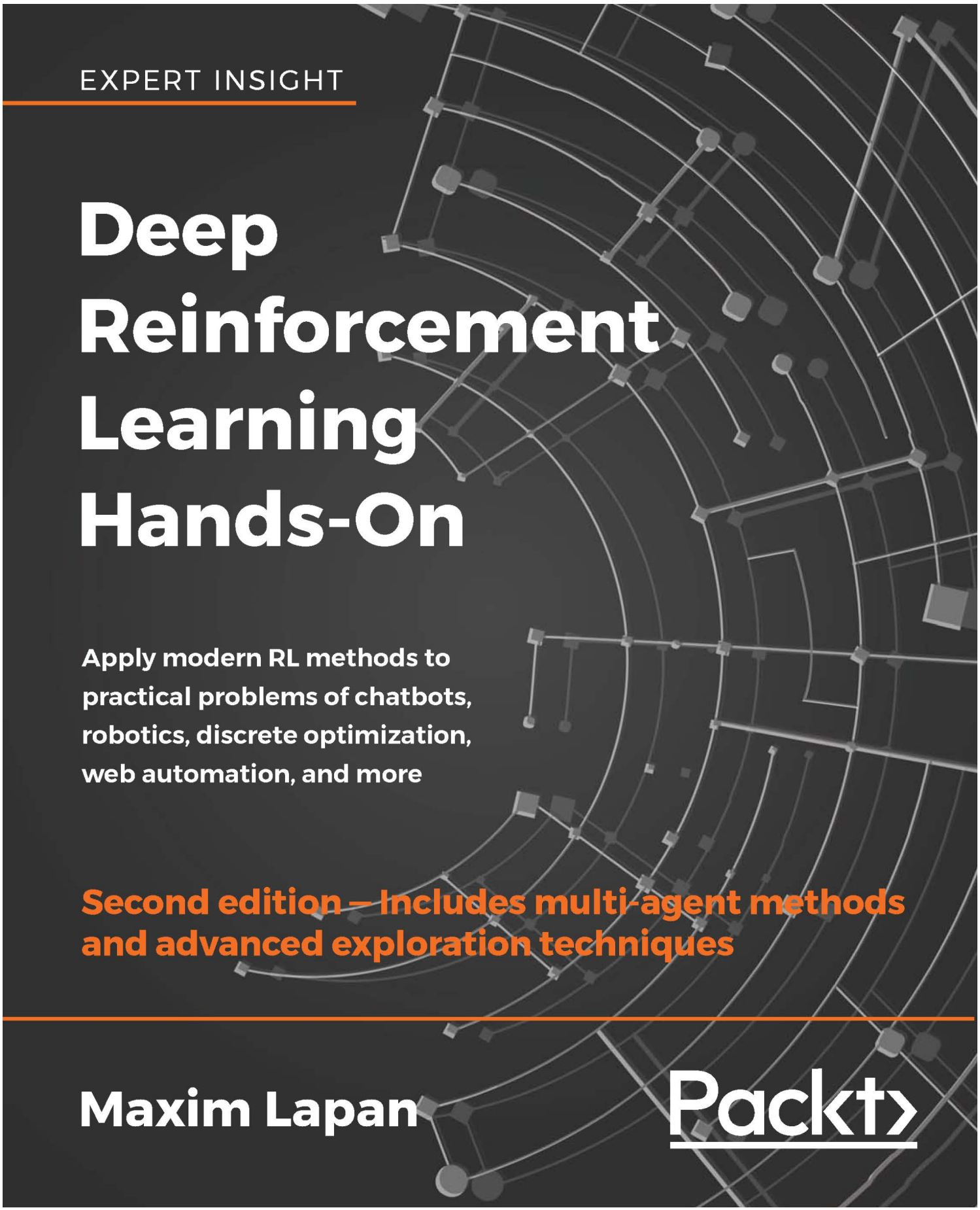
EXPERT INSIGHT
Deep Reinforcement Learning Hands-On
Apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more
Second edition – Includes multi-agent methods and advanced exploration techniques
Maxim Lapan
Packt

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

Packt

BIRMINGHAM - MUMBAI

Deep Reinforcement Learning Hands-On
Second Edition

Producer: Jonathan Malysiak

Acquisition Editor – Peer Reviews: Suresh Jain

Content Development Editors: Joanne Lovell and Chris Nelson

Technical Editor: Saby D'silva

Project Editor: Kishor Rit

Proofreader: Safis Editing

Indexer: Rekha Nair

Presentation Designer: Sandip Tadge

First published: June 2018

Second edition: January 2020

Production reference: 1300120

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-83882-699-4

www.packt.com

packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Learn better with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.Packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.Packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

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.

I'd like to thank my family: my wife, Olga, and my children, Ksenia, Julia, and Fedor, for their patience and support. It was a challenging time writing this book and it wouldn't have been possible without you, so thanks! Julia and Fedor did a great job of gathering samples for MiniWoB (*Chapter 16*, *Web Navigation*) and testing the Connect 4 agent's playing skills (*Chapter 23*, *AlphaGo Zero*).

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.

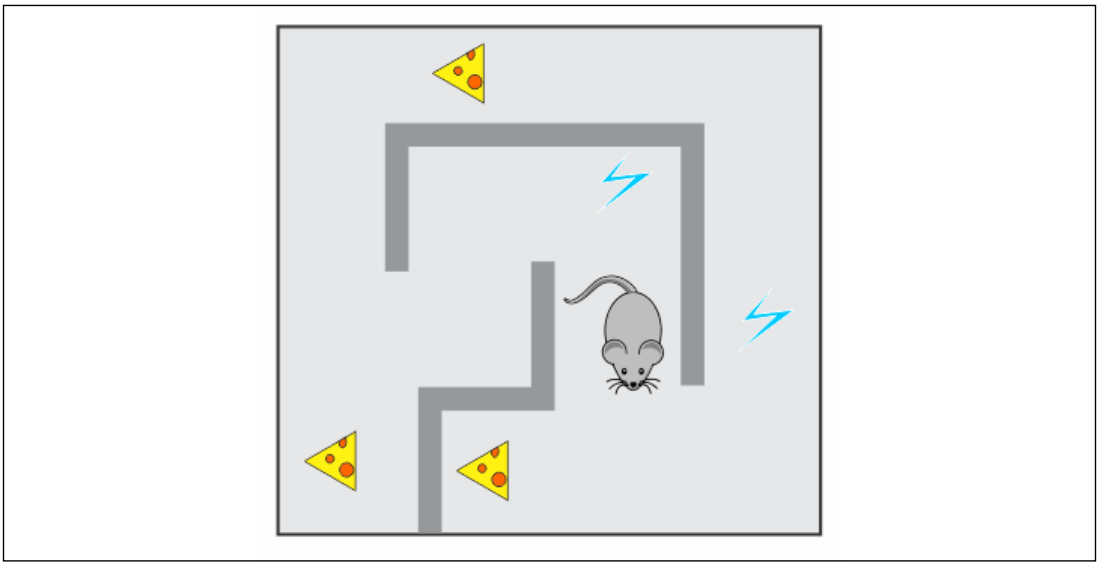

Figure 1.1: The robot mouse maze world

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:

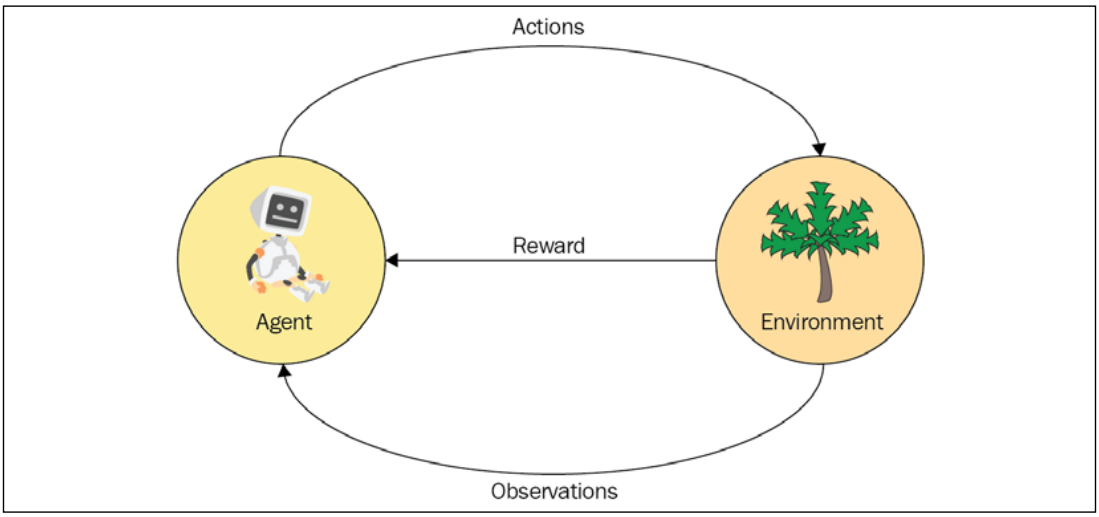

Figure 1.2: RL entities and their communication channels

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).

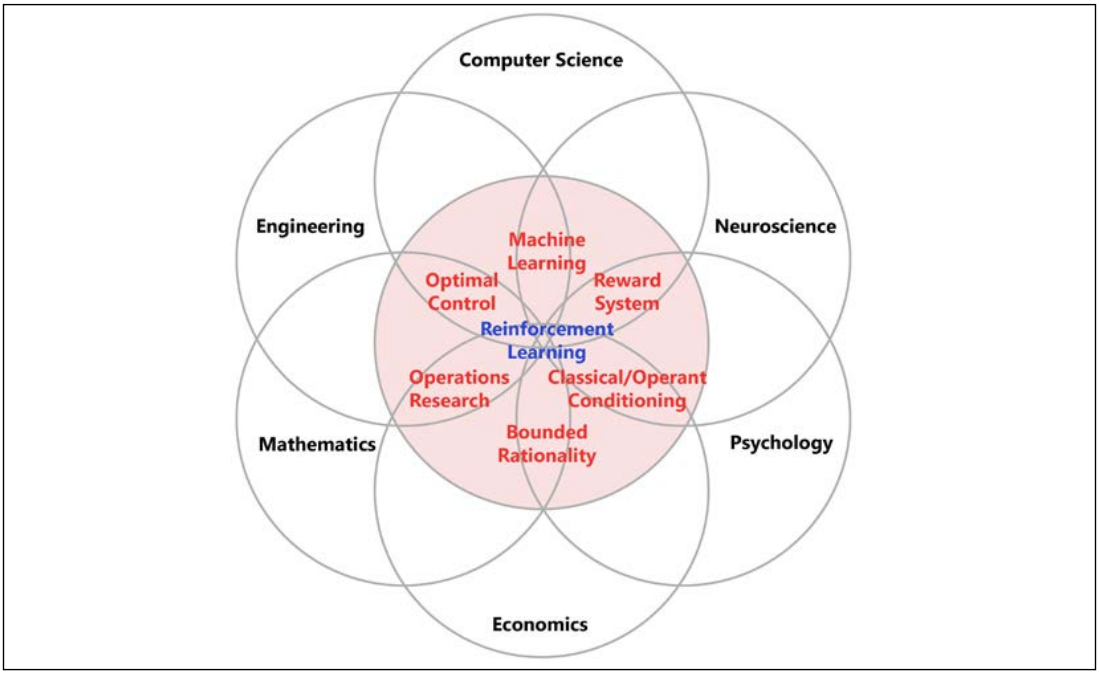

Figure 1.3: Various domains in RL

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.

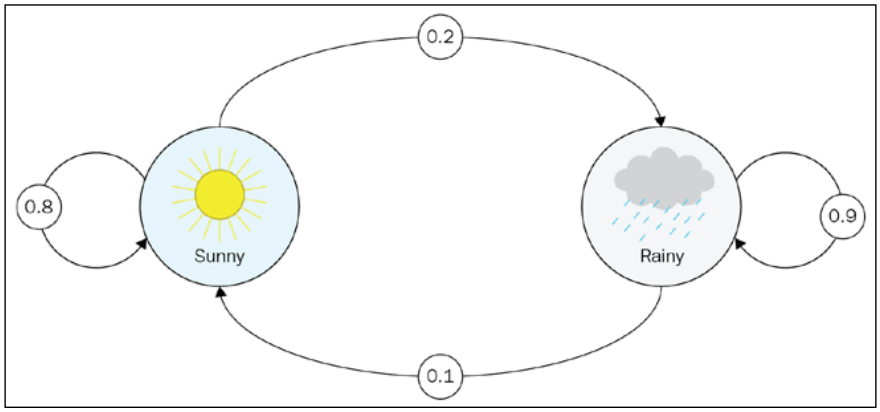

Figure 1.4: The sunny/rainy weather model

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:

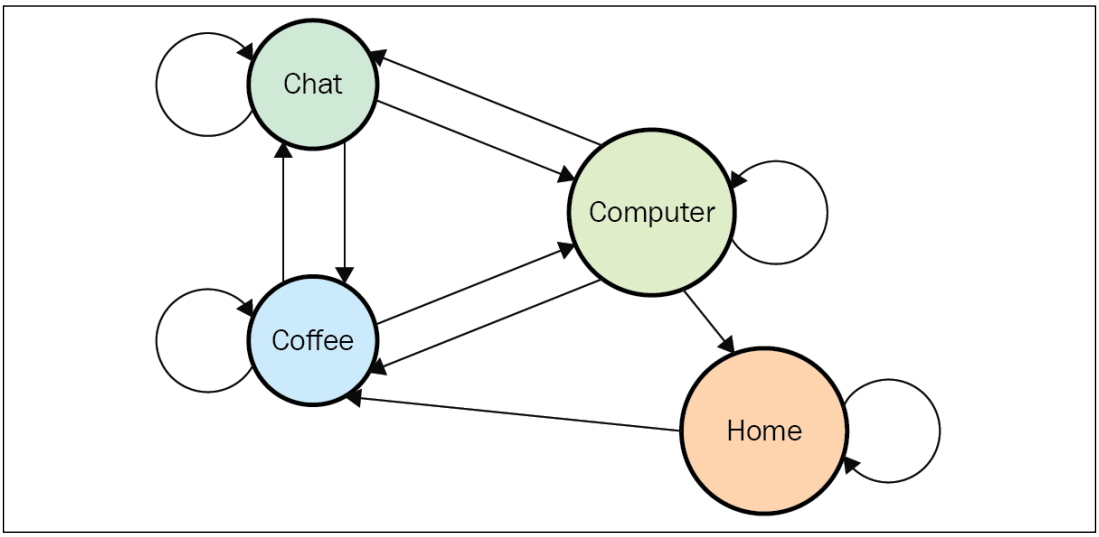

Figure 1.5: The state transition graph for our office worker

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:

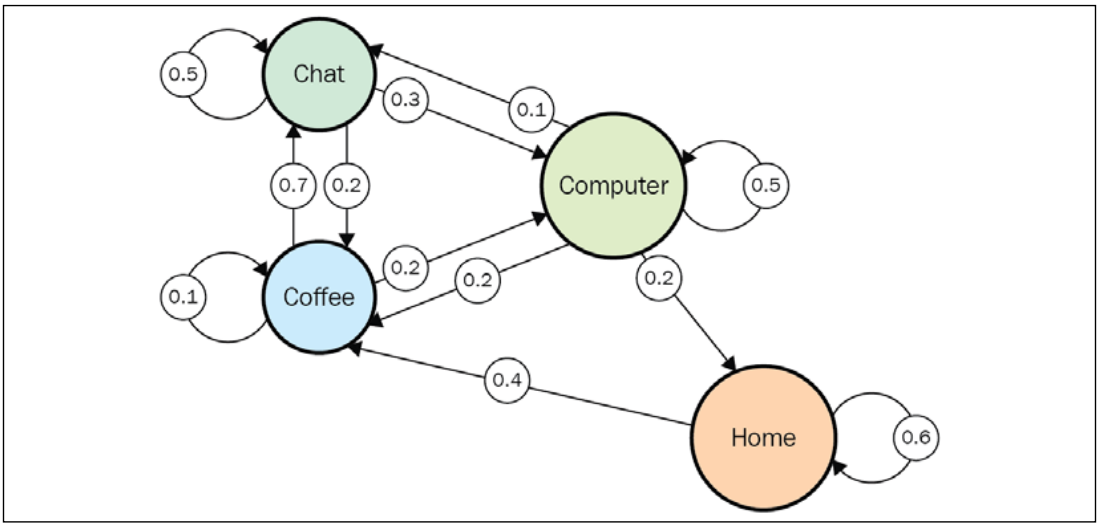

Figure 1.6: The state transition graph with transition probabilities

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:

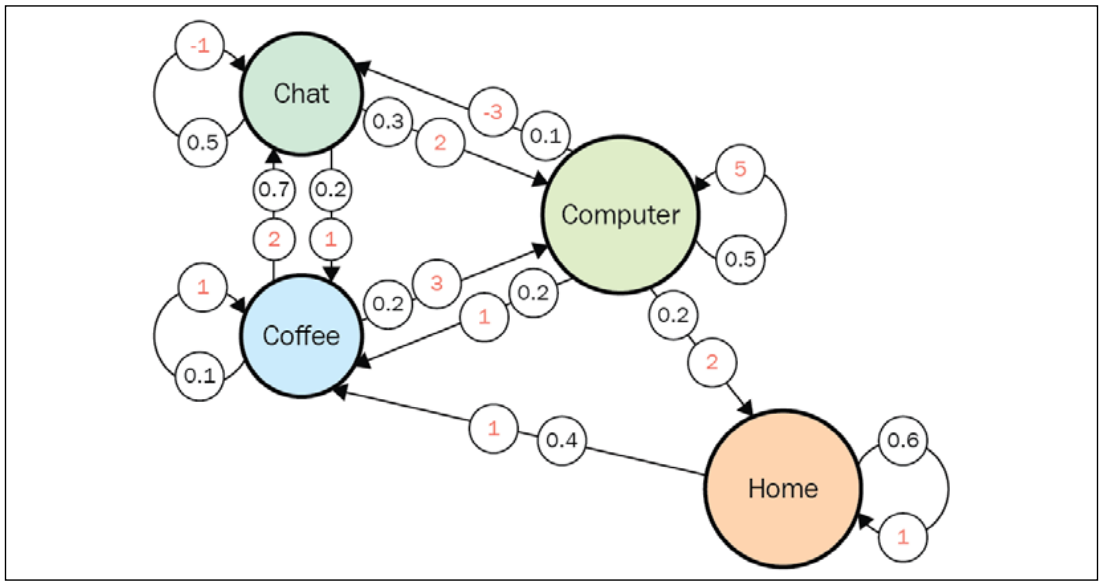

Figure 1.7: A state transition graph with transition probabilities (dark) and rewards (light)

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:

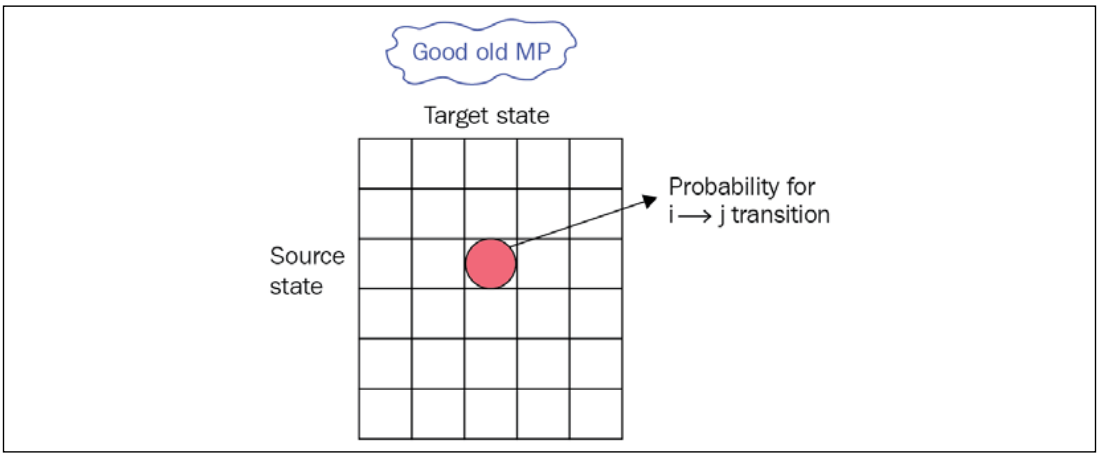

Figure 1.8: The transition matrix in square form

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:

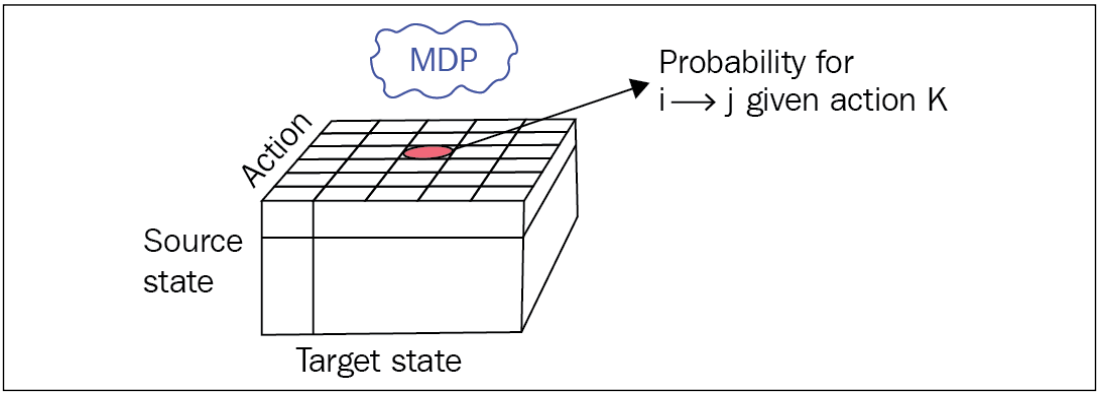

Figure 1.9: Transition probabilities for the MDP

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.

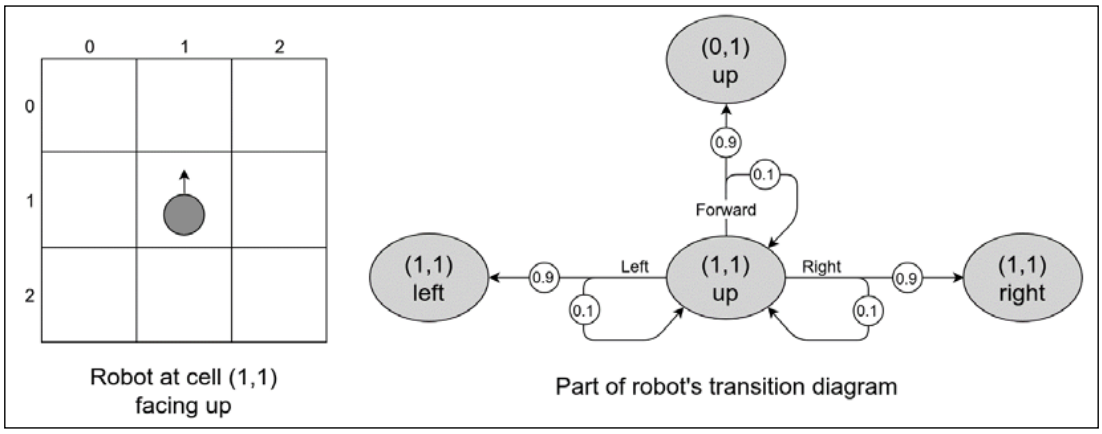

Figure 1.10: A grid world environment

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.