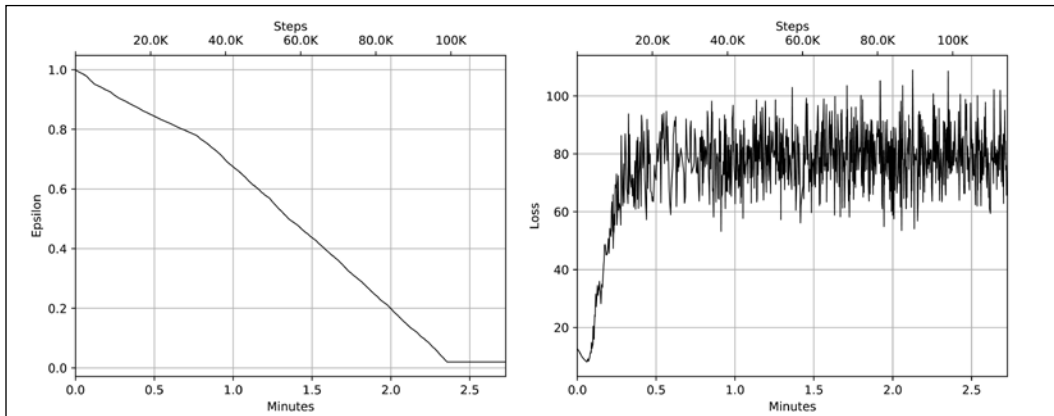
Figure 21.5: Epsilon and loss during the training

So, what should we do? If we want to stay with ε-greedy, the only option for us is to explore for longer. You can experiment with the hyperparameters of the `-p egreedy` mode, but I went to the extreme and implemented the `-p egreedy-long` hyperparameter set.

In this regime, we keep $\varepsilon = 1.0$ until we reach at least one episode with a total reward better than –200. Once this has happened, we start training the normal way, decreasing ε from 1.0 to 0.02 for subsequent 10^6 frames. As we don't do training, during the initial exploration phase, it normally runs five to 10 times faster. To start the training in this mode, we use the following command line: `$ ./mcar_dqn.py -n t1 -p egreedy-long`.

Unfortunately, even with this improvement of ε-greedy, it still failed to solve the environment. I left this version to run for five hours, but after 500k episodes, it still hadn't faced even a single example of the goal, so I gave up. Of course, you could try it for a longer period.

The DQN method with noisy networks

To apply the noisy networks approach to our MountainCar problem, we just need to replace one of two layers in our network with the `NoisyLinear` class, so our architecture will become as follows:

```
MountainCarNoisyNetDQN(
  (net): Sequential(
    (0): Linear(in_features=2, out_features=128, bias=True)
    (1): ReLU()
```

```
    (2): NoisyLinear(in_features=128, out_features=3, bias=True)
  )
)
```

The only difference between the `NoisyLinear` class and the version from *Chapter 8, DQN Extensions*, is that this version has an explicit method, `sample_noise()`, to update the noise tensors, so we need to call this method on every training iteration; otherwise, the noise will be constant during the training. This modification is needed for future experiments with policy-based methods, which require the noise to be constant during the relatively long period of trajectories. In any case, the modification is simple, and we just need to call this method from time to time. In the case of the DQN method, it is called on every training iteration. You can find the code of the `NoisyLinear` class in the module `Chapter21/lib/dqn_extra.py`.

The code is the same as before, so to activate the noisy networks, you need to run the training with the `-p noisynet` command line.

The following figures are the results that I got after almost three hours of training.
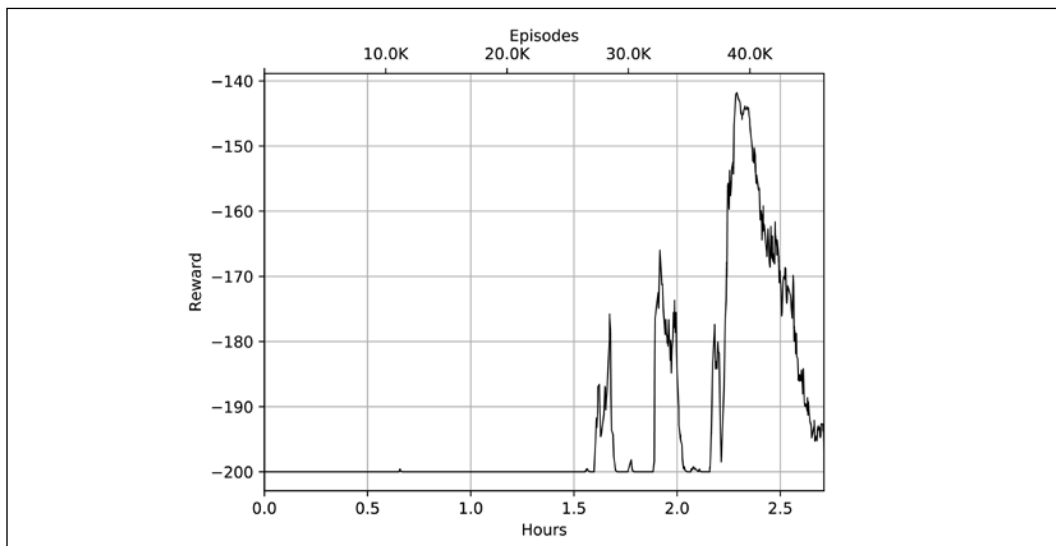
Figure 21.6: Rewards during the training of the DQN method + noisy networks
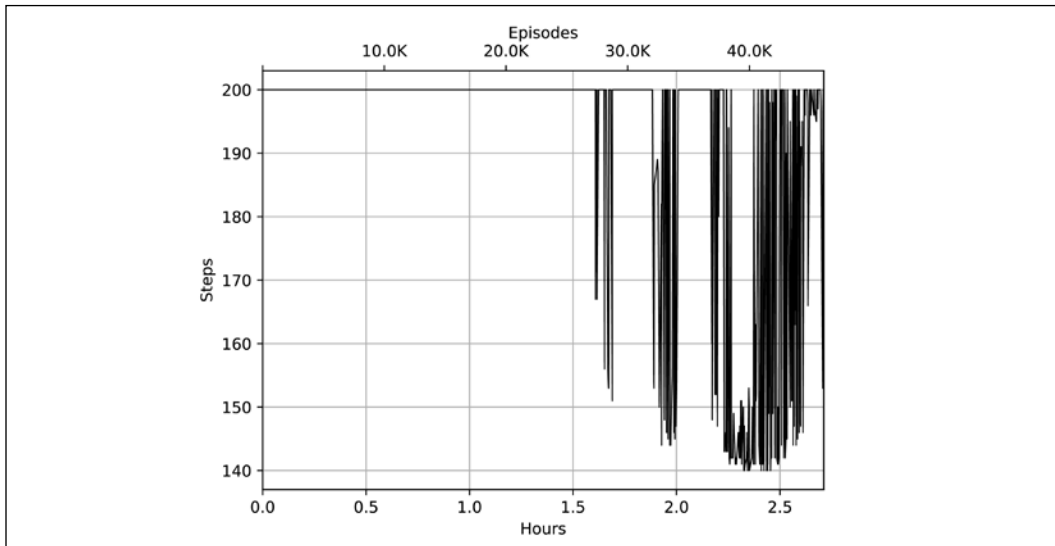
Figure 21.7: The length of training episodes
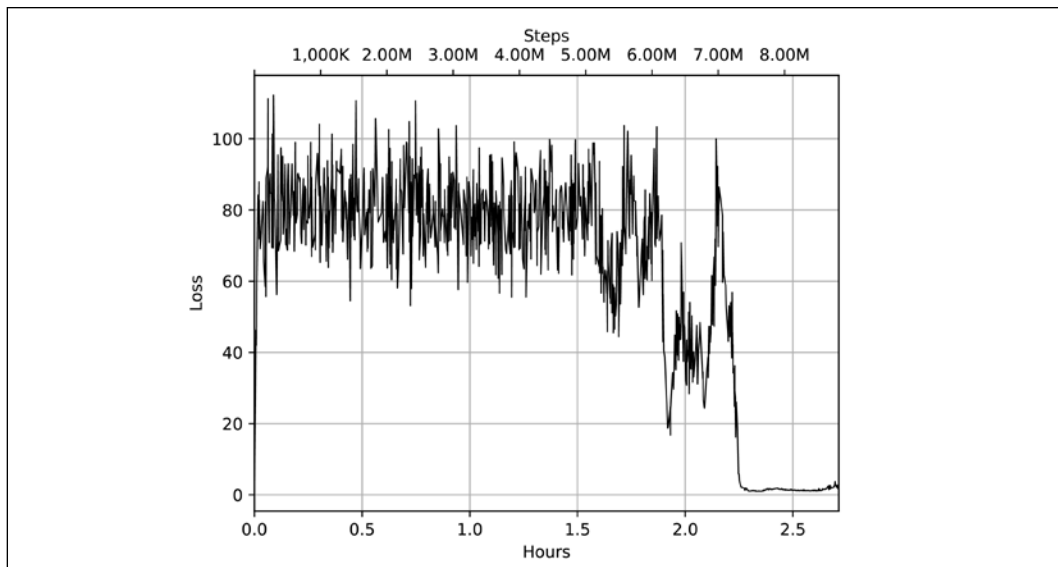
Figure 21.8: The loss during the training

As you can see, the training process wasn't able to reach the mean reward of –130 (as required in the code), but after just 20k episodes, we discovered the goal state, which is great progress in comparison to ε-greedy, which didn't find any single instance of the goal state after 500k episodes. In fact, the training process discovered the goal state after 10k episodes (the small bump on the line in *Figure 21.6*), but obviously, the method learned nothing from this sample.

In addition, it is obvious from the charts that the training process made three attempts to learn how to improve the policy but failed. Probably, this could be improved by hyperparameter tuning, for example, decreasing a learning rate and increasing the replay buffer size. Anyway, that's a nice improvement and shows the benefits of noisy networks over the traditional ε-greedy approach.

The DQN method with state counts

The last exploration technique that we will apply to the DQN method is count-based. As our state space is just two floating-point values, we will discretize the observation by rounding values to three digits after the decimal point, which should provide enough precision to distinguish different states from each other, but still group similar states together. For every individual state, we will keep the count of times we have seen this state before and use that to give an extra reward to the agent. For an off-policy method, it might not be the best idea to modify rewards during the training, but we will check the effect.

As before, I'm not going to provide the full source code; I will just emphasize the differences from the base version. First of all, we apply the wrapper to the environment to keep track of the counters and calculate the intrinsic reward value. The code of the wrapper is in the `Chapter21/lib/common.py` module and is shown here.

```
class PseudoCountRewardWrapper(gym.Wrapper):
    def __init__(self, env, hash_function = lambda o: o,
                 reward_scale: float = 1.0):
        super(PseudoCountRewardWrapper, self).__init__(env)
        self.hash_function = hash_function
        self.reward_scale = reward_scale
        self.counts = collections.Counter()
```

In the constructor, we take the environment we want to wrap, the optional hash function to be applied to the observations, and the scale of the intrinsic reward. We also create the container for our counters, which will map the hashed state into the count of times we have seen it.

```
def _count_observation(self, obs) -> float:
    h = self.hash_function(obs)
    self.counts[h] += 1
    return np.sqrt(1/self.counts[h])
```

Then, we define the helper function, which will calculate the intrinsic reward value of the state. It applies the hash to the observation, updates the counter, and calculates the reward using the formula we have already seen.

```
def step(self, action):
    obs, reward, done, info = self.env.step(action)
    extra_reward = self._count_observation(obs)
    return obs, reward + self.reward_scale * extra_reward, \
            done, info
```

The last method of the wrapper is responsible for the environment step, where we call the helper function to get the reward and return the sum of the extrinsic and intrinsic reward components.

To apply the wrapper, we need to pass to it the hashing function, which is very simple in our case.

```
def counts_hash(obs):
    r = obs.tolist()
    return tuple(map(lambda v: round(v, 3), r))
```

Probably, three digits are too much, so you can experiment with a different way of hashing states.

To start the training, pass `-p counts` to the training program. My result was a bit surprising: it took the method only 27 minutes and 9.3k episodes to solve the environment, which is quite impressive. The final piece of the output that I got looks like this:

```
Episode 9329: reward=-66.44, steps=93, speed=1098.5 f/s, elapsed=0:27:49
Episode 9330: reward=-72.30, steps=98, speed=1099.0 f/s, elapsed=0:27:49
Episode 9331: reward=-125.27, steps=158, speed=1099.5 f/s, elapsed=0:27:49
Episode 9332: reward=-125.61, steps=167, speed=1100.0 f/s, elapsed=0:27:49
Episode 9333: reward=-110.20, steps=132, speed=1100.5 f/s, elapsed=0:27:49
Episode 9334: reward=-105.95, steps=128, speed=1100.9 f/s, elapsed=0:27:49
```

```
Episode 9335: reward=-108.64, steps=129, speed=1101.3 f/s, elapsed=0:27:49
Episode 9336: reward=-64.87, steps=91, speed=1101.7 f/s, elapsed=0:27:49
Test done: got -89.000 reward after 89 steps, avg reward -131.206
Episode 9337: reward=-125.58, steps=158, speed=1101.0 f/s, elapsed=0:27:50
Episode 9338: reward=-154.56, steps=193, speed=1101.5 f/s, elapsed=0:27:50
Episode 9339: reward=-103.58, steps=126, speed=1101.9 f/s, elapsed=0:27:50
Episode 9340: reward=-129.93, steps=179, speed=1102.4 f/s, elapsed=0:27:50
Episode 9341: reward=-132.21, steps=170, speed=1097.8 f/s, elapsed=0:27:50
Episode 9342: reward=-125.93, steps=167, speed=1098.3 f/s, elapsed=0:27:50
Test done: got -88.000 reward after 88 steps, avg reward -129.046
Reward boundary has crossed, stopping training. Contgrats!
```

On the figures that follow, the dynamics of the training are shown.
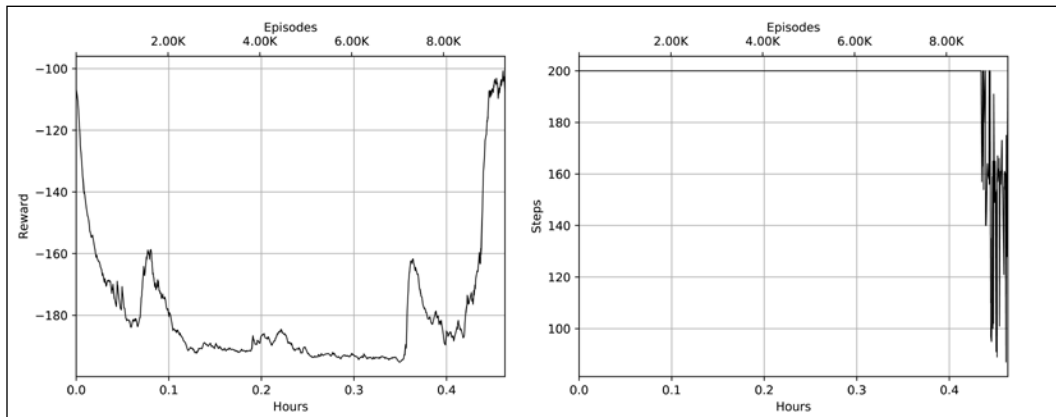
Figure 21.9: The reward and steps of the training episodes

As you can see, due to the intrinsic reward added, the training episode rewards were different from –200, but the amount of steps shows where the goal state was found.
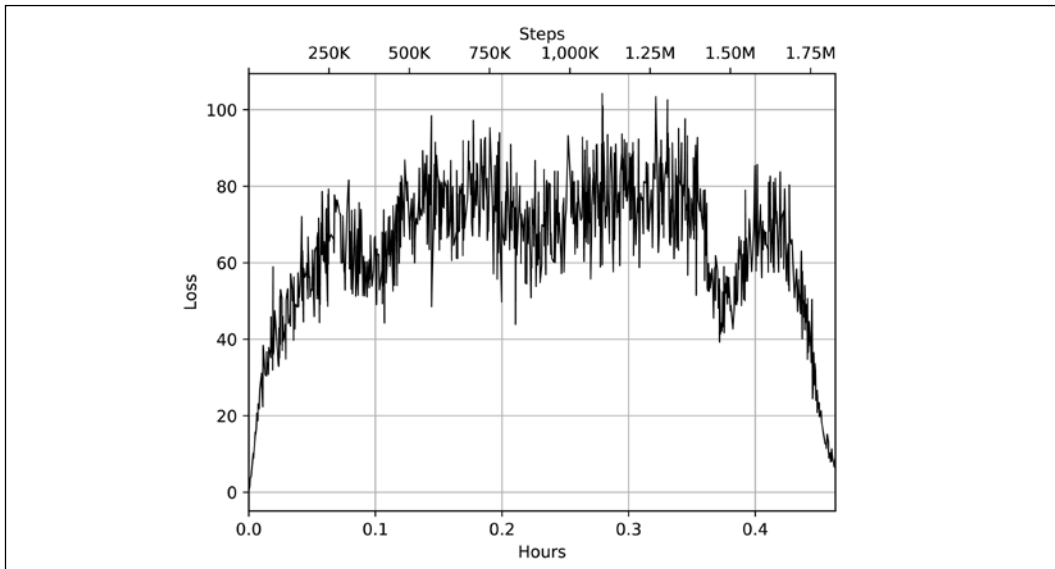
Figure 21.10: The loss during the training
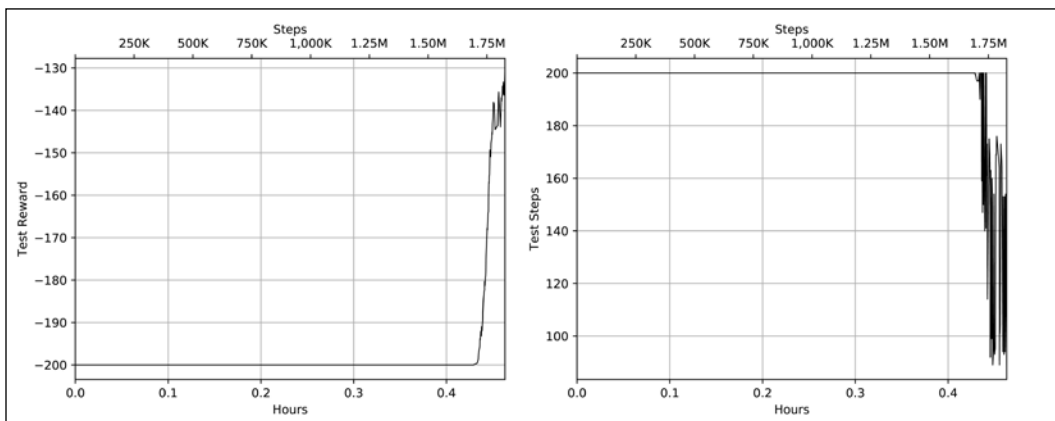
Figure 21.11: The reward and steps in the test episodes

On the preceding plots, the test episodes are shown. During the test, no intrinsic rewards were added.

The proximal policy optimization method

Another set of experiments that we will conduct with our MountainCar problem is related to the on-policy method proximal policy optimization (PPO), which we have covered before. There are several motivations for this choice. First of all, as you saw in the DQN method + noisy networks case, when good examples are rare, DQNs have trouble adopting them quickly. This might be solved by increasing the replay buffer size and switching to the prioritized buffer, or we could try on-policy methods, which adjust the policy immediately according to the obtained experience.

Another reason for choosing this method is the modification of the reward during the training. Count-based exploration and policy distillation introduce the intrinsic reward component, which might change over time. The value-based methods might be sensitive to the modification of the underlying reward as, basically, they will need to relearn values during the training. On-policy methods shouldn't have any problems with that, as an increase of the reward just puts more emphasis on a sample with higher reward in terms of the policy gradient.

Finally, it's just interesting to check our exploration strategies on both families of RL methods. To do this, in file `Chapter21/mcar_ppo.py` we have a PPO with various exploration strategies applied to MountainCar. The code is not very different from the PPO from *Chapter 19*, *Trust Regions – PPO, TRPO, ACKTR, and SAC*, so I'm not going to repeat it here.

To start the normal PPO without extra exploration tweaks, you should run the command `./mcar_ppo.py -n t1 -p ppo`.

As a reminder, PPO is in the policy gradient methods family, which limits Kullback-Leibler divergence between the old and new policy during the training, avoiding dramatic policy updates. Our network has two heads: the actor and the critic. The actor network returns the probability distribution over our actions (our policy) and the critic estimates the value of the state. The critic is trained using MSE loss, while the actor is driven by the PPO surrogate objective. In addition to those two losses, we regularize the policy by applying entropy loss scaled by the hyperparameter β. There is nothing new here so far. The following is the PPO network structure:

```
MountainCarBasePPO(
  (actor): Sequential(
    (0): Linear(in_features=2, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=3, bias=True)
  )
  (critic): Sequential(
```

```
    (0): Linear(in_features=2, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=1, bias=True)
  )
)
```

After almost six hours of training, the basic PPO was able to solve the environment. What follows are the final lines of the output:

```
Episode 185255: reward=-159, steps=159, speed=4290.6 f/s, elapsed=5:48:50
Episode 185256: reward=-154, steps=154, speed=4303.3 f/s, elapsed=5:48:50
Episode 185257: reward=-155, steps=155, speed=4315.9 f/s, elapsed=5:48:50
Episode 185258: reward=-95, steps=95, speed=4329.2 f/s, elapsed=5:48:50
Episode 185259: reward=-151, steps=151, speed=4342.6 f/s, elapsed=5:48:50
Episode 185260: reward=-148, steps=148, speed=4356.5 f/s, elapsed=5:48:50
Episode 185261: reward=-149, steps=149, speed=4369.7 f/s, elapsed=5:48:50
Test done: got -101.000 reward after 101 steps, avg reward -129.796
Reward boundary has crossed, stopping training. Contgrats!
```

As you can see, despite more than five hours of training, the PPO needed "just" 185k episodes to figure out the optimal behavior, which is worse than the DQN method with counter-based exploration, but still much better than the DQN method with ε-greedy exploration.
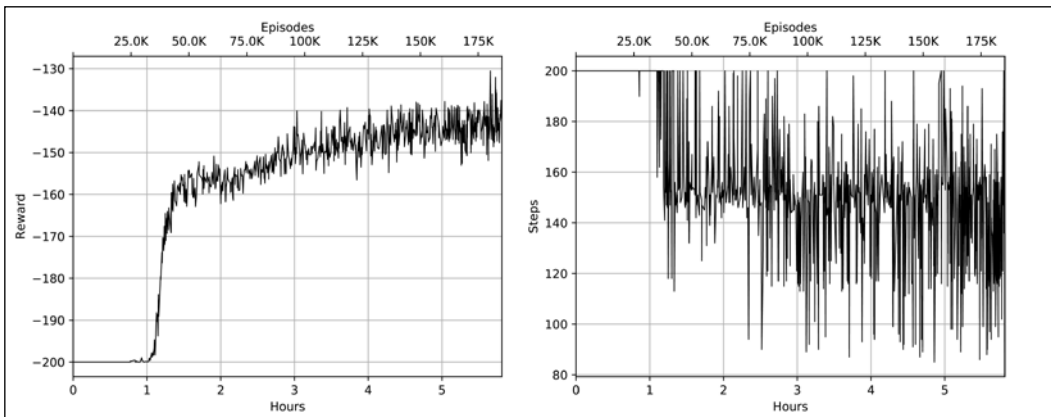
Figure 21.12: The reward and steps in the training episodes
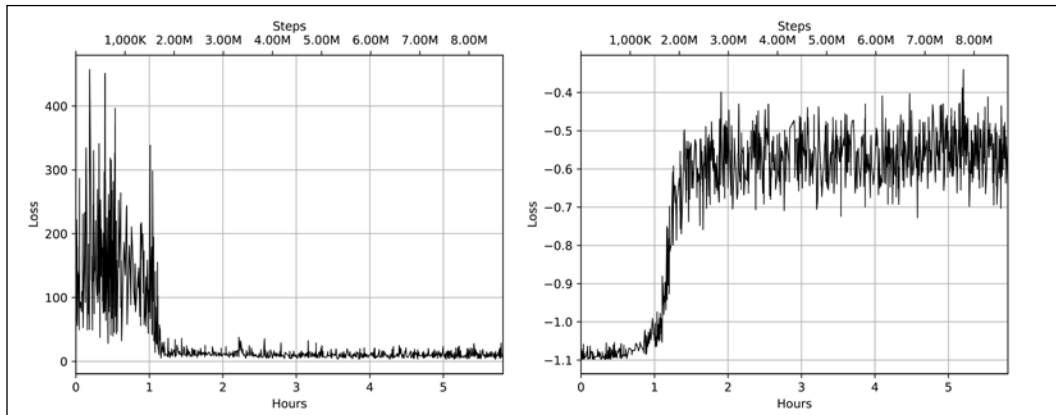
Figure 21.13: The total loss (left) and the entropy loss (right) during the training
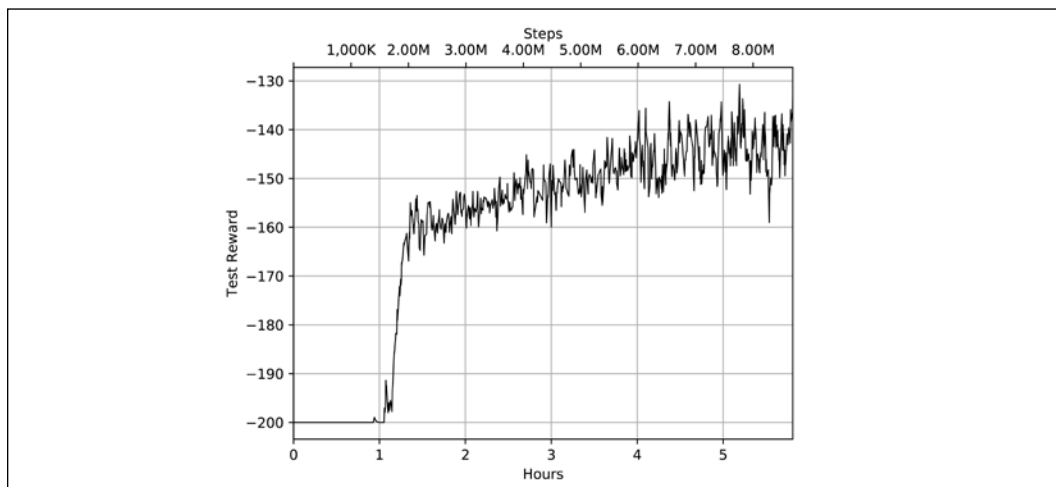
Figure 21.14: The dynamics of the test reward

From the preceding training dynamics charts, you can see that the PPO discovered the goal state quite soon, after just an hour of training and 25k episodes. During the rest of the training, it polished the policy, which might be an indication of too aggressive entropy regularization.

The PPO method with noisy networks

As with the DQN method, we can apply the noisy networks exploration approach to our PPO method. To do that, we need to replace the output layer of the actor with the `NoisyLinear` layer. Only the actor network needs to be affected, because we would like to inject the noisiness only into the policy and not in to the value estimation.

The new PPO network is shown here:

```
MountainCarNoisyNetsPPO(
  (actor): Sequential(
    (0): Linear(in_features=2, out_features=128, bias=True)
    (1): ReLU()
    (2): NoisyLinear(in_features=128, out_features=3, bias=True)
  )
  (critic): Sequential(
    (0): Linear(in_features=2, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=1, bias=True)
  )
)
```

There is one subtle nuance related to the application of noisy networks: where the random noise needs to be sampled. In *Chapter 8, DQN Extensions*, when you first met noisy networks, the noise was sampled on every `forward()` pass of the `NoisyLinear` layer. According to the original research paper, this is fine for off-policy methods, but for on-policy methods, it needs to be done differently. Indeed, when we train on-policy, we obtain the training samples produced by our current policy and calculate the policy gradient, which should push the policy toward improvement. The goal of noisy networks is to inject randomness, but as we have discussed, we prefer directed exploration over just a random change of policy after every step. With that in mind, our random component in the `NoisyLinear` layer needs to be updated not after every `forward()` pass, but much less frequently. In my code, I resampled the noise on every PPO batch, which was 2,048 transitions.

As before, to start the training, the same tool needs to be used and just parameter `-p noisynet` needs to be passed. In my experiment, it took slightly less than an hour to solve the environment.

```
Episode 29788: reward=-174, steps=174, speed=3817.1 f/s, elapsed=0:54:14
Episode 29789: reward=-166, steps=166, speed=3793.8 f/s, elapsed=0:54:14
Episode 29790: reward=-116, steps=116, speed=3771.2 f/s, elapsed=0:54:14
Episode 29791: reward=-154, steps=154, speed=3748.7 f/s, elapsed=0:54:14
Episode 29792: reward=-144, steps=144, speed=3727.0 f/s, elapsed=0:54:14
Test done: got -86.000 reward after 86 steps, avg reward -129.196
Reward boundary has crossed, stopping training. Contgrats!
```

The detailed dynamics of the training are shown on the charts that follow.
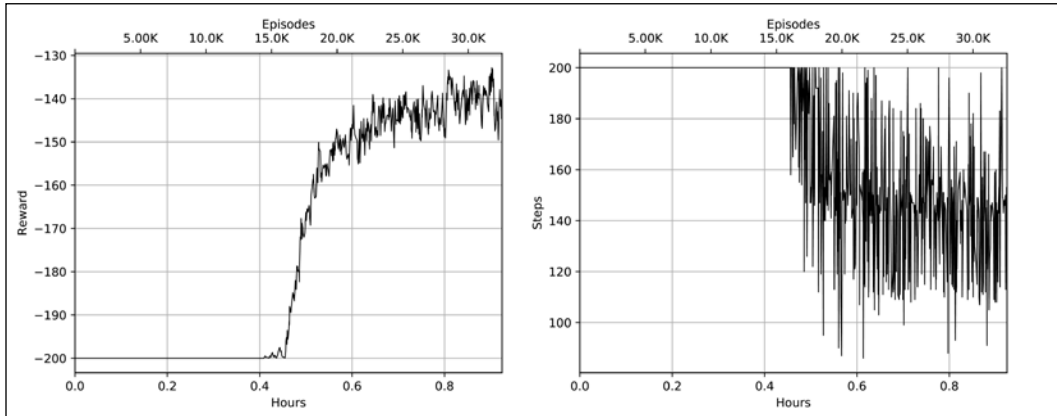
Figure 21.15: The reward and steps on the training episodes for the PPO method with noisy networks
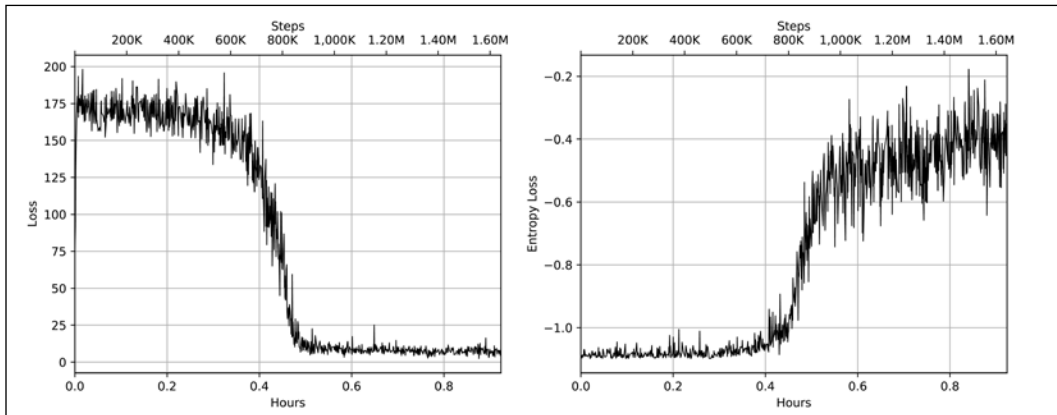
Figure 21.16: The total loss (left) and the entropy loss during the training (right)
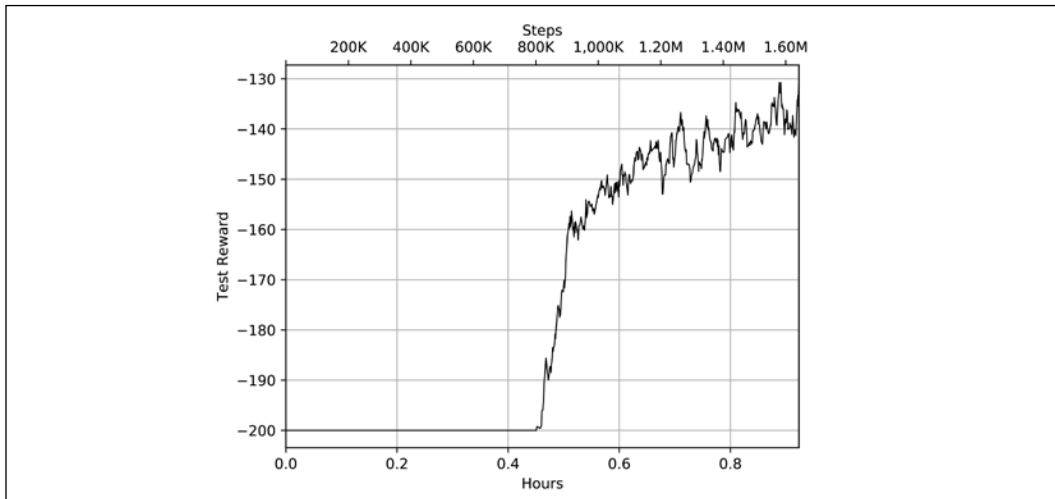
Figure 21.17: The reward obtained during the test

From those plots, it's obvious that noisy networks provided a major improvement to the training. This version discovered the goal state much sooner and the rest of the policy optimization was much more efficient.

The PPO method with count-based exploration

In this case, exactly the same count-based approach with three-digit hashing is implemented for the PPO method and can be triggered by passing `-p counts` to the training process.

In my experiments, the method was able to solve the environment in slightly more than an hour, and it required 50k episodes.

```
Episode 49473: reward=-134, steps=143, speed=4278.1 f/s, elapsed=1:17:43
Episode 49474: reward=-142, steps=152, speed=4274.6 f/s, elapsed=1:17:43
Episode 49475: reward=-130, steps=145, speed=4272.4 f/s, elapsed=1:17:43
Episode 49476: reward=-107, steps=117, speed=4270.2 f/s, elapsed=1:17:43
Episode 49477: reward=-109, steps=119, speed=4268.3 f/s, elapsed=1:17:43
Episode 49478: reward=-140, steps=148, speed=4266.3 f/s, elapsed=1:17:43
Episode 49479: reward=-142, steps=151, speed=4264.8 f/s, elapsed=1:17:43
Episode 49480: reward=-136, steps=146, speed=4259.0 f/s, elapsed=1:17:43
Test done: got -114.000 reward after 114 steps, avg reward -129.890
Reward boundary has crossed, stopping training. Contgrats!
```

The training plots follow. As before, due to the intrinsic reward component, the training reward values were higher than for the episodes during the testing.
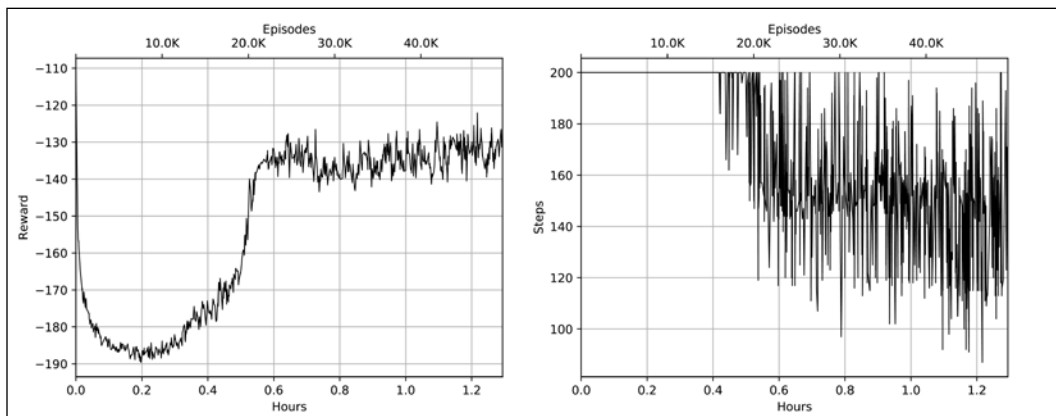
Figure 21.18: The reward and steps of the training episodes for the PPO method with count-based exploration
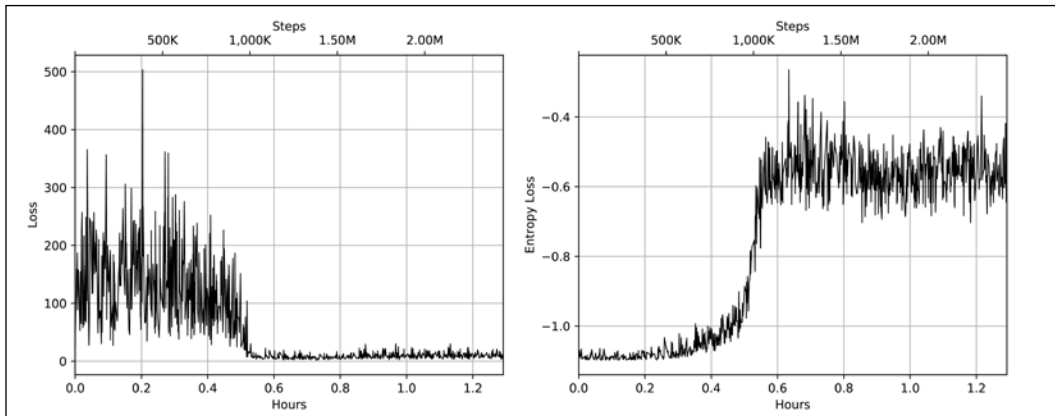
Figure 21.19: The total loss and entropy loss
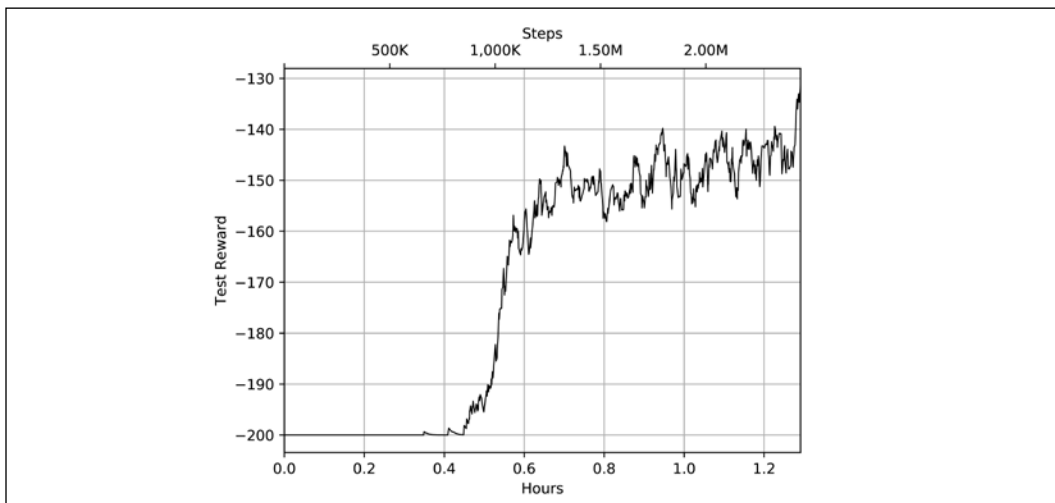
Figure 21.20: The reward for the test episodes

From these charts, you can see almost the same dynamics as with noisy networks.

The PPO method with network distillation

As the final exploration method in our MountainCar experiment, I implemented the network distillation method from the paper [4] cited earlier. In this method, two extra NNs are introduced. Both need to map the observation into one number, in the same way that our value head does. The difference is in the way they are used. The first NN is randomly initialized and kept untrained. This will be our *reference NN*. The second one is trained to minimize the MSE loss between the second and the first NN. In addition, the absolute difference between the outputs of NNs is used as the intrinsic reward component.

The idea behind this is simple. The better the agent has explored some state, the better our second NN will predict the output of the first one. This will lead to a smaller intrinsic reward being added to the total reward, which will decrease the policy gradient assigned to the sample.

In the paper, the authors suggested training separate value heads to predict separate intrinsic and extrinsic reward components, but for this example, I decided to keep it simple and just added both rewards in the wrapper, the same way that we did in the counter-based exploration method. This minimizes the amount of modifications in the code.

In terms of those extra NN architectures, I did a small experiment and tried several architectures for both NNs. The best results were obtained with the reference NN having three layers, but the trained NN having just one layer. This helps to prevent overfitting of the trained NN, as our observation space is not very large.

Both NNs are implemented in the class `MountainCarNetDistillery` in the module `Chapter21/lib/ppo.py`.

```python
class MountainCarNetDistillery(nn.Module):
    def __init__(self, obs_size: int, hid_size: int = 128):
        super(MountainCarNetDistillery, self).__init__()

        self.ref_net = nn.Sequential(
            nn.Linear(obs_size, hid_size),
            nn.ReLU(),
            nn.Linear(hid_size, hid_size),
            nn.ReLU(),
            nn.Linear(hid_size, 1),
        )
        self.ref_net.train(False)

        self.trn_net = nn.Sequential(
            nn.Linear(obs_size, 1),
```

```
        )

    def forward(self, x):
        return self.ref_net(x), self.trn_net(x)

    def extra_reward(self, obs):
        r1, r2 = self.forward(torch.FloatTensor([obs]))
        return (r1 - r2).abs().detach().numpy()[0][0]

    def loss(self, obs_t):
        r1_t, r2_t = self.forward(obs_t)
        return F.mse_loss(r2_t, r1_t).mean()
```

The code is simple: besides the `forward()` method, which returns the output from both NNs, the class includes two helper methods for intrinsic reward calculation and for getting the loss between two NNs.

To start the training, the argument `-p distill` needs to be passed to the `mcar_ppo.py` program. In my experiment, it took the code four hours to solve the environment, which is slower than other methods, but the slowdown is caused by the extra work we need to do. In terms of the amount of episodes, 63k were required, which is comparable to noisy networks and the count-based method. As usual, there might be some bugs and inefficiencies in my implementation, so you're welcome to improve it to make it faster and better.

```
Episode 62991: reward=-66, steps=122, speed=2468.0 f/s, elapsed=4:11:50
Episode 62992: reward=-42, steps=93, speed=2487.0 f/s, elapsed=4:11:50
Episode 62993: reward=-66, steps=116, speed=2505.1 f/s, elapsed=4:11:50
Episode 62994: reward=-70, steps=123, speed=2522.8 f/s, elapsed=4:11:50
Episode 62995: reward=-66, steps=118, speed=2540.2 f/s, elapsed=4:11:50
Episode 62996: reward=-70, steps=125, speed=2556.3 f/s, elapsed=4:11:50
Test done: got -118.000 reward after 118 steps, avg reward -129.809
Reward boundary has crossed, stopping training. Contgrats!
```

The convergence dynamics are shown on the charts that follow.
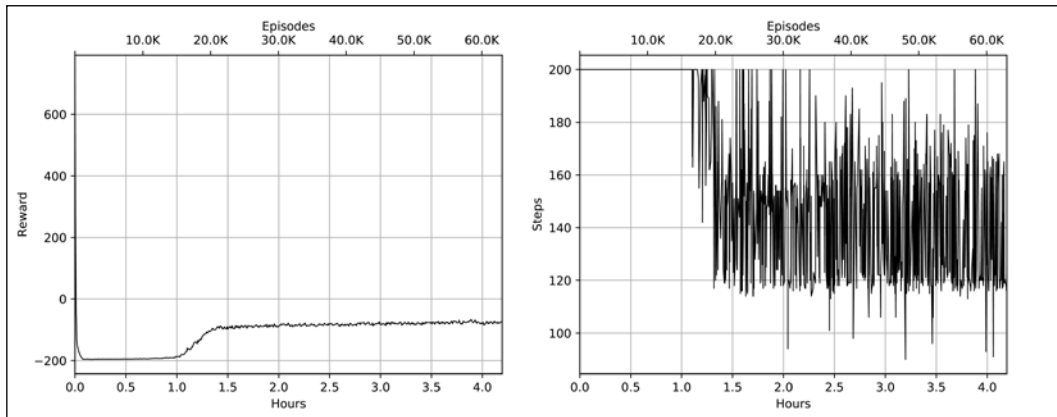
Figure 21.21: The reward and steps for the training episodes
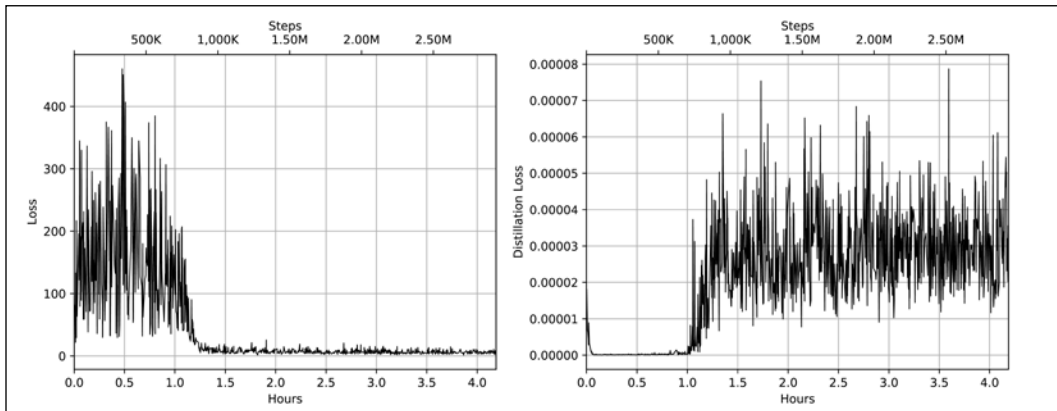
Figure 21.22: The total loss (left) and distillation loss (right) during the training
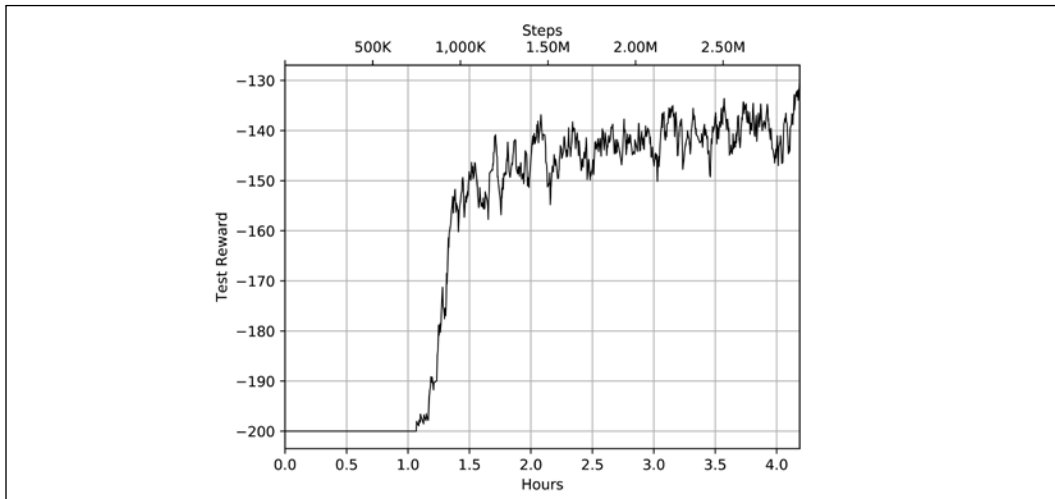
Figure 21.23: The reward for the test episodes

As before, due to the intrinsic reward component, the training episodes have a higher reward on the plots. From the distillation loss plot, it is clear that before the agent discovered the goal state, everything was boring and predictable, but once it had figured out how to end the episode earlier than 200 steps, the loss grew significantly.

Atari experiments

The MountainCar environment is a nice and fast way to experiment with exploration methods, but to conclude the chapter, I've included Atari versions of the DQN and PPO methods with the exploration tweaks we described. As the primary environment, I've used Seaquest, which is a game where the submarine needs to shoot fish and enemy submarines, and save aquanauts. This game is not as famous as Montezuma's Revenge, but it still might be considered as medium-hard exploration, because to continue the game, you need to control the level of oxygen. When it becomes low, the submarine needs to rise to the surface for some time. Without this, the episode will end after 560 steps and with a maximum reward of 20. But once the agent learns how to replenish the oxygen, the game might continue almost infinitely and bring to the agent a 10k-100k score. Surprisingly, traditional exploration methods struggle with discovering this; normally, training gets stuck at 560 steps, after which the oxygen runs out and the submarine dies.

The bad thing about Atari is that every experiment requires at least a day of training to check the effect, so my code and hyperparameters are very far from being the best, but they might be useful as a starting point for your own experiments. Of course, if you discover a way to improve the code, please share your findings on GitHub.

As before, there are two program files: `Chapter21/atari_dqn.py`, which implements the DQN method with ε-greedy and noisy networks exploration and `Chapter21/atari_ppo.py`, which is the PPO method with optional noisy networks and the network distillation method. To switch between hyperparameters, the command line `-p` needs to be used.

The following are the results that I got from a few runs of the code.

The DQN method with ε-greedy

Surprisingly, ε-greedy behaved quite well. With a 1M replay buffer and epsilon decayed from 1.0 to 0.02 for the first 1M steps, it was able to discover how to get the oxygen, and the training charts have almost no 560-steps boundary (which will be visible with other methods). After 13 hours of training, ε-greedy was able to reach the mean reward of 12-15, which is quite a good result. But the training was not very stable, which suggests that hyperparameter tuning might be needed to get better results.

In the DQN version, dueling, double DQN, and n-steps (with $n = 4$) were used.
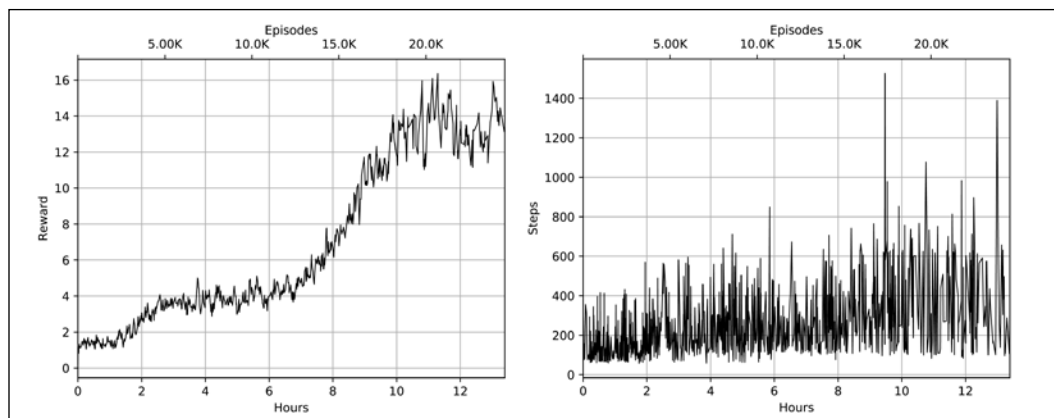
Figure 21.24: The reward and steps in the DQN method with ε-greedy on Seaquest

Unfortunately, I haven't done enough testing of the DQN method with noisy networks, which is implemented in the code and could be enabled with `-p noisynet`. It is possible that with enough tuning, it will behave much better.

Another direction for improvement might be the usage of the prioritized replay buffer, as some successful examples might be quite rare. Anyway, there are lots of directions to experiment with, which are left as exercises.

The classic PPO method

The classic PPO method with entropy regularization showed more stable training, and after 13 hours, it reached the mean reward of 18-19. At the same time, there was an obvious boundary in episodes' steps, which might become an issue with further policy improvements. It is likely that the entropy regularization coefficient, $\beta = 0.1$, is too high and should be lowered.
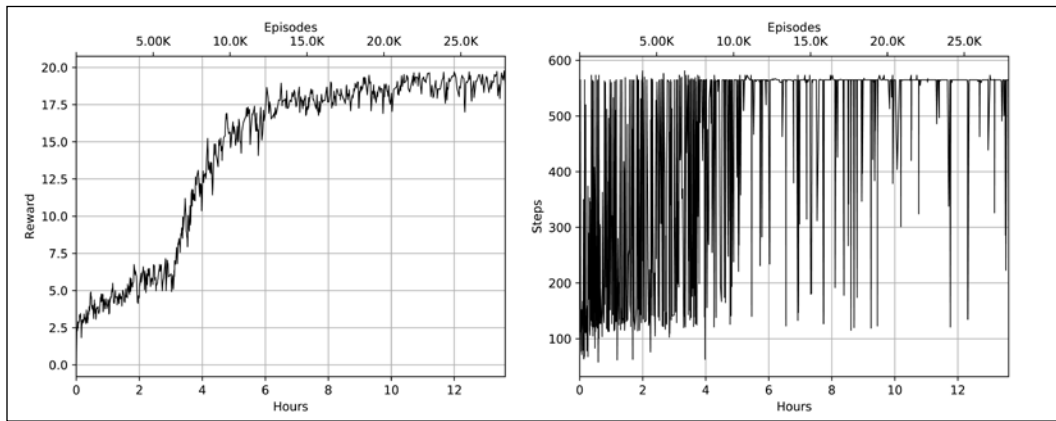
Figure 21.25: The training reward and steps for the PPO method
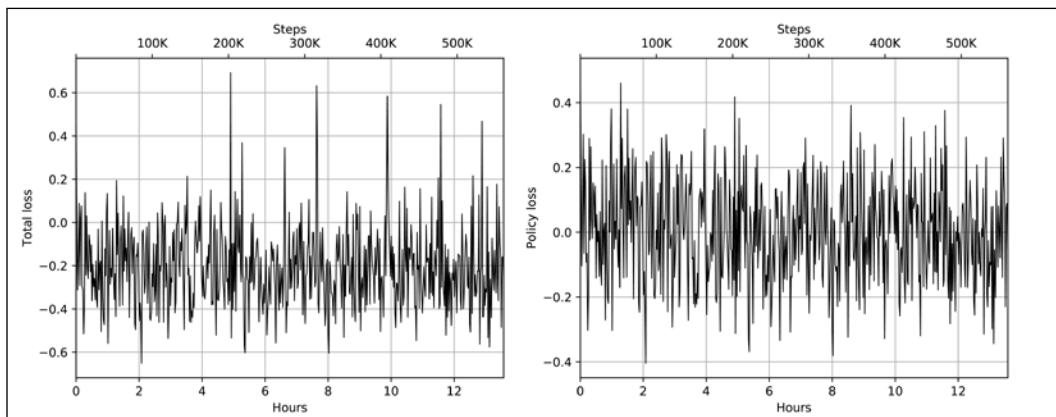
Figure 21.26: The total loss (left) and policy loss (right)
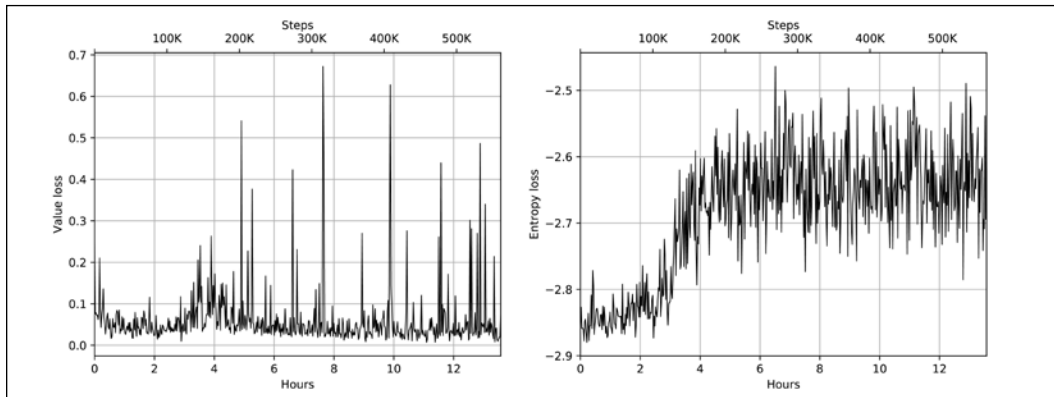
Figure 21.27: The value loss (left) and entropy loss (right)

High spikes in the value loss plot might suggest increasing the learning rate for the value head. At the moment, the common optimizer with the unified learning rate is used.

The PPO method with network distillation

The network distillation implementation follows the recommendations from the paper [4] and implements separate heads for predicting intrinsic and extrinsic reward components. The calculation for the reference values used to train those heads is the same as in the classic PPO method, it's just that different reward values are taken.

In terms of the results, they are quite similar to the classic PPO method; in 20 hours of training, the method was able to reach a test reward of 17-18, but the amount of steps in the episode was still strictly below 560.
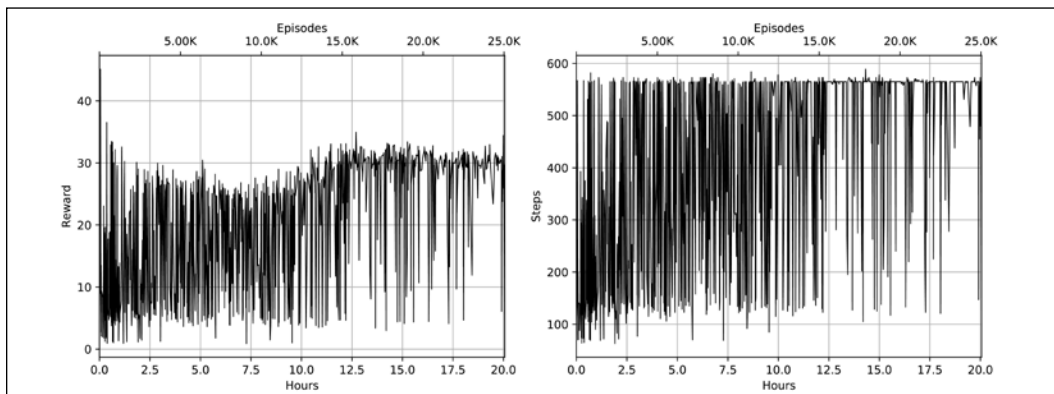
Figure 21.28: The reward and steps during the training
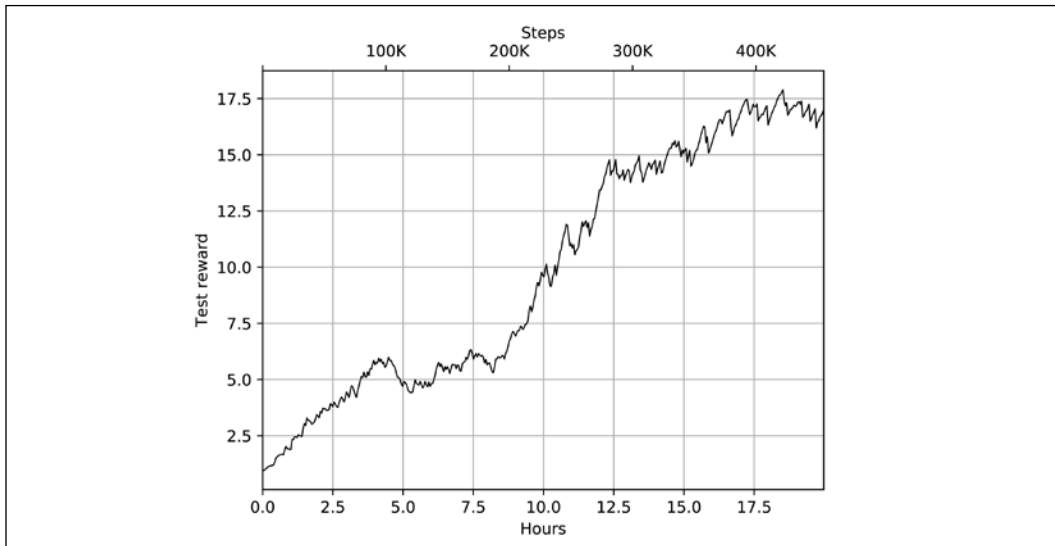
Figure 21.29: The test reward
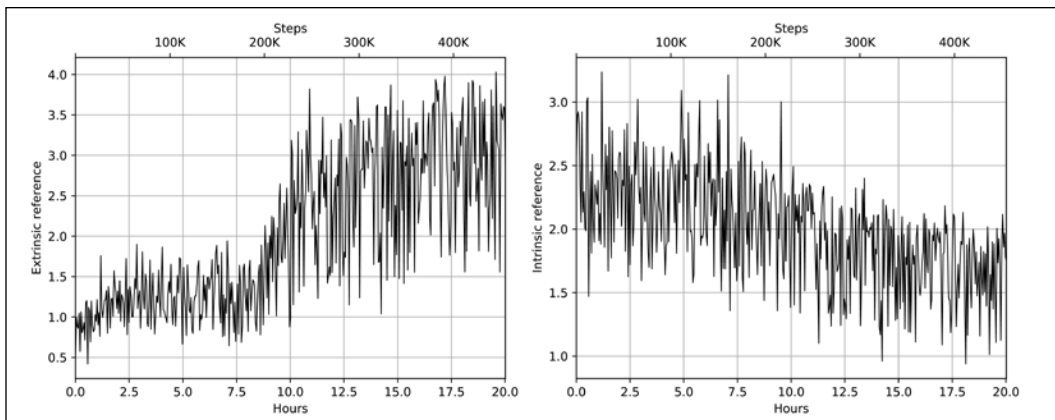
Figure 21.30: The extrinsic (left) and intrinsic (right) reference values during training

The PPO method with noisy networks

The code is very similar; two layers in the actor head are replaced with the
`NoisyLinear` class. The new noise values are sampled every new PPO batch.
During my experiments, one of the variants I ran with lower entropy regularization,
$\beta = 0.01$, was able to break the 560-steps barrier and play several episodes of length
1,000 with a reward of 21.

However, right after that discovery, the training process diverged. So, more experiments with hyperparameters are needed.
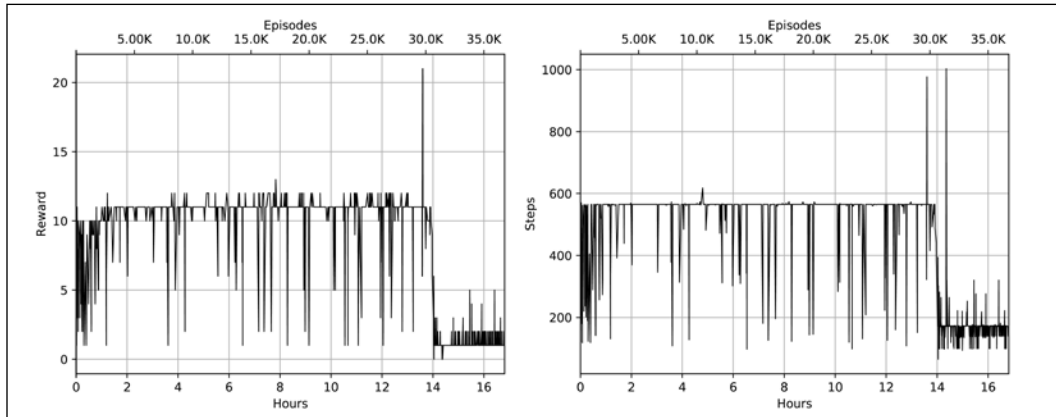
Figure 21.31: The reward and steps for training episodes
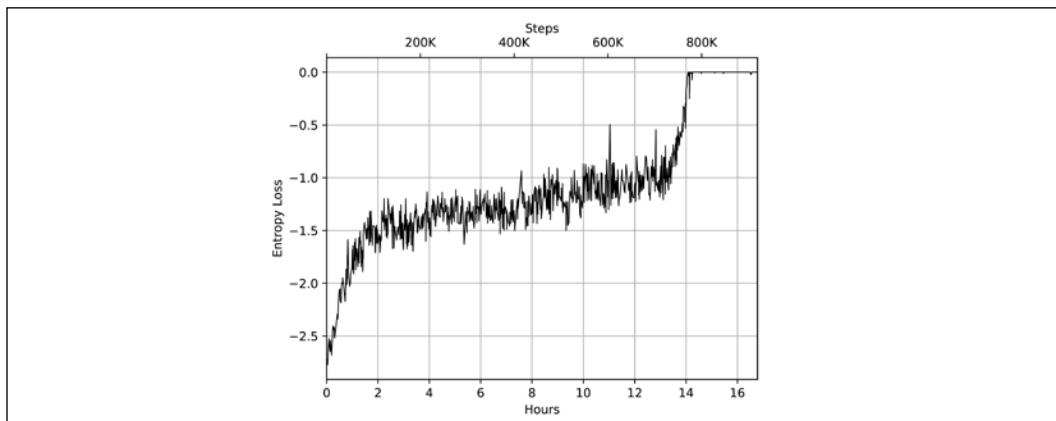
Figure 21.32: The entropy loss during the training

Summary

In this chapter, we discussed why ε-greedy exploration is not the best in some cases and checked alternative modern approaches for exploration. The topic of exploration is much wider and lots of interesting methods are left uncovered, but I hope you were able to get an overall impression of the new methods and the way they should be implemented.

In the next chapter, we will take a look at a different sphere of modern RL development: *model-based methods*.

References

1. *Strehl* and *Littman*, *An analysis of model-based Interval Estimation for Markov Decision Processes*, 2008: `https://www.sciencedirect.com/science/article/pii/S0022000008000767`

2. *Meire Fortunato*, et al, *Noisy Networks for Exploration* 2017, arxiv: 1706.10295

3. *Georg Ostrovski*, et al, *Count-Based Exploration with Neural Density Models*, 2017, arxiv:1703.01310v2

4. *Yuri Burda*, et al, *Exploration by random network distillation*, 2018, arxiv:1810.12894

22
Beyond Model-Free – Imagination

Model-based methods allow us to decrease the amount of communication with the environment by building a model of the environment and using it during training. In this chapter, we will:

- Take a brief look at the model-based methods in reinforcement learning (RL)
- Reimplement the model, described by DeepMind researchers in the paper *Imagination-Augmented Agents for Deep Reinforcement Learning* (`https://arxiv.org/abs/1707.06203`), that adds imagination to agents

Model-based methods

To begin, let's discuss the difference between the model-free approach that we have used in the book and model-based methods, including their strong and weak points and where they might be applicable.

Model-based versus model-free

In the *The taxonomy of RL methods* section in *Chapter 4*, *The Cross-Entropy Method*, we saw several different angles from which we can classify RL methods. We distinguished three main aspects:

- Value-based and policy-based
- On-policy and off-policy
- Model-free and model-based

There were enough examples of methods on both sides of the first and second categories, but all the methods that we have covered so far were 100% model-free. However, this doesn't mean that model-free methods are more important or better than their model-based counterparts. Historically, due to their **sample efficiency**, the model-based methods have been used in the robotics field and other industrial controls. This has also happened because of the cost of the hardware and the physical limitations of samples that can be obtained from a real robot. Robots with a large degree of freedom are not widely accessible, so RL researchers are more focused on computer games and other environments where samples are relatively cheap. However, the ideas from robotics are infiltrating RL, so, who knows, maybe the model-based methods will become more of a focus quite soon. For now, let's discuss where the differences between model-free and model-based lie.

In the names of both classes, "model" means the model of the environment, which could have various forms, for example, providing us with a new state and reward from the current state and action. All the methods covered so far put zero effort into predicting, understanding, or simulating the environment. What we were interested in was proper behavior (in terms of the final reward), specified directly (a policy) or indirectly (a value) given the observation. The source of the observations and reward was the environment itself, which in some cases could be very slow and inefficient.

In a model-based approach, we're trying to learn the model of the environment to reduce this "real environment" dependency. At the high level, the model is some kind of black box that approximates the real environment that we talked about in *Chapter 1*, *What Is Reinforcement Learning?*. If we have an accurate environment model, our agent can produce any number of trajectories that it needs, simply by using this model instead of executing the actions in the real world.

To some degree, the common playground of RL research is also just models of the real world; for example, MuJoCo or PyBullet are simulators of physics used to avoid building real robots with real actuators, sensors, and cameras to train our agents. The story is the same with Atari games or the TORCS car racing simulator: we use computer programs that model some processes, and these models can be executed quickly and cheaply. Even our CartPole example is an over-simplified approximation of a real cart with a stick attached. (By the way, in PyBullet and MuJoCo, there are more realistic CartPole versions with 3D actions and more accurate simulation.)

There are two motivations for using the model-based approach as opposed to model-free. The first and the most important one is sample efficiency caused by less dependency on the real environment. Ideally, by having an accurate model, we can avoid touching the real world and use only the trained model. In real applications, it is almost never possible to have the precise model of the environment, but even an imperfect model can significantly reduce the number of samples needed.

For example, in real life, you don't need an absolutely precise mental picture of some action (such as tying shoelaces or crossing the road), but this picture helps you in planning and predicting the outcome.

The second reason for a model-based approach is the **transferability** of the environment model across goals. If you have a good model for a robot manipulator, you can use it for a wide variety of goals without retraining everything from scratch.

There are a lot of details in this class of methods, but the aim of this chapter is to give you an overview and take a closer look at one particular research paper that has tried to combine both the model-free and model-based approaches in a sophisticated way.

Model imperfections

There is a serious issue with the model-based approach: when our model makes mistakes or is just inaccurate in some parts of the environment, the policy learned from this model may be totally wrong in real-life situations. To deal with this, we have several options. The most obvious option is to make the model better. Unfortunately, this can just mean that we need more observations from the environment, which is what we've tried to avoid. The more complicated and nonlinear the behavior of the environment, the worse the situation will be for modeling it properly.

Several ways have been discovered to tackle this issue. For example, there is the *local models* family of methods, when we replace one large environment model with a small regime-based set of models and train them using trust region tricks in the same way that trust region policy optimization (TRPO) does.

Another interesting way of looking at environment models is to augment model-free policy with model-based paths. In that case, we're not trying to build the best possible model of the environment, but just giving our agent extra information and letting it decide by itself whether the information will be useful during the training or not.

One of the first steps in that direction was carried out by DeepMind in their system UNREAL, which was described in the paper by *Max Jaderberg*, *Volodymyr Mnih*, and others called *Reinforcement Learning with Unsupervised Auxiliary Tasks*, which was published in 2016 (arXiv:1611.05397) [1]. In this paper, the authors augmented the **asynchronous advantage actor-critic** (**A3C**) agent with extra tasks learned in an unsupervised way during the normal training. The main tests of the agent were performed in a partially observable first-person view maze navigation problem, which is where the agent needs to navigate a Doom-like maze and obtain a reward for gathering things or executing other actions.

The novel approach of the paper was in artificially injecting extra auxiliary tasks not related to the usual RL methods' objectives of value or discounted reward. Those tasks were trained in an unsupervised way from observations and include the following:

- **An immediate reward prediction**: From the history of observations, the agent was asked to predict the immediate reward of the current step

- **Pixel control**: The agent was asked to communicate with the environment to maximize the change in its view

- **Feature control**: The agent was learning how to change specific features in its internal representation

Those tasks were not directly related to the agent's main objective of maximizing the total reward, but they allowed the agent to get a better representation of low-level features and allowed UNREAL to get better results. The first task of immediate reward prediction could be seen as a tiny environment model aiming to predict the reward. I'm not going to cover the UNREAL architecture in detail, but I recommend that you read the original paper.

The paper that we will cover in detail in this chapter was also published by DeepMind researchers [2]. In the paper, the authors augmented the model-free path of the standard A3C agent with the so-called "imagination module," which provides extra help for the agent to make decisions about actions.

The imagination-augmented agent

The overall idea of the new architecture, called **imagination-augmented agent** (**I2A**), is to allow the agent to imagine future trajectories using the current observations and incorporate these imagined paths into its decision process. The high-level architecture is shown in the following diagram:
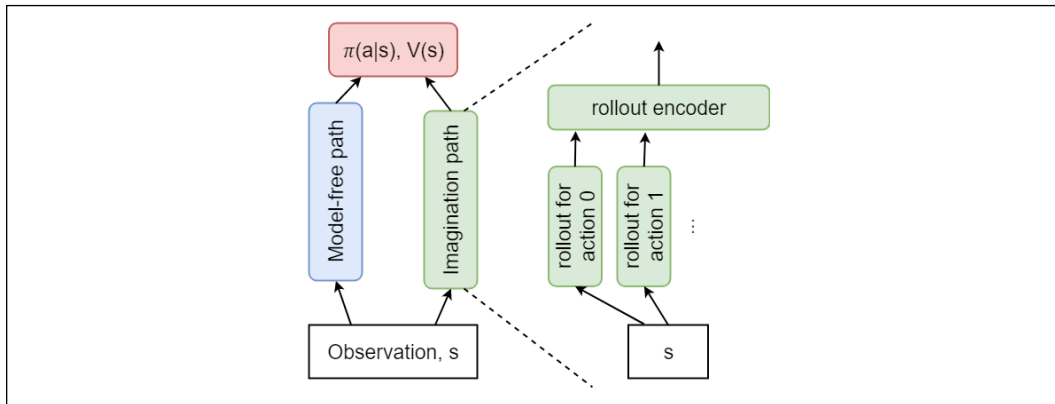
Figure 22.1: The I2A architecture

The agent consists of two different paths used to transform the input observation: model-free and imagination. Model-free is a standard set of convolution layers that transforms the input image in high-level features. The other path, imagination, consists of a set of trajectories imagined from the current observation. The trajectories are called *rollouts* and they are produced for every available action in the environment. Every rollout consists of a fixed number of steps into the future, and on every step, a special model, called the **environment model** (**EM**) (but not to be confused with the expectation maximization method), produces the next observation and predicted immediate reward from the current observation and the action to be taken.

Every rollout for every action is produced by taking the current observation into the EM and then feeding the predicted observation to the EM again N times. On the first step of the rollout, we know the action (as this is the action for which the rollout is generated), but on the subsequent steps, the action is chosen using the small *rollout policy network*, which is trained in conjunction with the main agent. The output from the rollout is N steps of imagined trajectory starting from the given action and continuing into the future according to the learned rollout policy. Every step of the rollout is imagined observation and predicted immediate reward.

All the steps from the single rollout are passed to another network, called the *rollout encoder*, which encodes them into a fixed-size vector. For every rollout, we get those vectors, concatenate them together, and feed them to the head of the agent, which produces the usual policy and value estimations for the A3C algorithm. As you can see, there are some moving parts here, so I've tried to visualize all of them in the following diagram for the case of two rollout steps and two actions in the environment. In the upcoming subsections, I will describe every network and the steps performed by the method in detail.
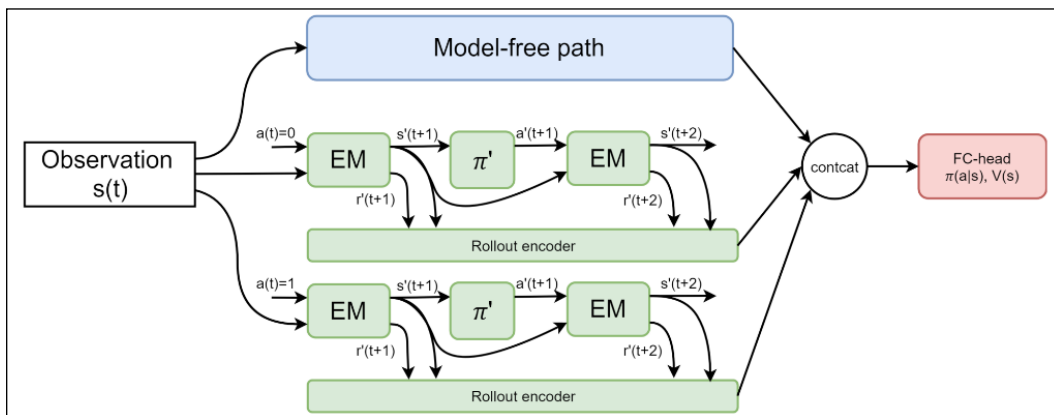
Figure 22.2: The imagination path architecture

The EM

The goal of the EM is to convert the current observation and the action into the next observation and the immediate reward. In the I2A paper [2], the authors tested the I2A model on two environments: the Sokoban puzzle and MiniPacman arcade game. In both cases, the observations were pixels, so the EM also returned the pixels, plus the float value for the reward. To incorporate the action into the convolution layers, the action was one-hot encoded and broadcast to match the observation pixels, one color plane per action. This transformation is illustrated in the following diagram:
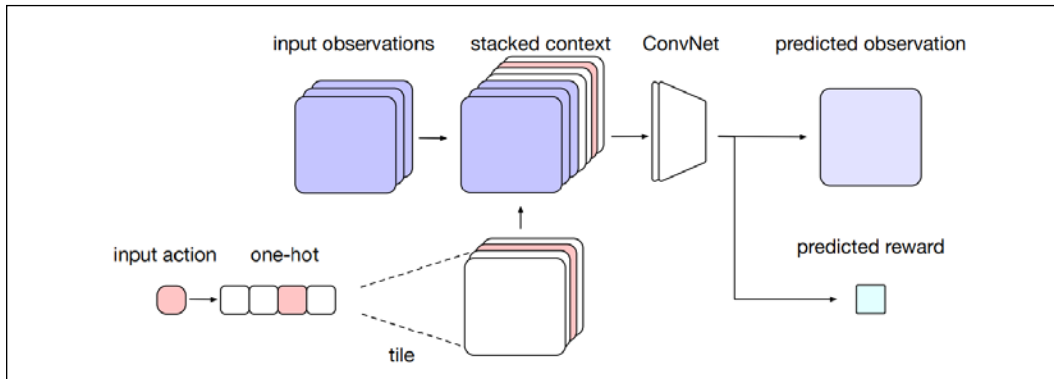
Figure 22.3: The EM structure

There are several possible ways that this EM could be trained. The paper's authors discovered that the fastest convergence is obtained by pretraining the EM using another, partially trained baseline agent as a source of environment samples.

The rollout policy

During the rollout steps, we need to make decisions about the action to be taken during our imagined trajectory. As mentioned, the action for the first step is set explicitly, as we produce an individual rollout trajectory for every action we have, but the subsequent steps require somebody to make this decision. Ideally, we would like those actions to be similar to our agent's policy, but we can't just ask our agent to produce the probabilities, as it will require rollouts to be created in the imagination path. To break this tie, a separate rollout policy network is trained to produce similar output to our main agent's policy. The rollout policy is a small network, with a similar architecture to A3C, that is trained in parallel to the main I2A network using a cross-entropy loss between the rollout policy network output and the output of the main network. In the paper, this training process is called "policy distillation."

The rollout encoder

The final component of the I2A model is the rollout encoder, which takes rollout steps (observation and reward pairs) as input and produces the fixed-sized vector, which embeds the information about the rollout. In this network, every rollout step was preprocessed with a small convolution network to extract the features from the observation, and these features were converted into a vector of fixed size by the long short-term memory (LSTM) network. The outputs from every rollout were concatenated together with features from the model-free path and used to produce the policy and the value estimation in the same way as in the A3C method.

The paper's results

As already mentioned, to check the effect of imagination in RL problems, the authors used two environments that require planning and making decisions about the future: randomly generated Sokoban puzzles and the MiniPacman game. In both environments, the imagination architecture showed better results than the baseline A3C agent.

In the rest of this chapter, we will apply the model to the Atari Breakout game and check the effect ourselves.

I2A on Atari Breakout

The training path of I2A is a bit complicated and includes a lot of code and several steps. To understand it better, let's start with a brief overview. In this example, we will implement the I2A architecture described in the paper [2], adopted to the Atari environments, and test it on the Breakout game. The overall goal is to check the training dynamics and the effect of imagination augmentation on the final policy.

Our example consists of three parts, which correspond to different steps in the training:

1. The baseline **advantage actor-critic** (**A2C**) agent in `Chapter22/01_a2c.py`. The resulting policy is used for obtaining observations of the EM.

2. The EM training in `Chapter22/02_imag.py`. It uses the model obtained on the previous step to train the EM in an unsupervised way. The result is the EM weights.

3. The final I2A agent training in `Chapter22/03_i2a.py`. In this step, we use the EM from step 2 to train a full I2A agent, which combines the model-free and rollouts paths.

Due to the size of the code, I'm not going to describe all of it here, but will focus instead on the important parts.

The baseline A2C agent

The first step in the training has two goals: to establish the baseline, which will be used to evaluate the I2A agent, and obtain the policy for the EM step. The EM is trained in an unsupervised way from the tuples (*s*, *a*, *s'*, *r*) obtained from the environment. So, the final quality of the EM is heavily dependent on the data it has been trained on. The closer the observations are to the data experienced by the agent during the real action, the better the final result will be.

The code is in `Chapter22/01_a2c.py` and `Chapter22/lib/common.py` and is the standard A2C algorithm that we have covered several times. To make the training data generation process reusable in I2A agent training, I haven't used PTAN library classes and have reimplemented data generation from scratch, which is in the `common.iterate_batches()` function and responsible for gathering observations from environments and calculating discounted rewards for the experienced trajectories.

This agent also has all hyperparameters set very close to the OpenAI Baseline A2C implementation, which I've used during debugging and the implementation of the agent. The only difference is the initialization of the initial weights (I rely on the standard PyTorch initialization of weights) and that the learning rate decreased from 7e-4 to 1e-4 to improve the stability of the training process.

For every 1,000 batches of training, a test of the current policy was performed, which consisted of three full episodes and five lives, each to be played by the agent. The mean reward and the number of steps were recorded, and the model was saved every time that a new best training reward had been achieved.

The configuration of the testing environment is different in two ways from environments used during the training. First of all, the test environment plays full episodes instead of per-life episodes, so the final reward on the test episodes is higher than the reward during the training. The second difference is that the test environment uses unclipped reward to make the test numbers interpretable. Clipping is a standard way to improve the stability of Atari training, as in some games, the raw score could have a large magnitude that negatively influences the estimated advantage variance.

Another difference from the classical Atari A2C agent is the number of frames given as an observation. Usually, four consequent frames are given, but from my experiments, I found out that the Breakout game has a very similar convergence on just two frames. Working with two frames is faster, so everywhere in this example the observation tensor has the dimensionality of `(2, 84, 84)`.

To make training repeatable, the fixed random seed is used in the baseline agent. This is done by the `common.set_seed` function, which sets the random seed for NumPy, Torch (both central processing unit (CPU) and CUDA), and every environment in the pool.

EM training

The EM is trained on data produced by the baseline agent, and you can specify any weight file saved on the previous step. It does not necessarily have to be the best model, as it just needs to be "good enough" to produce relevant observations.

The EM definition is in `Chapter22/lib/i2a.py`, the `EnvironmentModel` class, and has an architecture that mostly follows the model present in the I2A paper [2] for the Sokoban environment. The input to the model is an observation tensor that accompanies the action to be taken and is passed as an integer value. The action is one-hot encoded and broadcast to the observation tensor dimensionality. Then, both the broadcasted action and observation are concatenated along the "channels" dimension, giving the input tensor of `(6, 84, 84)`, as Breakout has four actions.

This tensor is processed with two convolution layers of 4×4 and 3×3, then the residual layer is used when the output is processed by a 3×3 convolution, the result of which is added to the input. The resulting tensor has been fed into two paths: one is deconvolution, producing the output observation, and another is a reward predicting path, consisting of two convolution layers and two fully connected layers.

The EM has two outputs: the immediate reward value, which is a single float, and the next observation. To reduce the dimensionality of the observation, the difference from the last observation is predicted. So, the output is a tensor `(1, 84, 84)`. Besides the decrease in the amount of values we need to predict, the difference has the benefit of being zero-centered and zero-valued for frames when nothing has changed, which will dominate during the Breakout gameplay, when only a few pixels usually change from frame to frame (the ball, the paddle, and the brick being hit). The architecture and code for the EM is shown in the following figure:
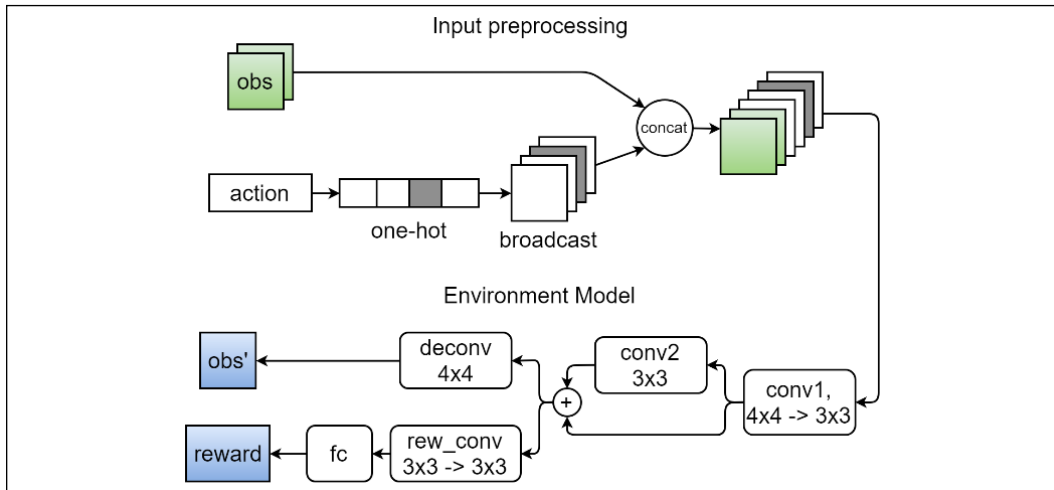
Figure 22.4: The EM architecture and its input preprocessing

The following is the model implementation from the `Chapter22/lib/i2a.py` module:

```
EM_OUT_SHAPE = (1, ) + common.IMG_SHAPE[1:]


class EnvironmentModel(nn.Module):
    def __init__(self, input_shape, n_actions):
        super(EnvironmentModel, self).__init__()

        self.input_shape = input_shape
        self.n_actions = n_actions

        n_planes = input_shape[0] + n_actions
        self.conv1 = nn.Sequential(
            nn.Conv2d(n_planes, 64, kernel_size=4,
                      stride=4, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU()
        )
```

The convolution part takes the input, which has color planes from the observation plus extra planes from the one-hot encoded action. As the model uses residual connection between both convolution layers, we will keep them as separate fields.

```python
        self.deconv = nn.ConvTranspose2d(
            64, 1, kernel_size=4, stride=4, padding=0)

        self.reward_conv = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3),
            nn.MaxPool2d(2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3),
            nn.MaxPool2d(2),
            nn.ReLU()
        )

        rw_conv_out = self._get_reward_conv_out(
            (n_planes, ) + input_shape[1:])
        self.reward_fc = nn.Sequential(
            nn.Linear(rw_conv_out, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )
```

The part after the convolution layers splits into two paths: the first one is deconvolution transformation, which generates the predicted observations of the next state, and the second generates the predicted reward of the state.

```python
    def _get_reward_conv_out(self, shape):
        o = self.conv1(torch.zeros(1, *shape))
        o = self.reward_conv(o)
        return int(np.prod(o.size()))

    def forward(self, imgs, actions):
        batch_size = actions.size()[0]
        act_planes_v = torch.FloatTensor(
            batch_size, self.n_actions, *self.input_shape[1:])
        act_planes_v.zero_()
        act_planes_v = act_planes_v.to(actions.device)
        act_planes_v[range(batch_size), actions] = 1.0
        comb_input_v = torch.cat((imgs, act_planes_v), dim=1)
        c1_out = self.conv1(comb_input_v)
        c2_out = self.conv2(c1_out)
        c2_out += c1_out
        img_out = self.deconv(c2_out)
```

```
        rew_conv = self.reward_conv(c2_out).view(batch_size, -1)
        rew_out = self.reward_fc(rew_conv)
        return img_out, rew_out
```

In the `forward()` method of our model, we combine all the parts together into the transformation shown in *Figure 22.4*.

The training process of the EM is simple and straightforward. The pool of 16 parallel environments is used to populate the batch of 64 samples. Every entry in a batch consists of the current observation, the next immediate observation, the action taken, and the immediate reward obtained. The final loss that is optimized is a sum of the observation loss and the reward loss. The observation loss is the **mean squared error** (**MSE**) loss between the predicted delta for the next observation and the real delta between the current and the next observation. The reward loss is again the MSE between rewards. To emphasize the importance of observation, the observation loss has a scale factor of 10.

The imagination agent

The final step in the training process is the I2A agent, which combines the model-free path with rollouts produced by the EM trained in the previous step.

The I2A model

The agent is implemented in the `I2A` class in the `Chapter22/lib/i2a.py` module.

```
class I2A(nn.Module):
    def __init__(self, input_shape, n_actions,
                 net_em, net_policy, rollout_steps):
        super(I2A, self).__init__()
```

The arguments of the constructor provide the shape of observations, the number of actions in the environment, and the two networks used during rollout: the EM and rollout policy and, finally, the count of steps to perform during rollouts. Both the EM and rollout policy networks are stored in a special way to prevent their weights from being included in the I2A network parameters.

```
        self.n_actions = n_actions
        self.rollout_steps = rollout_steps

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32,
                      kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
```

```
        nn.ReLU(),
        nn.Conv2d(64, 64, kernel_size=3, stride=1),
        nn.ReLU(),
    )
```

The preceding code specifies the model-free path, which produces the features from the observation. The architecture is a familiar Atari convolution.

```
conv_out_size = self._get_conv_out(input_shape)
fc_input = conv_out_size + ROLLOUT_HIDDEN * n_actions

self.fc = nn.Sequential(
    nn.Linear(fc_input, 512),
    nn.ReLU()
)
self.policy = nn.Linear(512, n_actions)
self.value = nn.Linear(512, 1)
```

The input of the layers that will produce the policy and the value of the agent is combined from the features obtained from the model-free path and encoded rollouts. Every rollout is represented with the ROLLOUT_HIDDEN constant (which equals 256), which is a dimensionality of the LSTM layer inside the RolloutEncoder class.

```
self.encoder = RolloutEncoder(EM_OUT_SHAPE)
self.action_selector = \
    ptan.actions.ProbabilityActionSelector()
object.__setattr__(self, "net_em", net_em)
object.__setattr__(self, "net_policy", net_policy)
```

The rest of the constructor creates the RolloutEncoder class (which will be described later in this section) and stores the EM and rollout policy networks. Neither of these networks is supposed to be trained together with the I2A agent, as the EM is not trained at all (it is pretrained on the previous step and remains fixed) and the rollout policy is trained with a separate policy distillation process. However, PyTorch 's Module class automatically registers and joins all fields assigned to the class. To prevent the EM and rollout policy networks from being merged into the I2A agent, we save their references via the __setattr__() call, which is a bit hacky, but does exactly what we need.

```
def _get_conv_out(self, shape):
    o = self.conv(torch.zeros(1, *shape))
    return int(np.prod(o.size()))

def forward(self, x):
    fx = x.float() / 255
    enc_rollouts = self.rollouts_batch(fx)
```

```
conv_out = self.conv(fx).view(fx.size()[0], -1)
fc_in = torch.cat((conv_out, enc_rollouts), dim=1)
fc_out = self.fc(fc_in)
return self.policy(fc_out), self.value(fc_out)
```

The `forward()` function looks simple, as most of the work here is inside the `rollouts_batch()` method. The next and last method of the I2A class is a bit more complicated. Originally, it was written to perform all rollouts sequentially, but that version was painfully slow. The new version of the code performs all rollouts at once, step by step, which increases the speed almost five times, but makes the code slightly more complicated.

```
def rollouts_batch(self, batch):
    batch_size = batch.size()[0]
    batch_rest = batch.size()[1:]
    if batch_size == 1:
        obs_batch_v = batch.expand(
            batch_size * self.n_actions, *batch_rest)
    else:
        obs_batch_v = batch.unsqueeze(1)
        obs_batch_v = obs_batch_v.expand(
            batch_size, self.n_actions, *batch_rest)
        obs_batch_v = obs_batch_v.contiguous()
        obs_batch_v = obs_batch_v.view(-1, *batch_rest)
```

At the beginning of the function, we take the batch of observations and we want to perform `n_actions` rollouts for every observation of the batch. So, we need to expand the batch of observations, repeating every observation `n_actions` times. The most efficient way of doing this is to use the PyTorch `expand()` method, which can repeat any tensor with 1 dimensionality and repeat it along this dimension any number of times. In case our batch consists of one single example, we just use this batch dimension; otherwise, we need to inject the extra unit dimension right after the batch dimension and then expand along it. Regardless of this, the resulting dimensionality of the `obs_batch_v` tensor is `(batch_size * n_actions, 2, 84, 84)`.

```
actions = np.tile(np.arange(0, self.n_actions,
                            dtype=np.int64), batch_size)
step_obs, step_rewards = [], []
```

After that, we need to prepare the array with the actions that we want the EM to take for every observation. As we repeated every observation `n_actions` times, our actions array will also have the form `[0, 1, 2, 3, 0, 1, 2, 3, ...]` (Breakout has four actions in total). In the `step_obs` and `step_rewards` lists, we will save observations and immediate rewards produced for every rollout step by the EM model.

This data will be passed to `RolloutEncoder` to be embedded into fixed-vector form.

```
for step_idx in range(self.rollout_steps):
    actions_t = torch.LongTensor(actions).to(batch.device)
    obs_next_v, reward_v = \
        self.net_em(obs_batch_v, actions_t)
```

Then, we start the loop for every rollout step. For every step, we ask the EM network to predict the next observation (returned as the delta to the current observation) and immediate reward. Subsequent steps will have actions selected using the rollout policy network.

```
step_obs.append(obs_next_v.detach())
step_rewards.append(reward_v.detach())
# don't need actions for the last step
if step_idx == self.rollout_steps-1:
    break
```

We store the observation delta and immediate reward in lists for `RolloutEncoder` and stop the loop if we're at the final rollout step. The early stop is possible as the rest of the code in the loop is supposed to select the actions, but for the last step we don't need actions at all.

```
# combine the delta from EM into new observation
cur_plane_v = obs_batch_v[:, 1:2]
new_plane_v = cur_plane_v + obs_next_v
obs_batch_v = torch.cat(
    (cur_plane_v, new_plane_v), dim=1)
```

To be able to use the rollout policy network, we need to create a normal observation tensor from the delta returned by the EM network. To do this, we take the last channel from the current observation, add the delta from the EM to it, creating a predicted frame, and then combine them into the normal observation tensor of shape (`batch_size * n_actions, 2, 84, 84`).

```
# select actions
logits_v, _ = self.net_policy(obs_batch_v)
probs_v = F.softmax(logits_v, dim=1)
probs = probs_v.data.cpu().numpy()
actions = self.action_selector(probs)
```

In the rest of the loop, we use the created observations batch to select the actions using the rollout policy network and convert the returned probability distribution into the action indices. Then, the loop continues to predict the next rollout step.

```
step_obs_v = torch.stack(step_obs)
step_rewards_v = torch.stack(step_rewards)
```

```
flat_enc_v = self.encoder(step_obs_v, step_rewards_v)
return flat_enc_v.view(batch_size, -1)
```

When we are done with all the steps, two lists, `step_obs` and `step_rewards`, will contain tensors for every step. Using the `torch.stack()` function, we join them on the new dimension. The resulting tensors have rollout steps as the first dimension and `batch_size * n_actions` as the second dimension. These two tensors are passed to `RolloutEncoder`, which produces an encoded vector for every entry in the second dimension. The output from the encoder is a tensor of `(batch_size*n_actions, encoded_len)`, and we want to concatenate encodings for different actions of the same batch sample together. To do this, we just reshape the output tensor to have `batch_size` as the first dimension, so the output from the function will have a `(batch_size, encoded_len*n_actions)` shape.

The Rollout encoder

The `RolloutEncoder` class accepts two tensors: observations of `(rollout_steps, batch_size, 1, 84, 84)` and rewards of `(rollout_steps, batch_size)`. It applies a **recurrent neural network** (**RNN**) along the rollout steps to convert every batch series into the encoded vector. Before the RNN, we have a preprocessor that extracts features from the observation delta given by the EM, and then the reward value is just appended to the features vector.

```
class RolloutEncoder(nn.Module):
    def __init__(self, input_shape, hidden_size=ROLLOUT_HIDDEN):
        super(RolloutEncoder, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32,
                      kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU(),
        )
```

The observation preprocessor has the same Atari convolution layers, except that the input tensor has a single channel, which is a delta between consecutive observations produced by the EM.

```
        conv_out_size = self._get_conv_out(input_shape)
        self.rnn = nn.LSTM(input_size=conv_out_size+1,
                           hidden_size=hidden_size,
                           batch_first=False)
```

The RNN of the encoder is an LSTM layer. The `batch_first=False` argument is a bit redundant (as the default value for the argument is also `False`), but is left here to remind us about the input tensor order, which is (`rollout_steps, batch_ size, conv_features+1`), so the time dimension has zero index.

```
def _get_conv_out(self, shape):
    o = self.conv(torch.zeros(1, *shape))
    return int(np.prod(o.size()))

def forward(self, obs_v, reward_v):
    # Input is in (time, batch, *) order
    n_time = obs_v.size()[0]
    n_batch = obs_v.size()[1]
    n_items = n_time * n_batch
    obs_flat_v = obs_v.view(n_items, *obs_v.size()[2:])
    conv_out = self.conv(obs_flat_v)
    conv_out = conv_out.view(n_time, n_batch, -1)
    rnn_in = torch.cat((conv_out, reward_v), dim=2)
    _, (rnn_hid, _) = self.rnn(rnn_in)
    return rnn_hid.view(-1)
```

The `forward()` function is obvious from the encoder architecture and first extracts the features from all `rollout_steps*batch_size` observations and then applies LSTM to the sequence. As an encoded vector of the rollout, we take the hidden state returned by the last step of the RNN.

The training of I2A

The training process has two steps: we train the I2A model in the usual A2C manner, and we do a distillation of the rollout policy using a separate loss. The distillation training is needed to approximate the I2A behavior by a smaller policy used during rollout steps to select actions. The actions chosen in imagined trajectories should be similar to the actions that the agent will choose in real situations. However, during the rollouts, we can't just use our main I2A model for action selection, as the main I2A model will need to do rollouts again. To break this contradiction, distillation is used, which is a very simple cross-entropy loss between the policy of the main I2A model during the training and the policy returned by the rollout policy network. This training step has a separate optimizer responsible only for rollouts' policy parameters.

The piece of the training loop that is responsible for distillation is given as follows. The array `mb_probs` contains the probabilities of actions chosen by the I2A model for observations, `obs_v`.

```
probs_v = torch.FloatTensor(mb_probs).to(device)
```

```
                   policy_opt.zero_grad()
                   logits_v, _ = net_policy(obs_v)
                   policy_loss_v = -F.log_softmax(logits_v, dim=1) * \
                                    probs_v.view_as(logits_v)
                   policy_loss_v = policy_loss_v.sum(dim=1).mean()
                   policy_loss_v.backward()
                   policy_opt.step()
```

Another step in the training that is supposed to train the I2A model is performed exactly the same way as we train the usual A2C, ignoring all the internals of the I2A model: the value loss is the MSE between the predicted and discounted reward approximated by the Bellman equation, while the **policy gradient** is approximated by the advantage multiplied by log probability of the chosen action.

Experiment results

In this section, we will take a look at the results of our multistep training process.

The baseline agent

To train the agent, run `Chapter22/01_a2c.py` with the optional `--cuda` flag to enable the graphics processing unit (GPU) and the required `-n` option with the experiment name used in TensorBoard and in a directory name to save the models.

```
Chapter22$ ./01_a2c.py --cuda -n tt
AtariA2C(
  (conv): Sequential(
    (0): Conv2d(2, 32, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (5): ReLU()
  )
  (fc): Sequential(
    (0): Linear(in_features=3136, out_features=512, bias=True)
    (1): ReLU()
  )
  (policy): Linear(in_features=512, out_features=4, bias=True)
  (value): Linear(in_features=512, out_features=1, bias=True)
```

```
)
4: done 13 episodes, mean_reward=0.00, best_reward=0.00, speed=696.72
9: done 12 episodes, mean_reward=0.00, best_reward=0.00, speed=721.23
10: done 2 episodes, mean_reward=1.00, best_reward=1.00, speed=740.74
13: done 6 episodes, mean_reward=0.00, best_reward=1.00, speed=735.17
```

In 300k training iterations, which took 15 hours, the A2C was able to reach the mean reward of 400 on test episodes with five lives and unclipped reward. The maximum test reward on three full episodes was 720. The following figures show the convergence plots.
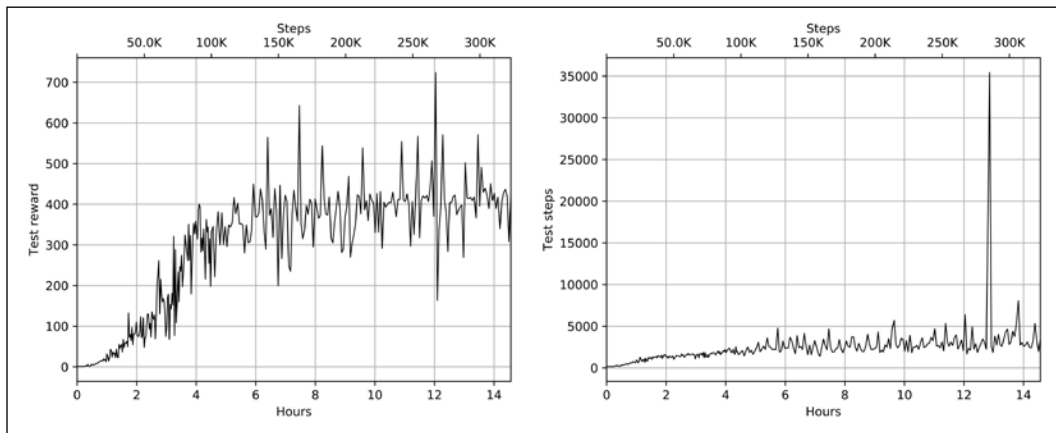
Figure 22.5: The A2C baseline: test episodes reward (left) and steps (right)
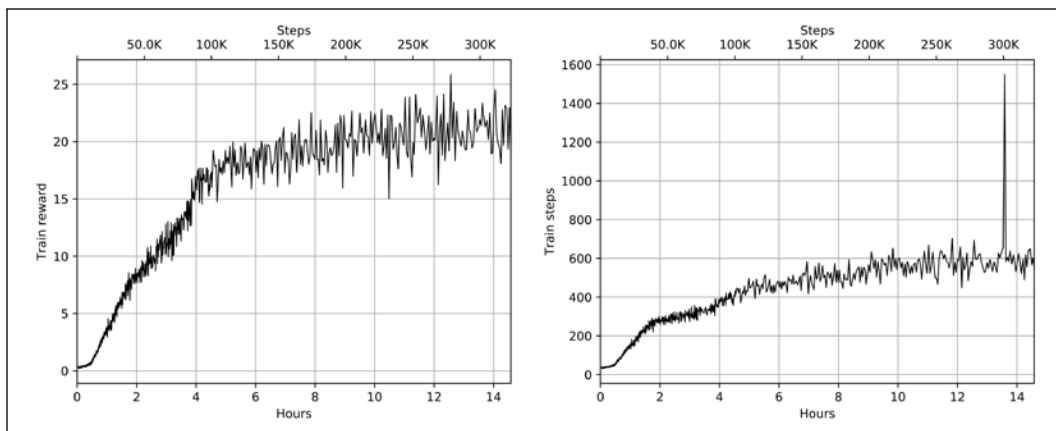
Figure 22.6: The reward (left) and steps (right) for training episodes (one game life)
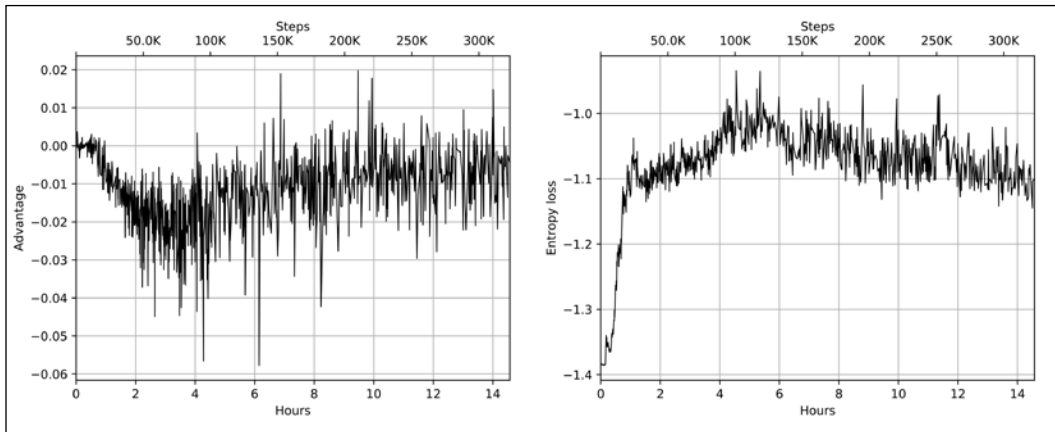
Figure 22.7: The advantage (left) and entropy loss (right) during the training
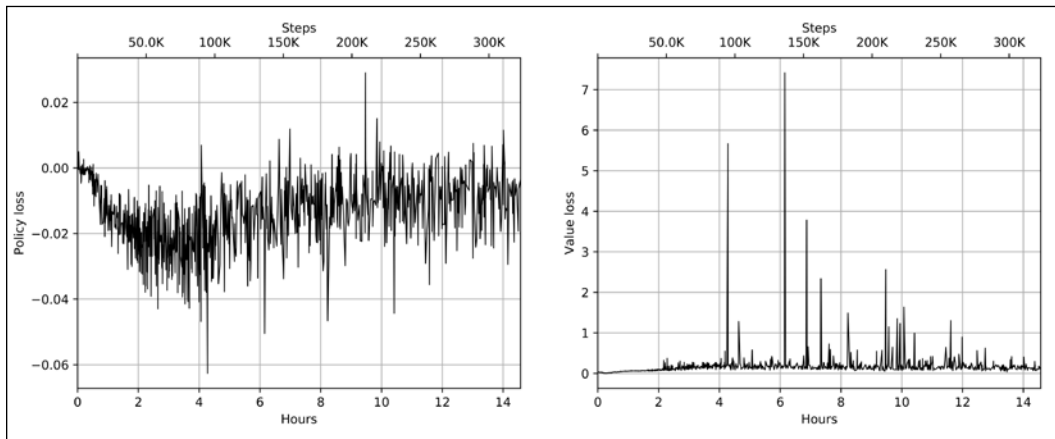
Figure 22.8: The policy loss (left) and value loss (right)

Training EM weights

To train the EM, you need to specify the policy produced during the baseline agent training. In my experiments, I took the policy from the partially trained agent to increase the potential diversity of the EM training data. Another way to increase the sample diversity would be to use several policies for data generation.

```
$ ./02_imag.py --cuda -m saves/01_a2c_t1/best_0400.333.dat -n t1
EnvironmentModel(
  (conv1): Sequential(
    (0): Conv2d(6,64,kernel_size=(4,4),stride=(4, 4),padding=(1,1))
    (1): ReLU()
```

```
    (2): Conv2d(64,64,kernel_size=(3,3),stride=(1,1),padding=(1,1))
    (3): ReLU()
  )
  (conv2): Sequential(
    (0): Conv2d(64,64,kernel_size=(3,3),stride=(1,1),padding=(1,1))
    (1): ReLU()
  )
  (deconv): ConvTranspose2d(64,1,kernel_size=(4,4),stride=(4,4))
  (reward_conv): Sequential(
    (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (1): MaxPool2d(kernel_size=2,stride=2,padding=0,dilation=1)
    (2): ReLU()
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1)
    (5): ReLU()
  )
  (reward_fc): Sequential(
    (0): Linear(in_features=576, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=1, bias=True)
  )
)
Best loss updated: inf -> 1.7988e-02
Best loss updated: 1.7988e-02 -> 1.1621e-02
Best loss updated: 1.1621e-02 -> 9.8923e-03
Best loss updated: 9.8923e-03 -> 8.6424e-03
...
```

In 200k training iterations, the loss almost stopped decreasing. The EM model with the smallest loss could be used for final training of the I2A model. The EM training dynamics follow.
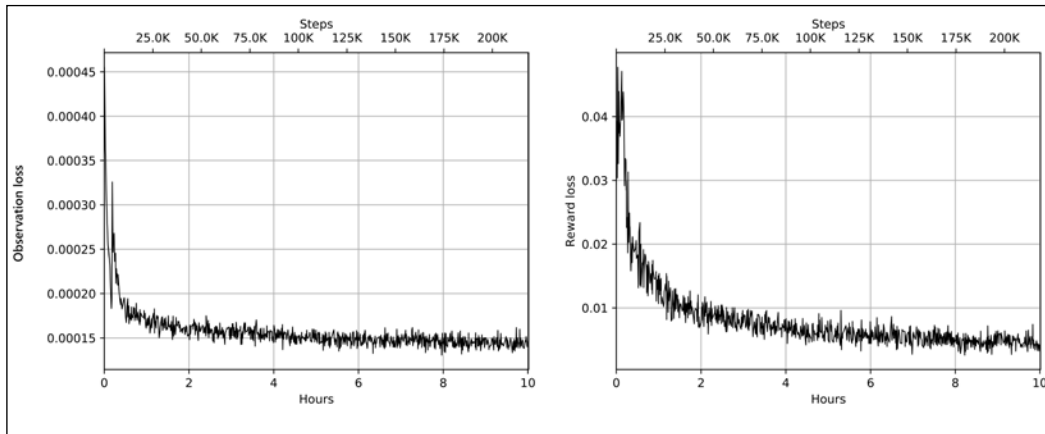
Figure 22.9: The EM training loss: observations (left) and reward (right)

Training with the I2A model

The imagination path comes with a significant computation cost, which is proportional to the number of rollout steps performed. I experimented with several values for this hyperparameter. For Breakout, there is not much difference between five and three steps, but the speed is almost two times faster.

```
$ ./03_i2a.py --cuda -n t1 --em saves/02_env_t1/best_9.78e-04.dat
I2A(
  (conv): Sequential(
    (0): Conv2d(2, 32, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (5): ReLU()
  )
  (fc): Sequential(
    (0): Linear(in_features=4160, out_features=512, bias=True)
    (1): ReLU()
  )
  (policy): Linear(in_features=512, out_features=4, bias=True)
  (value): Linear(in_features=512, out_features=1, bias=True)
  (encoder): RolloutEncoder(
```

```
  (conv): Sequential(
    (0): Conv2d(1, 32, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (5): ReLU()
  )
  (rnn): LSTM(3137, 256)
  )
)
2: done 1 episodes, mean_reward=0, best_reward=0, speed=160.41 f/s
4: done 12 episodes, mean_reward=0, best_reward=0, speed=190.84 f/s
7: done 1 episodes, mean_reward=0, best_reward=0, speed=169.94 f/s
...
```

In 300k training steps, which took more than two days on my hardware, I2A was able to reach the mean reward of 500 on the test, which shows better dynamics than the baseline. The maximum test reward on three full episodes was 750, which is also better than the 720 obtained by the baseline. The only serious downside of the I2A method is much lower performance, which is approximately four times slower than A2C. Following are figures for I2A training.
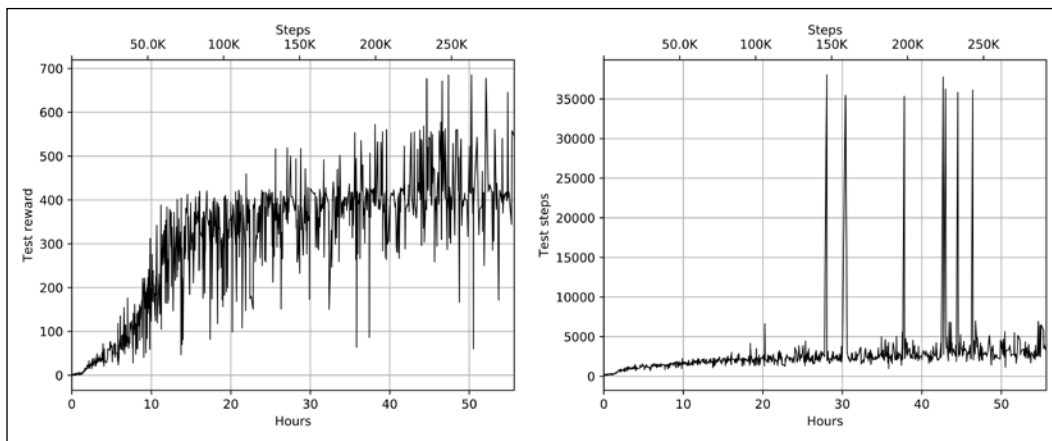
Figure 22.10: The reward (left) and steps (right) for the I2A test episodes
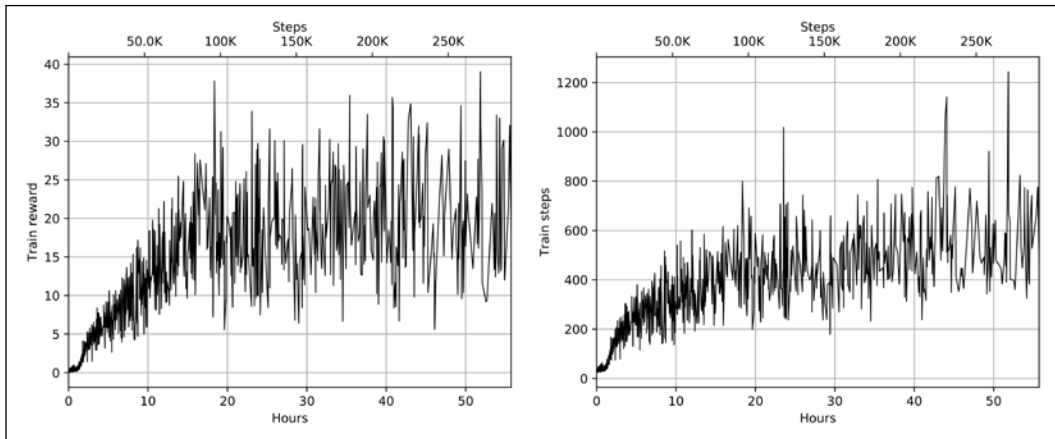
Figure 22.11: I2A training episodes (one life): the reward (left) and steps (right)
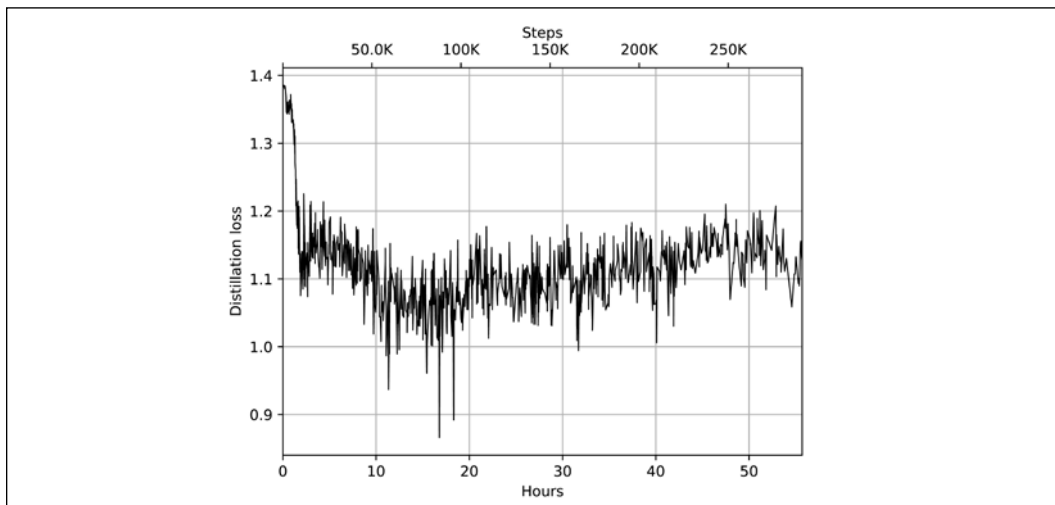
Figure 22.12: Distillation loss

I also ran an experiment with a single step in the rollout; surprisingly, the training dynamics between one step and three steps weren't much different, which may be a sign that in Breakout, the agent doesn't need to imagine the trajectory for too long to get the benefits from the EM. This is appealing, as with a single step, we don't need rollout policy at all (as the first step is always performed on all actions). An RNN is not needed either, which can significantly speed up the agent, pushing its performance close to the baseline A2C.

Summary

In this chapter, we discussed the model-based approach to RL and implemented one of the recent research architectures from DeepMind that augments the model of the environment. This model tries to join both model-free and model-based paths into one to allow the agent to decide which knowledge to use.

In the next chapter, we will take a look at a recent DeepMind breakthrough in the area of full-information games: *the AlphaGo Zero algorithm*.

References

1. *Reinforcement Learning with Unsupervised Auxiliary Tasks* by *Max Jaderberg, Volodymyr Mnih*, and others, (arXiv:1611.05397)

2. *Imagination-Augmented Agents for Deep Reinforcement Learning* by *Theophane Weber, Sebastien Racantiere*, and others, (arXiv:1707.06203)

23
AlphaGo Zero

We will now continue our discussion about model-based methods by exploring the cases when we have a model of the environment, but this environment is being used by two competing parties. This situation is very familiar in board games, where the rules of the game are fixed and the full position is observable, but we have an opponent who has the primary goal of preventing us from winning the game.

Recently, DeepMind proposed a very elegant approach to solving such problems. No prior domain knowledge is required, but the agent improves its policy only via self-play. This method is called **AlphaGo Zero**.

In this chapter, we will:

- Discuss the structure of the AlphaGo Zero method
- Implement the method for playing the game Connect 4

Board games

Most board games provide a setup that is different from an arcade scenario. The Atari game suite assumes that one player is making decisions in some environment with complex dynamics. By generalizing and learning from the outcome of their actions, the player improves their skills, increasing their final score. In a board game setup, however, the rules of the game are usually quite simple and compact. What makes the game complicated is the number of different positions on the board and the presence of an opponent with an unknown strategy who tries to win the game.

With board games, the ability to observe the game state and the presence of explicit rules opens up the possibility of analyzing the current position, which isn't the case for Atari. This analysis means taking the current state of the game, evaluating all the possible moves that we can make, and then choosing the best move as our action.

The simplest approach to evaluation is to iterate over the possible actions and recursively evaluate the position after the action is taken. Eventually, this process will lead us to the final position, when no more moves are possible. By propagating the game result back, we can estimate the expected value of any action in any position. One possible variation of this method is called **minimax**, which is when we are trying to make the strongest move, but our opponent is trying to make the worst move for us, so we are iteratively minimizing and maximizing the final game objective of walking down the tree of game states (which will be described in detail later).

If the number of different positions is small enough to be analyzed entirely, like in the tic-tac-toe game (which has only 138 terminal states), it's not a problem to walk down this game tree from any state that we have and figure out the best move to make.

Unfortunately, this brute-force approach doesn't work even for medium-complexity games, as the number of configurations grows exponentially. For example, in the game of draughts (also known as checkers), the total game tree has 5*1020 nodes, which is quite a challenge even for modern hardware. In the case of more complex games, like chess or Go, this number is much larger, so it's just not possible to analyze all the positions reachable from every state. To handle this, usually some kind of approximation is used, when we analyze the tree up to some depth. With a combination of careful search and stop criteria, called **tree pruning**, and the smart predefined evaluation of positions, we can make a computer program that plays complex games at a fairly good level.

In late 2017, DeepMind published an article in the journal *Nature* presenting a novel approach called AlphaGo Zero, which was able to achieve a superhuman level of play in complex games, like Go and chess, without any prior knowledge except the game rules. The agent was able to improve its policy by constantly playing against itself and reflecting on the outcomes. No large game databases, handmade features, or pretrained models were needed. Another nice property of the method is its simplicity and elegance.

In the example of this chapter, we will try to understand and implement this approach for the game Connect 4 (also known as four in a row or four in a line) to evaluate it ourselves.

The AlphaGo Zero method

In this section, we will discuss the structure of the method. The whole system contains several parts that need to be understood before we can implement them.

Overview

At a high level, the method consists of three components, all of which will be explained in detail later, so don't worry if something is not completely clear from this section:

- We constantly traverse the game tree using the **Monte Carlo tree search** (**MCTS**) algorithm, the core idea of which is to semi-randomly walk down the game states, expanding them and gathering statistics about the frequency of moves and underlying game outcomes. As the game tree is huge, both in terms of the depth and width, we don't try to build the full tree; we just randomly sample its most promising paths (that's the source of the method's name).

- At every moment, we have a *best player*, which is the model used to generate the data via self-play. Initially, this model has random weights, so it makes moves randomly, like a four-year-old just learning how chess pieces move. However, over time, we replace this best player with better variations of it, which generate more and more meaningful and sophisticated game scenarios. Self-play means that the same *current best* model is used on both sides of the board. This might not look very useful, as having the same model play against itself has an approximately 50% chance outcome, but that's actually what we need: samples of the games where our best model can demonstrate its best skills. The analogy is simple: it's usually not very interesting to watch a match between the outsider and the leader; the leader will win easily. What is much more fun and intriguing to see is when players of roughly equal skill compete. That's why the final in any championship attracts much more attention than the preceding matches: both teams or players in the final usually excel in the game, so they will need to play their best game to win.

- The third component in the method is the training process of the other, *apprentice* model, which is trained on the data gathered by the best model during self-play. This model could be compared to a kid sitting and constantly analyzing the chess games played by two adults. Periodically, we play several matches between this trained model and our current best model. When the trainee is able to beat the best model in the majority of games, we announce the trained model as the new best model and the process continues.

Despite the simplicity and even naivety of this, AlphaGo Zero was able to beat all the previous AlphaGo versions and became the best Go player in the world, without any prior knowledge except the game rules. After the paper called *Mastering the Game of Go Without Human Knowledge* [1] was published, DeepMind adapted the same method to fit chess and published the paper called *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm* [2], where the model trained from scratch beat Stockfish, which was the best chess program and took more than a decade for human experts to develop.

Now, let's check all three components of the method in detail.

MCTS

To understand what MCTS does, let's consider a simple subtree of the tic-tac-toe game, as shown in the following diagram. In the beginning, the game field is empty and the cross (X) needs to choose where to move. There are nine different options for the first move, so our root state has nine different branches leading to the corresponding states.
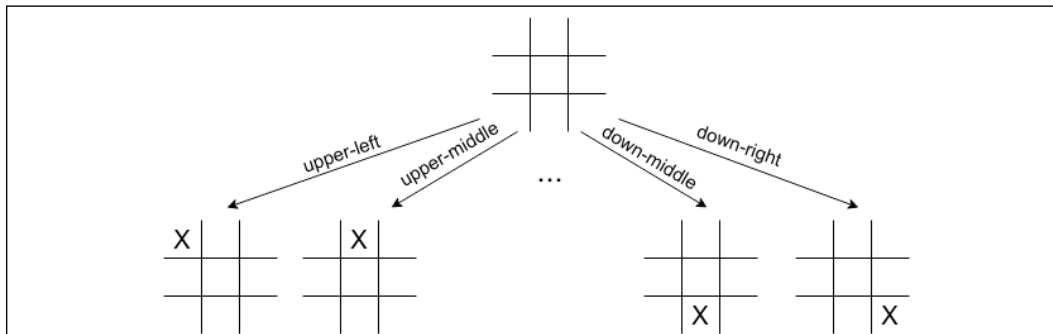
Figure 23.1: The game tree of tic-tac-toe

The number of possible actions at some particular game state is called the **branching factor**, and it shows the bushiness of the game tree. Of course, this is not constant and may vary, as some moves are not always doable. In the case of tic-tac-toe, the number of available actions could vary from nine at the beginning of the game to zero at the leaf nodes. The branching factor allows us to estimate how quickly the game tree grows, as every available action leads to another set of actions that could be taken.

For our example, after the cross has made its move, the nought (0) has eight alternatives at every nine positions, which makes 9*8 total positions at the second level of the tree. The total number of nodes in the tree can be up to 9! = 362880, but the actual number is less, as not all the games could be played to the maximum depth.

Tic-tac-toe is tiny, but if we consider larger games and, for example, think about the count of the first moves that white could make at the beginning of a chess game (which is 20) or the number of spots that the white stone could be placed at in Go (361 in total for a 19×19 game field), the number of game positions in the complete tree quickly becomes enormous. With every new level, the number of states gets multiplied by the average number of actions that we could perform on the previous level.

To deal with this combinatorial explosion, random sampling comes into play. In a general MCTS, we perform many iterations of depth-first search, starting at the current game state and either selecting the actions randomly or with some strategy, which should include enough randomness in its decisions. Every search is continued until the end state of the game, and then it is followed by updating the weights of the visited tree branches according to the game's outcome. This process is similar to the value iteration method, when we played the episodes and the final step of the episode influenced the value estimation of all the previous steps. This is a general MCTS, and there are many variants of this method related to expansion strategy, branch selection policy, and other details.

In AlphaGo Zero, a variant of MCTS is used. For every edge (representing the move from some position), this set of statistics is stored: a prior probability, $P(s, a)$, of the edge, a visit count, $N(s, a)$, and an action value, $Q(s, a)$. Each search starts from the root state following the most promising actions, selected using the utility value, $U(s, a)$, proportional to $Q(s,a) + \dfrac{P(s,a)}{1 + N(s,a)}$.

Randomness is added to the selection process to ensure enough exploration of the game tree. Every search could end up with two outcomes: the end state of the game is reached, or we face a state that hasn't been explored yet (in other words, has no statistics for values). In the latter case, the policy neural network (NN) is used to obtain the prior probabilities and the value of the state estimation, and the new tree node with $N(s, a) = 0$, $P(s, a) = p_{net}$ (which is a probability of the move returned by the network) and $Q(s, a) = 0$ is created. Besides the prior probability of the actions, the network returns the estimation of the game's outcome (or the value of the state) as seen from the current player.

As we have obtained the value (by reaching the final game state or by expanding the node using the NN), a process called the *backup of value* is performed. During the process, we traverse the game path and update statistics for every visited intermediate node; in particular, the visit count, $N(s, a)$, is incremented by one and $Q(s, a)$ is updated to include the game outcome from the current state perspective. As two players are exchanging moves, the final game outcome changes the sign in every backup step.

This search process is performed several times (in AlphaGo Zero's case, one to two thousand searches are performed), gathering enough statistics about the action to use the $N(s, a)$ counter as an action probability to be taken in the root node.

Self-play

In AlphaGo Zero, the NN is used to approximate the prior probabilities of the actions and evaluate the position, which is very similar to the **advantage actor-critic** (**A2C**) two-headed setup. On the input of the network, we pass the current game position (augmented with several previous positions) and return two values. The policy head returns the probability distribution over the actions, and the value head estimates the game outcome as seen from the player's perspective. This value is undiscounted, as moves in Go are deterministic. Of course, if you have stochasticity in a game, like in backgammon, some discounting should be used.

As has already been described, we're maintaining the current best network, which constantly self-plays to gather the training data for our apprentice network. Every step in each self-play game starts with several MCTSes from the current position to gather enough statistics about the game subtree to select the best action. The concrete selection depends on the move and our settings. For self-play games, which are supposed to produce enough variance in the training data, the first moves are selected in a stochastic way. However, after some number of steps (which is a hyperparameter in the method), action selection becomes deterministic, and we select the action with the largest visit counter, $N(s, a)$. In evaluation games (when we check the network being trained versus the current best model), all the steps are deterministic and selected solely on the largest visit counter.

Once the self-play game has been finished and the final outcome has become known, every step of the game is added to the training dataset, which is a list of tuples (s_t, π_t, r_t), where s_t is the game state, π_t is the action probabilities calculated from MCTS sampling, and r_t is the game's outcome from the perspective of the player at step t.

Training and evaluation

The self-play process between two clones of the current best network provides us with a stream of the training data, consisting of states, action probabilities, and position values obtained from the self-play games. With this at hand, our training is simple: we sample mini-batches from the replay buffer of training examples and minimize the mean squared error (MSE) between the value head prediction and the actual position value, as well as cross-entropy loss between predicted probabilities and sampled probabilities, π.

As mentioned earlier, once in several training steps the evaluation of the trained network is performed, which consists of playing several games between the current best and trained networks. Once the trained network becomes significantly better than the current best network, we copy the trained network into the best network and continue the process.

The Connect 4 bot

To see the method in action, let's implement AlphaGo Zero for Connect 4. The game is for two players with fields 6×7. Players have disks of two different colors, which they drop in turn into any of the seven columns. The disks fall to the bottom, stacking vertically. The game objective is to be the first to form a horizontal, vertical, or diagonal group of four disks of the same color. Two game situations are shown in the following diagram. In the first situation, the first player has just won, while in the second, the second player is going to form a group.
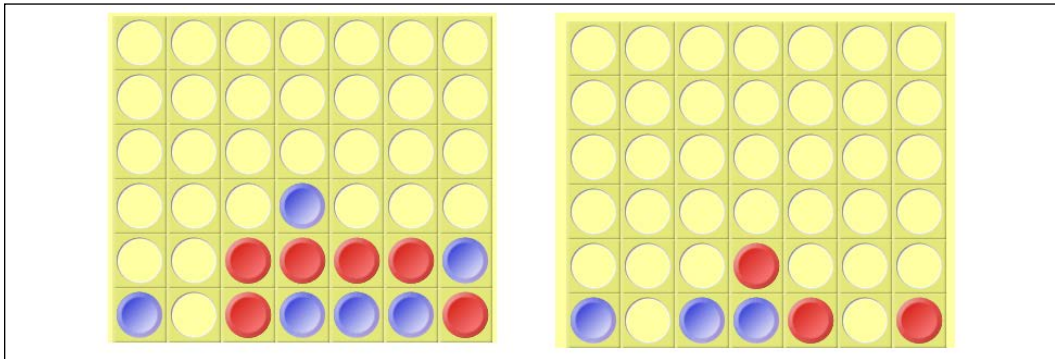
Figure 23.2: Two game positions in Connect 4

Despite its simplicity, this game has 4.5×10^{12} different game states, which is challenging for computers to solve with brute force. This example consists of several tools and library modules:

- `Chapter23/lib/game.py`: A low-level game representation that contains functions to make moves, encode, and decode the game state, and other game-related utilities.

- `Chapter23/lib/mcts.py`: The MCTS implementation that allows graphics processing unit-accelerated expansion of leaves and node backup. The central class here is also responsible for keeping the game node statistics, which are reused between the searches.

- `Chapter23/lib/model.py`: The NN and other model-related functions, such as the conversion between game states and the model's input and the playing of a single game.

- `Chapter23/train.py`: The main training utility that glues everything together and produces the model checkpoints of the new best networks.

- `Chapter23/play.py`: The tool that organizes the automated tournament between the model checkpoints. This accepts several model files and plays the given number of games against each other to form a leaderboard.

- `Chapter23/telegram-bot.py`: The bot for the Telegram chat platform that allows the user to play against any model file, keeping the statistics. This bot was used for human verification of the example's results.

The game model

The whole approach is based on our ability to predict the outcome of our actions; in other words, we need to be able to get the resulting game state after we execute some particular game move. This is a much stronger requirement than we had in Atari environments and Gym in general, where you can't specify any current state that you want to act from. So, we need a model of the game that encapsulates the game's rules and dynamics. Luckily, most board games have a simple and compact set of rules, which makes the model implementation a straightforward task.

In our case, the full game state of Connect 4 is represented by the state of the 6×7 game field cells and the indicator of who is going to move. What is important for our example is to make the game state representation occupy as little memory as possible, but still allow it to work efficiently. The memory requirement is dictated by the necessity of storing large numbers of game states during the MCTS. As our game tree is huge, the more nodes we're able to keep during the MCTS, the better our final approximation of move probabilities will be. So, potentially, we'd like to be able to keep millions, or maybe even billions, of game states in the memory.

With this in mind, the compactness of the game state representation could have a huge impact on memory requirements and the performance of our training process. However, the game state representation has to be convenient to work with, for example, when checking the board for a winning position, making a move, and finding all the valid moves from some state.

To keep this balance, two representations of the game field were implemented in `Chapter23/lib/game.py`. The first *encoded* form is very memory-efficient and takes only 63 bits to encode the full field, which makes it extremely fast and lightweight, as it fits in a machine world on 64-bit architectures.

Another *decoded* game field representation has the form of a list, with length seven, where every entry is a list of integers and keeps the disks in a particular column. This form takes much more memory, but it is more convenient to work with.

I'm not going to show the full code of `Chapter23/lib/game.py`, but if you need it, it's available in the repository. Here, let's just take a look at the list of the constants and functions that it provides:

```
GAME_ROWS = 6
GAME_COLS = 7
BITS_IN_LEN = 3
PLAYER_BLACK = 1
PLAYER_WHITE = 0
COUNT_TO_WIN = 4
INITIAL_STATE = encode_lists([[]] * GAME_COLS)
```

The first two constants in the preceding code define the dimensionality of the game field and are used everywhere in the code, so you can try to change them and experiment with a larger or smaller game. The `BITS_IN_LEN` value is used in state encoding functions and specifies how many bits are used to encode the height of the column (the number of disks present). In the 6×7 game, we could have up to six disks in every column, so three bits is enough to keep values from zero to seven. If you change the number of rows, you will need to adjust `BITS_IN_LEN` accordingly.

The `PLAYER_BLACK` and `PLAYER_WHITE` values define the values used in the *decoded* game representation and, finally, `COUNT_TO_WIN` sets the length of the group that needs to be formed to win the game. So, in theory, you can try to experiment with the code and train the agent for, say, five in a row on a 20×40 field by just changing four numbers in `game.py`.

The `INITIAL_STATE` value contains the *encoded* representation for an initial game state, which has `GAME_COLS` empty lists. The rest of the code is functions. Some of them are used internally, but some make an interface of the game used everywhere in the example. Let's list them quickly:

- `encode_lists(state_lists)`: This converts from a `decoded` to an `encoded` representation of the game state. The argument has to be a list of `GAME_COLS` lists, with the contents of the column specified in the bottom-to-top order. In other words, to drop a new disk on the top of the stack, we just need to append it to the corresponding list. The result of the function is an integer with 63 bits representing the game state.

- `decode_binary(state_int)`: This converts the integer representation of the field back into list form.

- `possible_moves(state_int)`: This returns the list with indices of columns that could be moved from the given encoded game state. The columns are numbered from zero to six, left to right.

- `move(state_int, col, player)`: The central function of the file, which provides game dynamics combined with a win/lose check. In arguments, it accepts the game state in the encoded form, the column to place the disk in, and the index of the player that moves. The column index has to be valid (be present in the result of `possible_moves(state_int)`), otherwise an exception will be raised. The function returns a tuple with two elements: a new game state in the encoded form after the move has been performed and a Boolean indicating the move leading to the win of the player. As a player can win only after their move, a single Boolean is enough. Of course, there is a chance of getting a draw state (when nobody won, but there are no possible moves remaining). Such situations have to be checked by calling the `possible_moves` function after the `move()` function.

- `render(state_int)`: This returns a list of strings representing the field's state. This function is used in the Telegram bot to send the field state to the user.

Implementing MCTS

MCTS is implemented in `Chapter23/lib/mcts.py` and represented by a single class, `MCTS`, which is responsible for performing a batch of MCTSes and keeping the statistics gathered during it. The code is not very large, but it still has several tricky pieces, so let's check it in detail.

```
class MCTS:
    def __init__(self, c_puct=1.0):
        self.c_puct = c_puct
        # count of visits, state_int -> [N(s, a)]
        self.visit_count = {}
        # total value of the state's act, state_int -> [W(s, a)]
        self.value = {}
        # average value of actions, state_int -> [Q(s, a)]
        self.value_avg = {}
        # prior probability of actions, state_int -> [P(s,a)]
        self.probs = {}
```

The constructor has no arguments except the `c_puct` constant, which is used in the node selection process and was mentioned in the original AlphaGo Zero paper [1] as *could be tweaked to increase exploration*, but I'm not redefining it anywhere and haven't experimented with it. The body of the constructor creates an empty container to keep statistics about the states.

The key in all of those dicts is the encoded game state (an integer), and values are lists, keeping the various parameters of actions that we have. The comments above every container have the same notations of values as in the AlphaGo Zero paper.

```
def clear(self):
    self.visit_count.clear()
    self.value.clear()
    self.value_avg.clear()
    self.probs.clear()
```

The preceding method clears the state without destroying the MCTS object, which happens when we switch the current best model to the new one and the gathered statistics become obsolete.

```
def find_leaf(self, state_int, player):
    states = []
    actions = []
    cur_state = state_int
    cur_player = player
    value = None
```

This method is used during the search to perform a single traversal of the game tree, starting from the root node given by the state_int argument and continuing to walk down until one of these two situations has been faced: we reach the final game state or a yet unexplored leaf has been found. During the search, we keep track of the visited states and the executed actions to be able to update the nodes' statistics later.

```
while not self.is_leaf(cur_state):
    states.append(cur_state)

    counts = self.visit_count[cur_state]
    total_sqrt = m.sqrt(sum(counts))
    probs = self.probs[cur_state]
    values_avg = self.value_avg[cur_state]
```

Every iteration of the loop processes the game state that we're currently at. For this state, we extract the statistics that we need to make the decision about the action.

```
if cur_state == state_int:
    noises = np.random.dirichlet(
        [0.03] * game.GAME_COLS)
    probs = [
        0.75 * prob + 0.25 * noise
        for prob, noise in zip(probs, noises)
    ]
score = [
```

```
            value + self.c_puct*prob*total_sqrt/(1+count)
            for value, prob, count in
                zip(values_avg, probs, counts)
        ]
```

The decision about the action is taken based on the `action` utility, which is a sum between $Q(s, a)$ and the prior probabilities scaled to the visit count. The root node of the search process has an extra noise added to the probabilities to improve the exploration of the search process. As we perform the MCTS from different game states along the self-play trajectories, this extra noise ensures that we have tried different actions along the path.

```
        invalid_actions = set(range(game.GAME_COLS)) - \
                          set(game.possible_moves(cur_state))
        for invalid in invalid_actions:
            score[invalid] = -np.inf
        action = int(np.argmax(score))
        actions.append(action)
```

As we have calculated the score for the actions, we need to mask invalid actions for the state. (For example, when the column is full, we can't place another disk on the top.) After that, the action with the maximum score is selected and recorded.

```
        cur_state, won = game.move(
            cur_state, action, cur_player)
        if won:
            value = -1.0
        cur_player = 1-cur_player
        # check for the draw
        moves_count = len(game.possible_moves(cur_state))
        if value is None and moves_count == 0:
            value = 0.0
    return value, cur_state, cur_player, states, actions
```

To finish the loop, we ask our game engine to make the move, returning the new state and the indication of whether the player won the game. The final game states (win, lose, or draw) are never added to the MCTS statistics, so they will always be leaf nodes. The function returns the game's value for the leaf player (or `None` if the final state hasn't been reached), the current player at the leaf state, the list of states we have visited during the search, and the list of the actions taken.

```
    def is_leaf(self, state_int):
        return state_int not in self.probs

    def search_batch(self, count, batch_size, state_int,
                     player, net, device="cpu"):
```

```
        for _ in range(count):
            self.search_minibatch(batch_size, state_int,
                                    player, net, device)
```

The main entry point to the MCTS class is the search_batch() function, which performs several batches of searches. Every search consists of finding the leaf of the tree, optionally expanding the leaf, and doing backup. The main bottleneck here is the expand operation, which requires the NN to be used to get the prior probabilities of the actions and the estimated game value. To make this expansion more efficient, we use mini-batches when we search for several leaves, but then perform expansion in a single NN execution. This approach has one disadvantage: as several MCTSes are performed in one batch, we don't get the same outcome as the execution of them serially.

Indeed, initially, when we have no nodes stored in the MCTS class, our first search will expand the root node, the second will expand some of its child nodes, and so on. However, one single batch of searches can expand only one root node in the beginning. Of course, later, individual searches in the batch could follow the different game paths and expand more, but in the beginning, mini-batch expansion is much less efficient in terms of exploration than a sequential MCTS.

To compensate for this, I still use mini-batches, but perform several of them.

```
        def search_minibatch(self, count, state_int, player,
                             net, device="cpu"):
        backup_queue = []
        expand_states = []
        expand_players = []
        expand_queue = []
        planned = set()
        for _ in range(count):
            value, leaf_state, leaf_player, states, actions = \
                self.find_leaf(state_int, player)
            if value is not None:
                backup_queue.append((value, states, actions))
            else:
                if leaf_state not in planned:
                    planned.add(leaf_state)
                    leaf_state_lists = game.decode_binary(
                        leaf_state)
                    expand_states.append(leaf_state_lists)
                    expand_players.append(leaf_player)
                    expand_queue.append((leaf_state, states,
                                            actions))
```

In the mini-batch search, we first perform the leaf search starting from the same state. If the search has found a final game state (in that case, the returned value will not be equal to `None`), no expansion is required and we save the result for a backup operation. Otherwise, we store the leaf for later expansion.

```
if expand_queue:
    batch_v = model.state_lists_to_batch(
        expand_states, expand_players, device)
    logits_v, values_v = net(batch_v)
    probs_v = F.softmax(logits_v, dim=1)
    values = values_v.data.cpu().numpy()[:, 0]
    probs = probs_v.data.cpu().numpy()
```

To expand, we convert the states into the form required by the model (there is a special function in the `model.py` library) and ask our network to return prior probabilities and values for the batch of states. We will use those probabilities to create nodes, and the values will be backed up on a final statistics update.

```
for (leaf_state, states, actions), value, prob in \
        zip(expand_queue, values, probs):
    self.visit_count[leaf_state] = [0]*game.GAME_COLS
    self.value[leaf_state] = [0.0]*game.GAME_COLS
    self.value_avg[leaf_state] = [0.0]*game.GAME_COLS
    self.probs[leaf_state] = prob
    backup_queue.append((value, states, actions))
```

Node creation is just storing zeros for every action in the visit count and action values (total and average). In prior probabilities, we store values obtained from the network.

```
for value, states, actions in backup_queue:
    cur_value = -value
    for state_int, action in zip(states[::-1],
                                 actions[::-1]):
        self.visit_count[state_int][action] += 1
        self.value[state_int][action] += cur_value
        self.value_avg[state_int][action] = \
            self.value[state_int][action] / \
            self.visit_count[state_int][action]
        cur_value = -cur_value
```

The backup operation is the core process in MCTS and it updates the statistics for a state visited during the search. The visit count of the taken actions is incremented, the total values are just summed, and the average values are normalized using visit counts.

It's very important to properly track the value of the game during the backup, as we have two opponents, and at every turn, the value changes the sign (as a winning position for the current player is a losing game state for the opponent).

```
def get_policy_value(self, state_int, tau=1):
    counts = self.visit_count[state_int]
    if tau == 0:
        probs = [0.0] * game.GAME_COLS
        probs[np.argmax(counts)] = 1.0
    else:
        counts = [count ** (1.0 / tau) for count in counts]
        total = sum(counts)
        probs = [count / total for count in counts]
    values = self.value_avg[state_int]
    return probs, values
```

The final function in the class returns the probability of actions and the action values for the game state, using the statistics gathered during the MCTS. There are two modes of probability calculation, specified by the τ parameter. If it equals to zero, the selection becomes deterministic, as we select the most frequently visited action.

In other cases, the distribution given by $\dfrac{N(s,a)^{\frac{1}{\tau}}}{\sum_k N(s,k)^{\frac{1}{\tau}}}$ is used, which, again, improves exploration.

The model

The NN used is a residual convolution network with six layers, which is a simplified version of the network used in the original AlphaGo Zero method. On the input, we pass the encoded game state, which consists of two 6×7 channels. The first has places with the current player's disks, and the second channel has 1.0 where the opponent has their disks. This representation allows us to make the network player invariant and analyze the position from the perspective of the current player.

The network consists of the common body with residual convolution filters. The features produced by them are passed to the policy and the value heads, which are the combination of a convolution layer and a fully connected layer. The policy head returns the logits for every possible action (the column that drops the disk) and a single-value float. The details are available in the `Chapter23/lib/model.py` file.

Besides the model, this file contains two functions. The first, with the name `state_lists_to_batch`, converts the batch of game states represented in lists into the model's input form. The second method has the `play_game` name and is very important for both the training and testing processes.

Its purpose is to simulate the game between two NNs, perform the MCTS, and optionally store the taken moves in a replay buffer.

```
def play_game(mcts_stores, replay_buffer, net1, net2,
              steps_before_tau_0, mcts_searches, mcts_batch_size,
              net1_plays_first=None, device="cpu"):
    if mcts_stores is None:
        mcts_stores = [mcts.MCTS(), mcts.MCTS()]
    elif isinstance(mcts_stores, mcts.MCTS):
        mcts_stores = [mcts_stores, mcts_stores]
```

The function accepts a lot of parameters:

- The MCTS class instance, which could be a single instance, a list of two instances, or None. We need to be flexible there to cover different usages of this function
- An optional replay buffer
- NNs to be used during the game
- The number of game steps that need to be taken before the τ parameter used for the action probability calculation will be changed from 1 to 0
- The amount of MCTSes to perform
- The MCTS batch size
- Which player acts first

```
        state = game.INITIAL_STATE
        nets = [net1, net2]
        if net1_plays_first is None:
            cur_player = np.random.choice(2)
        else:
            cur_player = 0 if net1_plays_first else 1
        step = 0
        tau = 1 if steps_before_tau_0 > 0 else 0
        game_history = []
```

Before the game loop, we initialize the game state and make a selection of the first player. If there is no information given about who will make the first move, this is chosen randomly.

```
        result = None
        net1_result = None

        while result is None:
            mcts_stores[cur_player].search_batch(
```

```
            mcts_searches, mcts_batch_size, state,
            cur_player, nets[cur_player], device=device)
        probs, _ = mcts_stores[cur_player].get_policy_value(
            state, tau=tau)
        game_history.append((state, cur_player, probs))
        action = np.random.choice(game.GAME_COLS, p=probs)
```

At every turn, we perform the MCTS to populate the statistics and then obtain the probability of actions, which will be sampled to get the action.

```
        if action not in game.possible_moves(state):
            print("Impossible action selected")
        state, won = game.move(state, action, cur_player)
        if won:
            result = 1
            net1_result = 1 if cur_player == 0 else -1
            break
        cur_player = 1-cur_player
        # check the draw case
        if len(game.possible_moves(state)) == 0:
            result = 0
            net1_result = 0
            break
        step += 1
        if step >= steps_before_tau_0:
            tau = 0
```

Then, the game state is updated using the function in the game engine module, and the handling of end-of-game situations is performed.

```
    if replay_buffer is not None:
        for state, cur_player, probs in reversed(game_history):
            replay_buffer.append(
                (state, cur_player, probs, result)
            )
            result = -result
    return net1_result, step
```

At the end of the function, we populate the replay buffer with probabilities for the action and the game result from the perspective of the current player. This data will be used to train the network.

Training

With all those functions in hand, the training process is a simple combination of them in the correct order. The training program is available in `Chapter23/train.py`, and it has logic that has already been described: in the loop, our current best model constantly plays against itself, saving the steps in the replay buffer. Another network is being trained on this data, minimizing the cross-entropy between the probabilities of actions sampled from MCTS and the result of the policy head. MSE between the value head predictions, about the game and the actual game result, is also added to the total loss.

Periodically, the network being trained and the current best network play 100 matches, and if the current network is able to win in more than 60% of them, the network's weights are synced. This process continues infinitely, hopefully, finding models that are more and more proficient in the game.

Testing and comparison

During the training process, the model's weights are saved every time the current best model is replaced with the trained model. As a result, we get multiple agents of various strengths. In theory, the skills of the later models should be better than preceding ones, but we would like to check this ourselves. To do this, there is a tool, `Chapter23/play.py`, that takes several model files and plays a tournament when every model plays a specified number of rounds with all others. The results table, with the number of wins achieved by every model, will represent the relative model's strength.

Another way of checking the performance of resulting agents is by playing them against humans. This was done by me, my kids (thanks Julia and Fedor!), and my friends, who played several matches against the selected models of various strengths. This was done using the bot written for Telegram Messenger, which allows the user to select the model to play against and keeps the global score table for all games. The bot is available in `Chapter23/telegram-bot.py` and it has the same requirements and installation process as the bot from *Chapter 14, Training Chatbots with RL*. (To get it up and running, you need a Telegram bot token to be created and placed in the config file.)

Connect 4 results

To make the training fast, I intentionally chose the hyperparameters of the training process to be small. For example, at every step of the self-play process, only 10 MCTSes were performed, each with a mini-batch size of eight. This, in combination with efficient mini-batch MCTS and the fast game engine, made training very fast.

Basically, after just one hour of training and 2,500 games played in the self-play mode, the produced model was sophisticated enough to be enjoyable to play against. Of course, the level of its play was well below even a kid's level, but it showed some rudimentary strategies and made mistakes in only every other move, which was good progress.

The training was left running for a day, which resulted in 60k games played by a best model and, in total, 105 best model rotations. The training dynamics are shown in the following charts. *Figure 23.3* shows the win ratio (win/loss for the current evaluated policy versus the current best policy).
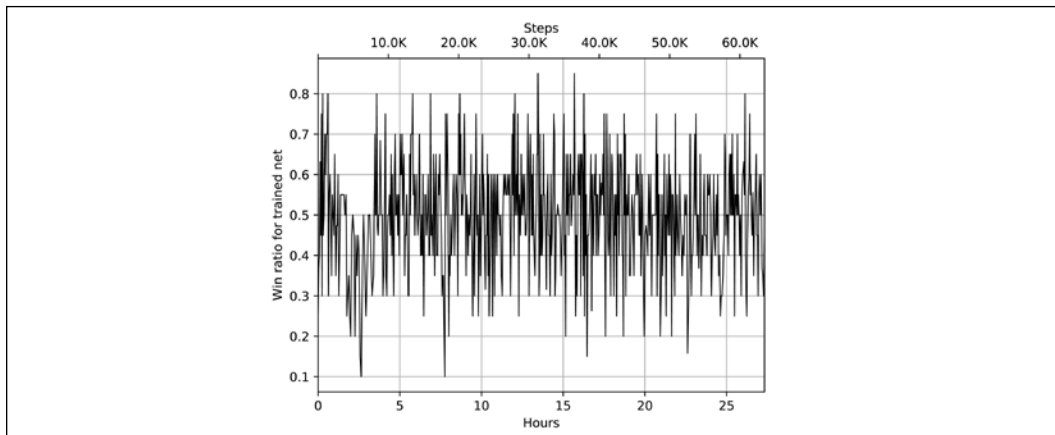
Figure 23.3: The win ratio for the evaluated network
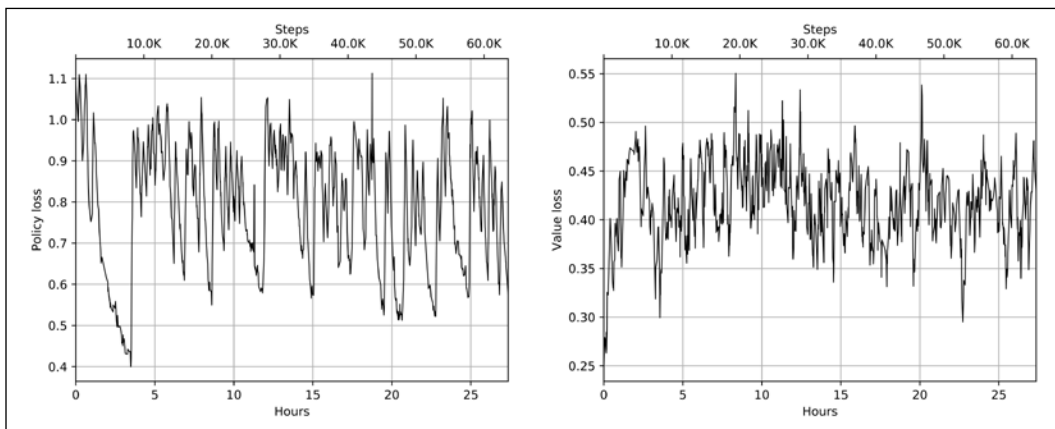
Figure 23.4: Policy loss (left) and value loss (right)

Figure 23.4 shows the loss components, and both have no clear trend. This is due to constant switches of the current best policy, which leads to constant retraining of the trained model.

The tournament verification was complicated by the number of different models, as several games needed to be played by each pair to estimate their strength. To handle this, all 105 models were separated into 10 groups (sorted by time), 100 games were played between all pairs in each group, and then two favorites from each group were selected for a final round. You can find the results in the `Chapter23/tournament` directory.

The chart with the scores obtained by the systems in the final round is shown here. The x axis is the model's index, while the y axis is the relative win ratio the model achieved in the final round:
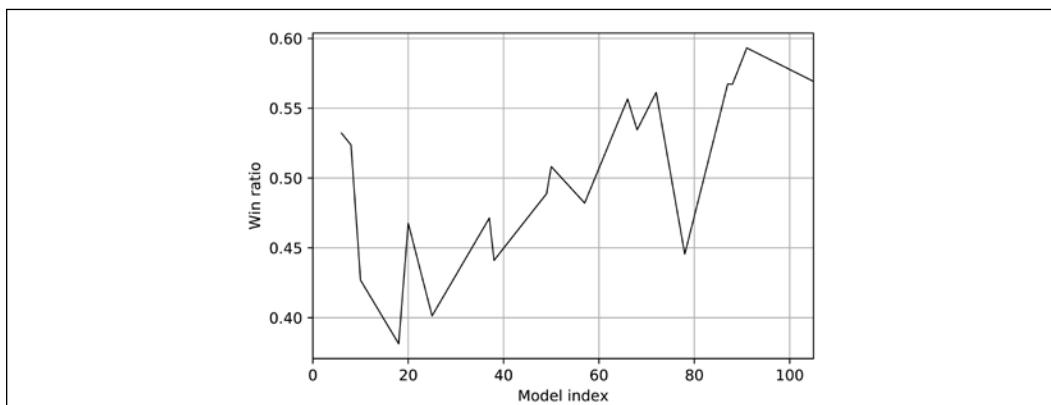
Figure 23.5: The win ratio among the best models saved

The chart shows a slightly noisy, but still consistent, uptrend in the model's success ratio. It is quite likely that with more training, an even better policy might be found.

The top 10 final leaderboard is shown here:

```
1.  best_091_51000.dat:    w=2254, l=1542, d=4
2.  best_105_62400.dat:    w=2163, l=1634, d=3
3.  best_087_48300.dat:    w=2156, l=1640, d=4
4.  best_088_48500.dat:    w=2155, l=1644, d=1
5.  best_072_39800.dat:    w=2133, l=1665, d=2
6.  best_066_37500.dat:    w=2115, l=1683, d=2
7.  best_068_37800.dat:    w=2031, l=1767, d=2
8.  best_006_01500.dat:    w=2022, l=1777, d=1
9.  best_008_01700.dat:    w=1990, l=1809, d=1
10. best_050_29900.dat:    w=1931, l=1869, d=0
```

Human verification was done in the first edition of this book. That time the best model was `best_008_02500.dat`, which was able to win 50% of the played games. The leaderboard is shown here.

Shmuma
/top

rl_ch18_bot

```
Leader board
            Luklusik: won=31, lost=9, draw=0
   best_008_02500.dat: won=8, lost=8, draw=0
              Shmuma: won=6, lost=6, draw=0
   best_019_11300.dat: won=2, lost=4, draw=0
   best_015_09200.dat: won=2, lost=4, draw=0
   best_005_01900.dat: won=2, lost=6, draw=0
              mbilan: won=2, lost=3, draw=0
   best_007_02400.dat: won=1, lost=0, draw=0
             Roman8i: won=1, lost=3, draw=0
   best_010_05900.dat: won=1, lost=0, draw=0
   best_001_00800.dat: won=1, lost=2, draw=0
   best_014_08600.dat: won=1, lost=0, draw=0
   best_004_01600.dat: won=1, lost=0, draw=0
   best_009_05700.dat: won=1, lost=2, draw=0
   best_002_00900.dat: won=1, lost=5, draw=0
   best_022_12200.dat: won=1, lost=5, draw=0
   best_061_34100.dat: won=0, lost=2, draw=0
                None: won=0, lost=1, draw=0
   best_047_28000.dat: won=0, lost=2, draw=0
```

Figure 23.6: The leaderboard of human verification. My daughter is dominating.

Summary

In this chapter, we implemented the AlphaGo Zero method, which was created by DeepMind to solve board games. The primary point of this method is to allow agents to improve their strength via self-play, without any prior knowledge from human games or other data sources.

In the next chapter, we will discuss another direction of practical RL: *discrete optimization problems*, which play an important role in various real-life problems, from schedule optimization to protein folding.

References

1. *Mastering the Game of Go Without Human Knowledge*, David Silver, Julian Schrittwieser, Karen Simonyan, and others, doi:10.1038/nature24270

2. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, David Silver, Thomas Hubert, Julian Schrittwieser, and others, arXiv:1712.01815

24

RL in Discrete Optimization

Next, we will explore the new field in reinforcement learning (RL) application: discrete optimization problems, which will be showcased using the famous Rubik's Cube puzzle.

In this chapter, we will:

- Briefly discuss the basics of discrete optimization
- Cover step by step the paper called *Solving the Rubik's Cube Without Human Knowledge*, by UCI researchers Stephen McAleer et al., 2018, arxiv: 1805.07470, which applies RL methods to the Rubik's Cube optimization problem
- Explore experiments that I've done in an attempt to reproduce the paper's results and directions for future method improvement

RL's reputation

The perception of deep RL is that it is a tool to be used mostly for game playing. This is not surprising given the fact that, historically, the first success in the field was achieved on the Atari game suite by DeepMind in 2015 (`https://deepmind.com/research/dqn/`). The Atari benchmark suite (`https://github.com/mgbellemare/Arcade-Learning-Environment`) turned out to be very successful for RL problems and, even now, lots of research papers use it to demonstrate the efficiency of their methods. As the RL field progresses, the classic 53 Atari games continue to become less and less challenging (at the time of writing, almost all the games have been solved with superhuman accuracy) and researchers are turning to more complex games, like StarCraft and Dota 2.

This perception, which is especially prevalent in the media, is something that I've tried to counterbalance in this book by accompanying Atari games with examples from other domains, including stock trading, chatbots and natural language processing (NLP) problems, web navigation automation, continuous control, board games, and robotics. In fact, RL's very flexible Markov decision process (MDP) model potentially could be applied to a wide variety of domains; computer games are just one convenient and attention-grabbing example of complicated decision-making.

In this chapter, I've tried to write a detailed description of the recent attempt to apply RL to the field of combinatorial optimization (`https://en.wikipedia.org/wiki/Combinatorial_optimization`) in the aforementioned paper (`https://arxiv.org/abs/1805.07470`). In addition, we will cover my implementation of the method described in the paper (which is in the directory `Chapter24` of the book's examples repository) and discuss directions to improve the method. I've intertwined the description of the paper's method with code pieces from my version to illustrate the concepts with concrete implementation.

Let's start with an overview of the Rubik's Cube and combinatorial optimization in general.

The Rubik's Cube and combinatorial optimization

I doubt it's possible to find a person who hasn't heard about the Rubik's Cube, so I'm not going to repeat the Wikipedia description (`https://en.wikipedia.org/wiki/Rubik%27s_Cube`) of this puzzle, but rather focus on the connections it has to mathematics and computer science. If it's not explicitly stated, by "cube" I mean the 3×3 classic Rubik's Cube. There are lots of variations based on the original 3×3 puzzle, but they are still far less popular than the classic invention.

Despite being quite simple in terms of mechanics and the task at hand, the cube is quite a tricky object in terms of all the transformations we can make by possible rotations of its sides. It was calculated that in total, the cube has ~4.33×10^{19} distinct states reachable by rotating it. That's only the states that are reachable without disassembling the cube; by taking it apart and then assembling it, you can get 12 times more states in total: ~5.19×10^{20}, but those "extra" states make the cube unsolvable without disassembling it.

All those states are quite intimately intertwined with each other through rotations of the cube's sides. For example, if we rotate the left side clockwise in some state, we get to the state from which rotation of the same side counterclockwise will destroy the effect of the transformation, and we will get into the original state.

But if we apply the left-side rotation three times in a row, the shortest path to the original state will be just a single rotation of the left side clockwise, but not three times counterclockwise (which is also possible, but just not optimal).

As the cube has six edges and each edge can be rotated in two directions, we have 12 possible rotations in total. Sometimes, half turns (which are two consecutive rotations in the same direction) are also included as distinct rotations, but for simplicity, we will treat them as two distinct transformations of the cube.

In mathematics, there are several areas that study objects of this kind. First of all, there is *abstract algebra*, a very broad division of math that studies abstract sets of objects with operations on top of them. In these terms, the Rubik's Cube is an example of a quite complicated group (`https://en.wikipedia.org/wiki/Group_theory`) with lots of interesting properties.

The cube is not only states and transformations: it's a puzzle, with the primary goal to find a sequence of rotations with the solved cube as the end point. Problems of this kind are studied using combinatorial optimization, which is a subfield of applied math and theoretical computer science. This discipline has lots of famous problems of high practical value, for example:

- The Traveling Salesman Problem (`https://en.wikipedia.org/wiki/Travelling_salesman_problem`): Finds the shortest closed path in a graph

- The protein folding simulation (`https://en.wikipedia.org/wiki/Protein_folding`): Finds possible 3D structures of protein

- Resource allocation: How to spread a fixed set of resources among consumers to get the best objective

What those problems have in common is a huge state space, which makes it infeasible to just check all possible combinations to find the best solution. Our "toy cube problem" also falls into the same category, because a state space of 4.33×10^{19} makes a brute force approach very impractical.

Optimality and God's number

What makes the combinatorial optimization problem tricky is that we're not looking for *any solution*; we're in fact interested in the *optimal solution* of the problem. The difference is obvious: right after the Rubik's Cube was invented, it was known how to reach the goal state (but it took Ernő Rubik about a month to figure out the first method of solving his own invention, which I guess was a frustrating experience). Nowadays, there are lots of different ways or *schemes* of cube solving: the beginner's method, the method by Jessica Fridrich (very popular among speedcubers), and so on.

All of them vary by the amount of moves to be taken. For example, a very simple beginner's method requires about 100 rotations to solve the cube using 5…7 sequences of rotations to be memorized. In contrast, the current world record in the speedcubing competition is solving the cube in 4.22 seconds, which requires much fewer steps, but more sequences to be memorized. The method by Fridrich requires about 55 moves on average, but you need to familiarize yourself with ~120 different sequences of moves.

Of course, the big question is: what is the shortest sequence of actions to solve any given state of the cube? Surprisingly, after 54 years of huge popularity of the cube, humanity still doesn't know the full answer to this question. Only in 2010 did a group of researchers from Google prove that the minimum number of moves needed to solve *any* cube state is 20. This number is also known as *God's number*. Of course, on average, the optimal solution is shorter, as only a bunch of states require 20 moves and one single state doesn't require any moves at all (the solved state). This result only proves the minimal amount of moves; it does not find the solution itself. How to find the optimal solution for any given state is still an open question.

Approaches to cube solving

Before the aforementioned paper was published, there were two major directions for solving the Rubik's Cube:

- By using group theory, it is possible to significantly reduce the state space to be checked. One of the most popular solutions using this approach is Kociemba's algorithm (`https://en.wikipedia.org/wiki/Optimal_ solutions_for_Rubik%27s_Cube#Kociemba's_algorithm`).
- By using brute force search accompanied by manually crafted heuristics, we can direct the search in the most promising direction. A vivid example of this is Korf's algorithm (`https://en.wikipedia.org/wiki/Optimal_ solutions_for_Rubik%27s_Cube#Korf's_algorithm`), which uses A* search with a large database of patterns to cut out bad directions.

The paper introduced a third approach: by training the neural network (NN) on lots of randomly shuffled cubes, it is possible to get the *policy* that will show us the direction to take toward the solved state. The training is done without any prior knowledge about the domain; the only thing needed is the cube itself (not the physical one, but the computer model of it). This is a contrast with the two preceding methods, which require lots of human knowledge about the domain and labor to implement them in the form of a computer code.

In the subsequent sections, we will take a detailed look at this new approach.

Data representation

In the beginning, let's start with *data representation*. In our cube problem, we have two entities to be encoded somehow: actions and states.

Actions

Actions are possible rotations that we can do from any given cube state and, as has already been mentioned, we have only 12 actions in total. For every side, we have two different actions, corresponding to the clockwise and counterclockwise rotation of the side (90° or −90°). One small, but very important, detail is that a rotation is performed from the position when the desired side is facing toward you. This is obvious for the *front* side, for example, but for the *back* side, it might be confusing due to the mirroring of the rotation.

The names of the actions are taken from the cube sides that we're rotating: *left*, *right*, *top*, *bottom*, *front*, and *back*. The first letter of a side's name is used. For instance, the rotation of the *right* side *clockwise* is named as *R*. There are different notations for counter-clockwise actions: sometimes they are denoted with an apostrophe (*R'*) or with a lowercase letter (*r*), or even with a tilde ($\tilde{R}$). The first and the last notations are not very practical for computer code, so in my implementation, I've used lowercase actions to denote counterclockwise rotations. For the right side we have two actions: *R* and *r*, and we have another two for the left side: *L* and *l*, and so on.

In my code, the action space is implemented using Python `enum` in `libcube/cubes/cube3x3.py`, in the class `Action`, where each action is mapped into the unique integer value. In addition, we describe the dictionary with reverse actions:

```
class Action(enum.Enum):
    R = 0
    L = 1
    T = 2
    D = 3
    F = 4
    B = 5
    r = 6
    l = 7
    t = 8
    d = 9
    f = 10
    b = 11

_inverse_action = {
    Action.R: Action.r,
```

```
        Action.r: Action.R,
        Action.L: Action.l,
        Action.l: Action.L,
        Action.T: Action.t,
        Action.t: Action.T,
        Action.D: Action.d,
        Action.d: Action.D,
        Action.F: Action.f,
        Action.f: Action.F,
        Action.B: Action.b,
        Action.b: Action.B
    }
```

States

A state is a particular configuration of the cube's colored stickers and, as I have already mentioned, the size of our state space is very large (4.33×10^{19} different states). But the amount of states is not the only complication we have; in addition, we have different objectives that we would like to meet when we choose a particular representation of a state:

- **Avoid redundancy**: In the extreme case, we can represent a state of the cube by just recording the colors of every sticker on every side. But if we just count the number of such combinations, we get $6^{6 \cdot 8} = 6^{48} \approx 2.5 \times 10^{37}$, which is significantly larger than our cube's state space size, which just means that this representation is highly redundant; for example, it allows all sides of the cube to have one single color (except the center cubelets). If you're curious how I got $6^{6 \cdot 8}$, this is simple: we have six sides of a cube, each having eight small cubes (we're not counting centers), so we have 48 stickers in total and each of them could be colored in one of six colors.

- **Memory efficiency**: As you will see shortly, during the training and, even more so, during the model application, we will need to keep in our computer's memory a large amount of different states of the cube, which might influence the performance of the process. So, we would like the representation to be as compact as possible.

- **Performance of the transformations**: On the other hand, we need to implement all the actions applied to the state, and those actions need to be taken quickly. If our representation is very compact in terms of memory (uses bit-encoding, for example), but requires us to do a lengthy unpacking process for every rotation of the cube's side, our training will become too slow.

- **NN friendliness**: Not every data representation is equally as good as an input for the NN. This is true not only for our case, but for machine learning in general; for example, in NLP, it is common to use bag-of-words or word embeddings; in computer vision, images are decoded from JPEG into raw pixels; random forests require data to be heavily feature-engineered; and so on.

In the paper, every state of the cube is represented as a 20×24 tensor with one-hot encoding. To understand how this is done and why it has this shape, let's start with the picture taken from the paper.

(c) Top view   (d) Bottom view
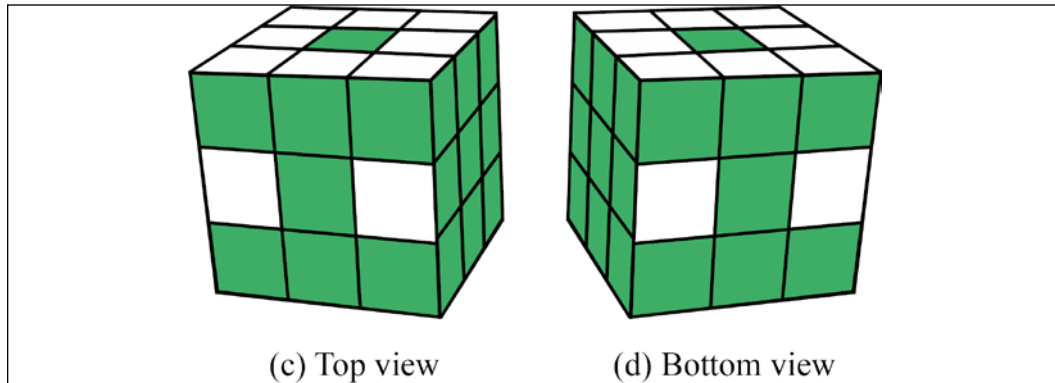
Figure 24.1: Stickers we need to track in the cube are marked in a lighter color

Here, with the light color, are marked the stickers of the cubelets that we need to track; the rest of the stickers (shown as darker) are redundant and there is no need to track them. As you know, a cube consists of three types of cubelets: eight corner cubelets with three stickers, 12 side cubelets with two stickers, and six central ones with a single sticker. It might not be obvious from first sight, but the central cubelets do not need to be tracked at all, as they can't change their relative position and can only rotate. So, in terms of the central cubelets, we need only to agree on the *cube alignment* and stick to it.

As an example, in my implementation, the white side is always on the top, the front is red, the left is green, and so on. This makes our state *rotation-invariant*, which basically means that all possible rotations of the cube as a whole are considered as the same state.

As the central cubelets are not tracked at all, on the figure, they are marked with the darker color. What about the rest? Obviously, every cubelet of a particular kind (corner or side) has a unique color combination of its stickers. For example, the assembled cube in my orientation (white on top, red on the front, and so on) has a top-left cubelet facing us with the following colors: green, white, and red.

There are no other corner cubelets with those colors (please check in case of any doubt). The same is true for the side cubelets.

Due to this, to find the position of some particular cubelet, we need to know the position of only one of its stickers. The selection of such stickers is completely arbitrary, but once they are selected, you need to stick to this. As shown on the preceding figure, we track eight stickers from the top side, eight stickers from the bottom, and four additional side stickers: two on the front face and two on the back. This gives us 20 stickers to be tracked.

Now, let's discuss where "24" in the tensor dimension comes from. In total, we have 20 different stickers to track, but in which positions could they appear due to cube transformations? It depends on the kind of cubelet we're tracking. Let's start with corner cubelets. In total, there are eight corner cubelets and cube transformations can reshuffle them in any order. So, any particular cubelet could end up in any of eight possible corners.

In addition, every corner cubelet could be rotated, so our "green, white, and red" cubelet could end up in three possible orientations:

1. White on top, green left, and red front
2. Green on top, red left, and white front
3. Red on top, white left, and green front

So, to precisely indicate the position and the orientation of the corner cubelet, we have $8 \times 3 = 24$ different combinations.

In the case of the 12 side cubelets, they have only two stickers, so there are only two orientations possible, which again, gives us 24 combinations, but they are obtained from a different calculation: $12 \times 2 = 24$. Finally, we have 20 cubelets to be tracked, eight corner and 12 side, each having 24 positions that it could end up in.

The very popular option to feed such data into an NN is one-hot encoding, when the concrete position of the object has 1, with other positions filled with 0. This gives us the final representation of the state as a tensor with the shape 20×24.

From the redundancy point of view, this representation is much closer to the total state space; the amount of possible combinations equals to $24^{20} \approx 4.02 \times 10^{27}$. It is still larger than the cube state space (it could be said that it is *significantly* larger, as the factor of 10^8 is a lot), but it is better than encoding all the colors of every sticker. This redundancy comes from tricky properties of cube transformations; for example, it is not possible to rotate one single corner cubelet (or flip one side cubelet) *leaving all others in their places*. Mathematical properties are well beyond the scope of this book, but if you're interested, I recommend the wonderful book by Alexander Frey and David Singmaster called *Handbook of Cubik Math*.

You might have noticed that the tensor representation of the cube state has one significant drawback: memory inefficiency. Indeed, by keeping the state as a floating-point tensor of 20 × 24, we're wasting 4 × 20 × 24 = 1,920 bytes of memory, which is a lot given the requirement to keep thousands of states during the training process and millions of them during the cube solving (as you will get to know shortly). To overcome this, in my implementation, I used *two representations*: one tensor is intended for NN input and another, much more compact, representation is needed to store different states for longer. This compact state is saved as a bunch of lists, encoding the permutations of corner and side cubelets, and their orientation. This representation is not only much more memory efficient (160 bytes) but also much more convenient for transformation implementation.

To illustrate this, what follows is the piece of the cube 3×3 library, `libcube/cubes/ cube3x3.py`, that is responsible for compact representation.

```
State = collections.namedtuple("State", field_names=[
    'corner_pos', 'side_pos', 'corner_ort', 'side_ort'])

initial_state = State(corner_pos=tuple(range(8)),
                      side_pos=tuple(range(12)),
                      corner_ort=tuple([0]*8),
                      side_ort=tuple([0]*12))
```

The variable `intial_state` is the encoding of the solved state of the cube. In it, corner and side stickers that we're tracking are in their original positions, and both orientation lists are zero, indicating the initial orientation of the cubelets.

The transformation of the cube is a bit complicated and includes lots of tables keeping the rearrangements of cubelets after different rotations are applied. I'm not going to put this code here; if you're curious, you can start with the function `transform(state, action)` in `libcube/cubes/cube3x3.py`. It also might be helpful to check unit tests of this code in `tests/libcube/cubes/test_cube3x3.py`.

Besides the actions and compact state representation and transformation, the module `cube3x3.py` includes the function that converts the compact representation of the cube state (as the `State` namedtuple) into the tensor form. This functionality is provided by the `encode_inplace()` method.

Another functionality implemented is the ability to render the compact state into human-friendly form by applying the `render()` function. It is very useful for debugging the transformation of the cube, but it's not used in the training code.

The training process

Now that you know how the state of the cube is encoded in a 20 × 24 tensor, let's talk about the NN architecture and how it is trained.

The NN architecture

On the figure that follows (taken from the paper), the network architecture is shown.
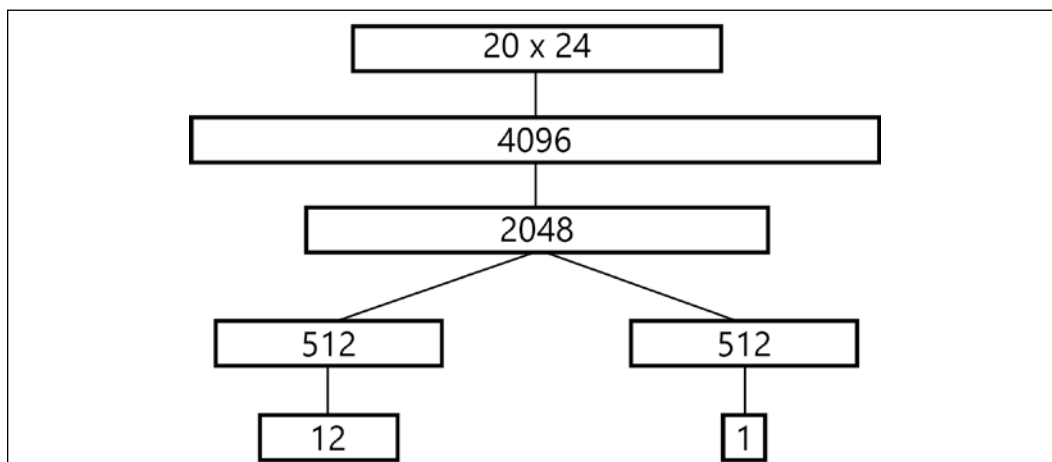
Figure 24.2: The NN architecture transforming the observation (top) to the action and value (bottom)

As the input, it accepts the already familiar cube state representation as a 20 × 24 tensor and produces two outputs:

- The policy, which is a vector of 12 numbers, representing the probability distribution over our actions.

- The value, a single scalar estimating the "goodness" of the state passed. The concrete meaning of a value will be discussed later.

Between the input and output, the network has several fully connected layers with exponential linear unit (ELU) activations. In my implementation, the architecture is exactly the same as in the paper, and the model is in the module `libcube/model.py`.

```
class Net(nn.Module):
    def __init__(self, input_shape, actions_count):
        super(Net, self).__init__()

        self.input_size = int(np.prod(input_shape))
        self.body = nn.Sequential(
            nn.Linear(self.input_size, 4096),
```

```
        nn.ELU(),
        nn.Linear(4096, 2048),
        nn.ELU()
    )
    self.policy = nn.Sequential(
        nn.Linear(2048, 512),
        nn.ELU(),
        nn.Linear(512, actions_count)
    )
    self.value = nn.Sequential(
        nn.Linear(2048, 512),
        nn.ELU(),
        nn.Linear(512, 1)
    )

def forward(self, batch, value_only=False):
    x = batch.view((-1, self.input_size))
    body_out = self.body(x)
    value_out = self.value(body_out)
    if value_only:
        return value_out
    policy_out = self.policy(body_out)
    return policy_out, value_out
```

Nothing is really complicated here, but the `forward()` call might be used in two modes: to get both the policy and the value, or whenever `value_only=True`, only the value. This saves us some computations in the case when only the value head's result is of interest.

The training

The network is quite simple and straightforward: the policy tells us what transformation we should apply to the state, and the value estimates how good the state is. But the big question still remains: how do we train the network?

The training method proposed in the paper has the name Autodidactic Iterations (ADI) and it has a surprisingly simple structure. We start with the goal state (the assembled cube) and apply the sequence of random transformations of some predefined length, *N*. This gives us a sequence of *N* states.

For each state, *s*, in this sequence, we carry out the following procedure:

1. Apply every possible transformation (12 in total) to *s*.
2. Pass those 12 states to our current NN, asking for the value output. This gives us 12 values for every substate of *s*.

3. The target value for s is calculated as $y_{v_i} = \max_a(v_s(a) + R(A(s,a)))$, where $A(s,a)$ is the state after the action, a, is applied to s and $R(s)$ equals 1 if s is the goal state and –1 otherwise.

4. The target policy for s is calculated using the same formula, but instead of max, we take argmax: $y_{p_i} = \text{argmax}_a(v_s(a) + R(A(s,a)))$. This just means that our target policy will have 1 at the position of the maximum value for the substate and 0 on all other positions.

This process is shown on the figure that follows, taken from the paper. The sequence of scrambles $x_0, x_1, \ldots x_N$, is generated, where the cube, x_i is shown expanded. For this state, x_i, we make targets for the policy and value heads from the expanded states by applying the preceding formulas.
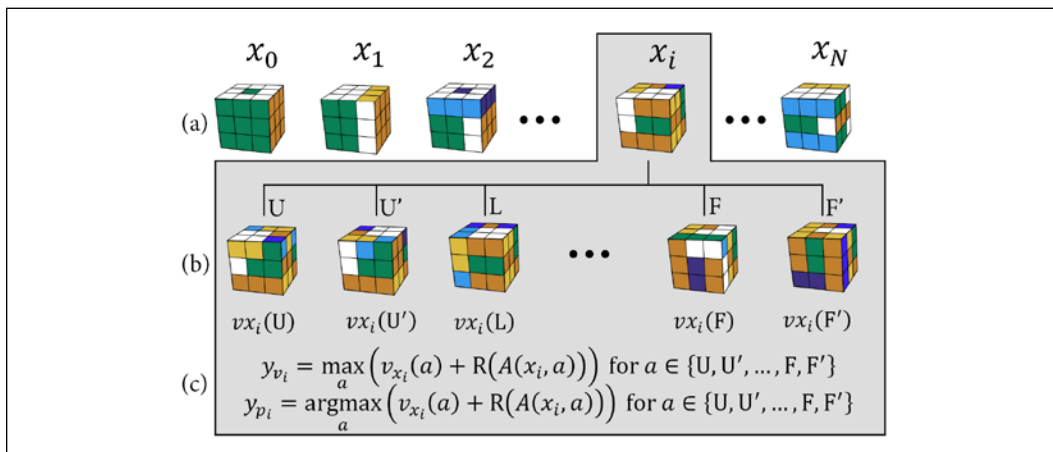
Figure 24.3: Data generation for training

Using this process, we can generate any amounts of training data that we want.

The model application

Okay, imagine that we have trained the model using the process just described. How should we use it to solve the scrambled cube? From the network's structure, you might imagine the obvious, but not very successful, way:

1. Feed the model the current state of the cube that we want to solve

2. From the policy head, get the largest action to perform (or sample it from the resulting distribution)

3. Apply the action to the cube

4. Repeat the process until the solved state has been reached

On paper, this method should work, but in practice, it has one serious issue: it doesn't! The main reason for that is our model's quality. Due to the size of the state space and the nature of the NNs, it just isn't possible to train an NN to return the exact optimal action for *any* input state all of the time. Rather than telling us what to do to get the solved state, our model shows us promising directions to explore. Those directions could bring us closer to the solution, but sometimes they could be misleading, just from the fact that this particular state has never been seen during the training. Don't forget, there are 4.33×10^{19} of them, so even with a graphics processing unit (GPU) training speed of hundreds of thousands of states per second, and after a month of training, we will only see a tiny portion of the state space, about 0.0000005%. So, a more sophisticated approach has to be used.

There is a family of very popular methods, called Monte Carlo tree search (MCTS), and one of these methods was covered in the last chapter. There are lots of variants of those methods, but the overall idea is simple and can be described in comparison with the well-known brute-force search methods, like breadth-first search (BFS) or depth-first search (DFS). In BFS and DFS, we perform the exhaustive search of our state space by trying all the possible actions and exploring all the states that we get from those actions. That behavior is the other extreme of the procedure described previously (when we have something that tells us where to go at every state).

MCTS offers something in between those extremes: we want to perform the search and we have some information about where we should go, but this information could be unreliable, noisy, or just wrong in some situations. However, sometimes, this information could show us the promising directions that could speed up the search process.

As I've mentioned, MCTS is a family of methods and they vary in their particular details and characteristics. In the paper, the method called Upper Confidence Bound 1 is used. The method operates on the tree, where the nodes are the states and the edges are actions connecting those states. The whole tree is enormous in most cases (4.33×10^{19} for our cube), so we can't try to build the whole tree, just some tiny portion of it.

In the beginning, we start with a tree consisting of a single node, which is our current state. On every step of the MCTS, we walk down the tree exploring some path in the tree, and there are two options we can face:

- Our current node is a leaf node (we haven't explored this direction yet)
- Our current node is in the middle of the tree and has children

In the case of a leaf node, we "expand" it by applying all the possible actions to the state. All the resulting states are checked for being the goal state (if the goal state of the solved cube has been found, our search is done).

The leaf state is passed to the model and the outputs from both the value and policy heads are stored for later use.

If the node is not the leaf, we know about its children (reachable states), and we have value and policy outputs from the network. So, we need to make the decision about which path to follow (in other words, which action is more promising to explore). This decision is not a trivial one and this is the exploration versus exploitation problem that we have covered previously in this book. On the one hand, our policy from the network says what to do. But what if it is wrong? This could be solved by exploring surrounding states, but we don't want to explore all the time (as the state space is enormous). So, we should keep the balance, and this has a direct influence on the performance and the outcome of the search process.

To solve this, for every state, we keep the counter for every possible action (there are 12 of them), which is incremented every time the action has been chosen during the search. To make the decision to follow a particular action, we use this counter; the more an action has been taken, the less likely it is to be chosen in the future.

In addition, the value returned by the model is also used in this decision-making. The value is tracked as the maximum from the current state's value and the value from its children. This allows the most promising paths (from the model perspective) to be seen from the parent's states.

To summarize, the action to follow from a non-leaf tree is chosen by using the following formula:

$$A_t = \text{argmax}_a(U_{s_t}(a) + W_{s_t}(a)) \text{, where } U_{s_t}(a) = cP_{s_t}(a)\frac{\sqrt{\sum_{a'} N_{s_t}(a')}}{1 + N_{s_t}(a)}$$

Here, $N_{s_t}(a)$ is a count of times that action a has been chosen in state s_t.

$P_{s_t}(a)$ is the policy returned by the model for state s_t and $W_{s_t}(a)$ is the maximum value returned by the model for all children states of s_t under the branch a.

This procedure is repeated until the solution has been found or our time budget has been exhausted. To speed up the process, MCTS is very frequently implemented in a parallel way, where several searches are performed by multiple threads. In that case, some extra loss could be subtracted from A_t to prevent multiple threads from exploring the same paths of the tree.

The final piece in solving the process picture is how to get the solution from the MCTS tree once we have reached the goal state. The authors of the paper experimented with two approaches:

- **Naïve**: Once we have faced the goal state, we use our path from the root state as the solution
- **The BFS way**: After reaching the goal state, BFS is performed on the MCTS tree to find the shortest path from the root to this state

According to the authors, the second method finds shorter solutions than the naïve version, which is not surprising, as the stochastic nature of the MCTS process can introduce cycles to the solution path.

The paper's results

The final result published in the paper is quite impressive. After 44 hours of training on a machine with three GPUs, the network learned how to solve cubes at the same level as (and sometimes better than) human-crafted solvers. The final model has been compared against the two solvers described earlier: the Kociemba two-stage solver and Korf. The method proposed in the paper is named **DeepCube**.

To compare efficiency, 640 randomly scrambled cubes were used in all the methods. The depth of the scramble was 1,000 moves. The time limit for the solution was an hour and both the DeepCube and Kociemba solvers were able to solve all of the cubes within the limit. The Kociemba solver is very fast, and its median solution time is just one second, but due to the hardcoded rules implemented in the method, its solutions are not always the shortest ones.

The DeepCube method took much more time, with the median time being about 10 minutes, but it was able to match the length of the Kociemba solutions or do better in 55% of cases. From my personal perspective, 55% is not enough to say that NNs are significantly better, but at least they are not worse.

On the figure that follows, taken from the paper, the length distributions for all the solvers are shown. As you can see, the Korf solver wasn't compared in 1,000 scramble test cases, due to the very long time needed to solve the cube. To compare the performance of DeepCube against the Korf solver, a much easier 15-step scramble test set was created.
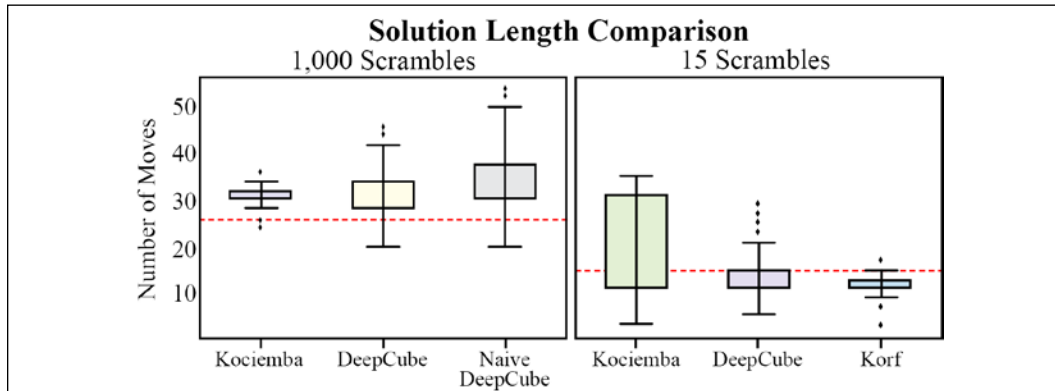
Figure 24.4: The length of solutions found by various solvers

The code outline

Okay, now let's switch to the code, which is in directory `Chapter24` in the book's repository. In this section, I'm going to give a quick outline of my implementation and the key design decisions, but before that, I have to emphasize the important points about the code to set up the correct expectations:

- I'm not a researcher, so the original goal of this code was just to reimplement the paper's method. Unfortunately, the paper has very few details about the exact hyperparameters used, so I had to guess and experiment a lot, and still, my results are very different from those published in the paper.

- At the same time, I've tried to implement everything in a general way to simplify further experiments. For example, the exact details about the cube state and transformations are abstracted away, which allows us to implement more puzzles similar to the 3×3 cube just by adding a new module. In my code, two cubes are implemented: 2×2 and 3×3, but any fully observable environment with a fixed set of predictable actions can be implemented and experimented with. The details are given in the next subsection.

- Code clarity and simplicity was put ahead of performance. Of course, when it was possible to improve performance without introducing much overhead, I did so. For example, the training process was sped up by a factor of five by just splitting the generation of the scrambled cubes and the forward network pass. But if the performance required refactoring everything into multi-GPU and multithreaded mode, I preferred to keep things simple. A very vivid example is the MCTS process, which is normally implemented as multithreaded code sharing the tree. It usually gets sped up several times, but requires tricky synchronization between processes. So, my version of MCTS is serial, with only trivial optimization of the batched search.

Overall, the code consists of the following parts:

1. The cube environment, which defines the observation space, the possible actions, and the exact representation of the state to the network. This part is implemented in the `libcube/cubes` module.

2. The NN part, which describes the model that we will train, the generation of training samples, and the training loop. It includes the training tool `train.py` and the module `libcube/model.py`.

3. The solver of cubes, including the `solver.py` utility and the module `libcube/mcts.py`, which implements MCTS.

4. Various tools used to glue up other parts, like configuration files with hyperparameters and tools used to generate cube problem sets.

Cube environments

As you have already seen, combinatorial optimization problems are quite large and diverse. Even the narrow area of cube-like puzzles includes a couple of dozen variations. The most popular ones are 2×2×2, 3×3×3, and 4×4×4 Rubik's Cubes, Square-1, Pyraminx, and tons of others (`https://ruwix.com/twisty-puzzles/`). At the same time, the method presented in the paper is quite general and doesn't depend on prior domain knowledge, the amount of actions, and the state space size. The critical assumptions imposed on the problem include:

- States of the environment need to be fully observable and observations need to distinguish states from each other. That's the case for the cube when we can see all the sides' states, but it doesn't hold true for most variants of poker, for example, when we can't see the cards of our opponent.

- The amount of actions needs to be discrete and finite. There are limited amounts of actions we can take with the cube, but if our action space is "rotate the steering wheel on angle $\alpha \in [-120° \dots 120°]$," we have a different problem domain here, as you have already seen in chapters devoted to the continuous control problems.

- We need to have a reliable model of the environment; in other words, we have to be able to answer questions like "what will be the result of applying action a_i to the state s_j?" Without this, both ADI and MCTS become non-applicable. This is a strong requirement and, for most of the problems, we don't have such a model or its outputs are quite noisy. On the other hand, in games like chess or Go, we have such a model: the rules of the game.

- In addition, our domain is deterministic, as the same action applied to the same state always ends up in the same final state. The counter example might be backgammon, when, on each turn, players roll the dice to get the amount of moves they can possibly make. I have a feeling that the methods should work even when we have stochastic actions, but I might be wrong.

To simplify the application of the methods to domains different from the 3×3 cube, all concrete environment details are moved to separate modules, communicating with the rest of the code via the abstract interface `CubeEnv`, which is described in the `libcube/cubes/_env.py` module. Its interface is shown and commented in the following code block.

```
class CubeEnv:
    def __init__(self, name, state_type, initial_state,
                 is_goal_pred, action_enum, transform_func,
                 inverse_action_func, render_func, encoded_shape,
                 encode_func):
        self.name = name
        self._state_type = state_type
        self.initial_state = initial_state
        self._is_goal_pred = is_goal_pred
        self.action_enum = action_enum
        self._transform_func = transform_func
        self._inverse_action_func = inverse_action_func
        self._render_func = render_func
        self.encoded_shape = encoded_shape
        self._encode_func = encode_func
```

The constructor of the class takes a bunch of arguments:

- The name of the environment.
- The type of the environment state.

- The instance of the initial (assembled) state of the cube.
- The predicate function to check that a particular state represents the assembled cube. For 3×3 cubes, this might look like an overhead, as we possibly could just compare this with the initial state passed in the `initial_state` argument, but cubes of size 2×2 and 4×4, for instance, might have multiple final states, so a separate predicate is needed to cover such cases.
- The enumeration of actions that we can apply to the state.
- The transformation function, which takes the state and the action and returns the resulting state.
- The inverse function, which maps every action into its inverse.
- The render function to represent the state in human-readable form.
- The shape of the encoded state tensor.
- The function to encode the compact state representation into an NN-friendly form.

As you can see, cube environments are not compatible with the Gym API, which I did intentionally to illustrate how you can step beyond Gym.

```
def __repr__(self):
    return "CubeEnv(%r)" % self.name

def is_goal(self, state):
    return self._is_goal_pred(state)

def transform(self, state, action):
    return self._transform_func(state, action)

def inverse_action(self, action):
    return self._inverse_action_func(action)

def render(self, state):
    return self._render_func(state)

def is_state(self, state):
    return isinstance(state, self._state_type)

def encode_inplace(self, target, state):
    return self._encode_func(target, state)
```

Some of the methods in the `CubeEnv` API are just wrappers around functions passed to the constructor. This allows the new environment to be implemented in a separate module, register itself in the environment registry, and provide the consistent interface to the rest of the code.

All the other methods in the class provide extended uniform functionality based on those primitive operations.

```python
def sample_action(self, prev_action=None):
    while True:
        res = self.action_enum(
            random.randrange(len(self.action_enum)))
        if prev_action is None or \
                self.inverse_action(res) != prev_action:
            return res
```

The preceding method provides the functionality of randomly sampling the random action. If the `prev_action` argument is passed, we exclude the reverse action from possible results, which is handy to avoid a generation of short loops, like *Rr* or *Ll*, for example.

```python
def scramble(self, actions):
    s = self.initial_state
    for action in actions:
        s = self.transform(s, action)
    return s
```

The method `scramble()` applies the list of actions to the initial state of the cube, returning the final state.

```python
def scramble_cube(self, scrambles_count, return_inverse=False,
                  include_initial=False):
    state = self.initial_state
    result = []
    if include_initial:
        assert not return_inverse
        result.append((1, state))
    prev_action = None
    for depth in range(scrambles_count):
        action = self.sample_action(prev_action=prev_action)
        state = self.transform(state, action)
        prev_action = action
        if return_inverse:
            inv_action = self.inverse_action(action)
            res = (depth+1, state, inv_action)
        else:
            res = (depth+1, state)
        result.append(res)
    return result
```

The preceding method is a bit lengthy and provides the functionality of randomly scrambling the cube, returning all the intermediate states. In the case of the `return_inverse` argument being `False`, the function returns the list of tuples with `(depth, state)` for every step of the scrambling process. If the argument is `True`, it returns a tuple with three values: `(depth, state, inv_action)`, which are needed in some situations.

```
def explore_state(self, state):
    res_states, res_flags = [], []
    for action in self.action_enum:
        new_state = self.transform(state, action)
        is_init = self.is_goal(new_state)
        res_states.append(new_state)
        res_flags.append(is_init)
    return res_states, res_flags
```

The method `explore_states` implements functionality for ADI and applies all the possible actions to the given cube state. The result is a tuple of lists in which the first list contains the expanded states, and the second has flags of those states as the goal state.

With this generic functionality, a similar environment might be implemented and plugged into the existing training and testing methods with very little boilerplate code. As an example, I have provided both the 2×2×2 cube and 3×3×3 cube that I used in my experiments. Their internals reside in `libcube/cubes/cube2x2.py` and `libcube/cubes/cube3x3.py`, which you can use as a base to implement your own environments of this kind.

Every environment needs to register itself by creating the instance of the `CubeEnv` class and passing the instance into the function `register()`, defined in `libcube/cubes/_env.py`. The following is the piece of code from the `cube2x2.py` module.

```
_env.register(_env.CubeEnv(
    name="cube2x2", state_type=State, initial_state=initial_state,
    is_goal_pred=is_initial, action_enum=Action,
    transform_func=transform, inverse_action_func=inverse_action,
    render_func=render, encoded_shape=encoded_shape,
    encode_func=encode_inplace))
```

Once this is done, the cube environment can be obtained by using the `libcube.cubes.get()` method, which takes the environment name as an argument. The rest of the code uses only the public interface of the `CubeEnv` class, which makes the code cube-type agnostic and simplifies the extensibility.

Training

The training process is implemented in the tool `train.py` and the module `libcube/model.py`, and it is a straightforward implementation of the training process described in the paper, with one difference: the code supports two methods of calculating the target values for the value head of the network. One of the methods is exactly how it was described in the paper and the other is my modification, which I'll explain in the details of the subsequent section.

To simplify the experimentation and make the results reproducible, all the parameters of the training are specified in a separate `.ini` file, which gives the following options for training:

- The name of the environment to be used; currently `cube2x2` and `cube3x3` are available.
- The name of the run, which is used in TensorBoard names and directories to save models.
- What target value calculation method in ADI will be used. I implemented two of them: one is described in the paper and then there is my modification, which, from my experiments, has more stable convergence.
- The training parameters: the batch size, the usage of CUDA, the learning rate, the learning rate decay, and others.

You can find the examples of my experiments in the `ini` folder in the repo. During the training, TensorBoard metrics of the parameters are written in the `runs` folder. Models with the best loss value are saved in the `saves` directory.

To give you an idea of what the configuration file looks like, the following is `ini/cube2x2-paper-d200.ini`, which defines the experiment for a 2×2 cube, using the value calculation method from the paper and a scramble depth of 200:

```
[general]
cube_type=cube2x2
run_name=paper

[train]
cuda=True
lr=1e-5
batch_size=10000
scramble_depth=200
report_batches=10
checkpoint_batches=100
lr_decay=True
lr_decay_gamma=0.95
lr_decay_batches=1000
```

To start the training, you need to pass the `.ini` file to the `train.py` utility; for example, this is how the preceding `.ini` file could be used to train the model.

```
$ ./train.py -i ini/cube2x2-paper-d200.ini -n t1
```

The extra argument `-n` gives the name of the run, which will be combined with the name in the `.ini` file to be used as the name of a TensorBoard series.

The search process

The result of the training is a model file with the network's weights. The file could be used to solve cubes using MCTS, which is implemented in the tool `solver.py` and the module `libcube/mcts.py`.

The solver tool is quite flexible and could be used in various modes:

1. To solve a single scrambled cube given as a comma-separated list of action indices, passed in the `-p` option. For example `-p 1,6,1` is a cube scrambled by applying the second action, then the seventh action, and finally, the second action again. The concrete meaning of the actions is environment-specific, which is passed with the `-e` option. You can find actions with their indices in the cube environment module. For example, the actions `1,6,1` for a 2×2 cube mean an *L,R',L* transformation.

2. To read permutations from a text file (one cube per line) and solve them. The file name is passed with the `-i` option. There are several sample problems available in the folder `cubes_tests`. You can generate your own random problem sets using the `gen_cubes_py` tool, which allows you to set the random seed, the depth of the scramble, and other options.

3. To generate a random scramble of the given depth and solve it.

4. To run a series of tests with increasing complexity (scramble depth), solve them, and write a CSV file with the result. This mode is enabled by passing the `-o` option and is very useful for evaluating the quality of the trained model, but it can take lots of time to complete. Optionally, plots with those test results are produced.

In all cases, you need to pass the environment name with the `-e` option and the file with the model's weights (`-m` option). In addition, there are other parameters, allowing you to tweak MCTS options and time or search step limits. You can find the names of those options in the code of `solver.py`.

The experiment results

Unfortunately, the paper provided no details about very important aspects of the method, like training hyperparameters, how deeply cubes were scrambled during the training, and the obtained convergence. To fill in the missing blanks, I did lots of experiments with various values of hyperparameters, but still my results are very different from those published in the paper. First of all, the training convergence of the original method is very unstable. Even with a small learning rate and a large batch size, the training eventually diverges, with the value loss component growing exponentially. Examples of this behavior are shown on the figure that follows.
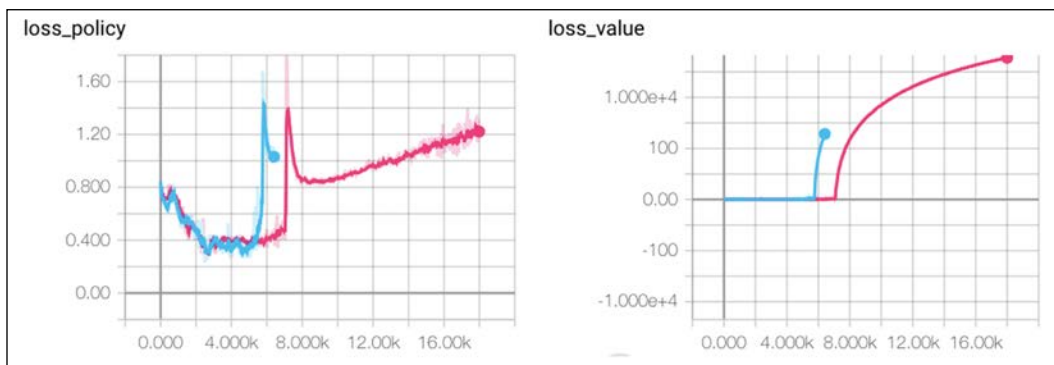
Figure 24.5: The policy loss (left) and value loss (right) of two runs of the paper's method

After several experiments with this, I came to the conclusion that this behavior is a result of the wrong value objective being proposed in the method. Indeed, in the formula $y_{v_i} = \max_a(v_s(a) + R(A(s, a)))$, the value $v_s(a)$ returned by the network is always added to the actual reward, $R(s)$, even for the goal state. With this, the actual values returned by the network could be anything: -100, 10^6, or 3.1415. This is not a great situation for NN training, especially with the mean squared error (MSE) objective.

To check this, I modified the method of the target value calculation by assigning zero target for the goal state:

$$
y_{v_i} = \begin{cases} \max_a \left(v_s(a) + R\big(A(s, a)\big) \right), & \text{if } s \text{ is not the goal} \\ 0, & \text{if } s \text{ is a goal state} \end{cases}
$$

This target could be enabled in the `.ini` file by specifying the parameter `value_targets_method` to be `zero_goal_value`, instead of the default `value_targets_method=paper`.

With this simple modification, the training process converged much quicker to stable values returned by the value head of the network. An example of convergence is shown on the following figures.
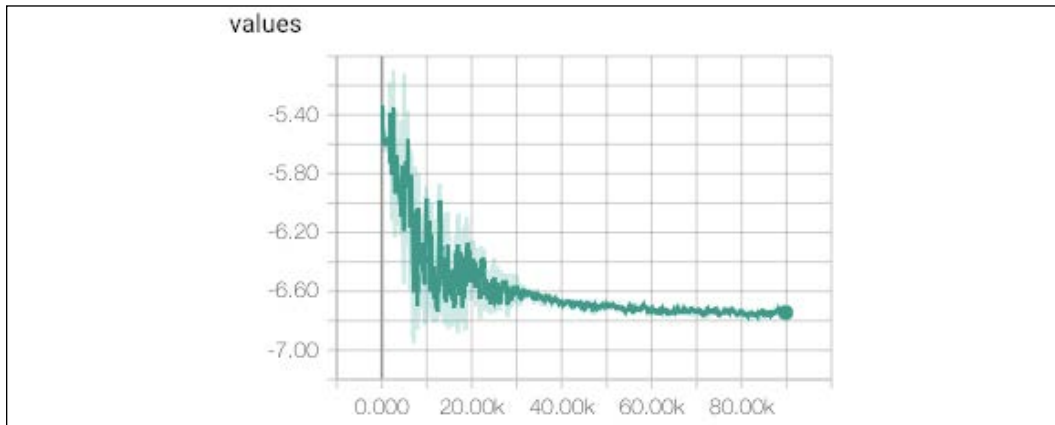
Figure 24.6: Values predicted by the value head during training
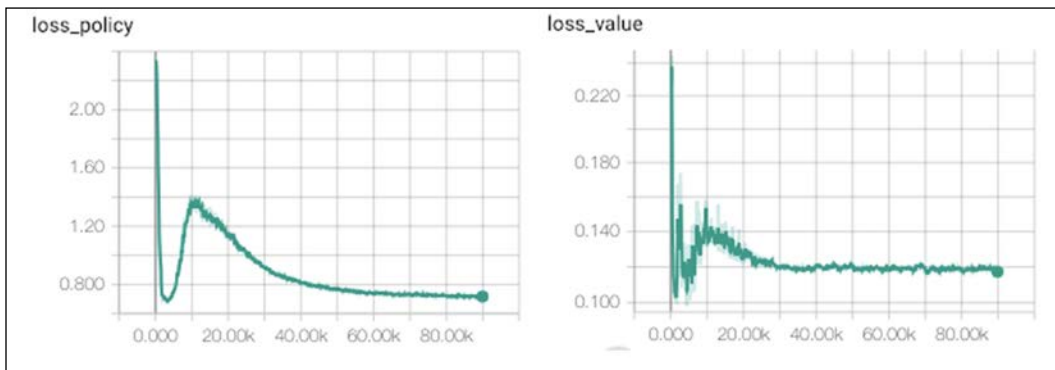
Figure 24.7: The policy loss (left) and value loss (right) after modifications

The 2×2 cube

In the paper, the authors reported training for 44 hours on a machine with three Titan Xp GPUs. During the training, their model saw 8 billion cube states. Those numbers correspond to the training speed ~50,000 cubes/second. My implementation shows 15,000 cubes/second on a single GTX 1080 Ti, which is comparable. So, to repeat the training process on a single GPU, we need to wait for almost six days, which is not very practical for experimentation and hyperparameter tuning.

To overcome this, I implemented a much simpler 2×2 cube environment, which takes just an hour to train. To reproduce my training, there are two `.ini` files in the repo:

- `ini/cube2x2-paper-d200.ini`: This uses the value target method described in the paper
- `ini/cube2x2-zero-goal-d200.ini`: The value target is set to zero for goal states

Both configurations use batches of 10k states and a scramble depth of 200, and the training parameters are the same. After the training, using both configurations, two models were produced (both are stored in the repo in the `models` directory):

- The paper's method: loss 0.18184
- The zero goal method: loss 0.014547

My experiments (using the `solver.py` utility) have shown that the model with a lower loss has a much higher success ratio of solving randomly scrambled cubes of increasing depth. The results for both models are shown on the figure that follows.
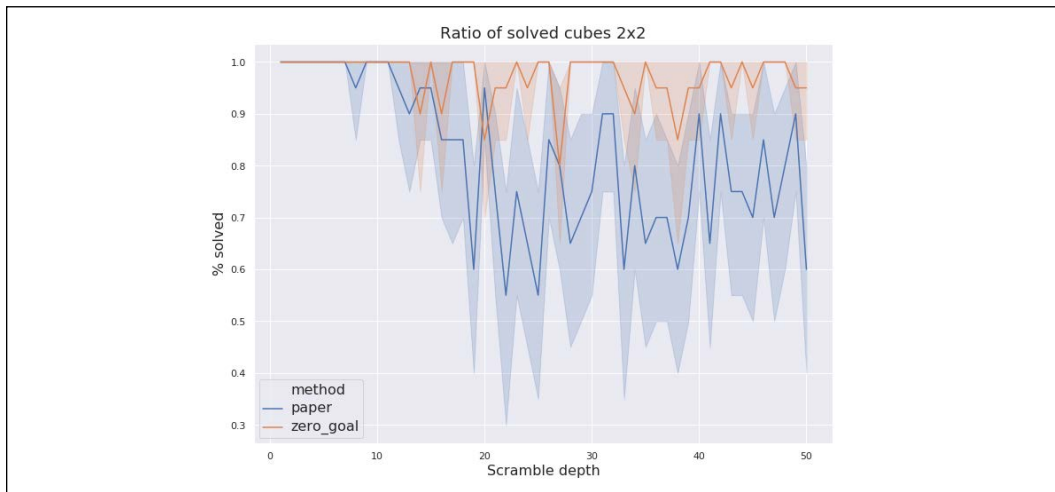
Figure 24.8: The ratio of solved cubes of various scramble depths

The next parameter to compare is the number of MCTS steps done during the solution. It is shown on the figure that follows. The zero goal model normally finds a solution in fewer steps. In other words, the policy it has learned is better.
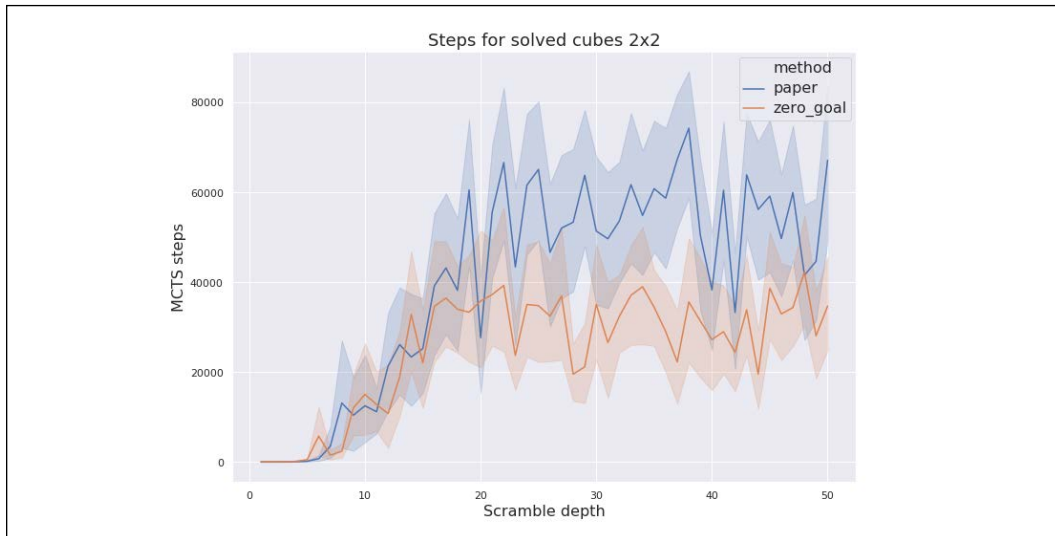
Figure 24.9: The number of MCTS steps for solved cubes of various depths

Finally, let's check the length of the found solutions. On the figure that follows, both the naïve and BFS solution lengths are plotted. From those plots, it can be seen that the naïve solutions are much longer (by a factor of 10) than solutions found by BFS. Such a difference might be an indication of untuned MCTS parameters, which could be improved. The zero goal model shows longer solutions than the paper's model, as the former fails to find longer solutions at all.
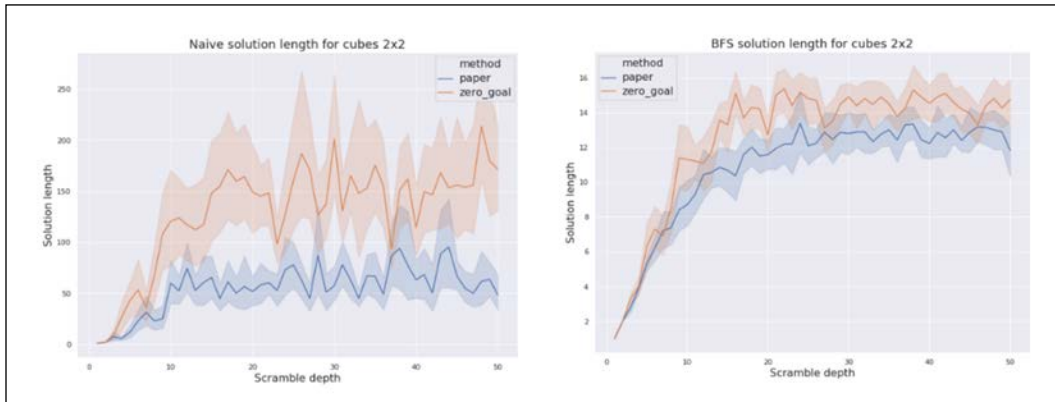
Figure 24.10: A comparison of naïve (left) and BFS (right) ways of solution generation

The 3×3 cube

The training of the 3×3 cube model is much more heavy, so I've probably just scratched the surface here. But even my limited experiments show that zero goal modifications to the training method greatly improve the training stability and resulting model quality. Training requires about 20 hours, so running lots of experiments requires time and patience.

My results are not as shiny as those reported in the paper: the best model I was able to obtain can solve cubes up to a scrambling depth of 12…15, but consistently fails at more complicated problems. Probably, those numbers could be improved with more central processing unit (CPU) cores plus parallel MCTS. To get the data, the search process was limited to 10 minutes and, for every scramble depth, five random scrambles were generated.
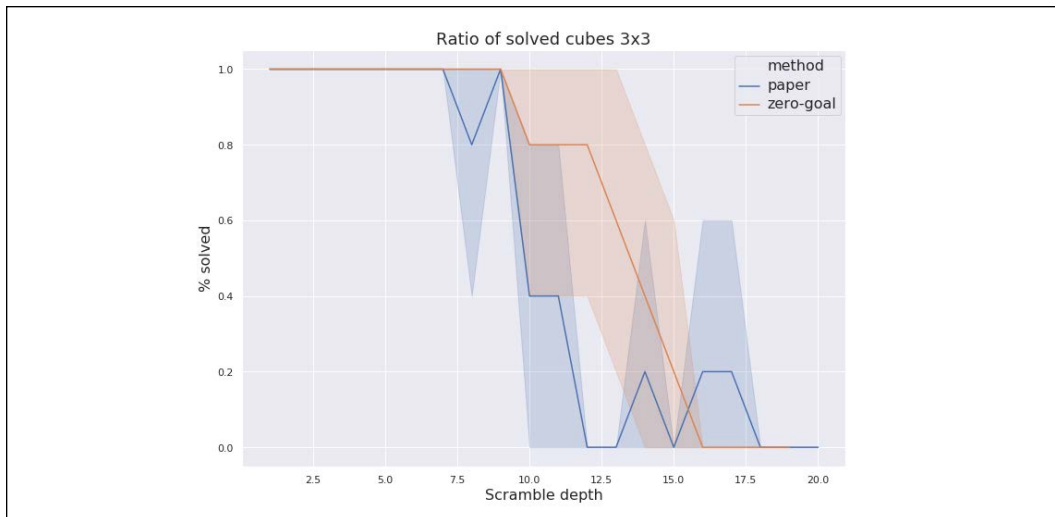
Figure 24.11: The ratio of solved 3×3 cubes by both methods

The preceding figure shows a comparison of solution rates for the method presented in the paper and the modified version with the zero value target.
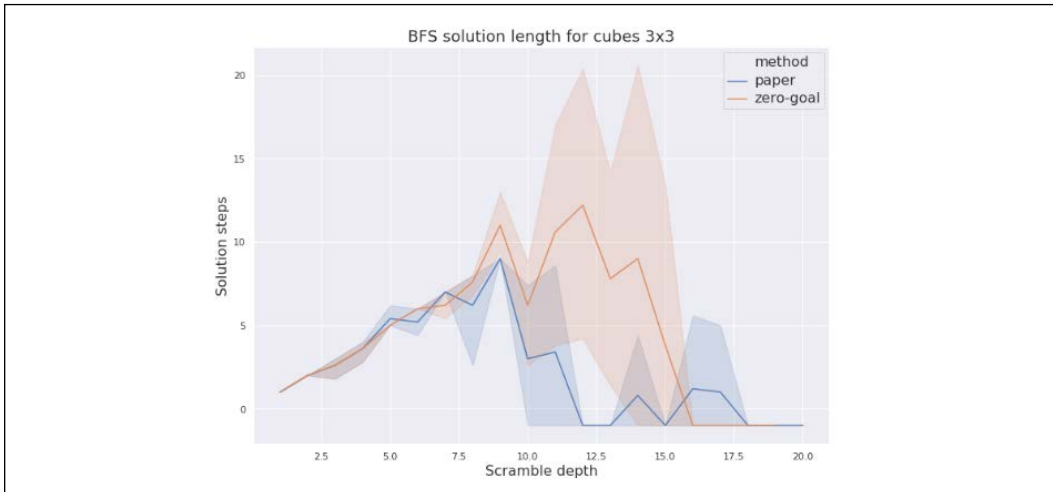
Figure 24.12: Solution lengths for both methods on a 3×3 cube

The final figure shows the length of the optimal solution found. There are two interesting issues that could be noted here. First of all, the length of solutions found by the zero target method in the 10…15 scramble depth range is *larger* than the scramble depth. This means that the model failed to find the scramble sequence used to generate the test problem, but still discovered some longer way to the goal state. Another observation is that for the depth range 12…18, the paper's method found solutions that were *shorter* than the scramble sequence. This could be due to a degenerative test sequence that was generated.

Further improvements and experiments

There are lots of directions and things that could be tried:

- More input and network engineering: the cube is a complicated thing, so simple feed-forward NNs may not be the best model. Probably, the network could greatly benefit from convolutions.

- Oscillations and instability during training might be a sign of a common RL issue with inter-step correlations. The usual approach is the *target network*, when we use the old version of the network to get bootstrapped values.

- The priority replay buffer might help the training speed.
- My experiments show that the samples' weighting (inversely proportional to the scramble depth) helps to get a better policy that knows how to solve slightly scrambled cubes, but might slow down the learning of deeper states. Probably, this weighting could be made adaptive to make it less aggressive in later training stages.
- Entropy loss could be added to the training to regularize our policy.
- The 2×2 cube model doesn't take into account the fact that the cube doesn't have central cubelets, so the whole cube could be rotated. This might not be very important for a 2×2 cube, as the state space is small, but the same observation will be critical for 4×4 cubes.
- More experiments are needed to get better training and MCTS parameters.

Summary

In this chapter, I described the application of RL methods to discrete optimization problems, using the Rubik's Cube as a well-known, but still challenging, problem. In the final chapter of the book, we will talk about *multi-agent problems in RL*.

25
Multi-agent RL

In the last chapter, we dived into discrete optimization problems. In this final chapter, we will discuss a relatively new direction of reinforcement learning (RL) and deep RL, which is related to the situations when multiple agents communicate in an environment.

In this chapter, we will:

- Start with an overview of the similarities and differences between the classical single-agent RL problem and multi-agent RL
- Cover the MAgent environment, which was implemented and open sourced by the Geek.AI UK/China research group
- Use MAgent to train models in different environments with several groups of agents

Multi-agent RL explained

The multi-agent setup is a natural extension of the familiar RL model that we covered in *Chapter 1*, *What Is Reinforcement Learning?*, In the normal RL setup, we have one agent communicating with the environment using the observation, reward, and actions. But in some problems, which often arise in reality, we have several agents involved in the environment interaction. To give some concrete examples:

- A chess game, when our program tries to beat the opponent
- A market simulation, like product advertisements or price changes, when our actions might lead to counter-actions from other participants
- Multiplayer games, like Dota2 or StarCraft II, when the agent needs to control several units competing with other players' units

If other agents are outside of our control, we can treat them as part of the environment and still stick to the normal RL model with the single agent. But sometimes, that's too limited and not exactly what we want. As you saw in *Chapter 23*, *AlphaGo Zero*, training via self-play is a very powerful technique, which might lead to good policies without much sophistication on the environment side.

In addition, as recent research shows, a group of simple agents might demonstrate collaborative behavior that is way more complex than expected. One such example is the OpenAI blog post (`https://openai.com/blog/emergent-tool-use/`) and paper (`https://arxiv.org/abs/1909.07528`) about the "hide-and-seek" game, when the group of agents collaborate and develop more and more sophisticated strategies and counter-strategies to win against another group of agents, for example "build a fence from available objects" and "use a trampoline to catch agents behind the fence."

Forms of communication

In terms of different ways that agents might communicate, they can be separated into two groups:

- Competitive: when two or more agents try to beat each other in order to maximize their reward. The simplest setup is a two-player game, like chess, backgammon or Atari Pong.
- Collaboration: when a group of agents needs to use joint efforts to reach some goal.

There are lots of examples that fall into one of the groups, but the most interesting and close to real-life scenarios normally are a mixture of both behaviors. There are tons of examples, starting from some board games that allow you to form allies, and going up to modern corporations, where 100% collaboration is assumed, but real life is normally much more complicated than that.

From the theoretical point of view, *game theory* has quite a developed foundation for both communication forms, but for the sake of brevity, I'm not going to deep dive into the field, which is large and has different terminology. The curious reader can find lots of books and courses about it. To give an example, the minimax algorithm is a well-known result of game theory, and you saw it modified in *Chapter 23*, *AlphaGo Zero*.

The RL approach

Multi-agent RL (sometimes abbreviated to MARL) is a very young field, but activity has been growing over time, so, it might be interesting to keep an eye on it. For instance, the recent interest of DeepMind and OpenAI in multiplayer strategy games (StarCraft II and Dota2) is an obvious step in this direction.

In this chapter, we will just take a quick glance at MARL and experiment a bit with simple environments, but, of course, if you find it interesting, there are lots of things you can experiment with. In our experiments, we will use a simple and straightforward approach, with agents sharing the policy that we are optimizing, but the observation will be given from the agent's standpoint and include information about the other agent's location. With that simplification, our RL methods will stay the same, and just the environment will require preprocessing and must handle the presence of multiple agents.

The MAgent environment

Before we jump into our first MARL example, I will describe our environment to experiment with.

Installation

If you want to play with MARL, your choice is a bit limited. All the environments that come with Gym support only one agent. There are some patches for Atari Pong, to switch it into two-player mode, but they are not standard and are an exception rather than the rule.

DeepMind, together with Blizzard, has made StarCraft II publicly available (`https://github.com/deepmind/pysc2`) and it makes for a very interesting and challenging environment for experimentation. However, for somebody who is taking their first steps in MARL, it might be too complex. In that regard, I found the MAgent environment from Geek.AI (`https://github.com/geek-ai/MAgent`) perfectly suitable: it is simple, fast, and has minimal dependency, but it still allows you to simulate different multi-agent scenarios for experimentation. It doesn't provide a Gym-compatible API, but we will implement it on our own.

All the examples in this chapter expect you to have MAgent cloned and installed in the `Chapter25/MAgent` directory. MAgent doesn't provide a Python package yet, so, to install it, you need to take the following steps:

- Run `git clone https://github.com/geek-ai/MAgent.git` in the `Chapter25` directory
- Build MAgent by running `bash build.sh` in the `Chapter25/MAgent` directory (you probably will need to install dependencies listed in `https://github.com/geek-ai/MAgent`)

An overview

The high-level concepts of MAgent are simple and efficient. It provides the simulation of a grid world that 2D agents inhabit. They can observe things around them (according to their perception length), move to some distance from their location, and attack other agents around them.

There might be different groups of agents with various characteristics and interaction parameters. For example, the first environment that we will consider is a predator-prey model, where "tigers" hunt "deer" and obtain reward for that. In the environment configuration, you can specify lots of aspects of the group, like perception, movement, attack distance, the initial health of every agent in the group, how much health they spend on movement and attack, and so on. Aside from the agents, the environment might contain walls, which are not crossable by the agents.

The nice thing about MAgent is that it is very scalable, as it is implemented in C++ internally, just exposing the Python interface. This means that the environment can have thousands of agents in the group, providing you with observations and processing the agents' actions.

A random environment

To quickly understand the MAgent API and logic, I've implemented a simple environment with "tiger" and "deer" agents, where both groups are driven by the random policy. It might not be very interesting from RL's perspective, but it will allow us to quickly learn the API to implement the Gym environment wrapper.

The example is in `Chapter25/forest_random.py` and it is short enough to be shown here in full.

```
import os
import sys
sys.path.append(os.path.join(os.getcwd(), "MAgent/python"))
```

As MAgent is not installed as a package, we need to tweak Python's import path before importing it.

```
import magent
from magent.builtin.rule_model import RandomActor

MAP_SIZE = 64
```

We import the main package provided by MAgent and the random actor class. In addition, we define the size of our environment, which is a 64×64 grid.

```
if __name__ == "__main__":
```

```
env = magent.GridWorld("forest", map_size=MAP_SIZE)
env.set_render_dir("render")
```

First of all, we create the environment, which is represented by the `GridWorld` class. In its constructor, we can pass various configuration parameters, tweaking the world's configuration. The first argument, `"forest"`, selects one of the predefined worlds in the MAgent code base, which is fully described in this source file: `https://github.com/geek-ai/MAgent/blob/master/python/magent/builtin/config/forest.py`. This configuration describes two groups of animals: tigers and deer. Tigers can attack deer, obtaining health points and rewards, but they spend energy that they need to replenish. Deer are not aggressive and have worse vision than tigers, but they recover their health over time (by eating grass, I suppose).

There are predefined configurations that are shipped with MAgent and you can always define your own. In the case of custom configuration, the `Config` class instance needs to be passed in the constructor.

The call `env.set_render_dir()` specifies the directory, which will be used to store the game's progress. This progress could be loaded by a separate utility, and we will take a look at how to do this later.

```
deer_handle, tiger_handle = env.get_handles()
models = [
    RandomActor(env, deer_handle),
    RandomActor(env, tiger_handle),
]
```

Once the environment is created, we need to get so-called "group handles" to provide us with access to the agent groups. For efficiency, all interactions with the environment are performed on a group level, rather than individually. In the preceding code, we get the handles of our groups and create two instances of the `RandomActor` class, which will select random actions for every group.

```
env.reset()
env.add_walls(method="random", n=MAP_SIZE * MAP_SIZE * 0.04)
env.add_agents(deer_handle, method="random", n=5)
env.add_agents(tiger_handle, method="random", n=2)
```

On the next step, we reset the environment and place key players in it. In MAgent terminology, `reset()` clears the grid completely, which is different from the Gym `reset()` call. The preceding code turns 4% of grid cells into unpassable walls by calling `env.add_walls()`, and randomly places five deer and two tigers.

```
v = env.get_view_space(tiger_handle)
r = env.get_feature_space(tiger_handle)
print("Tiger view: %s, features: %s" % (v, r))
```

```
        vv = env.get_view_space(deer_handle)
        rr = env.get_feature_space(deer_handle)
        print("Deer view: %s, features: %s" % (vv, rr))
```

Now, let's explore our observations a bit. In MAgent, the observation of every agent is divided into two parts: view space and feature space. View space is spatial and represents various information about the grid cells around the agent. If we run the code, it will output those lines with observation shapes:

Tiger view: (9, 9, 5), features: (20,)

Deer view: (3, 3, 5), features: (16,)

This means that every tiger gets a 9×9 matrix with five different planes of information. Deer are more short-sighted, so their observation is just 3×3. The observation always contains the agent in the center, so it shows the grid around this specific agent. The five planes of information include:

- Walls: 1 if this cell contains the wall and 0 otherwise
- Group 1 (the agent belongs to this): 1 if the cell contains agents from the agent's group and 0 otherwise
- Group 1 health: The relative health of the agent in this cell
- Group 2 enemy agents: 1 if there is an enemy in this cell
- Group 2 health: The relative health of the enemy or 0 if nothing is there.

If more groups are configured, the observation will contain more planes in the observation tensor. In addition, MAgent has a "minimap" functionality, which is disabled by default, that includes the "zoomed out" location of agents of every group. Group 1 and group 2 are relative to the agent's group, so, in the second plane, deer have information about other deer and for tigers this plane includes other tigers. This allows us to train a single policy for both groups if needed.

Another part of the observation is the so-called feature space. It is not spatial and is represented as just a vector of numbers. It includes one-hot encoding of the agent's ID, last action, last reward, and normalized position. Concrete details could be found in the MAgent source code, but they are not relevant at the moment.

Let's continue our code description.

```
        done = False
        step_idx = 0
        while not done:
            deer_obs = env.get_observation(deer_handle)
            tiger_obs = env.get_observation(tiger_handle)
            if step_idx == 0:
```

```
        print("Tiger obs: %s, %s" % (
            tiger_obs[0].shape, tiger_obs[1].shape))
        print("Deer obs: %s, %s" % (
            deer_obs[0].shape, deer_obs[1].shape))
    print("%d: HP deers:  %s" % (
        step_idx, deer_obs[0][:, 1, 1, 2]))
    print("%d: HP tigers: %s" % (
        step_idx, tiger_obs[0][:, 4, 4, 2]))
```

We start the step loop, where we get observations. The code will show the following:

```
Tiger obs: (2, 9, 9, 5), (2, 20)
Deer obs: (5, 3, 3, 5), (5, 16)
0: HP deers:  [1. 1. 1. 1. 1.]
0: HP tigers: [1. 1.]
```

As we have two tigers and five deer, the call `env.get_observation()` returns a NumPy array of the corresponding shape. This makes the communication with the environment very efficient, as the observation of the whole group is obtained at once.

Health points for both groups are taken from the center of the second plane, which includes information about the agent. So, the health vector shown is the health for the group.

```
        deer_act = models[0].infer_action(deer_obs)
        tiger_act = models[1].infer_action(tiger_obs)
        env.set_action(deer_handle, deer_act)
        env.set_action(tiger_handle, tiger_act)
```

In the next lines, we will ask the model to select the actions from the observations (which are taken randomly) and pass those actions to the environment. Once again, the actions are represented as a vector of integers for the entire group. Using this, we can have very large amounts of agents under our control.

```
        env.render()
        done = env.step()
        env.clear_dead()
        t_reward = env.get_reward(tiger_handle)
        d_reward = env.get_reward(deer_handle)
        print("Rewards: deer %s, tiger %s" % (d_reward, t_reward))
        step_idx += 1
```

In the rest of the loop, we do several things. First of all, we ask the environment to save information about the agents and their location to be explored later. Then we call `env.step()` to do one time step in the grid-world simulation. This function returns a single Boolean flag, which will become `True` once all the agents are dead. Then we get the rewards vector for the group, show it, and iterate the loop again.

If you start the program, it shows something like this:

```
Tiger view: (9, 9, 5), features: (20,)
Deer view: (3, 3, 5), features: (16,)
Tiger obs: (2, 9, 9, 5), (2, 20)
Deer obs: (5, 3, 3, 5), (5, 16)
0: HP deers:  [1. 1. 1. 1. 1.]
0: HP tigers: [1. 1.]
Rewards: deer [0. 0. 0. 0. 0.], tiger [1. 1.]
1: HP deers:  [1. 1. 1. 1. 1.]
1: HP tigers: [0.95 0.95]
Rewards: deer [0. 0. 0. 0. 0.], tiger [1. 1.]
2: HP deers:  [1. 1. 1. 1. 1.]
2: HP tigers: [0.9 0.9]
...
19: HP deers:  [1. 1. 1. 1. 1.]
19: HP tigers: [0.05 0.05]
Rewards: deer [0. 0. 0. 0. 0.], tiger [1. 1.]
20: HP deers:  [1. 1. 1. 1. 1.]
20: HP tigers: [0. 0.]
Rewards: deer [0. 0. 0. 0. 0.], tiger []
```

So, as you can see, tigers have their health decreased on every time step and after just 20 steps, all tigers are dead from starvation. As their actions are random, it is highly unlikely that they will eat a deer in one episode to live longer.

After the simulation completion, it stores the trace information in `render` directory, which is represented with two files, `config.json` and `video_1.txt`. To visualize them, you need to start the special program, which was built during the MAgent installation. It will act as a server for an HTML page, which will display the environment's progress. To start it, you need to run the `MAgent/build/render/render` tool, as shown in the following command:

```
Chapter25$ MAgent/build/render/render
[2019-10-20 19:47:38] [0x1076a65c0] Listening on port 9030
```

After that, you need to load with your browser the HTML file from the MAgent source to display the recorded information: `Chapter25/MAgent/build/render/index.html`. It will ask you for `config.json` and `video_1.txt` location, as shown in *Figure 25.1*:
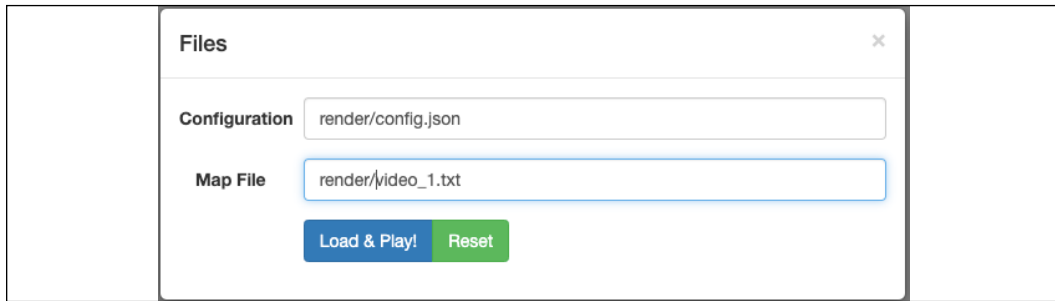
Figure 25.1: The MAgent render interface in browser

After you press the **Load & Play!** button, the recording will be loaded and the animation of the grid world shown. In our case, tigers and deer are moving randomly for 20 steps. The interface is shown in *Figure 25.2*.
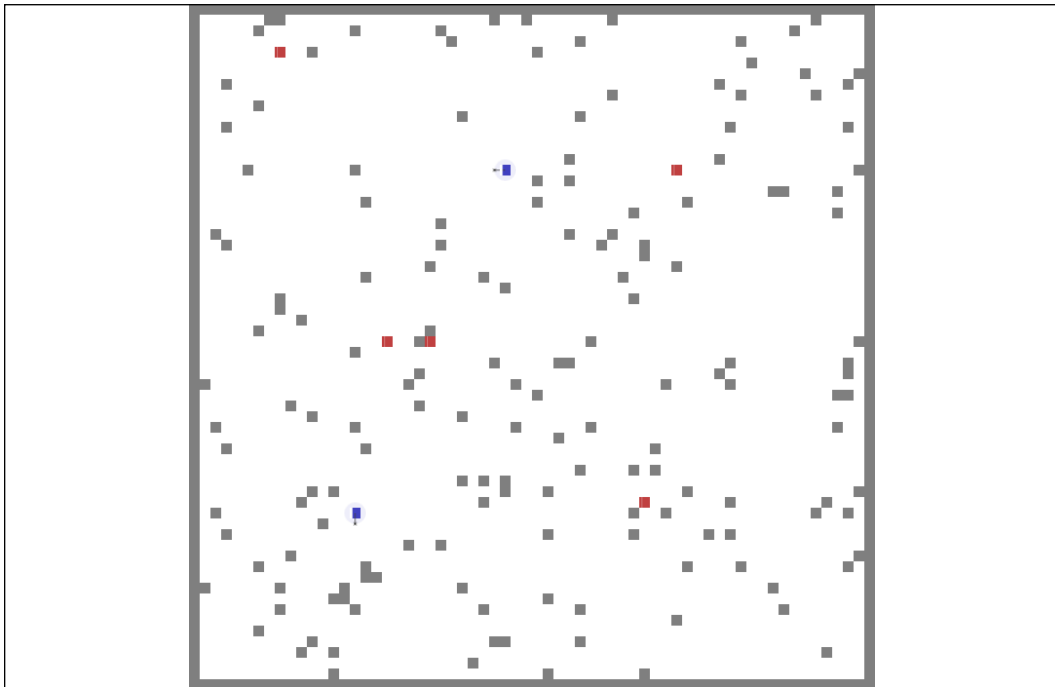
Figure 25.2: The grid world of our example

The rendering interface has hotkeys, which are displayed after the "h" button press. Using them, you can change the files, pause things, and show dialog with speed and progress.

Deep Q-network for tigers

In the previous example, both groups of agents were behaving randomly, which is not very interesting. Now we will apply the deep Q-network (DQN) model to the tiger group of agents to check whether they can learn some interesting policy. All of the agents share the network, so their behavior will be the same.

The training code is in `Chapter25/forest_tigers_dqn.py`, and it doesn't differ much from the other DQN versions from the previous chapters. To make the MAgent environment work with our classes, `gym.Env` wrapper was implemented in `Chapter25/lib/data.py` in class `MAgentEnv`. Let's check it to understand how it fits into the rest of the code.

```python
class MAgentEnv(VectorEnv):
    def __init__(self, env: magent.GridWorld, handle,
                 reset_env_func: Callable[[], None],
                 is_slave: bool = False,
                 steps_limit: Optional[int] = None):
        reset_env_func()
        action_space = self.handle_action_space(env, handle)
        observation_space = self.handle_obs_space(env, handle)

        count = env.get_num(handle)

        super(MAgentEnv, self).__init__(count, observation_space,
                                        action_space)
        self.action_space = self.single_action_space
        self._env = env
        self._handle = handle
        self._reset_env_func = reset_env_func
```

Our class is inherited from the `gym.vector.vector_env.VectorEnv` class, which allows two modes of vectorized environments: sync and async. Our communication is sync, but we will need to take into account this class implementation to subclass it properly. The constructor takes three arguments: the MAgent environment instance, the handle of the group that we will control, and the function `reset_env_func`, which has to reset the MAgent environment to the initial state (clear the grid, put walls in, and place agents). Helper methods are used to build action and observation space descriptions from the environment and handle.

```python
    @classmethod
    def handle_action_space(cls, env: magent.GridWorld,
                            handle) -> gym.Space:
        return spaces.Discrete(env.get_action_space(handle)[0])
```

We assume that the action space is just a discrete set of actions, so we get its size from the environment.

```
    @classmethod
    def handle_obs_space(cls, env: magent.GridWorld,
                         handle) -> gym.Space:
        v = env.get_view_space(handle)
        r = env.get_feature_space(handle)
        view_shape = (v[-1],) + v[:2]
        view_space = spaces.Box(low=0.0, high=1.0,
                                shape=view_shape)
        extra_space = spaces.Box(low=0.0, high=1.0, shape=r)
        return spaces.Tuple((view_space, extra_space))
```

The observation space is a bit more complicated. We have two parts to it: spatial, which will be processed with convolution networks, and a feature vector. Spatial features need to be rearranged to meet the PyTorch convention of (C, W, H). Then, we construct two `spaces.Box` instances and combine them using `spaces.Tuple`.

```
    @classmethod
    def handle_observations(cls, env: magent.GridWorld,
                            handle) -> List[Tuple[np.ndarray,
                                                  np.ndarray]]:
        view_obs, feats_obs = env.get_observation(handle)
        entries = view_obs.shape[0]
        if entries == 0:
            return []
        view_obs = np.array(view_obs)
        feats_obs = np.array(feats_obs)
        view_obs = np.moveaxis(view_obs, 3, 1)

        res = []
        for o_view, o_feats in zip(np.vsplit(view_obs, entries),
                                   np.vsplit(feats_obs, entries)):
            res.append((o_view[0], o_feats[0]))
        return res
```

This class method is responsible for building observations from the current environment state. It queries the observations, copies both components into NumPy arrays, changes the axis order, and splits both observations on the first dimension, converting them into a list of tuples. Every tuple in the returned list contains observations for every alive agent in the group. Those observations will be added to the replay buffer and sampled for training, so we need to split them into items.

```
    def reset_wait(self):
        self._steps_done = 0
        if not self._is_slave:
            self._reset_env_func()
        return self.handle_observations(self._env, self._handle)
```

The rest of the class uses helper class methods that we have defined. To reset the environment, we call the reset function and build the observation list.

```python
def step_async(self, actions):
    act = np.array(actions, dtype=np.int32)
    self._env.set_action(self._handle, act)
```

The step method consists of two parts: async, which sets the actions in the underlying MAgent environment, and sync, which does the actual step call and gathers the information we need from the `step()` method in the Gym environment.

```python
def step_wait(self):
    self._steps_done += 1
    if not self._is_slave:
        done = self._env.step()
        self._env.clear_dead()
        if self._steps_limit is not None and \
                self._steps_limit <= self._steps_done:
            done = True
    else:
        done = False

    obs = self.handle_observations(self._env, self._handle)
    r = self._env.get_reward(self._handle).tolist()
    dones = [done] * len(r)
    if done:
        obs = self.reset()
        dones = [done] * self.num_envs
        r = [0.0] * self.num_envs
    return obs, r, dones, {}
```

In `step_wait()`, we ask the MAgent environment to do a step in the simulation, then we clear dead agents from all groups and prepare the results. Special handling of partial groups is needed. As our agents might die during the episode, the length of observations and rewards might decrease over time. It's not an issue for PTAN classes, but we will need to take care of the `dones` array. If the episode ends, which will happen when all our agents die, we reset the episode and return the fresh observation.

That's all for the `MAgentEnv` class; the rest of the code is the same as before, so I'm not going to show it.

Training and results

To start the training, run `./forest_tigers_dqn.py -n run_name –cuda`. In about three hours of training, the tigers' test reward has reached the best score of 70, which is a significant improvement over the random baseline. Acting randomly, tigers die after 20 steps (thus, they obtain the reward of 20).

For every deer, the tigers obtain eight health points, which allows them to survive for an extra 16 steps. So, the reward of 70 for the group of 10 tigers means that they have eaten 31 deer during the episode, which is not that bad given that there were 50 deer in total and the sparsity of the deer on the map.

It's quite likely that the policy stopped improving just because of the limited view of the tigers. If you are curious, you can enable the minimap in the environment settings and experiment. With more information about the food's location, it's likely that the policy could be improved even more.

The following are the charts of the training process.
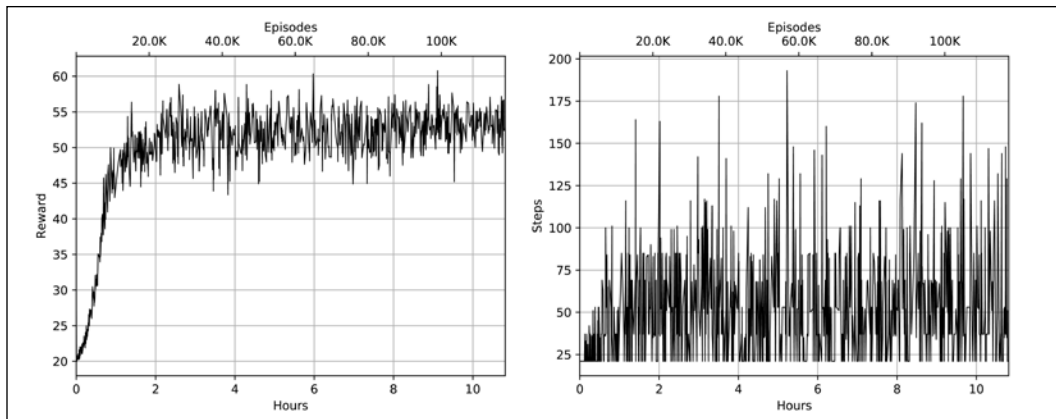
Figure 25.3: Average reward (left) and the steps of training episodes (right)

From the reward and steps charts, it's quite obvious that the training process stopped converging after 30k-40k episodes.
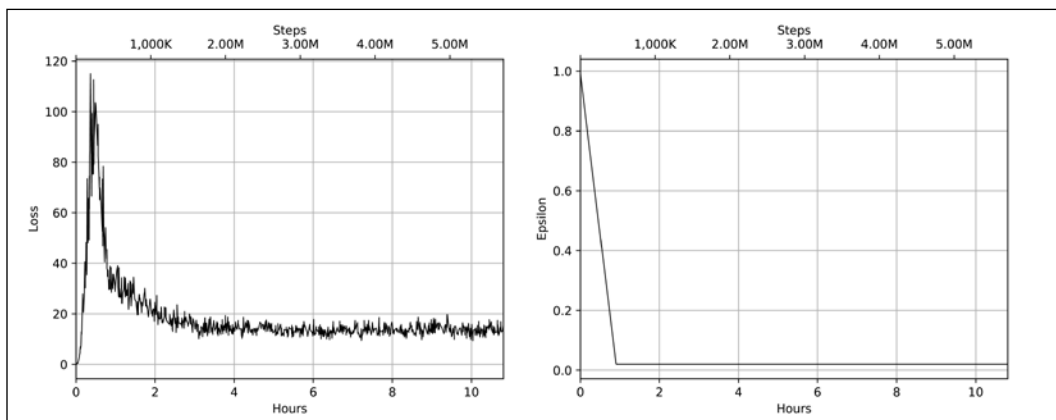
Figure 25.4: Training loss (left) and epsilon (right) during the training
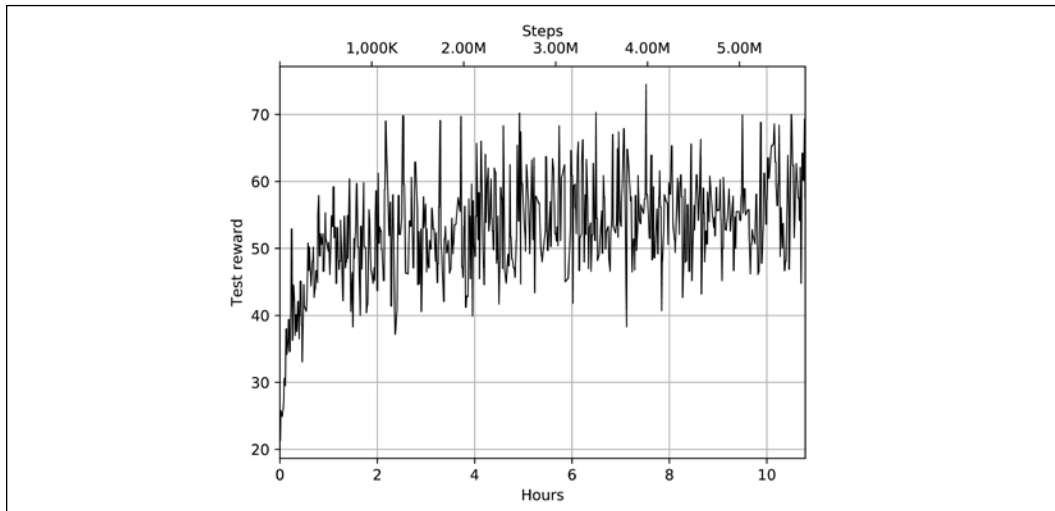
Figure 25.5: The reward value of test episodes

Every 10k training iterations, one full episode was played to check the current policy alone (without random actions introduced by epsilon-greedy exploration). Once the best reward is updated, the model can be saved into the file.

There is a tool, `Chapter25/forest_tigers_play.py`, that can load the saved policy and record the tigers in action. This utility loads the model and plays one episode, and then saves the recording file in the `Chapter25/render` directory. Recorded text files can be visualized in the browser using the record utility, as was already described. It's not possible to insert animation into this book, so the following are links to videos of the tigers' policy at different stages of the training.

- Score 23: `https://www.youtube.com/watch?v=NJPAGCGVeDM`. Tigers moved almost randomly; only one tiger was close enough to a deer to eat it and live a bit longer.

- Score 39: `https://www.youtube.com/watch?v=kkNrnIQ3j3U`. Looks like the tigers always moved left and ate everything they found. This allowed some of them to get better reward and live longer, but still, this policy is not the best.

- Score 67: `https://www.youtube.com/watch?v=W0aJ9mu1MwA`. This was the best policy that I got. Tigers were actively trying to eat deer. The last survivor even stole food from its slower colleague (at five seconds into the video).

Collaboration by the tigers

The second experiment that I implemented was designed to make the tigers' lives more complicated and encourage collaboration between them. The training and play code are the same; the only difference is in the MAgent environment's configuration. I took the `double_attack` configuration file from MAgent (`https://github.com/geek-ai/MAgent/blob/master/python/magent/builtin/config/double_attack.py`) and tweaked it to add the reward of 0.1 after every step for both tigers and deer. The following is the modified function `config_double_attack()` from Chapter25/lib/data.py:

```
def config_double_attack(map_size):
    gw = magent.gridworld
    cfg = gw.Config()
    cfg.set({"map_width": map_size, "map_height": map_size})
    cfg.set({"embedding_size": 10})
```

We create the configuration object and set the map dimensions. The embedding size is the dimensionality of the minimap, which is not enabled in this configuration.

```
    deer = cfg.register_agent_type("deer", {
        'width': 1, 'length': 1, 'hp': 5, 'speed': 1,
        'view_range': gw.CircleRange(1),
        'attack_range': gw.CircleRange(0),
        'step_recover': 0.2,
        'kill_supply': 8,
        'step_reward': 0.1,
    })
```

Then we register the group of agents for deer. They have a low health point and small observation range, and they can't attack, but they can recover over time. The parameter `step_reward` was added by me to get the reward from deer's actions.

```
    tiger = cfg.register_agent_type("tiger", {
        'width': 1, 'length': 1, 'hp': 10, 'speed': 1,
        'view_range': gw.CircleRange(4),
        'attack_range': gw.CircleRange(1),
        'damage': 1, 'step_recover': -0.2,
        'step_reward': 0.1,
    })
```

Now the second group: tigers. They have higher reward, a larger observation range, and can attack others. But they lose health with every step, so they need food.

```
    deer_group  = cfg.add_group(deer)
    tiger_group = cfg.add_group(tiger)
```

```
a = gw.AgentSymbol(tiger_group, index='any')
b = gw.AgentSymbol(tiger_group, index='any')
c = gw.AgentSymbol(deer_group,  index='any')
# tigers get reward when they attack a deer simultaneously
e1 = gw.Event(a, 'attack', c)
e2 = gw.Event(b, 'attack', c)
cfg.add_reward_rule(e1 & e2, receiver=[a, b], value=[1, 1])
return cfg
```

Next, we create a rule in the system that rewards two tigers for attacking a deer at the same time. Please note that they can still attack deer alone, but they just won't be rewarded for this. So, that's the model of "social" tigers.

To activate this mode, argument `--mode double_attack` needs to be passed to the training program, so, to train the previous example in collaboration mode, you need to run it as follows: `./forest_tigers_dqn.py -n run_name --mode double_attack --cuda`.

Another tweak I needed to make was related to the exploration. In the previous example, it was 10 tigers and 50 deer, which is enough when one tiger can eat the deer. But now, two tigers need to attack the deer simultaneously, which is much less likely. So, to simplify exploration, I increased the amount of both deer and tigers to 512 and 20, respectively. Otherwise, the process would have failed to converge due to the lack of samples with good reward. Of course, this is not the best solution and better exploration techniques could be used, but I've tried to keep the examples simple. You can experiment with the methods described in *Chapter 21*, *Advanced Exploration*.

The following are convergence charts that I got from training in this mode.
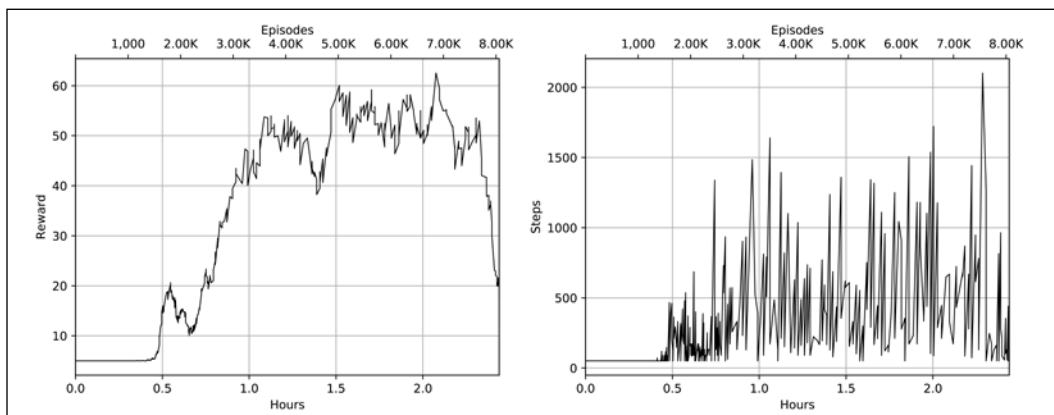
Figure 25.6: Reward and steps of the training episodes

In hindsight, it looks like I stopped the training too early, being scared by a fall in the mean reward. But maybe not; this fall could be a sign of bad hyperparameters, for example, replay size (as the number of agents was doubled, replay size also needs to be enlarged). But anyway, the training has brought us the policy to check.
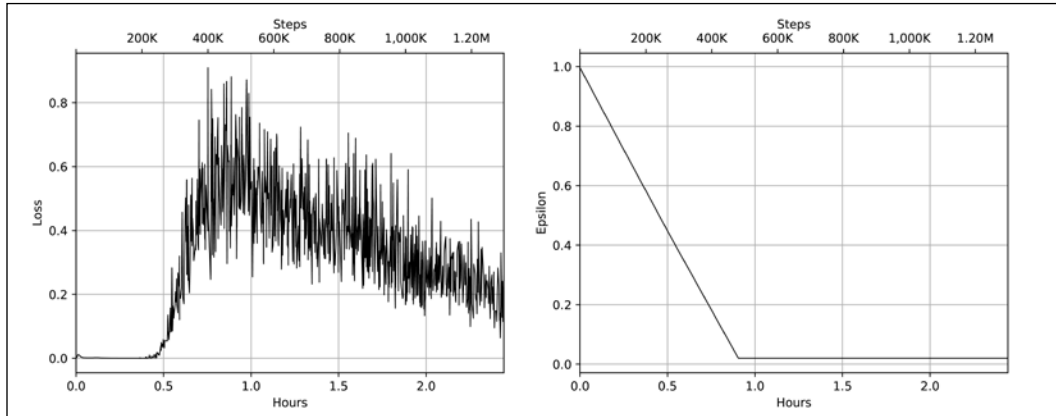
Figure 25.7: Training loss and epsilon values
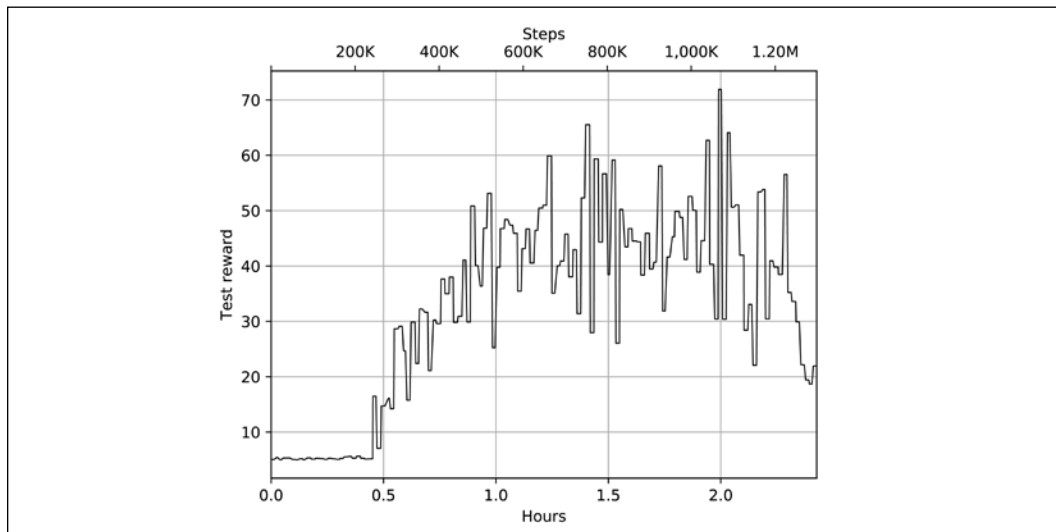
Figure 25.8: The reward value of test episodes

As before, every time the best reward from the test episode is obtained, the model file is saved. The following are visualizations of models obtained during the training:

- Score 5, small amount of agents: `https://youtu.be/_dtuac7ii7k`
- Score 5, large amount of agents: `https://youtu.be/GMyTEbKHATQ`

- Score 30, small amount of agents: `https://youtu.be/5CWiy4Ye57E`
- Score 30, large amount of agents: `https://youtu.be/M4IRD5sGi7U`
- Score 50, small amount of agents: `https://youtu.be/qr2rvCjpWis`
- Score 50, large amount of agents: `https://youtu.be/JiS06bF9hJM`

From these videos, it's obvious that due to my changes in the reward system (giving 0.1 reward on every step), the tigers haven't learned how to collaborate at all. Instead, they eat deer alone to live longer. Potentially, they could discover that attacking the deer together is more profitable, but this is hard from an exploration point of view.

We can also consider the third mode in the utility `./forest_tigers_dqn.py`, called `double_attack_nn`, which uses noisy networks and gives the reward only for situations when deer are killed by two tigers. But, unfortunately, after seven hours of training, this version still hadn't got any non-zero reward, so this `double_attack_nn` mode is left as an advanced exercise for the curious reader.

Training both tigers and deer

The next example is the scenario when both tigers and deer are controlled by different DQN models being trained simultaneously. Tigers are rewarded for living longer, which means eating more deer, as at every step in the simulation they lose health points. Deer are also rewarded on every timestamp.

The code is in `Chapter25/forest_both_dqn.py` and it is quite a simple extension of the previous example. For both groups of agents, we have a separate `Agent` class instance, which communicates with the environment. As the observation for both groups is different, we have two separate networks, replay buffers, and experience sources. On every training step, we sample batches from both replay buffers and then train both networks independently.

I'm not going to put the code here, as it differs from the previous example only in small details. If you are curious, you can check GitHub examples. The following are plots with the convergence results.
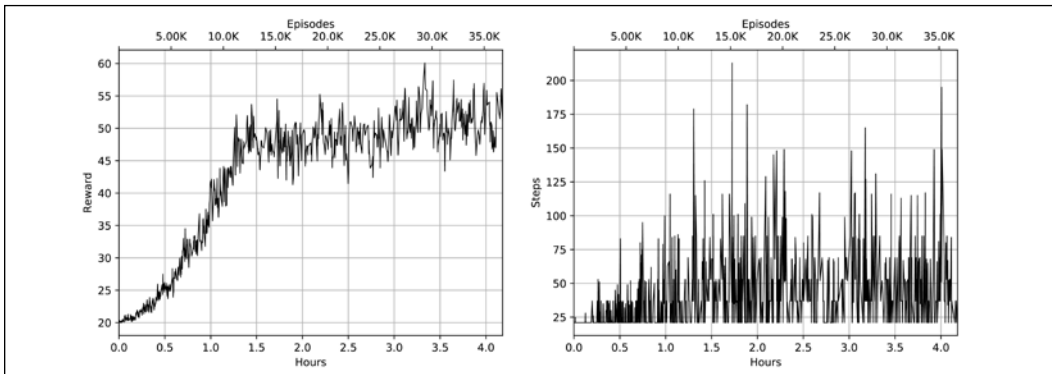
Figure 25.9: Reward and steps for tiger episodes during the training
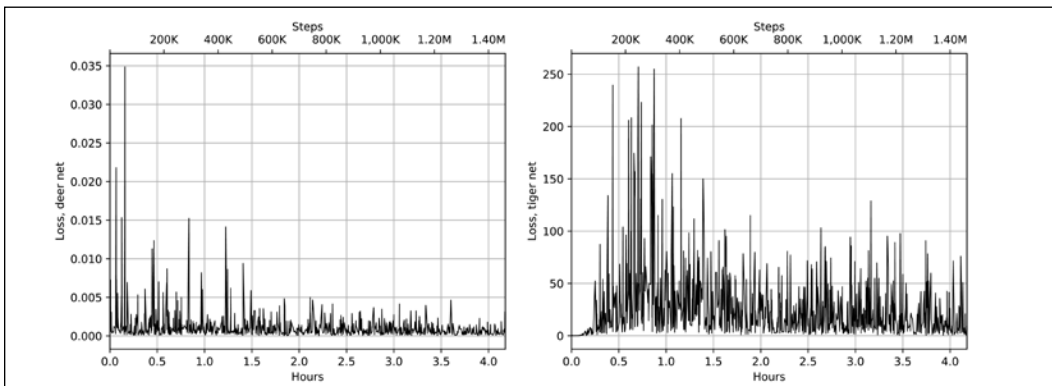
Figure 25.10: Training loss for the deer (left) and tiger networks (right)
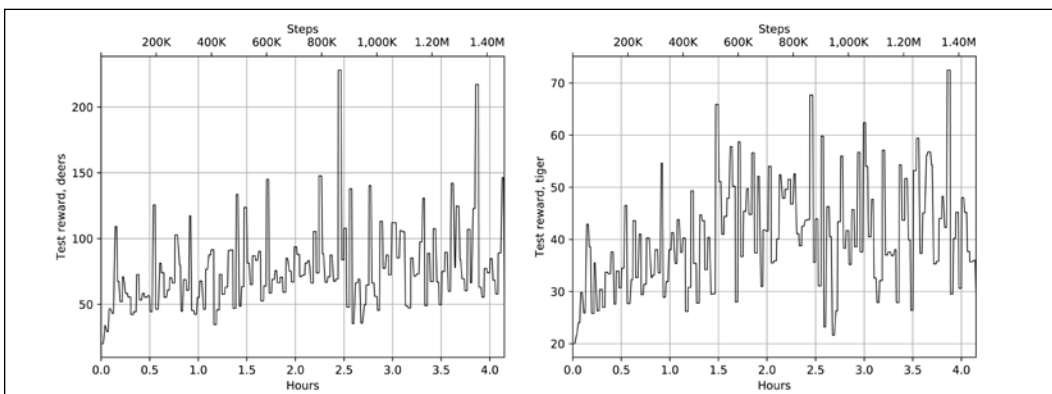
Figure 25.11: Test reward for deer (left) and tigers (right)

As you can see from *Figure 25.11*, deer are way more successful than tigers, which is not surprising, as the speed of both is the same, so the deer just need to move all the time and wait for the tigers to die from starvation. If you want, you can experiment with the environment settings, either by increasing the tigers' speed or by increasing the wall density. The following are recordings of some models that I made and the scores, which support this thinking:

- Model: tigers 72, deer 109: `https://youtu.be/u50_B5OcgXw`
- Model: tigers 72, deer 114: `https://youtu.be/rvWe0mJkkrw`
- Model: tigers 72, deer 228: `https://youtu.be/nAvqQ-yPlws`

As you can see, the deer policy just overfits to the "run in one direction" policy.

The battle between equal actors

The final example in this chapter is the situation when one policy drives fighting between two groups of identical agents. This version is implemented in `Chapter25/battle_dqn.py`. The code is straightforward and won't be put here.

I did only a couple of experiments with the code, so hyperparameters could be improved. In addition, you can experiment with the training process. In the code, both groups are driven by the same policy that we are optimizing, which may not be the best approach. Instead, you can experiment with an AlphaGo Zero style of training, when the best policy is used for one group and another group is driven by the policy that we are optimizing at the moment. Once the best policy starts to consistently lose, it is updated. In this case, the optimized policy may have time to learn all the tricks and weaknesses of the current best policy, which may start an improvement loop.

In my experiments, the training wasn't very stable, but it showed some progress. The following are a couple of recordings that I got from using the obtained policies. To play the battle between the two saved models, you can use the utility `Chapter25/battle_play.py`:

- Model with score –0.9 versus model with score –0.8: `https://youtu.be/sWxJMeC7psY`
- Model with score –0.8 versus model with score –0.7: `https://youtu.be/LArNodOT1oQ`
- Model with score –0.6 versus model with score –0.8: `https://youtu.be/LLn_DNVdyGU`

Summary

In this chapter, we just touched a bit on the very interesting and dynamic field of MARL. There are lots of things that you can try on your own using the MAgent environment or other environments (like PySC2).

My congratulations on reaching the end of the book! I hope that the book was useful and you enjoyed reading it as much as I enjoyed gathering the material and writing all the chapters. As a final word, I would like to wish you good luck in this exciting and dynamic area of RL. The domain is developing very rapidly, but with an understanding of the basics, it will become much simpler for you to keep track of the new developments and research in this field.

There are many very interesting topics left uncovered, such as partially observable Markov decision processes (where environment observations don't fulfill the Markov property) or recent approaches to exploration, such as the count-based methods. There has been a lot of recent activity around multi-agent methods, where many agents need to learn how to coordinate to solve a common problem.

I haven't mentioned the memory-based RL approach, where your agent can maintain some sort of a memory to keep its knowledge and experience. A great deal of effort is being put into increasing the RL sample efficiency, which will ideally be close to the human learning performance one day, but this is still a far-reaching goal at the moment.

Of course, it's not possible to cover the full domain in a small book, because new ideas appear almost every day. However, the goal of this book was to give you a practical foundation in the field, simplifying your own learning of the common methods.

Finally, I'd like to quote Volodymir Mnih's words from his talk, *Recent Advances and Frontiers in Deep RL*, from the Deep RL Bootcamp 2017: "The field of deep RL is very new and everything is still exciting. Literally, nothing is solved yet!"

Another Book You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:
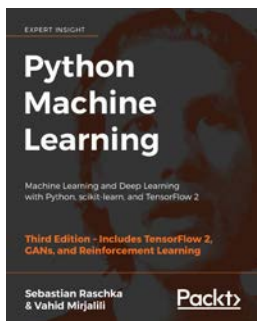
Python Machine Learning - Third Edition

Sebastian Raschka, Vahid Mirjalili

ISBN: 978-1-78995-575-0

- Master the frameworks, models, and techniques that enable machines to 'learn' from data
- Use scikit-learn for machine learning and TensorFlow for deep learning
- Apply machine learning to image classification, sentiment analysis, intelligent web applications, and more
- Build and train neural networks, GANs, and other models
- Discover best practices for evaluating and tuning models
- Predict continuous target outcomes using regression analysis
- Dig deeper into textual and social media data using sentiment analysis

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index