Addison Wesley Data & Analytics Series

Foundations of Deep Reinforcement Learning

Theory and Practice in Python

Laura Graesser
Wah Loon Keng

Praise for *Foundations of Deep Reinforcement Learning*

"This book provides an accessible introduction to deep reinforcement learning covering the mathematical concepts behind popular algorithms as well as their practical implementation. I think the book will be a valuable resource for anyone looking to apply deep reinforcement learning in practice."
—*Volodymyr Mnih, lead developer of DQN*

"An excellent book to quickly develop expertise in the theory, language, and practical implementation of deep reinforcement learning algorithms. A limpid exposition which uses familiar notation; all the most recent techniques explained with concise, readable code, and not a page wasted in irrelevant detours: it is the perfect way to develop a solid foundation on the topic."
—*Vincent Vanhoucke, principal scientist, Google*

"As someone who spends their days trying to make deep reinforcement learning methods more useful for the general public, I can say that Laura and Keng's book is a welcome addition to the literature. It provides both a readable introduction to the fundamental concepts in reinforcement learning as well as intuitive explanations and code for many of the major algorithms in the field. I imagine this will become an invaluable resource for individuals interested in learning about deep reinforcement learning for years to come."
—*Arthur Juliani, senior machine learning engineer, Unity Technologies*

"Until now, the only way to get to grips with deep reinforcement learning was to slowly accumulate knowledge from dozens of different sources. Finally, we have a book bringing everything together in one place."
—*Matthew Rahtz, ML researcher, ETH Zürich*

This page intentionally left blank

Foundations of Deep Reinforcement Learning

Foundations of Deep Reinforcement Learning

Theory and Practice in Python

Laura Graesser
Wah Loon Keng

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2019948417

Cover illustration by Wacomka/Shutterstock

ISBN-13: 978-0-13-517238-4
ISBN-10: 0-13-517238-1

3 20

For those people who make me feel that anything is possible
—Laura

For my wife Daniela
—Keng

This page intentionally left blank

Contents

Foreword

In April of 2019, OpenAI's Five bots played in a Dota 2 competition match against 2018 human world champions, OG. Dota 2 is a complex, multiplayer battle arena game where players can choose different characters. Winning a game requires strategy, teamwork, and quick decisions. Building an artificial intelligence to compete in this game, with so many variables and a seemingly infinite search space for optimization, seems like an insurmountable challenge. Yet OpenAI's bots won handily and, soon after, went on to win over 99% of their matches against public players. The innovation underlying this achievement was deep reinforcement learning.

Although this development is recent, reinforcement learning and deep learning have both been around for decades. However, a significant amount of new research combined with the increasing power of GPUs have pushed the state of the art forward. This book gives the reader an introduction to deep reinforcement learning and distills the work done over the last six years into a cohesive whole.

While training a computer to beat a video game may not be the most practical thing to do, it's only a starting point. Reinforcement learning is an area of machine learning that is useful for solving sequential decision-making problems—that is, problems that are solved over time. This applies to almost any endeavor—be it playing a video game, walking down the street, or driving a car.

Laura Graesser and Wah Loon Keng have put together an approachable introduction to a complicated topic that is at the forefront of what is new in machine learning. Not only have they brought to bear their research into many papers on the topic; they created an open source library, SLM Lab, to help others get up and running quickly with deep reinforcement learning. SLM Lab is written in Python on top of PyTorch, but readers only need familiarity with Python. Readers intending to use TensorFlow or some other library as their deep learning framework of choice will still get value from this book as it introduces the concepts and problem formulations for deep reinforcement learning solutions.

This book brings together the most recent research in deep reinforcement learning along with examples and code that the readers can work with. Their library also works with OpenAI's Gym, Roboschool, and the Unity ML-Agents toolkit, which makes this book a perfect jumping-off point for readers looking to work with those systems.

—*Paul Dix, Series Editor*

This page intentionally left blank

Preface

We first discovered deep reinforcement learning (deep RL) when DeepMind achieved breakthrough performance in the Atari arcade games. Using only images and no prior knowledge, artificial agents reached human-level performance for the first time.

The idea of an artificial agent learning by itself, through trial and error, without supervision, sparked something in our imaginations. It was a new and exciting approach to machine learning, and it was quite different from the more familiar field of supervised learning.

We decided to work together to learn about this topic. We read books and papers, followed online courses, studied code, and tried to implement the core algorithms. We realized that not only is deep RL conceptually challenging, but that implementation requires as much effort as a large software engineering project.

As we progressed, we learned more about the landscape of deep RL—how algorithms relate to each other and what their different characteristics are. Forming a mental model of this was hard because deep RL is a new area of research and the theoretical knowledge had not yet been distilled into a book. We had to learn directly from research papers and online lectures.

Another challenge was the large gap between theory and implementation. Often, a deep RL algorithm has many components and tunable hyperparameters that make it sensitive and fragile. For it to succeed, all the components need to work together correctly and with appropriate hyperparameter values. The implementation details required to get this right are not immediately clear from the theory, but are just as important. A resource that integrated theory and implementation would have been invaluable when we were learning.

We felt that the journey from theory to implementation could have been simpler than we found it, and we wanted to contribute to making deep RL easier to learn. This book is our attempt to do that. It takes an end-to-end approach to introducing deep RL—starting with intuition, then explaining the theory and algorithms, and finishing with implementations and practical tips. This is also why the book comes with a companion software library, SLM Lab, which contains implementations of all the algorithms discussed in it. In short, this is the book we wished existed when we were starting to learn about this topic.

Deep RL belongs to the larger field of reinforcement learning. At the core of reinforcement learning is function approximation; in deep RL, functions are learned using deep neural networks. Reinforcement learning, along with supervised and unsupervised learning, make up the three core machine learning techniques, and each technique differs in how problems are formulated and how algorithms learn from data.

In this book we focus exclusively on deep RL because the challenges we experienced are specific to this subfield of reinforcement learning. This bounds the scope of the book in two ways. First, it excludes all other techniques that can be used to learn functions in reinforcement learning. Second, it emphasizes developments between 2013 and 2019 even though reinforcement learning has existed since the 1950s. Many of the recent developments build from older research, so we felt it was important to trace the development of the main ideas. However, we do not intend to give a comprehensive history of the field.

This book is aimed at undergraduate computer science students and software engineers. It is intended to be an introduction to deep RL and no prior knowledge of the subject is required. However, we do assume that readers have a basic familiarity with machine learning and deep learning as well as an intermediate level of Python programming. Some experience with PyTorch is also useful but not necessary.

The book is organized as follows. Chapter 1 introduces the different aspects of a deep reinforcement learning problem and gives an overview of deep reinforcement learning algorithms.

Part I is concerned with policy-based and value-based algorithms. Chapter 2 introduces the first Policy Gradient method known as REINFORCE. Chapter 3 introduces the first value-based method known as SARSA. Chapter 4 discusses the Deep Q-Networks (DQN) algorithm and Chapter 5 focuses on techniques for improving it—target networks, the Double DQN algorithm, and Prioritized Experience Replay.

Part II focuses on algorithms which combine policy-based and value-based methods. Chapter 6 introduces the Actor-Critic algorithm which extends REINFORCE. Chapter 7 introduces Proximal Policy Optimization (PPO) which can extend Actor-Critic. Chapter 8 discusses synchronous and asynchronous parallelization techniques that are applicable to any of the algorithms in this book. Finally, all the algorithms are summarized in Chapter 9.

Each algorithm chapter is structured in the same way. First, we introduce the main concepts and work through the relevant mathematical formulations. Then we describe the algorithm and discuss an implementation in Python. Finally, we provide a configured algorithm with tuned hyperparameters which can be run in SLM Lab, and illustrate the main characteristics of the algorithm with graphs.

Part III focuses on the practical details of implementing deep RL algorithms. Chapter 10 covers engineering and debugging practices and includes an almanac of hyperparameters and results. Chapter 11 provides a usage reference for the companion library, SLM Lab. Chapter 12 looks at neural network design and Chapter 13 discusses hardware.

The final part of book, Part IV, is about environment design. It consists of Chapters 14, 15, 16, and 17 which treat the design of states, actions, rewards, and transition functions respectively.

The book is intended to be read linearly from Chapter 1 to Chapter 10. These chapters introduce all of the algorithms in the book and provide practical tips for getting them to work. The next three chapters, 11 to 13, focus on more specialized topics and can be read in any order. For readers that do not wish to go into as much depth, Chapters 1, 2, 3, 4, 6, and 10 are a coherent subset of the book that focuses on a few of the algorithms. Finally, Part IV contains a standalone set of chapters intended for readers with a particular interest in understanding environments in more depth or building their own.

SLM Lab [67], this book's companion software library, is a modular deep RL framework built using PyTorch [114]. SLM stands for Strange Loop Machine, in homage to Hofstadter's iconic book *Gödel, Escher, Bach: An Eternal Golden Braid* [53]. The specific examples from SLM Lab that we include use PyTorch's syntax and features for training neural networks. However, the underlying principles for implementing deep RL algorithms are applicable to other deep learning frameworks such as TensorFlow [1].

The design of SLM Lab is intended to help new students learn deep RL by organizing its components into conceptually clear pieces. These components also align with how deep RL is discussed in the academic literature to make it easier to translate from theory to code.

Another important aspect of learning deep RL is experimentation. To facilitate this, SLM Lab also provides an experimentation framework to help new students design and test their own hypotheses.

The SLM Lab library is released as an open source project on Github. We encourage readers to install it (on a Linux or MacOS machine) and run the first demo by following the instructions on the repository website `https://github.com/kengz/SLM-Lab`. A dedicated git branch "`book`" has been created with a version of code compatible with this book. A short installation instruction copied from the repository website is shown in Code 0.1.

Code 0.1 Installing SLM-Lab from the book git branch

```
# clone the repository
git clone https://github.com/kengz/SLM-Lab.git
cd SLM-Lab
# checkout the dedicated branch for this book
git checkout book
# install dependencies
./bin/setup
# next, follow the demo instructions on the repository website
```

We recommend you set this up first so you can train agents with algorithms as they are introduced in this book. Beyond installation and running the demo, it is not necessary to be familiar with SLM Lab before reading the algorithm chapters (Parts I and II)—we give all the commands to train agents where needed. We also discuss SLM Lab more extensively in Chapter 11 after shifting focus from algorithms to more practical aspects of deep reinforcement learning.

Acknowledgments

There are many people who have helped us finish this project. We thank Milan Cvitkovic, Alex Leeds, Navdeep Jaitly, Jon Krohn, Katya Vasilaky, and Katelyn Gleason for supporting and encouraging us. We are grateful to OpenAI, PyTorch, Ilya Kostrikov, and Jamromir Janisch for providing high-quality open source implementations of different components of deep RL algorithms. We also thank Arthur Juliani for early discussions on environment design. These resources and discussions were invaluable as we were building SLM Lab.

A number of people provided thoughtful and insightful feedback on earlier drafts of this book. We would like to thank Alexandre Sablayrolles, Anant Gupta, Brandon Strickland, Chong Li, Jon Krohn, Jordi Frank, Karthik Jayasurya, Matthew Rahtz, Pidong Wang, Raymond Chua, Regina R. Monaco, Rico Jonschkowski, Sophie Tabac, and Utku Evci for the time and effort you put into this. The book is better as a result.

We are very grateful to the Pearson production team—Alina Kirsanova, Chris Zahn, Dmitry Kirsanov, and Julie Nahil. Thanks to your thoughtfulness, care, and attention to detail the text has been greatly improved.

Finally, this book would not exist without our editor Debra Williams Cauley. Thank you for your patience and encouragement, and for helping us to see that writing a book was possible.

This page intentionally left blank

About the Authors

Laura Graesser is a research software engineer working in robotics at Google. She has a masters in computer science from New York University where she specialized in machine learning.

Wah Loon Keng is an AI engineer who applies deep reinforcement learning to industrial problems at Machine Zone. He has a background in theoretical physics and computer science.

Together, they have developed two deep reinforcement learning software libraries and have given a number of talks and tutorials on the subject.

1 Introduction to Reinforcement Learning

In this chapter we introduce the main concepts in reinforcement learning. We start by looking at some simple examples to build intuitions about the core components of a reinforcement learning problem—namely, an agent and an environment.

In particular, we will look at how an agent interacts with an environment to optimize an objective. We will then define these more formally and define reinforcement learning as a Markov Decision Process. This is the theoretical foundation of reinforcement learning.

Next, we introduce the three primary functions an agent can learn—a *policy*, *value functions*, and a *model*. We then see how learning these functions gives rise to different families of deep reinforcement learning algorithms.

Finally, we give a brief overview of deep learning, which is the function approximation technique used throughout this book, and discuss the main differences between reinforcement learning and supervised learning.

1.1 Reinforcement Learning

Reinforcement learning (RL) is concerned with solving sequential decision-making problems. Many real-world problems—playing video games, sports, driving, optimizing inventory, robotic control—can be framed in this way. These are things that humans and machines do.

When solving these problems, we have an objective or goal—such as winning the game, arriving safely at our destination, or minimizing the cost of building products. We take actions and get feedback from the world about how close we are to achieving the objective—the current score, distance to our destination, or price per unit. Reaching our goal typically involves taking many actions in sequence, each action changing the world around us. We observe these changes in the world as well as the feedback we receive before deciding on the next action to take as a response.

Imagine the following scenario: you are at a party where a friend brings out a flag pole and challenges you to balance it on your hand for as long as possible. If you have never held a flag pole before, your initial attempts will not be very successful. You may spend the first few moments trying to get a feel of the flag pole via trial and error—as it keeps falling over.

These mistakes allow you to collect valuable information and gain some intuition about how to balance the flag pole—where its center of gravity is, how fast it tilts over, how quickly you should adjust, at what angle it falls over, etc. You use this information to make corrections in your next attempts, improve, make further adjustments—and, before you know it, you can start balancing it for 5 seconds, 10 seconds, 30 seconds, 1 minute, and so on.

This process illustrates how reinforcement learning works. In reinforcement learning, you are what is called the "agent," and the flag pole and your surroundings are called an "environment." In fact, the first environment we will learn to solve with reinforcement learning is a toy version of this scenario called CartPole, shown in Figure 1.1. An agent controls a cart sliding along an axis in order to balance a pole upright for a given time. In reality, a human does much more—for example, you may apply your existing intuition about physics, or transfer skills from similar tasks such as balancing a tray full of drinks—but the problems are essentially the same in formulation.

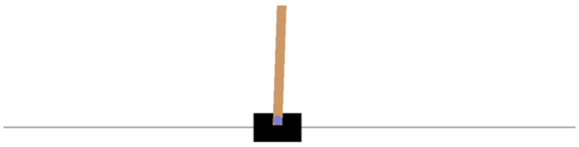


Figure 1.1 `CartPole-v0` is a simple toy environment. The objective is to balance a pole for 200 time steps by controlling the left-right motion of a cart.

Reinforcement learning studies problems of this form and methods by which artificial agents learn to solve them. It is a subfield of artificial intelligence that dates back to the optimal control theory and Markov decision processes (MDPs). It was first worked on by Richard Bellman in the 1950s in the context of dynamic programming and quasilinear equations [15]. We will see this name again when we study a famous equation in reinforcement learning—the Bellman equation.

RL problems can be expressed as a system consisting of an agent and an environment. An environment produces information which describes the state of the system. This is known as a *state*. An agent interacts with an environment by observing the state and using this information to select an *action*. The environment accepts the action and transitions into the next state. It then returns the next state and a *reward* to the agent. When the cycle of (*state* $\rightarrow$ *action* $\rightarrow$ *reward*) completes, we say that one *time step* has passed. The cycle repeats until the environment terminates, for example when the problem is solved. This entire process is described by the control loop diagram in Figure 1.2.

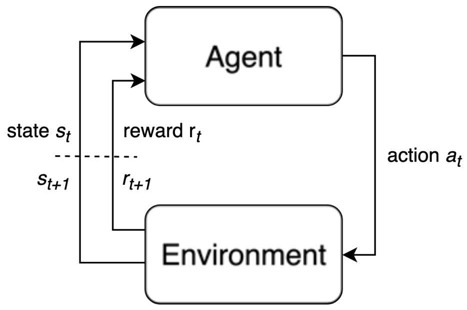


Figure 1.2 The reinforcement learning control loop

We call an agent's action-producing function a *policy*. Formally, a policy is a function which maps states to actions. An action will change the environment and affect what an agent observes and does next. The exchange between an agent and an environment unfolds in time—therefore it can be thought of as a sequential decision-making process.

RL problems have an *objective*, which is the sum of rewards received by an agent. An agent's goal is to maximize the objective by selecting good actions. It *learns* to do this by interacting with the environment in a process of trial and error, and uses the reward signals it receives to *reinforce* good actions.

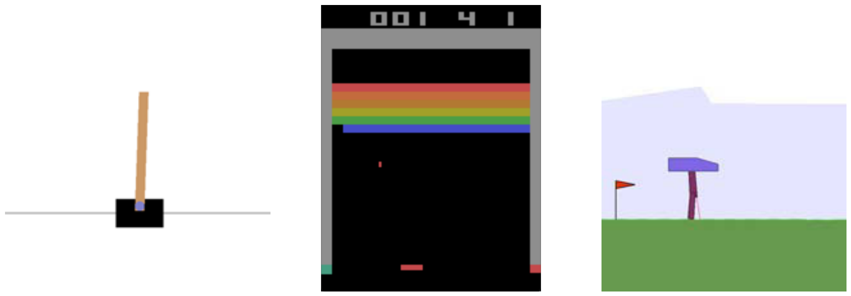
Agent and environment are defined to be mutually exclusive, so that the boundaries between the exchange of the state, action, and reward are unambiguous. We can consider the environment to be anything that is not the agent. For example, when riding a bike, we can have multiple but equally valid definitions of an agent and an environment. If we consider our entire body to be the agent that observes our surroundings and produces muscle movements as actions, then the environment is the bicycle and the road. If we consider our mental processes to be the agent, then the environment is our physical body, the bicycle, and the road, with actions being the neural signals sent from our brain to the muscles and states being the sensory inputs sent back to our brain.

Essentially, a reinforcement learning system is a feedback control loop where an agent and an environment interact and exchange signals, while the agent tries to maximize the objective. The signals exchanged are (s_t, a_t, r_t), which stand for state, action, and reward, respectively, and t denotes the time step in which these signals occurred. The (s_t, a_t, r_t) tuple is called an *experience*. The control loop can repeat forever[1] or terminate by reaching either a terminal state or a maximum time step $t = T$. The time horizon from $t = 0$ to when the environment terminates is called an *episode*. A *trajectory* is a sequence of experiences over an episode, $\tau = (s_0, a_0, r_0), (s_1, a_1, r_1), \ldots$. An agent typically needs many episodes to learn a good policy, ranging from hundreds to millions depending on the complexity of the problem.

Let's look at the three example reinforcement learning environments, shown in Figure 1.3, and how the states, actions, and rewards are defined. All the environments are

1. Infinite control loops exist in theory but not in practice. Typically, we assign a maximum time step T to an environment.

available through the OpenAI Gym [18] which is an open source library that provides a standardized set of environments.



(a) CartPole (b) Atari Breakout (c) BipedalWalker

Figure 1.3 Three example environments with different states, actions, and rewards. These environments are available in OpenAI Gym.

CartPole (Figure 1.3a) is one of the simplest reinforcement learning environments, first described by Barto, Sutton, and Anderson [11] in 1983. In this environment, a pole is attached to a cart that can be moved along a frictionless track. The main features of the environment are summarized below:

1. **Objective:** Keep the pole upright for 200 time steps.
2. **State:** An array of length 4 which represents: [cart position, cart velocity, pole angle, pole angular velocity]. For example, $[-0.034, 0.032, -0.031, 0.036]$.
3. **Action:** An integer, either 0 to move the cart a fixed distance to the left, or 1 to move the cart a fixed distance to the right.
4. **Reward:** $+1$ for every time step the pole remains upright.
5. **Termination:** When the pole falls over (greater than 12 degrees from vertical), or when the cart moves out of the screen, or when the maximum time step of 200 is reached.

Atari Breakout (Figure 1.3b) is a retro arcade game that consists of a ball, a bottom paddle controlled by an agent, and bricks. The goal is to hit and destroy all the bricks by bouncing the ball off the paddle. A player starts with five game lives, and a life is lost every time the ball falls off the screen from the bottom.

1. **Objective:** Maximize the game score.
2. **State:** An RGB digital image with resolution 160×210 pixels—that is, what we see on the game screen.
3. **Action:** An integer from the set $\{0, 1, 2, 3\}$ which maps to the game controller actions $\{$no-action, launch the ball, move right, move left$\}$.

4. **Reward:** The game score difference between consecutive states.
5. **Termination:** When all game lives are lost.

BipedalWalker (Figure 1.3c) is a continuous control problem where an agent uses a robot's lidar sensor to sense its surroundings and walk to the right without falling.

1. **Objective:** Walk to the right without falling.
2. **State:** An array of length 24 which represents: [hull angle, hull angular velocity, x-velocity, y-velocity, hip 1 joint angle, hip 1 joint speed, knee 1 joint angle, knee 1 joint speed, leg 1 ground contact, hip 2 joint angle, hip 2 joint speed, knee 2 joint angle, knee 2 joint speed, leg 2 ground contact, . . ., 10 lidar readings]. For example, [2.745e−03, 1.180e−05, −1.539e−03, −1.600e−02, . . ., 7.091e−01, 8.859e−01, 1.000e+00, 1.000e+00].
3. **Action:** A vector of four floating point numbers in the interval [−1.0, 1.0] which represents: [hip 1 torque and velocity, knee 1 torque and velocity, hip 2 torque and velocity, knee 2 torque and velocity]. For example, [0.097, 0.430, 0.205, 0.089].
4. **Reward:** Reward for moving forward to the right, up to a maximum of +300. −100 if the robot falls. Additionally, there is a small negative reward (movement cost) at every time step, proportional to the absolute torque applied.
5. **Termination:** When the robot body touches the ground or reaches the goal on the right side, or after the maximum time step of 1600.

These environments demonstrate some of the different forms that states and actions can take. In CartPole and BipedalWalker, the states are vectors describing properties such as positions and velocities. In Atari Breakout, the state is an image from the game screen. In CartPole and Atari Breakout, actions are single, discrete integers, whereas in BipedalWalker, an action is a continuous vector of four floating-point numbers. Rewards are always a scalar, but the range varies from task to task.

Having seen some examples, let's now formally describe states, actions, and rewards.

$$s_t \in \mathcal{S} \text{ is the state, } \mathcal{S} \text{ is the state space.} \tag{1.1}$$

$$a_t \in \mathcal{A} \text{ is the action, } \mathcal{A} \text{ is the action space.} \tag{1.2}$$

$$r_t = \mathcal{R}(s_t, a_t, s_{t+1}) \text{ is the reward, } \mathcal{R} \text{ is the reward function.} \tag{1.3}$$

The state space $\mathcal{S}$ is the set of all possible states in an environment. Depending on the environment, it can be defined in many different ways—as integers, real numbers, vectors, matrices, structured or unstructured data. Similarly, the action space $\mathcal{A}$ is the set of all possible actions defined by an environment. It can also take many forms, but is commonly defined as either a scalar or a vector. The reward function $\mathcal{R}(s_t, a_t, s_{t+1})$ assigns a positive, negative, or zero scalar to each transition (s_t, a_t, s_{t+1}). The state space, action space, and reward function are specified by the environment. Together, they define the (s, a, r) tuples which are the basic unit of information describing a reinforcement learning system.

1.2 Reinforcement Learning as MDP

Now, consider how an environment transitions from one state to the next using what is known as the *transition function*. In reinforcement learning, a transition function is formulated as a Markov decision process (MDP) which is a mathematical framework that models sequential decision making.

To understand why transition functions are represented as MDPs, consider a general formulation shown in Equation 1.4.

$$s_{t+1} \sim P\big(s_{t+1} \,|\, (s_0, a_0), (s_1, a_1), \ldots, (s_t, a_t)\big) \tag{1.4}$$

Equation 1.4 says that at time step t, the next state s_{t+1} is sampled from a probability distribution P conditioned on the entire history. The probability of an environment transitioning from state s_t to s_{t+1} depends on all of the preceding states s and actions a that have occurred so far in an episode. It is challenging to model a transition function in this form, particularly if episodes last for many time steps. Any transition function that we design would need to be able to account for a vast combination of effects that occurred at any point in the past. Additionally, this formulation makes an agent's action-producing function—its *policy*—significantly more complex. Since the entire history of states and actions is relevant for understanding how an action might change the future state of the world, an agent would need to take into account all of this information when deciding how to act.

To make the environment transition function more practical, we turn it into an MDP by adding the assumption that the transition to the next state s_{t+1} only depends on the previous state s_t and action a_t. This is known as the *Markov property*. With this assumption, the new transition function becomes the following:

$$s_{t+1} \sim P(s_{t+1} \,|\, s_t, a_t) \tag{1.5}$$

Equation 1.5 says that the next state s_{t+1} is sampled from a probability distribution $P(s_{t+1} \,|\, s_t, a_t)$. This is a simpler form of the original transition function. The Markov property implies that the current state and action at time step t contain sufficient information to fully determine the transition probability for the next state at $t + 1$.

Despite the simplicity of this formulation, it is still quite powerful. A lot of processes can be expressed in this form, including games, robotic control, and planning. This is because a state can be defined to include any necessary information required to make the transition function Markov.

For example, consider the Fibonacci sequence described by the formula $s_{t+1} = s_t + s_{t-1}$, where each term s_t is considered a state. To make the function Markov, we redefine the state as $s'_t = [s_t, s_{t-1}]$. Now the state contains sufficient information to compute the next element in the sequence. This strategy can be applied more generally to any system in which a finite set of k consecutive states contains sufficient information to transition to the next state. Box 1.1 contains more details on how states are defined in an MDP and in its generalization, an POMDP. Note that throughout this book, Boxes serve to provide in-depth details that may be skipped on first reading without a loss of understanding of the main subject.

Box 1.1 MDP and POMDP

So far, the concept of state has appeared in two places. First, the state is what is produced by an environment and observed by an agent. Let's call this the *observed state* s_t. Second, the state is what is used by transition function. Let's call this the environment's *internal state* s_t^{int}.

In an MDP, $s_t = s_t^{\text{int}}$, that is, the observed state is identical to the environment's internal state. The same state information that is used to transition an environment into the next state is also made available to an agent.

This is not always the case. The observed state may differ from the environment's internal state, $s_t \neq s_t^{\text{int}}$. In this case, the environment is described as a *partially observable MDP* (POMDP) because the state s_t exposed to the agent only contains partial information about the state of the environment.

In this book, for the most part, we forget about this distinction and assume that $s_t = s_t^{\text{int}}$. However, it is important to be aware of POMDPs for two reasons. First, some of the example environments we consider are not perfect MDPs. For example, in the Atari environment, the observed state s_t is a single RGB image which conveys information about object positions, lives, etc., but not object velocities. Velocities would be included in the environment's internal state since they are required to determine the next state given an action. In these cases, to achieve good performance, we will have to modify s_t to include more information. This is discussed in Chapter 5.

Second, many interesting real-world problems are POMDPs for many reasons, including sensor or data limitations, model error, and environment noise. A detailed discussion of POMDPs is beyond the scope of this book, but we will touch on them briefly when discussing network architecture in Chapter 12.

Finally, when discussing state design in Chapter 14, the distinction between s_t and s_t^{int} will be important because an agent learns from s_t. The information that is included in s_t and the extent to which it differs from s_t^{int} contributes to making a problem harder or easier to solve.

We are now in a position to present the MDP formulation of a reinforcement learning problem. An MDP is defined by a 4-tuple $\mathcal{S}, \mathcal{A}, P(.), \mathcal{R}(.)$, where

- $\mathcal{S}$ is the set of states.
- $\mathcal{A}$ is the set of actions.
- $P(s_{t+1} \mid s_t, a_t)$ is the state transition function of the environment.
- $\mathcal{R}(s_t, a_t, s_{t+1})$ is the reward function of the environment.

One important assumption underlying the reinforcement learning problems discussed in this book is that agents do not have access to the transition function, $P(s_{t+1} \mid s_t, a_t)$, or the reward function, $\mathcal{R}(s_t, a_t, s_{t+1})$. The only way an agent can get information about these functions is through the states, actions, and rewards it actually experiences in the environment—that is, the tuples (s_t, a_t, r_t).

To complete the formulation of the problem, we also need to formalize the concept of an objective which an agent maximizes. First, let's define the *return*[2] $R(\tau)$ using a trajectory from an episode, $\tau = (s_0, a_0, r_0), \ldots, (s_T, a_T, r_T)$:

$$R(\tau) = r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots + \gamma^T r_T = \sum_{t=0}^{T} \gamma^t r_t \tag{1.6}$$

Equation 1.6 defines the return as a discounted sum of the rewards in a trajectory, where $\gamma \in [0, 1]$ is the discount factor.

Then, the *objective* $J(\tau)$ is simply the expectation of the returns over many trajectories, shown in Equation 1.7.

$$J(\tau) = \mathbb{E}_{\tau \sim \pi}[R(\tau)] = \mathbb{E}_{\tau}\left[\sum_{t=0}^{T} \gamma^t r_t\right] \tag{1.7}$$

The return $R(\tau)$ is the sum of discounted rewards $\gamma^t r_t$ over all time steps $t = 0, \ldots, T$. The objective $J(\tau)$ is the return averaged over many episodes. The expectation accounts for stochasticity in the actions and the environment—that is, in repeated runs, the return may not always end up the same. Maximizing the objective is the same as maximizing the return.

The discount factor $\gamma \in [0, 1]$ is an important variable which changes the way future rewards are valued. The smaller γ, the less weight is given to rewards in future time steps, making it "shortsighted." In the extreme case with $\gamma = 0$, the objective only considers the initial reward r_0, as shown in Equation 1.8.

$$R(\tau)_{\gamma=0} = \sum_{t=0}^{T} \gamma^t r_t = r_0 \tag{1.8}$$

The larger γ, the more weight is given to rewards in future time steps: the objective becomes more "farsighted." If $\gamma = 1$, rewards from every time step are weighted equally, as shown in Equation 1.9.

$$R(\tau)_{\gamma=1} = \sum_{t=0}^{T} \gamma^t r_t = \sum_{t=0}^{T} r_t \tag{1.9}$$

For problems with *infinite* time horizon, we need to set $\gamma < 1$ to prevent the objective from becoming unbounded. For *finite* time horizon problems, γ is an important parameter as a problem may become more or less difficult to solve depending on the discount factor we use. We'll look at an example of this at the end of Chapter 2.

Having defined reinforcement learning as an MDP and the objective, we can now express the reinforcement learning control loop from Figure 1.2 as an MDP control loop in Algorithm 1.1.

2. We use R to denote return and reserve $\mathcal{R}$ for the reward function.

Algorithm 1.1 MDP control loop

```
1: Given an env (environment) and an agent:
2: for $episode = 0, \ldots, MAX\_EPISODE$ do
3:     state = env.reset()
4:     agent.reset()
5:     for $t = 0, \ldots, T$ do
6:         action = agent.act(state)
7:         state, reward = env.step(action)
8:         agent.update(action, state, reward)
9:         if env.done() then
10:            break
11:        end if
12:    end for
13: end for
```

Algorithm 1.1 expresses the interactions between an agent and an environment over many episodes and time steps. At the beginning of each episode, the environment and the agent are reset (lines 3–4). On reset, the environment produces an initial state. Then they begin interacting—an agent produces an action given a state (line 6), then the environment produces the next state and reward given the action (line 7), stepping into the next time step. The `agent.act-env.step` cycle continues until the maximum time step T is reached or the environment terminates. Here we also see a new component, `agent.update` (line 8), which encapsulates an agent's learning algorithm. Over multiple time steps and episodes, this method collects data and performs learning internally to maximize the objective.

This algorithm is generic to all reinforcement learning problems as it defines a consistent interface between an agent and an environment. The interface serves as a foundation for implementing many reinforcement learning algorithms under a unified framework, as we will see in SLM Lab, the companion library to this book.

1.3 Learnable Functions in Reinforcement Learning

With reinforcement learning formulated as an MDP, the natural question to ask is, what should an agent learn?

We have seen that an agent can learn an action-producing function known as a *policy*. However, there are other properties of an environment that can be useful to an agent. In particular, there are *three primary functions* to learn in reinforcement learning:

1. A policy, π, which maps state to action: $a \sim \pi(s)$
2. A value function, $V^{\pi}(s)$ or $Q^{\pi}(s, a)$, to estimate the expected return $\mathbb{E}_{\tau}[R(\tau)]$
3. The environment model,[3] $P(s' \mid s, a)$

A policy π is how an agent produces actions in the environment to maximize the objective. Given the reinforcement learning control loop, an agent must produce an action at every time step after observing a state s. A policy is fundamental to this control loop, since it generates the actions to make it run.

A policy can be stochastic. That is, it may probabilistically output different actions for the same state. We can write this as $\pi(a \mid s)$ to denote the probability of an action a given a state s. An action sampled from a policy is written as $a \sim \pi(s)$.

The value functions provide information about the objective. They help an agent understand how good the states and available actions are in terms of the expected future return. They come in two forms—the $V^{\pi}(s)$ and $Q^{\pi}(s, a)$ functions.

$$V^{\pi}(s) = \mathbb{E}_{s_0 = s, \tau \sim \pi}\left[\sum_{t=0}^{T} \gamma^t r_t\right] \tag{1.10}$$

$$Q^{\pi}(s, a) = \mathbb{E}_{s_0 = s, a_0 = a, \tau \sim \pi}\left[\sum_{t=0}^{T} \gamma^t r_t\right] \tag{1.11}$$

The value function V^{π} shown in Equation 1.10 evaluates how good or bad a state is. V^{π} measures the expected return from being in state s, assuming the agent continues to act according to its current policy π. The return $R(\tau) = \sum_{t=0}^{T} \gamma^t r_t$ is measured from the current state s to the end of an episode. It is a forward-looking measure, since all rewards received before state s are ignored.

To give some intuition for the value function V^{π}, let's consider a simple example. Figure 1.4 depicts a grid-world environment in which an agent can move from cell to cell vertically or horizontally. Each cell is a state with an associate reward, as shown on the left of the figure. The environment terminates when the agent reaches the goal state with reward $r = +1$.

On the right, we show the value $V^{\pi}(s)$ calculated for each state from the rewards using Equation 1.10, with $\gamma = 0.9$. The value function V^{π} always depends on a particular policy π. In this example, we chose a policy π which always takes the shortest path to the goal state. If we had chosen another policy—for example, one that always moves right—then the values would be different.

Here we can see the forward-looking property of the value function and its ability to help an agent differentiate between states that give the same reward. The closer an agent is to the goal state, the higher the value.

3. To make notation more compact, it is customary to write a successive pair of tuples (s_t, a_t, r_t), $(s_{t+1}, a_{t+1}, r_{t+1})$ as (s, a, r), (s', a', r'), where the prime symbol $'$ represents the next time step. We will see this throughout the book.

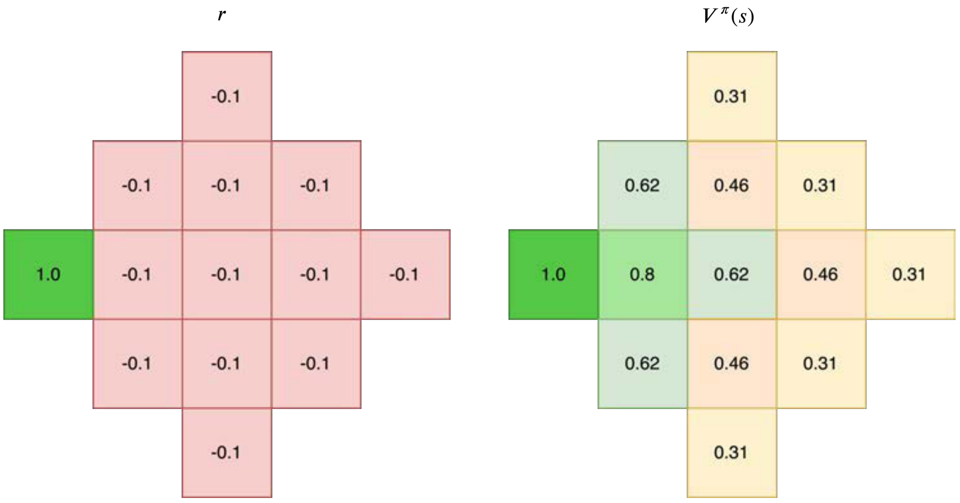


Figure 1.4 Rewards r and values $V^\pi(s)$ for each state s in a simple grid-world environment. The value of a state is calculated from the rewards using Equation 1.10 with γ = 0.9 while using a policy π that always takes the shortest path to the goal state with $r = +1$.

The Q-value function Q^π shown in Equation 1.11 evaluates how good or bad a state-action pair is. Q^π measures the expected return from taking action a in state s assuming that the agent continues to act according to its current policy, π. In the same manner as V^π, the return is measured from the current state s to the end of an episode. It is also a forward-looking measure, since all rewards received before state s are ignored.

We discuss the V^π and Q^π functions in more detail in Chapter 3. For the moment, you just need to know that these functions exist and can be used by agents to solve reinforcement learning problems.

The transition function $P(s' \mid s, a)$ provides information about the environment. If an agent learns this function, it is able to predict the next state s' that the environment will transition into after taking action a in state s. By applying the learned transition function, an agent can "imagine" the consequences of its actions without actually touching the environment. It can then use this information to plan good actions.

1.4 Deep Reinforcement Learning Algorithms

In RL, an agent learns functions to help it act and maximize the objective. This book is concerned with *deep* reinforcement learning (deep RL). This means that we use deep neural networks as the function approximation method.

In Section 1.3, we saw the three primary learnable functions in reinforcement learning. Correspondingly, there are three major families of deep reinforcement learning algorithms—*policy-based*, *value-based*, and *model-based* methods which learn policies, value functions, and models, respectively. There are also combined methods in which agents learn more than one of these functions—for instance, a policy and a value function, or a value function and a model. Figure 1.5 gives an overview of the major deep reinforcement learning algorithms in each family and how they are related.

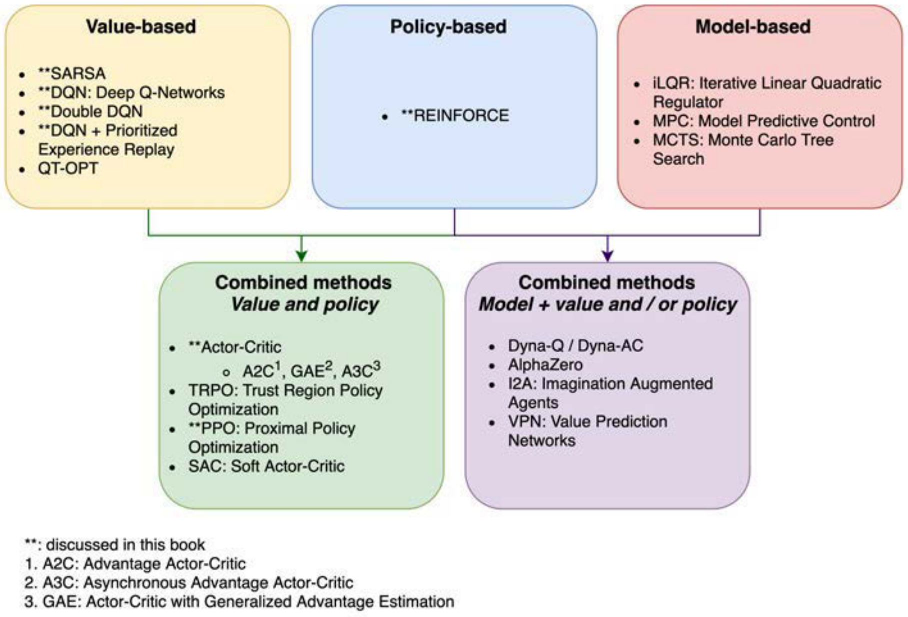

Value-based
**SARSA
**DQN: Deep Q-Networks
**Double DQN
**DQN + Prioritized Experience Replay
QT-OPT
Policy-based
**REINFORCE
Model-based
iLQR: Iterative Linear Quadratic Regulator
MPC: Model Predictive Control
MCTS: Monte Carlo Tree Search
Combined methods Value and policy
**Actor-Critic
A2C1, GAE2, A3C3
TRPO: Trust Region Policy Optimization
**PPO: Proximal Policy Optimization
SAC: Soft Actor-Critic
Combined methods Model + value and / or policy
Dyna-Q / Dyna-AC
AlphaZero
I2A: Imagination Augmented Agents
VPN: Value Prediction Networks
**: discussed in this book
1. A2C: Advantage Actor-Critic
2. A3C: Asynchronous Advantage Actor-Critic
3. GAE: Actor-Critic with Generalized Advantage Estimation

Figure 1.5 Deep reinforcement learning algorithm families

1.4.1 Policy-Based Algorithms

Algorithms in this family learn a policy π. Good policies should generate actions which produce trajectories τ that maximize an agent's objective, $J(\tau) = \mathbb{E}_{\tau \sim \pi}\left[\sum_{t=0}^{T} \gamma^t r_t\right]$. This approach is quite intuitive—if an agent needs to act in an environment, it makes sense to learn a policy. What constitutes a good action at a given moment depends on the state, so a policy function π takes a state s as input to produce an action $a \sim \pi(s)$. This means an agent can make good decisions in different contexts. REINFORCE [148], discussed in Chapter 2, is the most well known policy-based algorithm that forms the foundation of many subsequent algorithms.

A major advantage of policy-based algorithms is that they are a very general class of optimization methods. They can be applied to problems with any type of actions—discrete, continuous, or a mixture (multiactions). They also directly optimize for the thing an agent cares most about—the objective $J(\tau)$. Additionally, this class of methods is guaranteed to converge to a locally[4] optimal policy, as proven by Sutton et al. with the Policy Gradient Theorem [133]. One disadvantage of these methods is that they have high variance and are sample-inefficient.

1.4.2 Value-Based Algorithms

An agent learns either $V^\pi(s)$ or $Q^\pi(s,a)$. It uses the learned value function to evaluate (s,a) pairs and generate a policy. For example, an agent's policy could be to always select the action a in state s with the highest estimated $Q^\pi(s,a)$. Learning $Q^\pi(s,a)$ is far more common than $V^\pi(s)$ for pure value-based approaches because it is easier to convert into a policy. This is because $Q^\pi(s,a)$ contains information about paired states and actions whereas $V^\pi(s)$ just contains information about states.

SARSA [118], discussed in Chapter 3, is one of the older reinforcement learning algorithms. Despite its simplicity, SARSA incorporates many of the core ideas of value-based methods, so it is a good algorithm to study first in this family. However, it is not commonly used today due to its high variance and sample inefficiency during training. Deep Q-Networks (DQN) [88] and its descendants, such as Double DQN [141] and DQN with Prioritized Experience Replay (PER) [121], are much more popular and effective algorithms. These are the subjects of Chapters 4 and 5.

Value-based algorithms are typically more sample-efficient than policy-based algorithms. This is because they have lower variance and make better use of data gathered from the environment. However, there are no guarantees that these algorithms will converge to an optimum. In their standard formulation, they are also only applicable to environments with discrete action spaces. This has historically been a major limitation, but with more recent advances, such as QT-OPT [64], they can be effectively applied to environments with continuous action spaces.

1.4.3 Model-Based Algorithms

Algorithms in this family either learn a model of an environment's transition dynamics or make use of a known dynamics model. Once an agent has a model of the environment, $P(s' \mid s,a)$, it can "imagine" what will happen in the future by predicting the trajectory for a few time steps. If the environment is in state s, an agent can estimate how the state will change if it makes a sequence of actions $a_1, a_2, \ldots, a_n$ by repeatedly applying $P(s' \mid s,a)$, all without actually producing an action to change the environment. Hence, the predicted trajectory occurs in the agent's "head" using a model. An agent can complete many different trajectory predictions with different actions sequences, then examine these options to decide on the best action a to actually take.

Purely model-based approaches are most commonly applied to games with a target state, such as winning or losing in a game of chess, or navigation tasks with a goal state s^*.

4. Global convergence guarantee is still an open problem. Recently, it was proven for a subclass of problems known as linearized control. See the paper "Global Convergence of Policy Gradient Methods for Linearized Control Problems" by Fazel et al. (2018) [38].