Book draft

Mathematical Foundation of Reinforcement Learning

Shiyu Zhao

August 2022

Contents

Preface

This is a draft of a new book entitled "Mathematical Foundation of Reinforcement Learning". While there are many excellent studying materials, why do I write a new book on RL?

I have been teaching a graduate-level course on RL for three years. I will teach the course for the fourth time in the fall of 2022. Along with the teaching, I have been preparing this book as the lecture notes for my students. The main reason for me to write this book and develop this course is that I personally think the existing studying materials for RL are either too intuitive or too mathematical.

There are many excellent studying materials that introduce RL topics in intuitive ways. In these materials, the mathematics behind these topics is kept to a minimal level to adapt for a broader readership. Intuitive introductions are good in the sense that readers can grasp the ideas of a topic quickly. However, if the readers would like to understand a topic better, they have to dig out the mathematics that is scattered in technical papers and other materials, which is a huge barrier to their study. On the other hand, there are also many excellent mathematical introductions to RL. These materials, however, usually involve intense mathematics and may require the readers to have professional background on, for example, control theories.

This book aims to provide a mathematical but friendly introduction to the fundamental concepts, basic problems, and classical algorithms in RL. Some important features of this book are highlighted as follows.

1) The book introduces RL topics from a mathematical point of view in the hope that readers can better understand the mathematical root of an algorithm and hence why this algorithm is designed in the first place and why it works.
2) The depth of the mathematics is carefully controlled to an adequate level. The ways that the mathematics is presented are also carefully designed to ensure the book is friendly to read.
3) Many illustrative examples are given to help the readers better understand the topics. All the examples in this book are based on the grid-world task, which is very easy to understand and helpful in illustrating new concepts and algorithms.
4) When introducing an algorithm, the book aims to separate its core idea from the complications that may distract the readers. In this way, I hope that the readers can

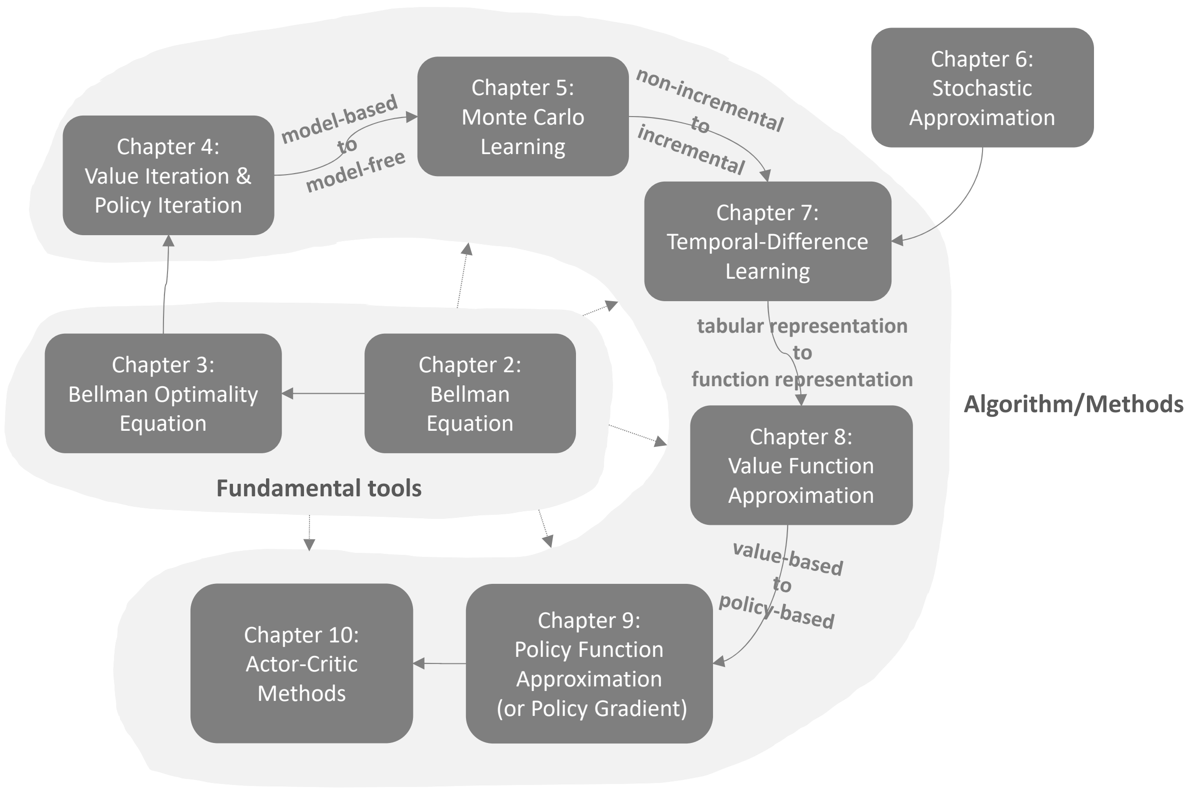


Figure 1: Relationship among the contents in different chapters. Chapter 2 introduces the Bellman equation, which is a fundamental tool for analyzing state values. Chapter 3 introduces the Bellman optimality equation, which is a special Bellman equation. Chapter 4 introduces the value iteration algorithm, which is an algorithm solving the Bellman optimality equation. Chapter 5 introduces Monte Carlo learning, which is an extension of the policy iteration algorithm introduced in Chapter 4. Chapter 6 introduces the basics of stochastic approximation, which lays a foundation for introducing temporal-difference learning in Chapter 7. Chapter 8 extends the tabular temporal-difference learning methods to the case of value function approximation. While Chapter 9 switches to policy iteration, Chapter 10 introduces actor-critic methods, which are a combination of the contents in Chapter 8 and Chapter 9.

better grasp the core idea of an algorithm.

5) This book includes a Q&A section at the end of each chapter. This is motivated by the frequently asked questions on the Internet. I also sometimes contribute to answering some questions online. Although the answers to many frequently asked questions can be found in the main text of the book, they may not be easy to find. Therefore, I believe it is beneficial to list these questions and answers explicitly.

6) The contents of the book are organized coherently. Each chapter is built based on the preceding chapter and lays a necessary foundation for the consequent chapters. The relationship among the contents of different chapters are shown in Figure 1.

This book is aimed at senior undergraduate students, graduate students, researchers, and practitioners who are interested in RL. It does NOT require the readers have any background on RL because it starts by introducing the very basic concepts of RL. However, if the readers already have some background in RL, I believe the book can also help them to understand some topics deeper or provide them with different perspectives.

This book, however, requires the readers to have some knowledge of probability theory and linear algebra. Some basics of the required mathematics are also included in the appendix of this book.

This book has not been finalized yet. A few more chapters will be added. The slides and videos for my course will also be uploaded online. I am collecting feedback about this book. Any advice from the readers will be appreciated. Any feedback can be sent to zhaoshiyu@westlake.edu.cn.

Chapter 1

Basic Concepts

This chapter introduces the basic concepts of reinforcement learning (RL) that will be used throughout this book. We first introduce these concepts by using examples and then formalize them in the context of the Markov decision processes.

1.1 A grid-world example

Consider the example shown in Figure 1.1, where a robot moves in a grid world. The robot or called *agent* can move across adjacent cells in the grid. Each time it can only occupy a single cell. The white cells are *accessible* and the orange ones are *forbidden* to enter. There is a *target* cell that the robot would like to reach. We will use such kind of grid world examples throughout this book considering that they are intuitive to illustrate new concepts and algorithms.

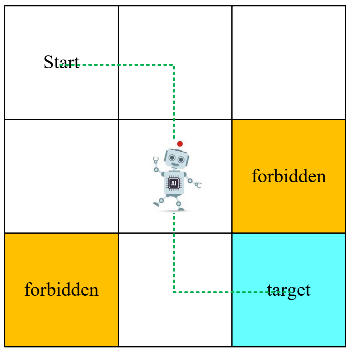


Figure 1.1: The grid-world example used throughout the book.

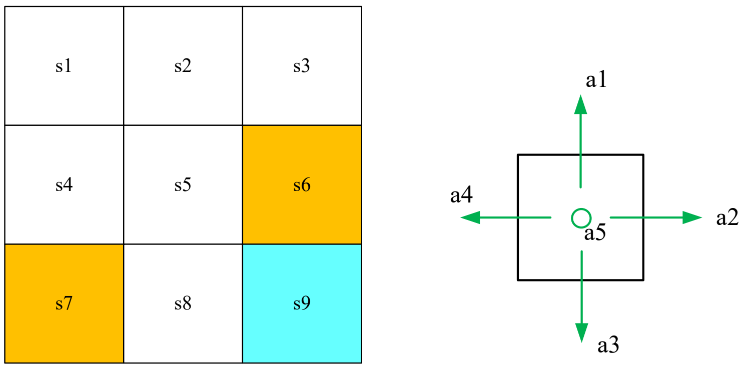
The ultimate goal of the agent is to find a "good" policy to reach the target cell starting from any initial cell. How to define the goodness of a policy? The intuition is that the agent should reach the target without entering the forbidden cells, taking unnecessary detours, or colliding with the boundary of the grid.

It would be trivial to plan a path to reach the target cell if the agent knows the map of the grid world. The task becomes nontrivial if the agent does not know any information about the environment in advance. Then, the agent has to interact with the environment to find a good policy to reach the target cell by trial-and-error. To do that, we introduce some necessary concepts in the rest of the chapter.

1.2 State and action

The first concept to be introduced is called *state*, which describes the status of the agent with respect to the environment. In the grid-world example, the state corresponds to the location of the agent. Since there are nine possible locations/cells, there are nine states as well. They are indexed as $s_1, s_2, \ldots, s_9$. The set of all the states is called *state space*, denoted as $\mathcal{S} = \{s_1, \ldots, s_9\}$ (see Figure 1.2(a)).

For each state, the agent has five possible *actions* to take: move upwards, rightwards, downwards, leftwards, and stay unchanged. The five actions are denoted as $a_1, a_2, \ldots, a_5$, respectively (see Figure 1.2(b)). The set of all the actions is called *action space*. In particular, the action space of s_i is denoted as $\mathcal{A}(s_i)$. Here, we suppose that different states have the same action space: $\mathcal{A}(s_i) = \mathcal{A} = \{a_1, \ldots, a_5\}$.



(a) States (b) Actions

Figure 1.2: (a) In the grid-world example, there are nine states $\{s_1, \ldots, s_9\}$. (b) Each state has five possible actions $\{a_1, a_2, a_3, a_4, a_5\}$.

It should be noted that different states may have different action spaces in general. For instance, taking a_1 or a_4 at state s_1 would collide with the boundary of the grid. Then, we can remove these two actions from the state space of s_1 and hence $\mathcal{A}(s_1) = \{a_2, a_3, a_5\}$. Removing useless actions from the action space is favorable to simplify policy search processes in RL. However, such removal is based on the knowledge that some actions are not good. Such knowledge sometimes can be obtained easily and sometimes must be learned. In fact, the basic purpose of RL is to find out which actions are good and which are not. In this book, we consider the most general and challenging case that no actions are removed from the action space.

1.3 State transition

When taking an action, the agent may move from one state to another. Such a process is called *state transition*. For example, if the agent is at state s_1 and takes action a_2 (that is to move rightwards), then the agent would move to state s_2. Such a process can be expressed as

$$s_1 \xrightarrow{a_2} s_2.$$

What is the next state when the agent attempts to go out of the boundary, for example, taking action a_1 at state s_1? The answer is that the agent will get bounced back because it is impossible for the agent getting out of the state space. Hence, we have $s_1 \xrightarrow{a_1} s_1$.

What is the next state when the agent attempts to enter the forbidden cells, for example, taking action a_2 at state s_5? There are two different scenarios. The first is that, although s_6 is forbidden, it is still accessible but with a severe penalty. In this case, the next state is s_6 and hence the state transition process is $s_5 \xrightarrow{a_2} s_6$. The second scenario is that s_6 is physically not assessable. For example, it is surrounded by walls. The agent will get bounced back to s_5 if it attempts to move rightwards. In this case, the state transition is $s_5 \xrightarrow{a_2} s_5$. Which scenario should we consider? The answer depends on the physical environment. Since the grid-world example is based on numerical simulation, we can choose either. Here, we consider the first scenario that the forbidden cells are still accessible, though stepping into them may get punished. This scenario is more general and interesting.

Since we are considering a simulation task, we can define the state transition process whatever we like. If the system involves physical interaction, the state transition process would be determined by physical laws.

How to describe the state transition process? The state transition process is defined for each state and their associated actions. Therefore, it can be represented using a table as in Table 1.1. In this table, each row represents a state and each column represents an action.

	a_1 (upwards)	a_2 (rightwards)	a_3 (downwards)	a_4 (leftwards)	a_5 (unchanged)
s_1	s_1	s_2	s_4	s_1	s_1
s_2	s_2	s_3	s_5	s_1	s_2
s_3	s_3	s_3	s_6	s_2	s_3
s_4	s_1	s_5	s_7	s_4	s_4
s_5	s_2	s_6	s_8	s_4	s_5
s_6	s_3	s_6	s_9	s_5	s_6
s_7	s_4	s_8	s_7	s_7	s_7
s_8	s_5	s_9	s_8	s_7	s_8
s_9	s_6	s_9	s_9	s_8	s_9

Table 1.1: A tabular representation of state transition. Each cell indicates the next state to transit to after the agent taking an action at a state.

Though intuitive, the tabular representation is only able to describe deterministic state transition processes. It is necessary to use rigorous mathematics to describe state transition in a general way. For example, for s_1 and a_2, the conditional probability distribution is

$$\begin{aligned}
p(s_1|s_1, a_2) &= 0,\\
p(s_2|s_1, a_2) &= 1,\\
p(s_3|s_1, a_2) &= 0,\\
p(s_4|s_1, a_2) &= 0,\\
p(s_5|s_1, a_2) &= 0,
\end{aligned}$$

which indicates that, if taking a_2 at s_1, the probability for the agent to move to s_2 is one and the others is zero. As a result, taking action a_2 at s_1 will certainly take the agent to s_2. Preliminaries of conditional probability are given in the appendix. Readers are strongly advised to get familiar with probability theory first because it is necessary to study RL and will be used throughout this book.

It is noted that the state transition in the above example is *deterministic*. In general, state transition can be *stochastic*. For instance, if there are random wind gusts across the grid, when taking action a_2 at s_1, the agent may be blew to s_5 instead of s_2. In this case, we have $p(s_5|s_1, a_2) > 0$. However, we merely consider the deterministic case for the sake of simplicity.

1.4 Policy

A *policy* tells the agent which actions to take at each state.

Intuitively, policies can be depicted as arrows as shown in Figure 1.3. Following the policy starting from different initial states, the agent would generate some trajectories across different states as shown in Figure 1.3.

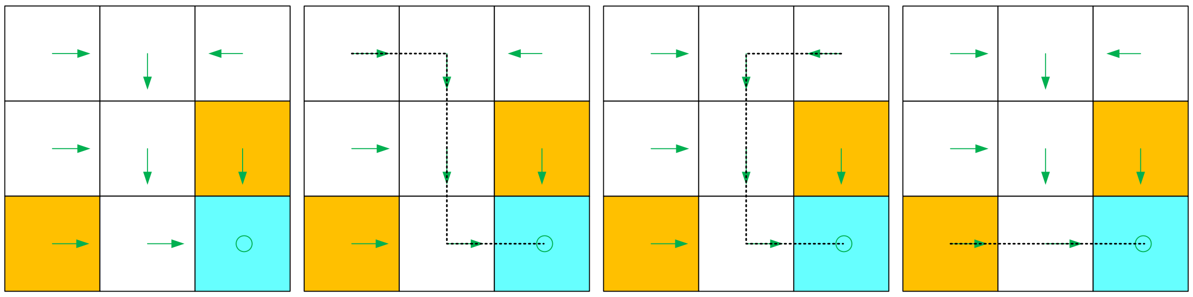


Figure 1.3: A policy represented by arrows and some trajectories obtained starting from different states.

Mathematically, policies can be described by conditional probabilities. Denote the policy in Figure 1.3 as $\pi(a|s)$, which is a conditional probability distribution function defined for *every* state-action pair. For example, the policy for s_1 as shown in Figure 1.3

is

$$
\begin{aligned}
\pi(a_1|s_1) &= 0,\\
\pi(a_2|s_1) &= 1,\\
\pi(a_3|s_1) &= 0,\\
\pi(a_4|s_1) &= 0,\\
\pi(a_5|s_1) &= 0,
\end{aligned}
$$

which indicates that the probability of taking action a_2 at state s_1 is one and the others zero. Therefore, the agent would definitely select action a_2 at s_1.

The above policy is *deterministic*. Policies may be *stochastic* in general. For example, the policy shown in Figure 1.4 is stochastic: at state s_1, the agent may take actions to go either rightwards or downwards. The probabilities of taking the two actions are the same as 0.5. In this case, the policy is

$$
\begin{aligned}
\pi(a_1|s_1) &= 0,\\
\pi(a_2|s_1) &= 0.5,\\
\pi(a_3|s_1) &= 0.5,\\
\pi(a_4|s_1) &= 0,\\
\pi(a_5|s_1) &= 0.
\end{aligned}
$$

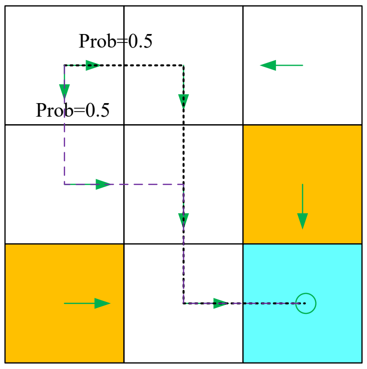


Figure 1.4: A stochastic policy. At state s_1, the agent may move rightwards or downwards with the equal probability of 0.5.

Policies represented by conditional probabilities can be intuitively described as tables. For example, Table 1.2 represents the stochastic policy depicted in Figure 1.4. The entry at the ith row and jth column is the probability to take the jth action at the ith state. Such kind of representation is called *tabular representation*, which will be used throughout this book. However, towards the end of this book, we will introduce another way to represent policies as parameterized functions instead of tables.

	a_1 (upwards)	a_2 (rightwards)	a_3 (downwards)	a_4 (leftwards)	a_5 (unchanged)
s_1	0	0.5	0.5	0	0
s_2	0	0	1	0	0
s_3	0	0	0	1	0
s_4	0	1	0	0	0
s_5	0	0	1	0	0
s_6	0	0	1	0	0
s_7	0	1	0	0	0
s_8	0	1	0	0	0
s_9	0	0	0	0	1

Table 1.2: A tabular representation of a policy. Each entry indicates the probability of taking an action at a state.

1.5 Reward

Reward is one of the most unique concepts in RL.

After taking an action, the agent would get a reward as feedback from the environment. The reward can be a positive, negative, or zero real number. Different rewards have different impacts on the policy that the agent would eventually learn. In particular, by a positive reward, we encourage the agent to take that action. By a negative reward, we discourage the agent to take that action. What if the agent receives a zero reward after taking an action? A zero reward can be interpreted as encouragement because the agent loses nothing if taking that action.

In the grid-world example, the rewards are designed as follows:

– If the agent attempts to get out of the boundary, let $r_{\text{boundary}} = -1$.

– If the agent attempts to enter a forbidden cell, let $r_{\text{forbidden}} = -1$.

– If the agent reaches the target cell, let $r_{\text{target}} = +1$.

– Otherwise, the agent gets a reward of $r = 0$.

Reward can be interpreted as a human-machine interface, with which we can guide the agent to behave as we expect. For example, with the above designed rewards, we can expect that the agent will try to avoid getting out of the boundary or stepping into the forbidden cells. The basic idea behind it is that optimal policies aim to collect rewards as great as possible. At this moment, it is still unclear how to define the optimality of policies. It will be discussed when we study the Bellman optimality equation.

Designing appropriate rewards is an important step of RL. This step is, however, nontrivial for complex tasks since it may require us to have a good understanding of the problem. Nevertheless, designing rewards and using RL to solve a complex problem may still be much easier than solving the problem using other approaches that require an even deeper understanding of the problem and professional background. This is one of the reasons why RL can attract a more broad range of users.

Reward transition is the process of getting a reward after taking an action. This process can be intuitively represented as a table as shown in Table 1.3. Each row of the table corresponds to a state and each column corresponds to an action. The value in each cell of the table indicates the reward that can be obtained by taking an action at a state.

One question that beginners may ask is, if given the table of rewards, whether we can find good policies by simply selecting the actions with the greatest rewards. The answer is no. That is because the above rewards are *immediate rewards* that can be obtained after taking an action. In order to determine a good policy, we need to consider the total reward obtained in a long run (see next section). It is often the case that an action that has the greatest immediate reward may not lead to the greatest total reward.

	a_1 (upwards)	a_2 (rightwards)	a_3 (downwards)	a_4 (leftwards)	a_5 (unchanged)
s_1	r_{boundary}	0	0	r_{boundary}	0
s_2	r_{boundary}	0	0	0	0
s_3	r_{boundary}	r_{boundary}	$r_{\text{forbidden}}$	0	0
s_4	0	0	$r_{\text{forbidden}}$	r_{boundary}	0
s_5	0	$r_{\text{forbidden}}$	0	0	0
s_6	0	r_{boundary}	r_{target}	0	$r_{\text{forbidden}}$
s_7	0	0	r_{boundary}	r_{boundary}	$r_{\text{forbidden}}$
s_8	0	r_{target}	r_{boundary}	$r_{\text{forbidden}}$	0
s_9	$r_{\text{forbidden}}$	r_{boundary}	r_{boundary}	0	r_{target}

Table 1.3: A tabular representation of reward transition. Here, the reward transition is deterministic and each cell indicates how much reward can be obtained after the agent taking an action at a given state.

Though intuitive, the tabular representation is only able to describe deterministic reward transition processes. A more powerful way is to use conditional probabilities $p(r|s,a)$ to describe general reward transition. For example, for state s_1, we have

$$p(r=-1|s_1,a_2)=1, \quad p(r\neq -1|s_1,a_2)=0,$$

which indicates that, if taking a_2 at s_1, it is certain that the agent would get $r=-1$. Again, in this example, the reward transition process is deterministic. In general, it can be stochastic. For example, if a student studies hard, he/she would get a positive reward in general (for example, higher grades in exams), but how much is uncertain.

One problem that may confuse beginners is whether a reward should depend on the action taken or the next state reached. A mathematical rephrase of this problem is whether we should use $p(r|s,a)$ or $p(r|s,s')$ to represent reward transition. Here, s' is the next state reached after leaving the state s. In the grid-world examples, since the state transition is deterministic, the two ways happen to be equivalent. For example, $p(r|s_1,a_2)=p(r|s_1,s_2)$ since the agent taking a_2 at s_1 will certainly lead to s_2. However, when the state transition is stochastic, the two ways are not equivalent anymore. More importantly, we should always use $p(r|s,a)$ instead of $p(r|s,s')$ because we want to punish or encourage an action instead of the next state. For example, suppose the current state is s_1. Although taking actions a_1 and a_5 will both lead to the next state as s_1, taking a_1 is worse than a_5 because a_1 attempts to collide to the boundary and should be given negative rewards.

1.6 Trajectory, return, and episode

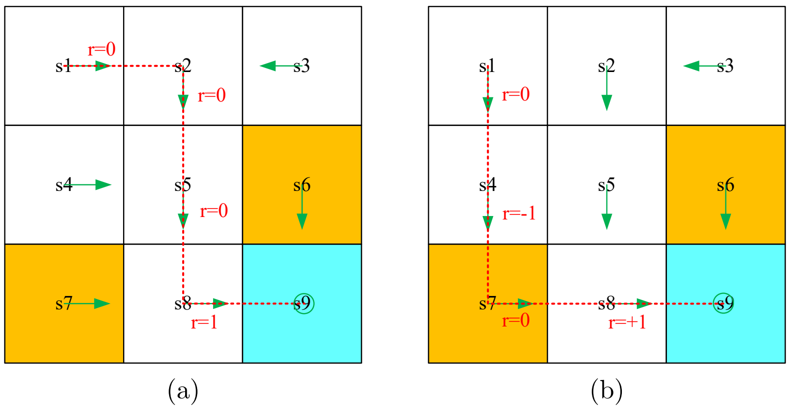


Figure 1.5: Trajectories obtained following two policies. The trajectories are indicated by red dashed lines.

A *trajectory* is a state-action-reward chain.

For example, given the policy shown in Figure 1.5(a), starting from s_1, the agent follows a trajectory as

$$s_1 \xrightarrow[r=0]{a_2} s_2 \xrightarrow[r=0]{a_3} s_5 \xrightarrow[r=0]{a_3} s_8 \xrightarrow[r=1]{a_2} s_9.$$

The *return* of this trajectory is the sum of all the rewards collected along the trajectory:

$$\text{return} = 0 + 0 + 0 + 1 = 1. \tag{1.1}$$

Return is also sometimes called *total rewards* or *cumulative rewards.*

Return can be used to evaluate the "goodness" of policies. For example, we can compare the two policies in Figure 1.5 by comparing their returns. In particular, starting from s_1, the return obtained by the left policy is 1 as calculated above. For the right policy, starting from s_1 gives the trajectory as

$$s_1 \xrightarrow[r=0]{a_3} s_4 \xrightarrow[r=-1]{a_3} s_7 \xrightarrow[r=0]{a_2} s_8 \xrightarrow[r=+1]{a_2} s_9.$$

The corresponding return is:

$$\text{return} = 0 - 1 + 0 + 1 = 0.$$

The values of the returns for the two policies indicate that the left policy is better than the right one since its return is greater. This mathematical conclusion is also consistent with the intuition that the right policy is not good since it passes through a forbidden cell.

While reward only reflects the encouragement or discouragement for taking a single action, return can be used to evaluate a sequence of actions in a long run. A return consists of the *immediate reward* and the *delayed reward*. Here, the immediate reward is the reward obtained after taking an action at the initial state; the delayed reward is the sum of the rewards obtained after leaving the initial state. Although the immediate reward may be negative, the delayed reward may be positive. Thus, which actions to take should be determined by the return (i.e., total reward) rather than the immediate reward to avoid short-sighted decisions. As we will see in the next chapter, return plays an important role to evaluate different policies.

The return in (1.1) is defined for a finite-length trajectory. Return can also be defined for infinitely long trajectories. For example, the trajectory in Figure 1.5 stops after reaching s_9. Such a stop relies on a stop criterion: that is, the agent stops moving after reaching the target state. Since the policy is also well defined for the target state s_9, the agent does not have to stop after reaching s_9. Then, we obtain the following infinitely long trajectory:

$$s_1 \xrightarrow[r=0]{a_2} s_2 \xrightarrow[r=0]{a_3} s_5 \xrightarrow[r=0]{a_3} s_8 \xrightarrow[r=1]{a_2} s_9 \xrightarrow[r=1]{a_5} s_9 \xrightarrow[r=1]{a_5} s_9 \dots$$

The direct summation of the rewards obtained along this trajectory is

$$\text{return} = 0 + 0 + 0 + 1 + 1 + 1 + \cdots = \infty,$$

which unfortunately diverges. Therefore, we need to introduce the concept of *discounted return* for infinitely long trajectories. In particular, the discounted return is the sum of the discounted rewards:

$$\text{discounted return} = 0 + \gamma 0 + \gamma^2 0 + \gamma^3 1 + \gamma^4 1 + \gamma^5 1 + \dots, \tag{1.2}$$

where $\gamma \in (0, 1)$ is called the *discount rate*. When $\gamma \in (0, 1)$, (1.2) is finite and can be calculated as

$$\text{discounted return} = \gamma^3(1 + \gamma + \gamma^2 + \dots) = \gamma^3 \frac{1}{1-\gamma}.$$

The reason that we consider discounted return is twofold. First, it removes the stop criterion and the mathematics is more elegant. Second, the discount rate can be used to adjust the emphasis on near or far future rewards. In particular, if γ is close to 0, then the user put more emphasis on the reward obtained in the near future. The resulting policy would be short-sighted. If γ is close to 1, then the agent put more emphasis on the far future rewards. In this case, the resulting policy would dare to take risks of getting negative rewards in the near future. These points can be well demonstrated later by the examples in Section 3.5 in Chapter 3.

One important notion that was not explicitly mentioned in the above discussion is *episode*. When interacting with the environment following a policy, the agent may stop at some *terminal states*. The resulting trajectory is called an *episode* or a *trial*. If the environment or policy is stochastic, we would obtain different episodes starting from the same state. However, if everything is deterministic, we may always obtain the same episode starting from the same state.

An episode is usually assumed to be a finite trajectory. Tasks with episodes are called *episodic tasks*. However, some tasks may have no terminal states, meaning the interaction with the environment will never end. Such tasks are called *continuing tasks*. In fact, we can treat episodic and continuing tasks in a unified mathematical way by converting episodic tasks to continuing tasks. The key is to well define the process after reaching the target/terminal state. Specifically, after reaching the target or terminal state in an episodic task, the agent can continue taking actions. We can treat the target/terminal state in two ways.

First, if we treat it as a special state, we can specially design its action space or state transition such that the agent stays at this state forever. Such states are called *absorbing states*, meaning that the agent would never leave the state once it is reached. For example, for the target state s_9, we can specify $\mathcal{A}(s_9) = \{a_5\}$ or set $\mathcal{A}(s_9) = \{a_1, \ldots, a_5\}$ but $p(s_9|s_9, a_i) = 1$ for all $i = 1, \ldots, 5$. We can also set the reward obtained after reaching s_9 as always zero.

Second, if we treat the target state as a normal one, we can simply set its action space the same as others and the agent may leave the state. Since a positive reward of $r = 1$ can be obtained every time s_9 is reached, the agent will eventually learn to stay at s_9 forever to collect more rewards. Of course, when the episode is infinitely long and the reward of staying at s_9 is positive, a discount rate must be used to calculate the discounted return to avoid divergence. In this book, we consider the second scenario where the target state is treated as a normal state whose action space is $\mathcal{A}(s_9) = \{a_1, \ldots, a_5\}$.

1.7 Markov decision process

The previous sections of this chapter have illustrated some fundamental concepts in RL by examples. This section presents these concepts in a more formal way under the framework of the Markov decision processes (MDP).

MDP is a general framework to describe stochastic dynamical systems. The key ingredients of an MDP are listed below.

– Sets:

- State set: the set of all states, denoted as $\mathcal{S}$.
- Action set: a set of actions, denoted as $\mathcal{A}(s)$, is associated for each state $s \in \mathcal{S}$.
- Reward set: a set of rewards, denoted as $\mathcal{R}(s, a)$, is associated for each state action pair (s, a).

– Model:

  - State transition probability: at state s, taking action a, the probability to transit to state s' is $p(s'|s, a)$.
  - Reward transition probability: at state s, taking action a, the probability to get reward r is $p(r|s, a)$.

– Policy: at state s, the probability to choose action a is $\pi(a|s)$.

– Markov property: One key property of MDPs is the *Markov property*, which refers to the memoryless property of a stochastic process. Mathematically, it means

$$\begin{aligned} p(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0, a_0) &= p(s_{t+1}|s_t, a_t), \\ p(r_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0, a_0) &= p(r_{t+1}|s_t, a_t). \end{aligned} \tag{1.3}$$

That is, the next state or reward merely depends on the present state and action rather than the previous states or actions. Equation (1.3) indicates the property of *conditional independence* of random variables. Preliminaries to the probability theory can be found in the appendix. Markov property is important for deriving the fundamental Bellman equation of MDPs as shown in the next chapter.

The state transition, reward transition, and policy can be all stochastic, although they are sometimes deterministic in our illustrative examples. Here, $p(s'|s, a)$ and $p(r|s, a)$ for all s, a are called the *model* or *dynamics* of an MDP. We will show later in the book that there are *model-based* and *model-free* RL algorithms. Moreover, the model can be either *stationary* or *nonstatinary*, or in other words, time-variant or time-invariant. In stationary environments, the models do not change over time; in nonstationary environments, the models may vary over time. For instance, in the example of grid world, if some forbidden areas may pop up or disappear in the grid, such an environment is nonstationary.

The reader may have also heard about the Markov process (MP). What is the difference between MDP and MP? The answer is that, once the policy in an MDP is fixed, the MDP degenerates to an MP. In the literature on stochastic processes, a Markov process is also called a Markov chain if it is discrete-time and the number of states is finite or countable [1]. In our book, the Markov process and Markov chain are used interchangeably when the context is clear. In the first half of this book, we consider *finite* MDPs where the numbers of states, actions, and reward values are all finite. This is the simplest case that should be well understood first of all. Towards the end of this book, we will consider the case of *infinite* states or actions.

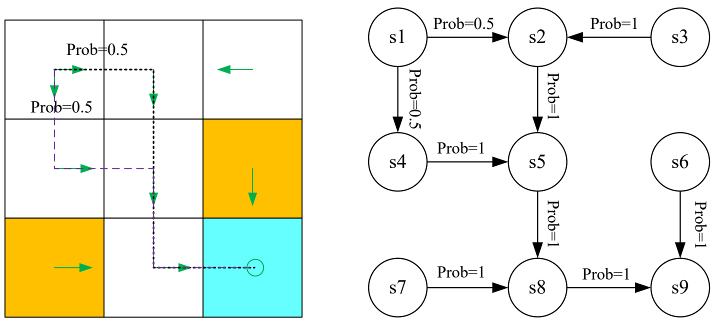
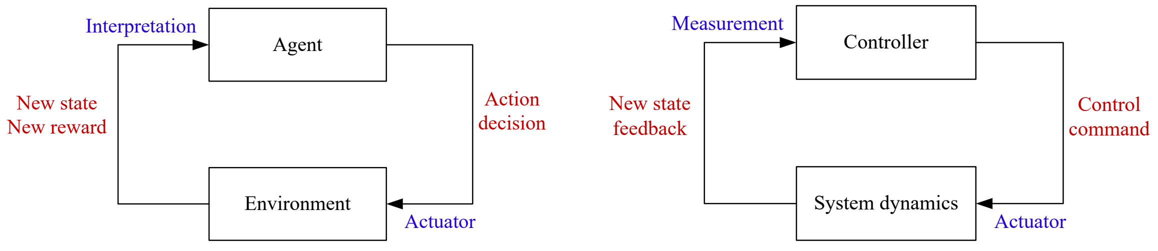


Figure 1.6: Abstraction of the grid-world example as a Markov process.

The grid-world example in Figure 1.6 can be abstracted as a *Markov decision process* (MDP), where the circles represent states and the links with arrows represent the state transition.

Finally, RL can be described as an agent-environment interaction process (see Figure 1.7). The *agent* is a decision-maker that can sense its own state, maintain policies, and execute actions. Everything outside of the agent is regarded as the *environment*. In the grid-world examples, the agent and environment refer to the robot and grid world, respectively. As depicted by Figure 1.7, after the agent decides to take an action, the actuator would execute such a decision. Then, the state of the agent would be changed and a reward can be obtained. By using interpreters, the agent can interpret the new state and reward to make the next decision. Thus, a closed-loop is formed.

If the reader is familiar with control theory, the agent-environment paradigm of RL is very similar to that of control systems. See Figure 1.7. In addition to different names of the ingredients, the key difference between RL and automatic control is the reward, which is a favorable human-machine interface in RL.



(a) Diagram of RL (b) Diagram of control systems

Figure 1.7: Diagrams of RL and control systems.

1.8 Summary

This chapter introduced some basic concepts in RL. We used intuitive grid-world examples to demonstrate these concepts and then formalized them in the framework of MDP. In this chapter, we know that RL can be described as an agent-environment interaction process. The agent is a composition of a sensor, decision-maker, and actuator. That is, the agent is able to sense its own state, maintain a policy, and execute actions. The dynamics of the environment are described by state and reward probability transition probabilities. Through these probabilities, taking an action would change the state of the agent and, in the meantime, generate a reward signal. Such a reward is the feedback for the action taken and can be used by the agent to adjust its policy. For more information about the Markov decision processes, readers are advised to refer to [1, 2].

Chapter 2

State Value and Bellman Equation

This chapter will mainly introduce a core concept and an important tool to analyze the concept. The core concept is state value as well as action value. The important tool is the Bellman equation, which characterizes the relationship among the values of different states and can be used to calculate state values. The contents of this chapter are very fundamental in reinforcement learning (RL).

2.1 Motivating examples: Why return is important?

In the last chapter, we introduced the concept of return. In fact, return plays a fundamental role in RL. We next use examples to demonstrate.

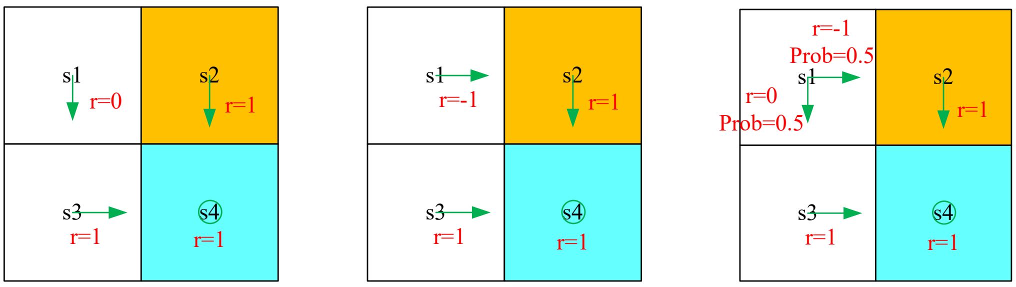


Figure 2.1: Examples to demonstrate the importance of return. The three examples have the same problem setup but different policies.

Consider the three policies as shown in Figure 2.1. The problem setup of the three examples is exactly the same, but the policies are different. Intuitively, the leftmost policy is the best because the agent starting from s_1 can avoid the forbidden area to reach the target. The middle policy is intuitively worse because the agent starting from s_1 moves to the forbidden area. The rightmost policy is in between because it has a probability of 0.5 to go rightwards to the forbidden area.

While the above analysis is based on intuition, a question that immediately follows is: can we use mathematics to describe such intuition? The answer relies on the notion of return.

Suppose the starting state is s_1. Following the first policy, the resulting trajectory is $s_1 \to s_3 \to s_4 \to s_4$. The corresponding discounted return is

$$\begin{aligned}\text{return}_1 &= 0 + \gamma 1 + \gamma^2 1 + \dots \\ &= \gamma(1 + \gamma + \gamma^2 + \dots) \\ &= \frac{\gamma}{1-\gamma}.\end{aligned}$$

where $\gamma \in (0,1)$ is the discount rate. Following the second policy, the trajectory is $s_1 \to s_2 \to s_4 \to s_4$. The discounted return is

$$\begin{aligned}\text{return}_2 &= -1 + \gamma 1 + \gamma^2 1 + \dots \\ &= -1 + \gamma(1 + \gamma + \gamma^2 + \dots) \\ &= -1 + \frac{\gamma}{1-\gamma}.\end{aligned}$$

Following the third policy, there are two possible trajectories. One is $s_1 \to s_3 \to s_4 \to s_4$ and the other is the trajectory is $s_1 \to s_2 \to s_4 \to s_4$. The probability of taking either of them is 0.5. Then, the average of the discounted returns that can be obtained starting from s_1 is

$$\begin{aligned}\text{return}_3 &= 0.5\left(-1 + \frac{\gamma}{1-\gamma}\right) + 0.5\left(\frac{\gamma}{1-\gamma}\right) \\ &= -0.5 + \frac{\gamma}{1-\gamma}.\end{aligned}$$

By comparing the returns of the three policies, we can easily see that

$$\text{return}_1 > \text{return}_3 > \text{return}_2$$

for any value of γ. The above inequality suggests that the first policy is the best and the second policy is the worst, which is exactly the same as our intuition.

The above example demonstrates that return can be used to evaluate different policies: a policy is better if the return obtained following that policy is greater. Of course, this is a naive idea. Such an idea will be formalized when we define state value functions. Finally, it is notable that return_3 does not comply with the definition of return strictly because it is more like an expected value. It will become clear that return_3 is actually a state value.

2.2 Motivating example: How to calculate return?

While we have demonstrated the importance of return, a question that immediately follows is how to calculate the return of a policy starting from different states. There are two ways to do that.

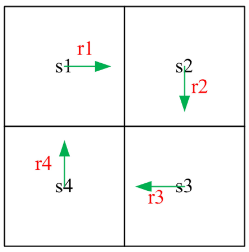


Figure 2.2: An example to demonstrate how to calculate return.

The first is simply by definition: return is defined as the discounted summation of rewards along a trajectory. Consider the example in Figure 2.2. Let v_i denote the return obtained starting from s_i for $i = 1, \dots, 4$. Then, the returns starting from the four states in Figure 2.2 can be respectively calculated as

$$\begin{aligned} v_1 &= r_1 + \gamma r_2 + \gamma^2 r_3 + \dots, \\ v_2 &= r_2 + \gamma r_3 + \gamma^2 r_4 + \dots, \\ v_3 &= r_3 + \gamma r_4 + \gamma^2 r_1 + \dots, \\ v_4 &= r_4 + \gamma r_1 + \gamma^2 r_2 + \dots. \end{aligned} \tag{2.1}$$

The second way is by the idea of *bootstrapping*. By observing the expressions of the returns in (2.1), we can rewrite it to

$$\begin{aligned} v_1 &= r_1 + \gamma(r_2 + \gamma r_3 + \dots) = r_1 + \gamma v_2, \\ v_2 &= r_2 + \gamma(r_3 + \gamma r_4 + \dots) = r_2 + \gamma v_3, \\ v_3 &= r_3 + \gamma(r_4 + \gamma r_1 + \dots) = r_3 + \gamma v_4, \\ v_4 &= r_4 + \gamma(r_1 + \gamma r_2 + \dots) = r_4 + \gamma v_1. \end{aligned} \tag{2.2}$$

The above equations indicate that the calculation of v_1 relies on the value of v_2. Similarly, v_2 relies on v_3, v_3 relies on v_4, and finally v_4 relies on v_1. This reflects the idea of bootstrapping, which is to obtain something from itself.

At first glance, bootstrapping is an endless loop because the calculation of an unknown value relies on another unknown value. In fact, bootstrapping is easier to understand if we view it from a mathematical perspective. In particular, the equations in (2.2) can be reformed to a linear matrix-vector equation:

$$\underbrace{\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}}_{\mathbf{v}} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \end{bmatrix} + \begin{bmatrix} \gamma v_2 \\ \gamma v_3 \\ \gamma v_4 \\ \gamma v_1 \end{bmatrix} = \underbrace{\begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \end{bmatrix}}_{\mathbf{r}} + \gamma \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}}_{\mathbf{P}} \underbrace{\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}}_{\mathbf{v}},$$

which can be written in short as

$$\mathbf{v} = \mathbf{r} + \gamma \mathbf{P}\mathbf{v}.$$

Thus, the value of $\mathbf{v}$ can be calculated easily as $\mathbf{v} = (\mathbf{I} - \gamma \mathbf{P})^{-1}\mathbf{r}$.

In fact, (2.2) is the Bellman equation for this simple example. Though simple, (2.2) demonstrates the core idea of the Bellman equation: that is, the return obtained starting from one state depends on those starting from other states. The idea of bootstrapping and the Bellman equation for general scenarios will be formalized in the following sections.

2.3 State value

Although return can be used to evaluate if a policy is good or not, it does not apply to stochastic systems because starting from a state may lead to different trajectories and hence different returns. Motivated by this problem, we define the *mean* of all possible returns starting from a state as the *state value.*

The mathematical definition of state value is derived as follows. First of all, we need to introduce some necessary notations. Consider a sequence of time steps $t = 0, 1, 2, \ldots$. At time t, the agent is at state S_t and the action taken following a policy π is A_t. The next state is S_{t+1} and the immediate reward obtained is R_{t+1}. This process can be expressed concisely as

$$S_t \xrightarrow{A_t} S_{t+1}, R_{t+1}$$

Note that $S_t, S_{t+1}, A_t, R_{t+1}$ are all *random variables.* In particular, $S_t, S_{t+1} \in \mathcal{S}$, $A_t \in \mathcal{A}(S_t)$, and $R_{t+1} \in \mathcal{R}(S_t, A_t)$. It is worth mentioning that the reward obtained after the agent takes action A_t can be also denoted R_t instead of R_{t+1}. Mathematically, it does not make any difference.

Starting from t, we can obtain a state-action-reward trajectory:

$$S_t \xrightarrow{A_t} S_{t+1}, R_{t+1} \xrightarrow{A_{t+1}} S_{t+2}, R_{t+2} \xrightarrow{A_{t+2}} S_{t+3}, R_{t+3} \ldots.$$

By definition, the discounted return along the trajectory is

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots$$

where $\gamma \in (0, 1)$ is a discount rate. Note that G_t is a random variable since $R_{t+1}, R_{t+2}, \ldots$ are all random variables.

Since G_t is a random variable, we can calculate its expectation (or called expected value or mean):

$$v_\pi(s) \doteq \mathbb{E}[G_t | S_t = s]$$

Here, $v_\pi(s)$ is called the *state-value function* or simply *state value* of s. Some important remarks are given below.

– $v_\pi(s)$ depends on s because its definition is a conditional expectation with the condition that the state starts from $S_t = s$. As its name suggests, it represents the "value" of a state, which is the expected value of the rewards that can be obtained starting from s.

– $v_\pi(s)$ depends on π because the trajectories are generated by following the policy π. For a different policy, the state value may be different.

The relationship between state value and return is further clarified as follows. When everything (system model and policy) is deterministic, the value of a state is equal to the return obtained starting from that state. In the presence of randomness, the returns of different trajectories would be different. In this case, the state value is the mean of these returns. While Section 2.1 demonstrates that return can be used to evaluate policies, a more general way is to use state value to evaluate policies: policies generating greater state values are better. More details about optimal policies will be given in the next chapter.

2.4 Bellman equation

We now introduce the Bellman equation, a mathematical tool to analyze state values. In one word, the Bellman equation is a set of linear equations describing the relationship among the values of all the states.

We next derive the Bellman equation. First of all, note that G_t can be rewritten as

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) \\ &= R_{t+1} + \gamma G_{t+1}, \end{aligned}$$

where $G_{t+1} = R_{t+2} + \gamma R_{t+3} + \dots$. This equation establishes the relationship between G_t and G_{t+1}. Then, the state value can be written as

$$\begin{aligned} v_\pi(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}[R_{t+1}|S_t = s] + \gamma\mathbb{E}[G_{t+1}|S_t = s]. \end{aligned} \tag{2.3}$$

The first term, $\mathbb{E}[R_{t+1}|S_t = s]$, in (2.3) is the expectation of the immediate reward that can be obtained starting from s. By using the law of total expectation, it can be calculated as

$$\begin{aligned} \mathbb{E}[R_{t+1}|S_t = s] &= \sum_a \pi(a|s)\mathbb{E}[R_{t+1}|S_t = s, A_t = a] \\ &= \sum_a \pi(a|s) \sum_r p(r|s,a)r. \end{aligned} \tag{2.4}$$

The second term, $\mathbb{E}[G_{t+1}|S_t = s]$, in (2.3) is the expectation of the future reward. It can be calculated as

$$\begin{aligned}
\mathbb{E}[G_{t+1}|S_t = s] &= \sum_{s'} \mathbb{E}[G_{t+1}|S_t = s, S_{t+1} = s']p(s'|s) \\
&= \sum_{s'} \mathbb{E}[G_{t+1}|S_{t+1} = s']p(s'|s) \qquad \text{(due to conditional independence)} \\
&= \sum_{s'} v_\pi(s')p(s'|s) \\
&= \sum_{s'} v_\pi(s') \sum_a p(s'|s,a)\pi(a|s). \qquad (2.5)
\end{aligned}$$

The above derivation uses the fact that $\mathbb{E}[G_{t+1}|S_t = s, S_{t+1} = s'] = \mathbb{E}[G_{t+1}|S_{t+1} = s']$. This is due to the conditional independence property thanks to the memoryless Markov property that the future behavior totally depends on the present state. Substituting (2.4)-(2.5) into (2.3) yields

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}[R_{t+1}|S_t = s] + \gamma\mathbb{E}[G_{t+1}|S_t = s], \\
&= \underbrace{\sum_a \pi(a|s) \sum_r p(r|s,a)r}_{\text{mean of immediate rewards}} + \underbrace{\gamma \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)v_\pi(s')}_{\text{mean of future rewards}}, \\
&= \sum_a \pi(a|s)\left[\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a)v_\pi(s')\right], \quad \text{for all } s \in \mathcal{S}. \qquad (2.6)
\end{aligned}$$

This equation is called the *Bellman equation*, which characterizes the relationship of state values. It is a fundamental tool for designing and analyzing RL algorithms.

At first glance, the Bellman equation is quite complex. In fact, it has a clear structure.

– $v_\pi(s)$ and $v_\pi(s')$ are state values to be calculated. It may be confusing to beginners how to calculate the unknown $v_\pi(s)$ given that it relies on another unknown $v_\pi(s')$. It must be noted that the Bellman equation is a set of linear equations rather than a single equation. If we put these equations together, it would be clear how to calculate all the state values. Details will be given in Section 2.5.

– $\pi(a|s)$ is a given policy. Since state values can be used to evaluate a policy, calculating the state values from the Bellman equation is called *policy evaluation*, which is an important step for many RL algorithms as we will see later in the book.

– $p(r|s,a)$ and $p(s'|s,a)$ represent the dynamic model. We will first show how to calculate the state values given the model and later show how to do that without the model.

2.4.1 Illustrative examples

We next use examples to demonstrate how to manually write out the Bellman equation and calculate the state values step by step.

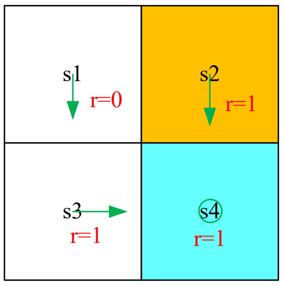


Figure 2.3: An example to demonstrate the Bellman equation. The policy in this example is deterministic.

Consider the first example shown in Figure 2.3 where the policy is deterministic. First, consider state s_1. Under the policy, the probability to take actions is $\pi(a = a_3|s_1) = 1$ and $\pi(a \neq a_3|s_1) = 0$. The state transition probability is $p(s' = s_3|s_1, a_3) = 1$ and $p(s' \neq s_3|s_1, a_3) = 0$. The reward probability is $p(r = -1|s_1, a_3) = 1$ and $p(r \neq -1|s_1, a_3) = 0$. Substituting them into the Bellman equation gives

$$v_\pi(s_1) = 0 + \gamma v_\pi(s_3),$$

Similarly, it can be obtained that

$$\begin{aligned} v_\pi(s_2) &= 1 + \gamma v_\pi(s_4), \\ v_\pi(s_3) &= 1 + \gamma v_\pi(s_4), \\ v_\pi(s_4) &= 1 + \gamma v_\pi(s_4). \end{aligned}$$

Solving the above equations one by one from the last to the first gives

$$\begin{aligned} v_\pi(s_4) &= \frac{1}{1-\gamma}, \\ v_\pi(s_3) &= \frac{1}{1-\gamma}, \\ v_\pi(s_2) &= \frac{1}{1-\gamma}, \\ v_\pi(s_1) &= \frac{\gamma}{1-\gamma}. \end{aligned}$$

If $\gamma = 0.9$, then

$$\begin{aligned} v_\pi(s_4) &= \frac{1}{1-0.9} = 10, \\ v_\pi(s_3) &= \frac{1}{1-0.9} = 10, \\ v_\pi(s_2) &= \frac{1}{1-0.9} = 10, \\ v_\pi(s_1) &= \frac{0.9}{1-0.9} = 9. \end{aligned}$$

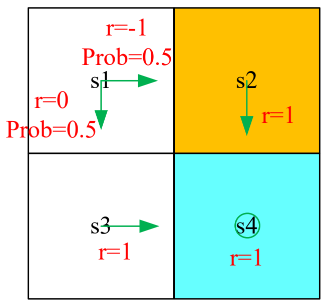


Figure 2.4: An example to demonstrate the Bellman equation. The policy in this example is stochastic.

Consider the second example shown in Figure 2.4 where the policy is stochastic. We next write out the Bellman equation and then compare this policy with the one in Figure 2.3.

At state s_1, the probabilities to go right and down are equal to 0.5. Mathematically, the probability to take actions is $\pi(a = a_2|s_1) = 0.5$ and $\pi(a = a_3|s_1) = 0.5$. The state transition probability is deterministic since $p(s' = s_3|s_1, a_3) = 1$ and $p(s' = s_2|s_1, a_2) = 1$. The reward probability is also deterministic since $p(r = 0|s_1, a_3) = 1$ and $p(r = -1|s_1, a_2) = 1$. Therefore, we have

$$v_\pi(s_1) = 0.5[0 + \gamma v_\pi(s_3)] + 0.5[-1 + \gamma v_\pi(s_2)]$$

Similarly, it can be obtained that

$$\begin{aligned} v_\pi(s_2) &= 1 + \gamma v_\pi(s_4), \\ v_\pi(s_3) &= 1 + \gamma v_\pi(s_4), \\ v_\pi(s_4) &= 1 + \gamma v_\pi(s_4). \end{aligned}$$

Solving the above equations one by one from the last to the first gives

$$\begin{aligned} v_\pi(s_4) &= \frac{1}{1-\gamma}, \\ v_\pi(s_3) &= \frac{1}{1-\gamma}, \\ v_\pi(s_2) &= \frac{1}{1-\gamma}, \\ v_\pi(s_1) &= 0.5[0 + \gamma v_\pi(s_3)] + 0.5[-1 + \gamma v_\pi(s_2)], \\ &= -0.5 + \frac{\gamma}{1-\gamma}. \end{aligned}$$

Substituting $\gamma = 0.9$ yields

$$\begin{aligned} v_\pi(s_4) &= 10, \\ v_\pi(s_3) &= 10, \\ v_\pi(s_2) &= 10, \\ v_\pi(s_1) &= -0.5 + 9 = 8.5. \end{aligned}$$

If we compare the two policies in the above two examples, the first policy is better since $v_{\pi_1}(s_1) = 9 > v_{\pi_2}(s_1) = 8.5$. This is also consistent with the intuition that the policy in the second example is not good since the agent may move to the forbidden cell from s_1.

2.4.2 Alternative expressions of the Bellman equation

In addition to the expression in (2.6), the reader may also encounter other expressions of the Bellman equation in the literature. We next give another two alternative expressions.

First, it follows from the law of total probability that

$$\begin{aligned} p(s'|s,a) &= \sum_r p(s',r|s,a), \\ p(r|s,a) &= \sum_{s'} p(s',r|s,a). \end{aligned}$$

Then, equation (2.6) can be rewritten as

$$v(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a) \left[r + \gamma v(s') \right], \tag{2.7}$$

where $\sum_{s',r} = \sum_{s'} \sum_r$. This is the same as the one used in [3].

Second, in some problems, the next state s' and the reward r is one-to-one matched. As a result, we have $p(r|s,a) = p(s'|s,a)$, substituting which into (2.6) gives

$$v(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) \left[r + \gamma v(s') \right]. \tag{2.8}$$

The assumption of $p(r|s,a) = p(s'|s,a)$ is valid for some simple problems. For example, in the grid-world examples, the next state and the immediate reward are one-to-one matched. However, it may not be true in general. Hence, the expression in (2.9) is not as general as the one in (2.6).

2.5 Solving state values from the Bellman equation

Calculating the state values of a given policy is a fundamental problem in RL. This problem is often referred to as *policy evaluation*, an important step in many RL algorithms. In this section, we show how to obtain state values by solving the Bellman equation.

The Bellman equation in (2.6) is an *elementwise form*. It indicates that the value of a state depends on the values of some other states. Since such an equation is valid for every state, there are $|\mathcal{S}|$ equations like this. If we put all these equations together, we obtain a set of linear equations, which can be expressed concisely in a *matrix-vector form*. Then, it will be much more clear to see how to solve the state values. The matrix-vector form is elegant and will be used frequently to analyze the Bellman equation. Next, we first present the matrix-vector form of the Bellman equation and then introduce two ways to solve the equation.

2.5.1 Matrix-vector form of the Bellman equation

To derive the matrix-vector form, we first rewrite the Bellman equation in (2.6) as

$$v_\pi(s) = r_\pi(s) + \gamma \sum_{s'} p_\pi(s'|s) v_\pi(s'), \tag{2.9}$$

where

$$r_\pi(s) \triangleq \sum_a \pi(a|s) \sum_r p(r|s,a) r, \qquad p_\pi(s'|s) \triangleq \sum_a \pi(a|s) \sum_{s'} p(s'|s,a).$$

Here, $r_\pi(s)$ denotes the mean of the immediate rewards that can be obtained starting from s under policy π, and $p_\pi(s'|s)$ is the probability jumping from s to s' under policy π.

In order to write in a matrix form, suppose that the states are indexed as s_i ($i = 1, \ldots, n$). For state s_i, the Bellman equation is

$$v_\pi(s_i) = r_\pi(s_i) + \gamma \sum_{s_j} p_\pi(s_j|s_i) v_\pi(s_j).$$

Let $v_\pi = [v_\pi(s_1), \ldots, v_\pi(s_n)]^T \in \mathbb{R}^n$, $r_\pi = [r_\pi(s_1), \ldots, r_\pi(s_n)]^T \in \mathbb{R}^n$, and $P_\pi \in \mathbb{R}^{n\times n}$ where $[P_\pi]_{ij} = p_\pi(s_j|s_i)$. Then we have the matrix-vector form as

$$v_\pi = r_\pi + \gamma P_\pi v_\pi. \tag{2.10}$$

Here, v_π is the unknown to be solved and r_π, P_π are known.

To illustrate, consider the policy in Figure 2.5. The matrix-vector form of the Bellman equation is

$$\underbrace{\begin{bmatrix} v_\pi(s_1) \\ v_\pi(s_2) \\ v_\pi(s_3) \\ v_\pi(s_4) \end{bmatrix}}_{v_\pi} = \underbrace{\begin{bmatrix} r_\pi(s_1) \\ r_\pi(s_2) \\ r_\pi(s_3) \\ r_\pi(s_4) \end{bmatrix}}_{r_\pi} + \gamma \underbrace{\begin{bmatrix} p_\pi(s_1|s_1) & p_\pi(s_2|s_1) & p_\pi(s_3|s_1) & p_\pi(s_4|s_1) \\ p_\pi(s_1|s_2) & p_\pi(s_2|s_2) & p_\pi(s_3|s_2) & p_\pi(s_4|s_2) \\ p_\pi(s_1|s_3) & p_\pi(s_2|s_3) & p_\pi(s_3|s_3) & p_\pi(s_4|s_3) \\ p_\pi(s_1|s_4) & p_\pi(s_2|s_4) & p_\pi(s_3|s_4) & p_\pi(s_4|s_4) \end{bmatrix}}_{P_\pi} \underbrace{\begin{bmatrix} v_\pi(s_1) \\ v_\pi(s_2) \\ v_\pi(s_3) \\ v_\pi(s_4) \end{bmatrix}}_{v_\pi}.$$

Substituting the specific values into the equation gives

$$\begin{bmatrix} v_\pi(s_1) \\ v_\pi(s_2) \\ v_\pi(s_3) \\ v_\pi(s_4) \end{bmatrix} = \begin{bmatrix} 0.5(0) + 0.5(-1) \\ 1 \\ 1 \\ 1 \end{bmatrix} + \gamma \begin{bmatrix} 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_\pi(s_1) \\ v_\pi(s_2) \\ v_\pi(s_3) \\ v_\pi(s_4) \end{bmatrix}.$$

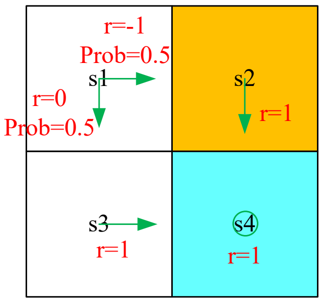


Figure 2.5: An example to demonstrate the matrix-vector form of the Bellman equation.

2.5.2 Closed-form solution

Since $v_\pi = r_\pi + \gamma P_\pi v_\pi$ is a simple linear equation, its *closed-form solution* can be easily obtained as

$$v_\pi = (I - \gamma P_\pi)^{-1} r_\pi.$$

The matrix $(I - \gamma P_\pi)^{-1}$ has some interesting properties.

- $(I - \gamma P_\pi)^{-1} \geq I$, indicating that every element of $(I - \gamma P_\pi)^{-1}$ is nonnegative and more specifically no less than that of the identity matrix. This fact is because P_π has nonnegative entries and hence $(I - \gamma P_\pi)^{-1} = I + \gamma P_\pi + \gamma^2 P_\pi^2 + \cdots \geq I \geq 0$. In this book, $\geq$ or $\leq$ is elementwise.
- If a vector $r \geq 0$, then $(I - \gamma P_\pi)^{-1} r \geq r \geq 0$. The proof is as follows. $(I - \gamma P_\pi)^{-1} r = r + \gamma P_\pi r + \gamma^2 P_\pi^2 r + \cdots \geq r$.
- If $r_1 \geq r_2$, then $(I - \gamma P_\pi)^{-1} r_1 \geq (I - \gamma P_\pi)^{-1} r_2$. This property directly follows from the second one.

2.5.3 Iterative solution

Although the closed-form solution is useful for theoretical analysis, it is not applicable in practice because it involves a matrix inverse operation, which still needs to calculate by other numerical algorithms. In fact, we can directly solve the Bellman equation using the following iterative algorithm:

$$v_{k+1} = r_\pi + \gamma P_\pi v_k.$$

This algorithm generates a sequence of intermediate values $\{v_0, v_1, v_2, \dots\}$, where $v_0 \in \mathbb{R}^n$ is an initial guess of v_π. It holds that

$$v_k \to v_\pi = (I - \gamma P_\pi)^{-1} r_\pi, \qquad \text{as } k \to \infty.$$

Interested readers may see the proof below.

Proof. Define the error as $\delta_k = v_k - v_\pi$. We only need to show $\delta_k \to 0$. Substituting $v_{k+1} = \delta_{k+1} + v_\pi$ and $v_k = \delta_k + v_\pi$ into $v_{k+1} = r_\pi + \gamma P_\pi v_k$ gives

$$\delta_{k+1} + v_\pi = r_\pi + \gamma P_\pi(\delta_k + v_\pi),$$

which can be rewritten as

$$\begin{aligned}\delta_{k+1} &= -v_\pi + r_\pi + \gamma P_\pi \delta_k + \gamma P_\pi v_\pi,\\ &= \gamma P_\pi \delta_k - v_\pi + r_\pi + \gamma P_\pi v_\pi,\\ &= \gamma P_\pi \delta_k.\end{aligned}$$

As a result,

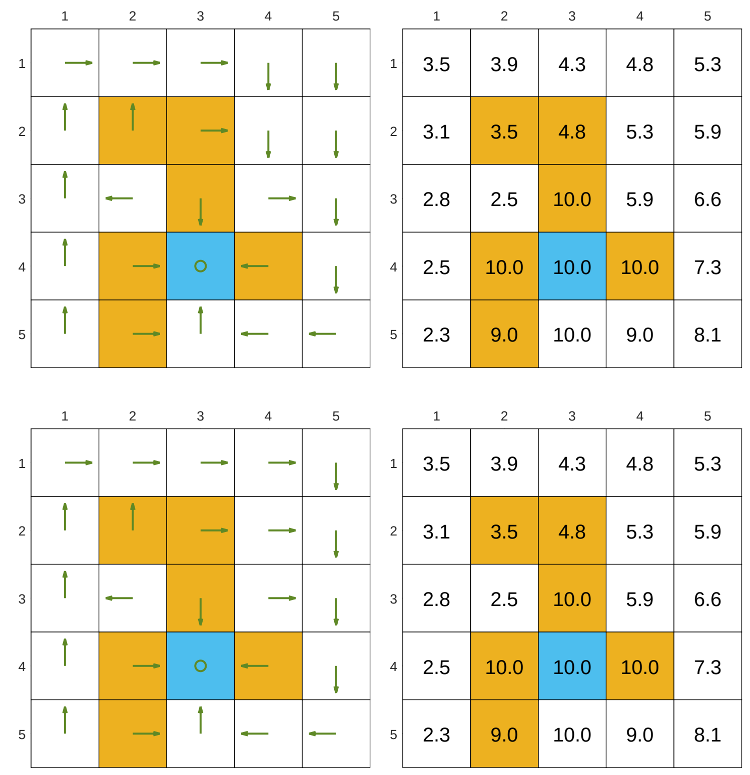
$$\delta_{k+1} = \gamma P_\pi \delta_k = \gamma^2 P_\pi^2 \delta_{k-1} = \cdots = \gamma^{k+1} P_\pi^{k+1} \delta_0.$$

Since P_π is a nonnegative stochastic matrix satisfying $P_\pi \mathbf{1} = \mathbf{1}$ where $\mathbf{1} = [1, \dots, 1]^T$, we have $0 \le P_\pi^k \le 1$ for any k. That is, every entry of P_π^k is no greater than 1. On the other hand, since $\gamma < 1$, we know $\gamma^k \to 0$ and hence $\delta_{k+1} = \gamma^{k+1} P_\pi^{k+1} \delta_0 \to 0$ as $k \to \infty$. □

2.5.4 Illustrative examples

We next show some grid-world examples to demonstrate the algorithms introduced above. The orange cells represent forbidden areas. The blue cell represents the target area. The reward setting is $r_{\text{boundary}} = r_{\text{forbidden}} = -1$ and $r_{\text{target}} = 1$. Here, the discount rate is $\gamma = 0.9$.

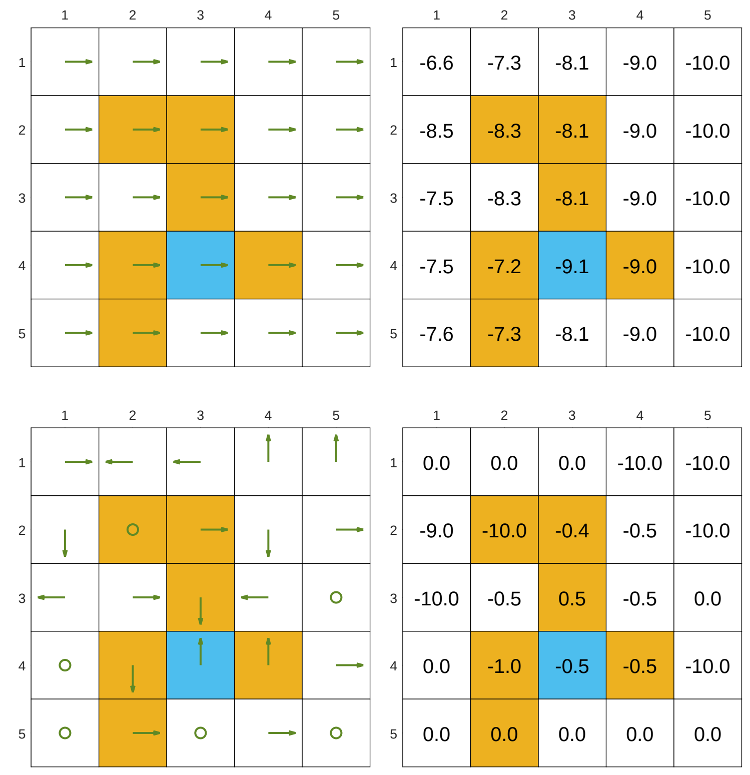
Figure 2.6(a) shows two "good" policies and their corresponding state values. The two policies have exactly the same state values, although they are different in terms of the



	1	2	3	4	5
1	3.5	3.9	4.3	4.8	5.3
2	3.1	3.5	4.8	5.3	5.9
3	2.8	2.5	10.0	5.9	6.6
4	2.5	10.0	10.0	10.0	7.3
5	2.3	9.0	10.0	9.0	8.1

	1	2	3	4	5
1	3.5	3.9	4.3	4.8	5.3
2	3.1	3.5	4.8	5.3	5.9
3	2.8	2.5	10.0	5.9	6.6
4	2.5	10.0	10.0	10.0	7.3
5	2.3	9.0	10.0	9.0	8.1

(a) Two "good" policies and their state values. The state values of the two policies are exactly the same although the two policies are different in terms of the top two states in the fourth column.



	1	2	3	4	5
1	-6.6	-7.3	-8.1	-9.0	-10.0
2	-8.5	-8.3	-8.1	-9.0	-10.0
3	-7.5	-8.3	-8.1	-9.0	-10.0
4	-7.5	-7.2	-9.1	-9.0	-10.0
5	-7.6	-7.3	-8.1	-9.0	-10.0

	1	2	3	4	5
1	0.0	0.0	0.0	-10.0	-10.0
2	-9.0	-10.0	-0.4	-0.5	-10.0
3	-10.0	-0.5	0.5	-0.5	0.0
4	0.0	-1.0	-0.5	-0.5	-10.0
5	0.0	0.0	0.0	0.0	0.0

(b) Two "bad" policies and their state values. The state values are smaller than those of the "good" policies.

Figure 2.6: Examples of policies and the corresponding state values.

top two states in the fourth column. For the two specific states, the agent either moving rightwards or moving downwards makes no difference in terms of the state values.

Figure 2.6(b) shows two "bad" policies and their corresponding state values. The two policies are bad because the actions of many states are not reasonable intuitively. Such intuition is supported by the state values. As can be seen, the state values of these two policies are much lower than those of the good policies in Figure 2.6(a).

2.6 From state value to action value

While we have introduced the concept of state value, we now turn to *action value*, which indicates the "value" of taking an action. Action value is a very important concept. As we will see later in the book, we often care about action values more than state values because action values can be used to generate optimal policies.

The definition of action value is

$$q_\pi(s,a) \doteq \mathbb{E}[G_t|S_t=s, A_t=a]$$

for any state $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. It is clear that the action value is the average return that can be obtained after taking an action. Note that $q_\pi(s,a)$ depends on a state-action pair (s,a) instead of an action alone.

What is the relationship between action value and state value? First, it follows from the properties of conditional expectation that

$$\underbrace{\mathbb{E}[G_t|S_t=s]}_{v_\pi(s)} = \sum_a \underbrace{\mathbb{E}[G_t|S_t=s, A_t=a]}_{q_\pi(s,a)} \pi(a|s).$$

Hence,

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a). \tag{2.11}$$

As a result, the value of a state is the expectation of the action values associated to that state. Second, since the state value is given by $v_\pi(s) = \sum_a \pi(a|s)\Big[\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a)v_\pi(s')\Big]$ as in the Bellman equation, comparing it with (2.11) leads to

$$q_\pi(s,a) = \sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a)v_\pi(s'). \tag{2.12}$$

It can be seen that the action value consists of two terms. The first term is the mean of the immediate rewards and the second term is the mean of the future rewards.

It is interesting that both (2.11) and (2.12) describe the relationship between state value and action value. They are the two sides of the same coin: (2.11) shows how to obtain state values from action values, whereas (2.12) shows the reverse that is how to obtain action values from state values.

2.6.1 Illustrative examples

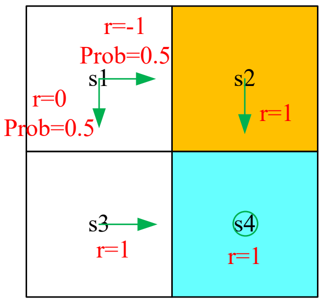


Figure 2.7: An example to demonstrate the calculation of action values.

We next give an example to illustrate the calculation of action values and discuss a common mistake that beginners may make.

Consider the example in Figure 2.7. The policy is stochastic. For the sake of simplicity, we only consider the action values of s_1. In particular, taking action a_2 at state s_1 would get a total reward of

$$q_\pi(s_1, a_2) = -1 + \gamma v_\pi(s_2),$$

where s_2 is the next state. Similarly, it can be obtained that

$$q_\pi(s_1, a_3) = 0 + \gamma v_\pi(s_3).$$

A common mistake that beginners may make is about computing the action values of a_1, a_4, a_5. One may say that, since the policy would not take the actions of a_1, a_4, a_5, we have $q_\pi(s_1, a_1) = q_\pi(s_1, a_4) = q_\pi(s_1, a_5) = 0$, or we do not need to calculate them. In fact, the action value at a specific state does not depend on the policy at that specific state. In particular, since the next state is still s_1 after taking a_1, a_4, or a_5 at s_1, we have

$$\begin{aligned} q_\pi(s_1, a_1) &= -1 + \gamma v_\pi(s_1), \\ q_\pi(s_1, a_4) &= -1 + \gamma v_\pi(s_1), \\ q_\pi(s_1, a_5) &= 0 + \gamma v_\pi(s_1). \end{aligned}$$

The reason that we care about the values of those actions that may not be taken by the current policy is that these actions may be good and missed by the current policy. Therefore, we have to explore the values of these actions.

Finally, after computing the action values, we can also calculate the state value following (2.12):

$$
\begin{aligned}
v_\pi(s_1) &= 0.5q_\pi(s_1,a_2) + 0.5q_\pi(s_1,a_3),\\
&= 0.5[0+\gamma v_\pi(s_3)] + 0.5[-1+\gamma v_\pi(s_2)].
\end{aligned}
$$

2.6.2 Bellman equation in terms of action values

The Bellman equation that we introduced previously is defined based on state values. In fact, it can also be expressed in terms of action values.

Substituting (2.11) into (2.12) gives

$$
q_\pi(s,a) = \sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) \sum_{a'\in\mathcal{A}(s')} \pi(a'|s') q_\pi(s',a'), \tag{2.13}
$$

which is an equation of action values. Suppose each state has the same number of actions. The matrix-vector form of (2.13) is

$$
q_\pi = \tilde{r} + \gamma P \Pi q_\pi, \tag{2.14}
$$

where $q_\pi \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ is the action value vector indexed by state-action pairs. In particular, the (s,a)th element is $[q_\pi]_{(s,a)} = q_\pi(s,a)$. Here, $\tilde{r} \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ is the immediate reward vector indexed by state-action pairs. In particular,

$$
[\tilde{r}]_{(s,a)} = \sum_r p(r|s,a)r.
$$

Here, $P \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|\times|\mathcal{S}|}$ is the probability transition matrix, whose row is indexed by state-action pairs and column indexed by states. In particular,

$$
[P]_{(s,a),s'} = p(s'|s,a)
$$

and $\Pi \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}||\mathcal{A}|}$ describes the policy π. In particular,

$$
\Pi_{s',(s',a')} = \pi(a'|s')
$$

and the other entries of Π are zero. Π is a block diagonal matrix with each block as a $1\times|\mathcal{A}|$ vector.

Compared to the Bellman equation in terms of state values, the one in terms of action values has some unique features. For example, $\tilde{r}$ and P are independent of the policy and merely determined by the state model. The policy is totally contained in Π. It can also be verified that (2.14) is also a contraction mapping and hence has a unique solution, which can be solved iteratively. More details can be found in [4].

2.7 Summary

State value is the most important concept introduced in this chapter. Mathematically, it is the expectation or mean value of the returns that the agent can obtain starting from a state. The values of different states are related to each other. That is, the value of state s relies on the values of some other states, which may further rely on the value of state s itself. Such a phenomenon might be the most confusing part for beginners. It is related to an important concept called bootstrapping, which means calculating something from itself. Although bootstrapping may be confusing intuitively, it is crystal if we examine the matrix-vector form of the Bellman equation. In particular, the Bellman equation is a set of linear equations that describe the relationship among the values of all states. If we put all the equations together into a matrix-vector form, it will be clear that solving the state values is simply solving a linear equation.

Since state values can be used to evaluate if a policy is good or not, the process of solving the state values from the Bellman equation is called policy evaluation. As we will see later in this book, policy evaluation is an important step in many RL algorithms.

Finally, another important concept, action value, describes the value of taking one action at a state. As we will see later in this book, action value plays a more direct role than state value when we attempt to find optimal policies.

The Bellman equation is not restricted to the RL domain. Instead, it widely exists in many fields such as control theories and operation research. In different fields, the Bellman equation may have different expressions. In this book, the Bellman equation is studied under the discrete Markov decision process. A complete treatment of this topic can be found in [2].

2.8 Q&A

– Q: What is the relationship between state value and return?

  A: The value of a state is the mean of the returns that can be obtained if the agent starts from that state.

– Q: Why do we care about state value?

  A: State value can be used to evaluate the "goodness" of a policy. In fact, the optimal policies are defined based on state values. This point will be clearer in the next chapter when we introduce the Bellman optimality equation.

– Q: Why do we care about the Bellman equation?

  A: The Bellman equation describes the relationship among the values of all the states. It is the tool to analyze the state values.

– Q: Why solving the Bellman equation is called policy evaluation?

A: Solving the Bellman equation is to solve the state values. Since state values can be used to evaluate the “goodness” of a policy, solving the Bellman equation can be interpreted as evaluating the policy.

– Q: Why do we need to study the matrix-vector form of the Bellman equation?

A: The Bellman equation actually refers to a set of linear equations established for all the states. In order to solve state values, we must put all the linear equations together. The matrix-vector form is a concise expression of these linear equations.

– Q: Why do we care about action values?

A: That is simply because the ultimate goal of reinforcement learning is to find out which actions are more valuable. This point will be clearer in the following chapters.

– Q: What is the relationship between state value and action value?

A: On the one hand, the value of a state is the mean of the action values of that state. On the other hand, the value of an action relies on the values of the next states that the agent may transit to after taking the action.

– Q: Why do we care about the values of those actions that the given policy would not take?

A: Although some actions would not be taken by the given policy, it does not mean these actions are not good. On the contrary, it is possible that the given policy is not good and misses the best action. Therefore, we must keep exploring to check the values of these actions.

Chapter 3

Optimal State Value and Bellman Optimality Equation

The ultimate goal of reinforcement learning (RL) is to seek *optimal policies*. It is, therefore, important to define "optimal". In this chapter, we will introduce a core concept and an important tool. The core concept is optimal state value, based on which we can define optimal policies. The important tool is the Bellman optimality equation, from which we can solve the optimal state values as well as optimal policies. The contents of this chapter are fundamentally important for understanding the model-based RL algorithms that will be introduced in the next chapter. Be prepared that this chapter is a little mathematically intensive. However, it is worth because many fundamental and algorithmic questions will be clearly answered.

3.1 Motivating example: how to improve policies?

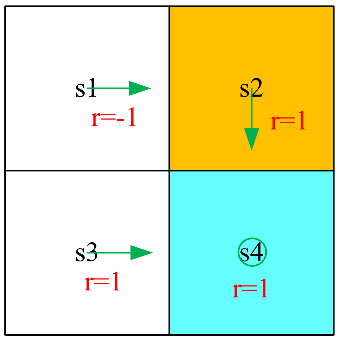


Figure 3.1: An example to demonstrate policy improvement.

Consider the example shown in Figure 3.1, where the target state is s_4. This policy is not good, because the policy at s_1 suggests moving rightwards to the forbidden area. How can we improve the given policy to make it better? The answer lies in state values and action values.

First, we calculate the state values of the given policy. In particular, the Bellman equation of this policy is

$$\begin{aligned}
v_\pi(s_1) &= -1 + \gamma v_\pi(s_2),\\
v_\pi(s_2) &= +1 + \gamma v_\pi(s_4),\\
v_\pi(s_3) &= +1 + \gamma v_\pi(s_4),\\
v_\pi(s_4) &= +1 + \gamma v_\pi(s_4).
\end{aligned}$$

Let $\gamma = 0.9$. It can be calculated that

$$\begin{aligned}
&v_\pi(s_4) = v_\pi(s_3) = v_\pi(s_2) = 10,\\
&v_\pi(s_1) = 8.
\end{aligned}$$

Second, we calculate the action values for state s_1:

$$\begin{aligned}
q_\pi(s_1, a_1) &= -1 + \gamma v_\pi(s_1) = 6.2,\\
q_\pi(s_1, a_2) &= -1 + \gamma v_\pi(s_2) = 8,\\
q_\pi(s_1, a_3) &= 0 + \gamma v_\pi(s_3) = 9,\\
q_\pi(s_1, a_4) &= -1 + \gamma v_\pi(s_1) = 6.2,\\
q_\pi(s_1, a_5) &= 0 + \gamma v_\pi(s_1) = 7.2.
\end{aligned}$$

It is notable that action a_3 (moving downwards) has the greatest action value. That is

$$q_\pi(s_1, a_3) \geq q_\pi(s_1, a_i), \quad \text{for all } i \in \{1, 2, 3, 4, 5\}.$$

Therefore, if we update the policy so that it selects a_3 at s_1, then the updated policy becomes better since moving downwards at s_1 can avoid the forbidden area.

This example illustrates that, if we update the policy to select the action with the *greatest action value*, we could find a better policy. This is the basic idea of many RL algorithms. Of course, this example is very simple in the sense that the given policy is not good only for state s_1. If the policy is also not good for the other states, will selecting the action with the greatest action value still generate a better policy? Moreover, whether there exist optimal or best policies? What does an optimal policy look like? We will answer all of these questions in this chapter.

3.2 Optimal state value and optimal policy

The goal of reinforcement learning is to find out optimal policies. It is, therefore, important to define what optimal policy is. While state values can be used to evaluate policies, they actually can also be used to define optimal policies. In particular, consider two given policies π_1 and π_2. If

$$v_{\pi_1}(s) \geq v_{\pi_2}(s), \quad \text{for all } s \in \mathcal{S}.$$

π_1 is said "better" than π_2. That is the state value of π_1 is greater than or equal to that of π_2 for any state. Furthermore, if a policy is better than all the other possible policies, then this policy is optimal. This idea is formally stated below.

Definition 3.1 (Optimal policy and optimal state value)**.** *A policy π^* is optimal if $v_{\pi^*}(s) \geq v_\pi(s)$ for all $s \in \mathcal{S}$ and for any other policy π. The state values of π^* are the optimal state values.*

This definition indicates that an optimal policy has the greatest state value for every state compared to all the other policies. The definition also leads to many questions:

– Existence: Does the optimal policy exist?

– Uniqueness: Is the optimal policy unique?

– Stochasticity: Is the optimal policy stochastic or deterministic?

– Algorithm: How to obtain the optimal policy and the optimal state values?

Be patient. We will answer these questions one by one in the rest of the chapter.

3.3 Bellman optimality equation

In order to understand optimal state values and optimal policies, we need to explore a special Bellman equation called *Bellman optimality equation* (BOE). We first directly present this equation and then analyze why it describes optimal state values and optimal policies.

The BOE is

$$\begin{aligned} v(s) &= \max_\pi \sum_a \pi(a|s) \left(\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a)v(s') \right), \\ &= \max_\pi \sum_a \pi(a|s) q(s,a), \quad \text{for all } s \in \mathcal{S}. \end{aligned} \tag{3.1}$$

where $v(s), v(s')$ are unknowns and

$$q(s,a) \doteq \sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a)v(s').$$

The BOE is an elegant and powerful tool for analyzing optimal policies. However, it looks tricky at first glance. For example, there are two unknowns $v(s)$ and $\pi(a|s)$ in the equation. It may be confusing how to solve unknowns from one equation. Also, although this equation is a Bellman equation, it is nontrivial to see that since it involves a maximization problem on the right-hand side. We also need to answer the following fundamental questions about the BOE.

– Existence: does this equation have solutions?

– Uniqueness: is the solution unique?

– Algorithm: how to solve this equation?

– Optimality: how is the solution related to optimal policy?

Once we can answer these questions, we will understand optimal state values and optimal policies clearly. At this moment, it is better to temporarily forget about the RL interpretations of the symbols in the BOE and study the equation from a pure algebraic perspective. The rest of the section is important but, in the meantime, a little mathematically intensive. The readers are suggested to read selectively.

3.3.1 Maximization on the right-hand side of the BOE

First of all, we clarify how to solve the maximization problem on the right-hand side of the BOE. At first glance, it may be confusing to beginners how can we solve two unknowns $v(s)$ and $\pi(a|s)$ from one equation. In fact, the two unknowns can be solved one by one. This idea is illustrated by the following simple example.

Example 3.1. *Consider two variables $x, a \in \mathbb{R}$. Suppose they satisfy*

$$x = \max_a (2x - 1 - a^2).$$

This equation has two unknowns. To solve them, first consider the right hand side. Regardless the value of x, $\max_a(2x-1-a^2) = 2x-1$ where the maximization is achieved when $a = 0$. Second, when $a = 0$, the equation becomes $x = 2x - 1$, which leads to $x = 1$. Therefore, $a = 0$ and $x = 1$ are the solution of the equation.

From the above example, we know that we can first fix one variable and solve the maximization problem. We now come back to the maximization problem on the right-hand side of the BOE. The BOE in (3.1) can be written in short as

$$v(s) = \max_\pi \sum_a \pi(a|s) q(s,a), \quad s \in \mathcal{S}.$$

Inspired by Example 3.1, we can first fix $q(s,a)$ for all $a \in \mathcal{A}(s)$ and then find the optimal π. How to do that? The following example can demonstrate the basic idea.

Example 3.2. *Suppose $x_1, x_2, x_3 \in \mathbb{R}$ are given. Find c_1^*, c_2^*, c_3^* solving*

$$\max_{c_1,c_2,c_3} c_1x_1 + c_2x_2 + c_3x_3.$$

where $c_1 + c_2 + c_3 = 1$ *and* $c_1, c_2, c_3 \geq 0$.

Without loss of generality, suppose $x_3 \geq x_1, x_2$. *Then, the optimal solution is* $c_3^* = 1$ *and* $c_1^* = c_2^* = 0$. *That is because for any* c_1, c_2, c_3

$$x_3 = (c_1 + c_2 + c_3)x_3 = c_1x_3 + c_2x_3 + c_3x_3 \geq c_1x_1 + c_2x_2 + c_3x_3.$$

Inspired by the above example, considering that $\sum_a \pi(a|s) = 1$, we have

$$\sum_a \pi(a|s)q(s,a) \leq \sum_a \pi(a|s) \max_{a\in\mathcal{A}(s)} q(s,a) = \max_{a\in\mathcal{A}(s)} q(s,a),$$

where the equality is achieved when

$$\pi(a|s) = \begin{cases} 1 & a = a^* \\ 0 & a \neq a^* \end{cases}$$

where $a^* = \arg\max_a q(s,a)$. Therefore, when $\{q(s,a)\}_{a\in\mathcal{A}(s)}$ are fixed, the optimal policy $\pi(s)$ is to choose the action that corresponds to the greatest value of $q(s,a)$.

3.3.2 Matrix-vector form of the BOE

The BOE refers to a set of equations that are defined for all states. If we put these equations together, we can have a concise matrix-vector form, which will be extensively used in this chapter.

The matrix-vector form of the BOE is

$$v = \max_\pi (r_\pi + \gamma P_\pi v),$$

where $v \in \mathbb{R}^{|\mathcal{S}|}$. The structure of r_π and P_π are the same as those in the matrix-vector form of a normal Bellman equation:

$$[r_\pi]_s \triangleq \sum_a \pi(a|s) \sum_r p(r|s,a)r, \qquad [P_\pi]_{s,s'} = p(s'|s) \triangleq \sum_a \pi(a|s) \sum_{s'} p(s'|s,a).$$

Here, $\max_\pi$ is performed elementwise. Furthermore, denote the right hand side as

$$f(v) \doteq \max_\pi (r_\pi + \gamma P_\pi v).$$

Then, the BOE becomes

$$v = f(v). \tag{3.2}$$

Therefore, the BOE is expressed as a simple nonlinear equation of v. In the rest, we show how to solve this nonlinear equation and what kind of properties it has.

3.3.3 Contraction mapping theorem

Since the BOE can be expressed as a nonlinear equation $v = f(v)$, we next introduce the contraction mapping theorem, which is a useful tool to analyze general nonlinear equations. This theorem is also known as the fixed-point theorem. Readers who already know this theorem can skip this part.

Consider a function $f(x)$ where $x \in \mathbb{R}^d$ and $f : \mathbb{R}^d \to \mathbb{R}^d$. A point x^* is called a *fixed point* if

$$f(x^*) = x^*.$$

The interpretation is that the map of x^* is itself. That is why it is called "fixed". The function f is a *contraction mapping* (or contractive function) if there exists $\gamma \in (0, 1)$ such that

$$\|f(x_1) - f(x_2)\| \leq \gamma \|x_1 - x_2\|$$

for any $x_1, x_2 \in \mathbb{R}$. In this book, $\|\cdot\|$ denotes a vector or matrix norm.

Example 3.3. *Here are three examples to demonstrate the concept of fixed point and contraction mapping.*

- $x = f(x) = 0.5x$, $x \in \mathbb{R}$.

  It is easy to verify that $x = 0$ *is a fixed point since* $0 = 0.5 \cdot 0$. *Moreover,* $f(x) = 0.5x$ *is a contraction mapping because* $\|0.5x_1 - 0.5x_2\| = 0.5\|x_1 - x_2\| \leq \gamma\|x_1 - x_2\|$ *for any* $\gamma \in [0.5, 1)$.

- $x = f(x) = Ax$, *where* $x \in \mathbb{R}^n, A \in \mathbb{R}^{n\times n}$ *and* $\|A\| \leq \gamma < 1$.

  It is easy to verify that $x = 0$ *is a fixed point since* $0 = A0$. *To see the contraction property,* $\|Ax_1 - Ax_2\| = \|A(x_1 - x_2)\| \leq \|A\|\|x_1 - x_2\| \leq \gamma\|x_1 - x_2\|$. *Therefore,* $f(x) = Ax$ *is a contraction mapping.*

- $x = f(x) = 0.5 \sin x$, $x \in \mathbb{R}$.

  It is easy to see that $x = 0$ *is a fixed point since* $0 = 0.5 \sin 0$. *Moreover, it follows from the* mean value theorem *that*

  $$\left|\frac{0.5 \sin x_1 - 0.5 \sin x_2}{x_1 - x_2}\right| = |0.5 \cos x_3| \leq 0.5, \quad x_3 \in [x_1, x_2].$$

  As a result, $|0.5 \sin x_1 - 0.5 \sin x_2| \leq 0.5|x_1 - x_2|$ *and hence* $f(x) = 0.5 \sin x$ *is a contraction mapping.*

The relationship between the fixed point and the contraction property is given in the following classic theorem.

Theorem 3.1 (Contraction mapping theorem)**.** *For any equation that has the form of* $x = f(x)$ *where* x *and* $f(x)$ *are real vectors, if* f *is a contraction mapping, then*

– *Existence: There exists a fixed point x^* satisfying $f(x^*) = x^*$.*

– *Uniqueness: The fixed point x^* is unique.*

– *Algorithm: Consider the iterative process:*

$$x_{k+1} = f(x_k),$$

where $k = 0, 1, 2, \ldots$. Then, $x_k \to x^*$ as $k \to \infty$ for any initial guess x_0. Moreover, the convergence rate is exponentially fast.

The contraction mapping theorem not only tells if the solution of a nonlinear equation exists but also suggests a numerical algorithm solving the equation. The proof of the theorem is given below. Readers who are not interested can skip the proof.

Example 3.4. *Let's revisit the three examples: $x = 0.5x$, $x = Ax$, and $x = 0.5\sin x$. While it has been shown that the right-hand side of the three equations are all contraction mapping, it follows from the contraction mapping theorem that they have a unique fixed point, which can be easily verified to be $x^* = 0$. Moreover, the fixed points of the three equations can be solved respectively by*

$$\begin{aligned} x_{k+1} &= 0.5x_k, \\ x_{k+1} &= Ax_k, \\ x_{k+1} &= 0.5\sin x_k, \end{aligned}$$

given any initial guess x_0.

Proof of the contraction mapping theorem

Part 1: the convergence of $\{x_k = f(x_{k-1})\}_{k=1}^\infty$.

The proof relies on *Cauchy sequences*. A sequence $x_1, x_2, \cdots \in \mathbb{R}$ is called *Cauchy* if for any small $\varepsilon > 0$, there exist N such that $\|x_m - x_n\| < \varepsilon$ for all $m, n > N$. The intuitive interpretation is that, in a Cauchy sequence, there exists a finite integer N such that all the elements after N are sufficiently close to each other. Cauchy sequences are important because a Cauchy sequence will converge to a limit. Its convergence property will be used to prove the contraction mapping theorem. It is worth mentioning that we must have $\|x_m - x_n\| < \varepsilon$ for all $m, n > N$. If we simply have $x_{n+1} - x_n \to 0$, it is insufficient to claim it is a Cauchy sequence. For example, it holds that $x_{n+1} - x_n \to 0$ for $x_n = \sqrt{n}$, but apparently $x_n = \sqrt{n}$ diverges.

We next show that $\{x_k = f(x_{k-1})\}_{k=1}^\infty$ is a Cauchy sequence and hence converges. First, since f is a contraction mapping, we have

$$\|x_{k+1} - x_k\| = \|f(x_k) - f(x_{k-1})\| \leq \gamma\|x_k - x_{k-1}\|.$$

Similarly, we have $\|x_k - x_{k-1}\| \leq \gamma\|x_{k-1} - x_{k-2}\|, \dots, \|x_2 - x_1\| \leq \gamma\|x_1 - x_0\|$. As a result, we have

$$\begin{aligned}
\|x_{k+1} - x_k\| &\leq \gamma\|x_k - x_{k-1}\| \\
&\leq \gamma^2\|x_{k-1} - x_{k-2}\| \\
&\ \vdots \\
&\leq \gamma^k\|x_1 - x_0\|.
\end{aligned}$$

Since $\gamma < 1$, we know that $\|x_{k+1} - x_k\|$ converges to zero exponentially fast as $k \to \infty$ given any x_1, x_0. It is worth mentioning that the convergence of $\{\|x_{k+1} - x_k\|\}$ is not sufficient to imply the convergence of $\{x_k\}$. Therefore, we need to further consider $\|x_m - x_n\|$ for any $m > n$. In particular,

$$\begin{aligned}
\|x_m - x_n\| &= \|x_m - x_{m-1} + x_{m-1} - \dots - x_{n+1} + x_{n+1} - x_n\| \\
&\leq \|x_m - x_{m-1}\| + \dots + \|x_{n+1} - x_n\| \\
&\leq \gamma^{m-1}\|x_1 - x_0\| + \dots + \gamma^n\|x_1 - x_0\| \\
&= \gamma^n(\gamma^{m-1-n} + \dots + 1)\|x_1 - x_0\| \\
&\leq \gamma^n(1 + \dots + \gamma^{m-1-n} + \gamma^{m-n} + \gamma^{m-n+1} + \dots)\|x_1 - x_0\| \\
&= \frac{\gamma^n}{1-\gamma}\|x_1 - x_0\|. \qquad (3.3)
\end{aligned}$$

As a result, for any ε, we can always find N such that $\|x_m - x_n\| < \varepsilon$ for all $m, n > N$. Therefore, this sequence is Cauchy and hence it converges to a limit point denoted as $x^* = \lim_{k\to\infty} x_k$.

Part 2: we show that the limit $x^* = \lim_{k\to\infty} x_k$ is a fixed point. To see that, since

$$\|f(x_k) - x_k\| = \|x_{k+1} - x_k\| \leq \gamma^k\|x_1 - x_0\|,$$

we know $\|f(x_k) - x_k\|$ converges to zero exponentially fast. Hence in the limit we have $f(x^*) = x^*$.

Part 3: we show that the fixed point is unique. To see that, suppose there is another fixed point x' satisfying $f(x') = x'$. Then,

$$\|x' - x^*\| = \|f(x') - f(x^*)\| \leq \gamma\|x' - x^*\|.$$

Since $\gamma < 1$, this inequality holds if and only if $\|x' - x^*\| = 0$. Therefore, $x' = x^*$.

Part 4: we show that x_k converges to x^* exponentially fast. Recall that $\|x_m -$

$x_n\| \leq \frac{\gamma^n}{1-\gamma}\|x_1 - x_0\|$ as proven in (3.3). Since m can be arbitrarily large, we have

$$\|x^* - x_n\| = \lim_{m\to\infty} \|x_m - x_n\| \leq \frac{\gamma^n}{1-\gamma}\|x_1 - x_0\|.$$

Since $\gamma < 1$, the error converges to zero exponentially fast. The smaller γ is, the faster the convergence is.

3.3.4 Contraction property of the right-hand side of the BOE

We next show that $f(v)$ in the BOE (3.2) is a contraction mapping. Then, the contraction mapping theorem introduced in the last subsection can be applied to solve the BOE immediately.

Theorem 3.2 (Contraction property of $f(v)$)**.** *The function $f(v)$ in the BOE is a contraction mapping satisfying*

$$\|f(v_1) - f(v_2)\| \leq \gamma\|v_1 - v_2\|,$$

where $v_1, v_2 \in \mathbb{R}^{|\mathcal{S}|}$ are any two vectors and $\gamma \in (0,1)$ is the discount rate.

Thanks to the contraction mapping property, the BOE can be analyzed by the contraction mapping theorem. The details of the analysis will be given in the next section. The proof of the contraction property of the BOE is given below.

Proof of Theorem 3.2

Consider any two vectors $v_1, v_2 \in \mathbb{R}^{|\mathcal{S}|}$, suppose $\pi_1^* \doteq \arg\max_\pi(r_\pi + \gamma P_\pi v_1)$ and $\pi_2^* \doteq \arg\max_\pi(r_\pi + \gamma P_\pi v_2)$. Then,

$$\begin{aligned}
f(v_1) &= \max_\pi(r_\pi + \gamma P_\pi v_1) = r_{\pi_1^*} + \gamma P_{\pi_1^*} v_1 \geq r_{\pi_2^*} + \gamma P_{\pi_2^*} v_1, \\
f(v_2) &= \max_\pi(r_\pi + \gamma P_\pi v_2) = r_{\pi_2^*} + \gamma P_{\pi_2^*} v_2 \geq r_{\pi_1^*} + \gamma P_{\pi_1^*} v_2,
\end{aligned}$$

where $\geq$ is elementwise. As a result,

$$\begin{aligned}
f(v_1) - f(v_2) &= r_{\pi_1^*} + \gamma P_{\pi_1^*} v_1 - (r_{\pi_2^*} + \gamma P_{\pi_2^*} v_2) \\
&\leq r_{\pi_1^*} + \gamma P_{\pi_1^*} v_1 - (r_{\pi_1^*} + \gamma P_{\pi_1^*} v_2) \\
&= \gamma P_{\pi_1^*}(v_1 - v_2).
\end{aligned}$$

Similarly, it can be shown that $f(v_2) - f(v_1) \leq \gamma P_{\pi_2^*}(v_2 - v_1)$, which implies $f(v_1) - f(v_2) \geq \gamma P_{\pi_2^*}(v_1 - v_2)$. Therefore,

$$\gamma P_{\pi_2^*}(v_1 - v_2) \leq f(v_1) - f(v_2) \leq \gamma P_{\pi_1^*}(v_1 - v_2). \tag{3.4}$$

Define

$$z \doteq \max\left\{|\gamma P_{\pi_2^*}(v_1 - v_2)|, |\gamma P_{\pi_1^*}(v_1 - v_2)|\right\} \in \mathbb{R}^{|\mathcal{S}|}.$$

By definition, $z \geq 0$. Here, $\max(\cdot)$, $|\cdot|$, and $\geq$ are all elementwise. On the one hand, it is easy to see that

$$-z \leq \gamma P_{\pi_2^*}(v_1 - v_2) \leq f(v_1) - f(v_2) \leq \gamma P_{\pi_1^*}(v_1 - v_2) \leq z,$$

which implies

$$|f(v_1) - f(v_2)| \leq z.$$

Since $|f(v_1) - f(v_2)| \leq z$ elementwise, it follows that

$$\|f(v_1) - f(v_2)\|_\infty \leq \|z\|_\infty, \tag{3.5}$$

where $\|\cdot\|_\infty$ is the sup-norm, which is the maximum absolute value of the elements of a vector. Here, the inequality is still valid if the sup-norm is replaced by other norms such as $\|\cdot\|_2$ or $\|\cdot\|_1$.

On the other hand, suppose z_i is the ith entry of z, and p_i^T and q_i^T are the ith row of $P_{\pi_1^*}$ and $P_{\pi_2^*}$,respectively. Then,

$$z_i = \max\{\gamma|p_i^T(v_1 - v_2)|, \gamma|q_i^T(v_1 - v_2)|\}.$$

Since p_i is a vector with all the elements nonnegative and the sum of the elements is equal to one, it follows that

$$|p_i^T(v_1 - v_2)| \leq p_i^T|v_1 - v_2| \leq \|v_1 - v_2\|_\infty.$$

Similarly, we have $|q_i^T(v_1 - v_2)| \leq \|v_1 - v_2\|_\infty$. Therefore, $z_i \leq \gamma\|v_1 - v_2\|_\infty$ and hence

$$\|z\|_\infty = \max_i |z_i| \leq \gamma\|v_1 - v_2\|_\infty.$$

Substituting this inequality to (3.5) gives

$$\|f(v_1) - f(v_2)\|_\infty \leq \gamma\|v_1 - v_2\|_\infty,$$

which concludes the contraction property of $f(v)$.

The proof of the contraction property also suggests another useful fact. That is, $f(v)$ is monotonically non-decreasing.

Corollary 3.1 (Monotone property of $f(v)$). *For any $v_1, v_2 \in \mathbb{R}^n$, $f(v_1) \geq f(v_2)$ if $v_1 \geq v_2$, and $f(v_1) \leq f(v_2)$ if $v_1 \leq v_2$, where $\geq$ and $\leq$ are elementwise.*

Proof of Corollary 3.1

If $v_1 \geq v_2$, then $P_{\pi_2^*}(v_1 - v_2) \geq 0$ because $P_{\pi_2^*}$ is a nonnegative matrix. Thus, (3.4) implies that

$$0 \leq \gamma P_{\pi_2^*}(v_1 - v_2) \leq f(v_1) - f(v_2) \leq \gamma P_{\pi_1^*}(v_1 - v_2)$$

and hence $f(v_1) \geq f(v_2)$. Similarly, if $v_1 \leq v_2$, (3.4) implies that

$$\gamma P_{\pi_2^*}(v_1 - v_2) \leq f(v_1) - f(v_2) \leq \gamma P_{\pi_1^*}(v_1 - v_2) \leq 0$$

and hence $f(v_1) \leq f(v_2)$.

The monotone property of $f(v)$ will be used frequently later in the analysis of policy optimality. It should be noted that the converse of Corollary 3.1 is not true. That is, $f(v_1) \geq f(v_2)$ may not imply $v_1 \geq v_2$, because it is possible that some elements of v_1 may be greater than their counterparts in v_2 while the others may be less.

3.4 Solutions of the BOE

With the preparation in the last section, we are ready now to solve the BOE. There are two unknowns in the BOE: v and π. We solve the two unknowns one by one as follows.

First, we solve the unknown v in the BOE.

Suppose v^* is a solution to $v = f(v) = \max_\pi(r_\pi + \gamma P_\pi v)$. Then, it holds that

$$v^* = \max_\pi(r_\pi + \gamma P_\pi v^*).$$

Clearly, v^* is a fixed point because $v^* = f(v^*)$. Then, the contraction mapping theorem suggests the following results.

Theorem 3.3 (Existence, Uniqueness, and Algorithm). *For the BOE $v = f(v) = \max_\pi(r_\pi + \gamma P_\pi v)$, there always exists a unique solution v^*, which can be solved iteratively by*

$$v_{k+1} = f(v_k) = \max_\pi(r_\pi + \gamma P_\pi v_k), \quad k = 0, 1, \ldots.$$

The sequence $\{v_k\}$ converges to v^* exponentially fast given any initial guess v_0.

Proof. Since $f(v)$ is a contraction mapping as proven in Theorem 3.2, all the results here follow directly from the contraction mapping theorem. □

This theorem answers some fundamental questions.

– Existence: The solution v^* to the BOE always exists.

– Uniqueness: The solution v^* is always unique.

– Algorithm: v^* can be solved iteratively by the iterative algorithm in Theorem 3.3.

The iterative algorithm given in Theorem 3.3 is an important RL algorithm. This algorithm, which has a name called *value iteration*, will be studied in detail in the next chapter. In this chapter, we merely treat this algorithm as a numerical algorithm solving the BOE and focus on the fundamental properties of the BOE.

Second, we solve the unknown π in the BOE. Suppose v^* has been obtained and

$$\pi^* = \arg\max_{\pi}(r_\pi + \gamma P_\pi v^*). \tag{3.6}$$

Then, v^* and π^* satisfy

$$v^* = r_{\pi^*} + \gamma P_{\pi^*} v^*.$$

Therefore, $v^* = v_{\pi^*}$ is a state value of π^* and hence the BOE is the Bellman equation under the policy π^*.

Up to now, we merely know that v^* is the solution to the BOE. Whether it is the optimal state value is still unclear. The following theorem shows that v^* is the optimal state value and π^* is an optimal policy.

Theorem 3.4 (Optimality)**.** *For any policy π, it holds that*

$$v^* = v_{\pi^*} \geq v_\pi,$$

where v_π is the state value of π.

Proof of optimality

Since

$$\begin{aligned} v_\pi &= r_\pi + \gamma P_\pi v_\pi, \\ v^* &= \max_\pi(r_\pi + \gamma P_\pi v^*) = r_{\pi^*} + \gamma P_{\pi^*} v^* \geq r_\pi + \gamma P_\pi v^*, \end{aligned}$$

we have

$$v^* - v_\pi \geq (r_\pi + \gamma P_\pi v^*) - (r_\pi + \gamma P_\pi v_\pi) = \gamma P_\pi (v^* - v_\pi).$$

Using the above inequality recursively gives $v^* - v_\pi \geq \gamma P_\pi(v^* - v_\pi) \geq \gamma^2 P_\pi^2(v^* - v_\pi) \geq \cdots \geq \gamma^n P_\pi^n(v^* - v_\pi)$. It follows that

$$v^* - v_\pi \geq \lim_{n\to\infty} \gamma^n P_\pi^n(v^* - v_\pi) = 0,$$

where the last equality is due to $\gamma < 1$ and P_π^n is a nonnegative matrix with all elements no greater than 1 (because $P_\pi^n \mathbf{1} = \mathbf{1}$).

Finally, we know that the two unknowns v and π in the BOE correspond to the optimal state value and optimal policy, respectively. This is the reason why it is important to study the BOE.

What does an optimal policy look like? The following theorem shows that a deterministic greedy policy is optimal.

Theorem 3.5 (Greedy optimal policy). *For any $s \in \mathcal{S}$, the deterministic greedy policy*

$$\pi^*(a|s) = \begin{cases} 1 & a = a^*(s) \\ 0 & a \neq a^*(s) \end{cases} \tag{3.7}$$

is an optimal policy solving the BOE. Here,

$$a^*(s) = \arg\max_a q^*(a, s),$$

where

$$q^*(s,a) \doteq \sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a)v^*(s').$$

Proof. The matrix-vector form of the optimal policy is $\pi^* = \arg\max_\pi(r_\pi + \gamma P_\pi v^*)$. Its elementwise form is

$$\pi^*(s) = \arg\max_\pi \sum_a \pi(a|s) \underbrace{\left(\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a)v^*(s')\right)}_{q^*(s,a)}, \quad s \in \mathcal{S}.$$

It is clear that $\sum_a \pi(a|s)q^*(s,a)$ is maximized if $\pi(s)$ selects the action with the greatest value of $q^*(s,a)$. □

The policy in (3.7) is called *greedy*, because it seeks the actions with the greatest $q^*(s,a)$.

Finally, we discuss the last two questions about the BOE.

– Uniqueness of optimal policies: is the optimal policy π^* unique? Although the value of v^* is unique, the optimal policy corresponding to v^* may not be unique. It can be easily verified by counterexamples. For example, the two policies shown in Figure 3.2 are both optimal.

– Stochasticity of optimal policies: is the optimal policy stochastic or deterministic? The optimal policy can be either stochastic or deterministic as demonstrated in Figure 3.2. However, it is certain that there always exists a deterministic optimal policy as shown in Theorem 3.5.

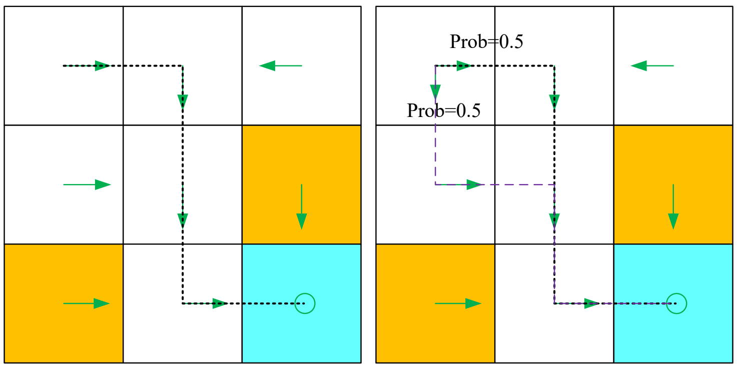


Figure 3.2: Examples to demonstrate that optimal policies may not unique.

3.5 Factors that influence optimal policies

The BOE is a powerful tool to analyze optimal policies. We next further use the BOE to study what kind of factors can influence optimal policies. This question can be answered by observing the elementwise form of the BOE:

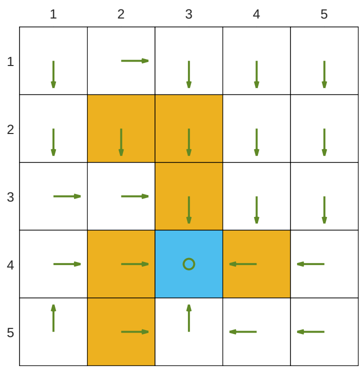
$$v(s) = \max_{\pi} \sum_a \pi(a|s) \left(\sum_r p(r|s,a) r + \gamma \sum_{s'} p(s'|s,a) v(s') \right), \quad s \in \mathcal{S}.$$

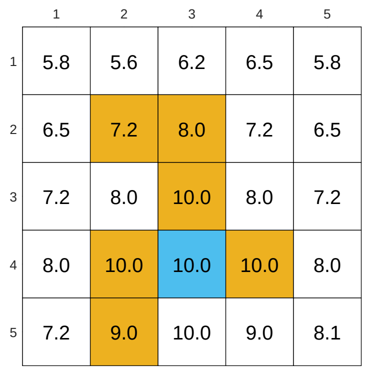
The parameters in this equation include the immediate reward r, the discount rate γ, and the system model $p(s'|s,a), p(r|s,a)$. While the system model is fixed in general, we next discuss how the optimal policy changes if we change the values of r and γ. All the optimal policies presented in this section can be obtained by the algorithm in Theorem 3.3. The details of the algorithm are given in the next chapter. This chapter mainly focuses on fundamental problems rather than algorithms.

A baseline example

Consider the example in Figure 3.3. The reward setting is $r_{\text{boundary}} = r_{\text{forbidden}} = -1$ and $r_{\text{target}} = 1$. Besides, the agent receives a reward of $r = 0$ for every step of movements. The discount rate is selected as $\gamma = 0.9$.

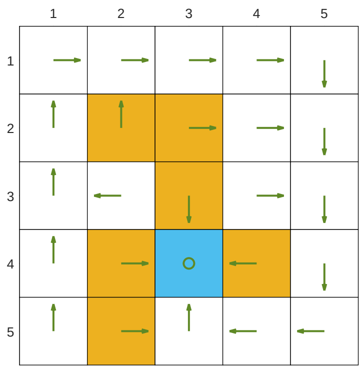
With the above parameters, the optimal policy and optimal state values are given in Figure 3.3(a). It is interesting to note that the agent is not afraid of passing through forbidden areas to reach the target area. For example, starting from the state at (row=4, column=1), the agent has two options to reach the target area. The first is to avoid all

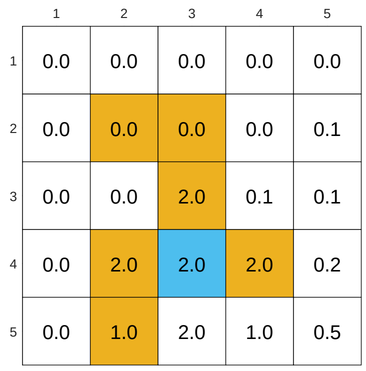




	1	2	3	4	5
1	5.8	5.6	6.2	6.5	5.8
2	6.5	7.2	8.0	7.2	6.5
3	7.2	8.0	10.0	8.0	7.2
4	8.0	10.0	10.0	10.0	8.0
5	7.2	9.0	10.0	9.0	8.1

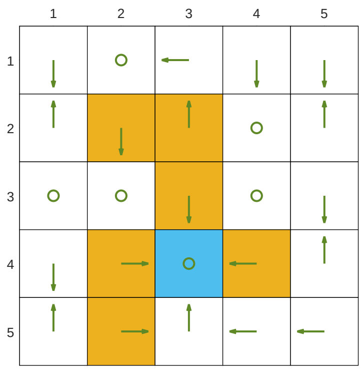
(a) Baseline example: $r_{\text{boundary}} = r_{\text{forbidden}} = -1$, $r_{\text{target}} = 1$, $\gamma = 0.9$

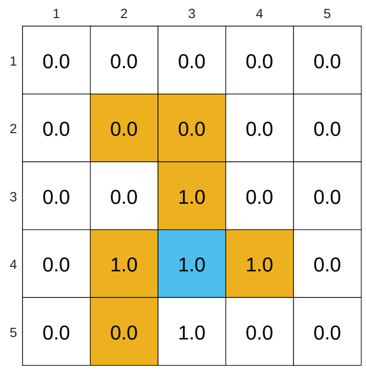




	1	2	3	4	5
1	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.1
3	0.0	0.0	2.0	0.1	0.1
4	0.0	2.0	2.0	2.0	0.2
5	0.0	1.0	2.0	1.0	0.5

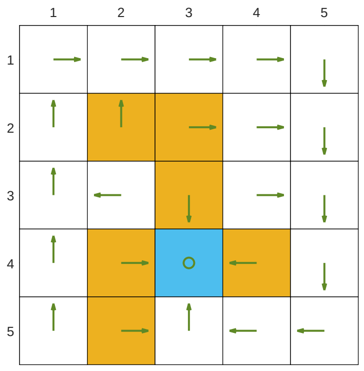
(b) The discount rate is $\gamma = 0.5$. Other parameters are the same as (a).

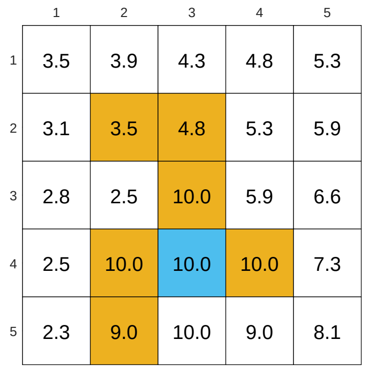




	1	2	3	4	5
1	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	1.0	0.0	0.0
4	0.0	1.0	1.0	1.0	0.0
5	0.0	0.0	1.0	0.0	0.0

(c) The discount rate is $\gamma = 0$. Other parameters are the same as (a).





	1	2	3	4	5
1	3.5	3.9	4.3	4.8	5.3
2	3.1	3.5	4.8	5.3	5.9
3	2.8	2.5	10.0	5.9	6.6
4	2.5	10.0	10.0	10.0	7.3
5	2.3	9.0	10.0	9.0	8.1

(d) $r_{\text{forbidden}} = -10$. Other parameters are the same as (a).

Figure 3.3: Optimal policies and optimal state values given different values of parameters.

the forbidden areas and travel a long distance to the target area. The second is to pass through the forbidden areas. Although the agent gets negative rewards when entering forbidden areas, the overall cumulative reward of the second trajectory is greater than that of the first trajectory.

The impact of discount rate

If we change the discount rate from $\gamma = 0.9$ to $\gamma = 0.5$ and keep other parameters unchanged, the optimal policy is shown in Figure 3.3(b). It is interesting to see that the agent would not take risks in this case. It instead avoids all the forbidden areas to reach the target. That is simply because the agent becomes short-sighted. It would not take any risk to get punished even though it can get greater rewards afterward.

In the extreme case where we set $\gamma = 0$, the optimal policy is given in Figure 3.3(c). In this case, the agent will not be able to reach the target area. That is because the optimal policy for each state is extremely short-sighted and merely selects the action that has the greatest immediate reward (instead of the greatest return). As a result, only those states that are adjacent to the target have nonzero state values and can select correct actions. Those states that are not adjacent to the target randomly select an action that has the greatest rewards.

The impact of reward signals

If we want to strictly prohibit the agent from entering any forbidden area, we can increase the punishment. For instance, if the reward of entering a forbidden area is changed from -1 to -10, the optimal policy in this case would avoid all the forbidden areas (see Figure 3.3(d)).

Changing rewards does not always lead to different optimal policies. One important fact is that optimal policies are *invariant* to affine transformations of rewards. In other words, if we scale up or down all the rewards or add the same number to all the rewards, the optimal policy would remain the same.

Theorem 3.6 (Optimal policy invariance)**.** *Consider a Markov decision process with* $v^* \in \mathbb{R}^{|\mathcal{S}|}$ *as the optimal state value satisfying* $v^* = \max_\pi(r_\pi + \gamma P_\pi v^*)$*. If every reward* r *is changed by an affine transformation to* $ar + b$*, where* $a, b \in \mathbb{R}$ *and* $a > 0$*, then the corresponding optimal state value* v' *is also an affine transformation of* v^**:*

$$v' = av^* + \frac{b}{1-\gamma}\mathbf{1}, \tag{3.8}$$

where $\gamma \in (0, 1)$ *is the discount rate and* $\mathbf{1} = [1, \dots, 1]^T$*. Consequently, the optimal policies are invariant to the affine transformation of the reward signals.*

Proof of the optimal policy invariance theorem

For any policy π, define $r_\pi = [\dots, r_\pi(s), \dots]^T$ where

$$r_\pi(s) = \sum_a \pi(a|s) \sum_r p(r|s,a)r, \quad s \in \mathcal{S}.$$

If $r \to ar+b$, then $r_\pi(s) \to ar_\pi(s)+b$ and hence $r_\pi \to ar_\pi + b\mathbf{1}$, where $\mathbf{1} = [1, \dots, 1]^T$. In this case, the BOE becomes

$$v' = \max_\pi (ar_\pi + b\mathbf{1} + \gamma P_\pi v'). \tag{3.9}$$

We next solve the new BOE in (3.9). To do that, we verify that $v' = av^* + k\mathbf{1}$ with $k = b/(1-\gamma)$ is the solution of (3.9). In particular, substituting $v' = av^* + k\mathbf{1}$ into (3.9) gives

$$av^* + k\mathbf{1} = \max_\pi [ar_\pi + b\mathbf{1} + \gamma P_\pi(av^* + k\mathbf{1})] = \max_\pi (ar_\pi + b\mathbf{1} + a\gamma P_\pi v^* + k\gamma\mathbf{1}),$$

where the last equation is due to $P_\pi \mathbf{1} = \mathbf{1}$. The above equation can be rewritten as

$$av^* = \max_\pi (ar_\pi + a\gamma P_\pi v^*) + b\mathbf{1} + k\gamma\mathbf{1} - k\mathbf{1},$$

which is equivalent to

$$b\mathbf{1} + k\gamma\mathbf{1} - k\mathbf{1} = 0.$$

Since $k = b/(1-\gamma)$, the above equation is valid and hence $v' = av^* + k\mathbf{1}$ is the solution to (3.9). Since (3.9) is the BOE, v' is also the unique solution. Finally, since v' is an affine transformation of v^*, the relative relationship among the action values remain the same. Hence, v' would lead to the same optimal policies as v^*.

Designing appropriate rewards in RL is of practical challenge. Interested readers may see more discussion in [5].

Avoiding meaningless detour

In the reward setting, the agent would receive a reward of $r = 0$ for every step unless it enters a forbidden or the target area or attempts to get out of the boundary. Since a zero reward is not punishment, would the optimal policy take meaningless detours before reaching the target?

To be specific, consider the examples in Figure 3.4, where the target cell is the right bottom one. The two policies here are the same except for state s_2. In particular, the agent would move downwards at s_2 following the first policy in Figure 3.4(a) and move downwards following the second policy in Figure 3.4(b). As a result, the resulting trajectory is $s_2 \to s_4$ by the first policy and $s_2 \to s_1 \to s_3 \to s_4$ by the second policy. It is noted that the second policy takes a detour before reaching the target area.

Does this detour matter? If we merely consider the immediate rewards, taking this detour would not matter, because no negative immediate rewards will be obtained. However, if we consider the discounted return, then this detour matters. In particular, for the first policy, the discounted return is

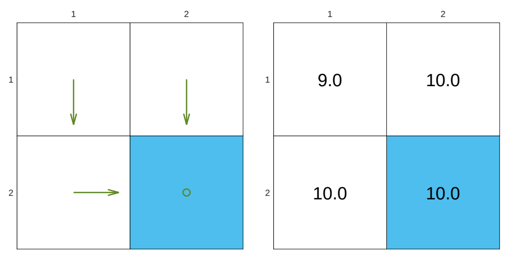
$$\text{return} = 1 + \gamma 1 + \gamma^2 1 + \cdots = 1/(1-\gamma) = 10.$$

As a comparison, the discounted return for the second policy is

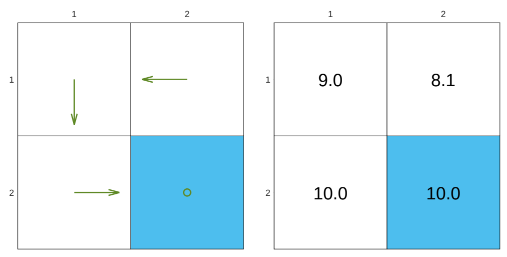
$$\text{return} = 0 + \gamma 0 + \gamma^2 1 + \gamma^3 1 + \cdots = \gamma^2/(1-\gamma) = 8.1.$$

It is clear that the shorter the trajectory is, the greater the return is. Therefore, although the immediate reward of every step does not encourage the agent to approach the target as quickly as possible, the discount rate does.

A misunderstanding that beginners may have is that adding a negative reward (say -1) on top of the rewards for every move is necessary to encourage the agent to reach the target as quickly as possible. This is a mistake because adding the same amount of reward on top of all rewards is an affine transformation. We have shown that affine transformation of the rewards would not change the optimal policies. As we analyzed above, due to the discount rate, optimal policies would not take meaningless detours before reaching the target even though a detour may not get any negative immediate rewards.



(a) Optimal policy



(b) Not optimal

Figure 3.4: Examples illustrating that optimal policies do not take meaningless detours in the presence of discount rate.

An interesting pattern of the spatial distribution of the state values is that the states close to the target have greater state values, whereas those far away have less values. This pattern can be observed from all the examples shown in Figure 3.3. This interesting pattern can also be explained by using the discount rate. That is, if a state has to travel a longer trajectory to the target, its state value would be less due to the discount rate.

3.6 Summary

The core concept in this chapter is optimal policy, which is defined based on optimal state values. In particular, a policy is optimal if its state values are greater than or equal to those of all the other policies. In order to analyze optimal policies, we have to study the BOE. This equation is a nonlinear equation with a nice contraction property. Based on this property, we can apply the contraction mapping theorem to solve this equation. We proved that the solution to the BOE always exists and is unique. Its solution is the optimal state value, which is the greatest state value that can be achieved by any policy. The corresponding optimal policies may not be unique. While optimal policies may be either stochastic or deterministic, a nice property is that there always exist deterministic greedy optimal policies.

The contents in this chapter are important for thoroughly understanding many fundamental ideas of RL. For example, Theorem 3.3 suggests an iterative algorithm for solving the BOE. This algorithm is exactly the value iteration algorithm introduced in the next chapter.

3.7 Q&A

– Q: What is the definition of optimal policy?

  A: A policy is optimal if the corresponding state values are greater than or equal to all the other policies. It should be noted that this specific definition of optimality is valid only for value-based RL algorithms. When policies are approximated by functions, different metrics will be used to define optimal policies. Details will be given when we introduce policy gradient algorithms.

– Q: Do optimal policies exist?

  A: Yes. There always exist optimal policies according to the BOE.

– Q: Are optimal policies unique?

  A: No. There may exist multiple or infinite optimal policies which have the same optimal state values.

– Q: Are optimal policies stochastic or deterministic?

  A: An optimal policy can be either deterministic or stochastic. A nice fact is that there always exist deterministic greedy optimal policies.

– Q: How to obtain an optimal policy?

  A: Solving the BOE is one way. The value iteration algorithm as introduced in the next chapter is an algorithm solving the BOE. Of course, many other RL algorithms can also obtain optimal policies as shown later in this book.

– Q: Why is the Bellman optimality equation important?

  A: It is important to study the BOE because it characterizes both optimal policies and optimal state values.

– Q: Is the Bellman optimality equation a Bellman equation?

  A: Yes. The Bellman optimality equation is a special Bellman equation. The corresponding policy is optimal. The corresponding state value is the optimal state value.

– Q: Is the solution to the Bellman optimality equation unique?

  A: The Bellman optimality equation has two unknowns. The first is a value and the second is a policy. The solution to the value, which is the optimal state value, is unique. The solution to the policy, which is an optimal policy, may not be unique.

– Q: What is the key property of the Bellman optimality equation for us analyzing its solution?

  A: The key property is that the right-hand side of the Bellman optimality equation is a contraction mapping. As a result, we can apply the contraction mapping theorem or called the fixed-point theorem to analyze its solution.

– Q: What is the general impact on the optimal policies if we reduce the value of the discount rate?

  A: The optimal policy will become short-sighted when we reduce the discount rate. That is, the agent does not dare to take high risks even though they may get greater cumulative rewards afterward.

– Q: What if we set the discount rate to zero?

  A: The agent would become extremely short-sighted. The optimal policy would be purely based on immediate rewards. That is the agent would take the action that has the greatest immediate reward, even though that action is not good in the long run.

– Q: If we increase all the rewards by the same amount, will the optimal state value change? Will the optimal policy change?

  A: Increasing all the rewards by the same amount is an affine transformation of the rewards, which would not affect the optimal policies. However, the optimal state value will increase as shown in (3.8).

– Q: If we hope that the optimal policy can avoid meaningless detours before reaching the target, should we add a negative reward to every step so that the agent would reach the target as quickly as possible?

  A: No, it is useless to introduce an additional negative reward to every step. Such an operation is an affine transformation of the rewards, which would not change the optimal policies. In fact, the discount rate can do the job to encourage the agent to reach the target as soon as possible. That is because meaningless detours would increase the length of the trajectory and hence reduce the discounted return.

Chapter 4

Value Iteration and Policy Iteration

With the preparation in the previous chapters, we are ready now to present the first reinforcement learning (RL) algorithms: *value iteration* and *policy iteration*. They have a common name called *dynamic programming*. The algorithms introduced in this chapter are *model-based* since they require knowing the exact probability model of the environment. They are the simplest RL algorithms. Understanding them is the foundation for understanding *model-free* RL algorithms that will be introduced later in the book.

This chapter contains three parts. The first part introduces the value iteration algorithm. This algorithm is exactly the algorithm suggested by the contraction mapping theorem to solve the Bellman optimality equation. Here, we introduce the implementation details of this algorithm. The second part introduces the policy iteration algorithm. This algorithm is fundamental in RL. Its idea is widely used by many model-free RL algorithms. The third part introduces truncated policy iteration, which is an algorithm that can unify value iteration and policy iteration. We will show that value iteration and policy iteration are two special cases of truncated policy iteration.

4.1 Value iteration

This section introduces the value iteration algorithm, one classic model-based RL algorithm. This algorithm is exactly the algorithm suggested by the contraction mapping theorem to solve the Bellman optimality equation as introduced in the last chapter (Theorem 3.3). For a quick reference, it is

$$v_{k+1} = \max_{\pi}(r_\pi + \gamma P_\pi v_k), \quad k = 0, 1, 2, \dots$$

It is guaranteed by Theorem 3.3 that v_k will converge to the optimal state value and π_k will converge to an optimal policy. This algorithm is iterative. In order to implement it, we can further decompose every iteration into two steps.

– The first step in every iteration is called *policy update*. Mathematically, it is to find a policy solving the following optimization problem:

$$\pi_{k+1} = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_k).$$

– The second step is called *value update*. Mathematically, it is to substitute π_{k+1} and do the following operation:

$$v_{k+1} = r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_k.$$

The above description of the value iteration algorithm relies on the matrix-vector form of the Bellman optimality equation. However, in order to implement the algorithm, we need to introduce the elementwise form of the algorithm. The matrix-vector form is useful to understand the core idea of the algorithm, whereas the elementwise form is good for explaining the implementation details.

4.1.1 Elementwise form and implementation

We next show how to implement the value iteration algorithm in detail.

– First, the elementwise form of the *policy update step* $\pi_{k+1} = \arg\max_\pi(r_\pi + \gamma P_\pi v_k)$ is

$$\pi_{k+1}(s) = \arg\max_{\pi} \sum_a \pi(a|s) \underbrace{\left(\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) v_k(s')\right)}_{q_k(s,a)}, \quad s \in \mathcal{S}.$$

The optimal policy solving the above optimization problem is

$$\pi_{k+1}(a|s) = \begin{cases} 1 & a = a_k^*(s), \\ 0 & a \neq a_k^*(s). \end{cases}$$

where $a_k^*(s) = \arg\max_a q_k(a,s)$. If $a_k^*(s) = \arg\max_a q_k(a,s)$ has multiple solutions, we can simply select any of them. Since the new policy π_{k+1} selects an action with the greatest value of $q_k(s,a)$, such a policy is greedy.

– Second, the elementwise form of the *value update step* $v_{k+1} = r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_k$ is

$$v_{k+1}(s) = \sum_a \pi_{k+1}(a|s) \underbrace{\left(\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) v_k(s')\right)}_{q_k(s,a)}, \quad s \in \mathcal{S}.$$

Since π_{k+1} is a greedy policy, the above equation is simply

$$v_{k+1}(s) = \max_a q_k(a,s).$$

The details of the implementation are summarized in the pseudocode.

Pseudocode: Value iteration algorithm

Initialization: The probability model $p(r|s,a)$ and $p(s'|s,a)$ for all (s,a) are known. Initial guess v_0.
Aim: Search for the optimal state value and an optimal policy solving the Bellman optimality equation.

```
While the state value has not converged, for the kth iteration, do
    For every state $s \in \mathcal{S}$, do
        For every action $a \in \mathcal{A}(s)$, do
            q-value: $q_k(s,a) = \sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a)v_k(s')$
        Maximum action value: $a_k^*(s) = \arg\max_a q_k(a,s)$
        Policy update: $\pi_{k+1}(a|s) = 1$ if $a = a_k^*$, and $\pi_{k+1}(a|s) = 0$ otherwise
        Value update: $v_{k+1}(s) = \max_a q_k(a,s)$
```

One problem of value iteration that may confuse beginners is whether v_k is a state value. The answer is no although v_k converges to the optimal state value eventually. That is because v_k is not guaranteed to satisfy any Bellman equation. For example, we do not have $v_k = r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_k$ or $v_k = r_{\pi_k} + \gamma P_{\pi_k} v_k$ in general. When representing a state value, we will explicitly indicate the corresponding policy in the subscript. For example, v_{π_k} is the state value of policy π_k. Instead, v_k is not a state value and simply the kth intermediate value of an iterative process. Since v_k is not a state value, q_k is not an action value either. This might be one of the most confusing problems for beginners studying this algorithm. It is easier if we simply treat them as intermediate variables that emerge when solving the Bellman optimality equation.

4.1.2 Illustrative examples

We next present an example to illustrate the implementation details of the value iteration algorithm.

This example is a two-by-two grid with one forbidden area and the target area is s_4. The reward setting is $r_{\text{boundary}} = r_{\text{forbidden}} = -1$, $r_{\text{target}} = 1$. The discount rate is $\gamma = 0.9$.

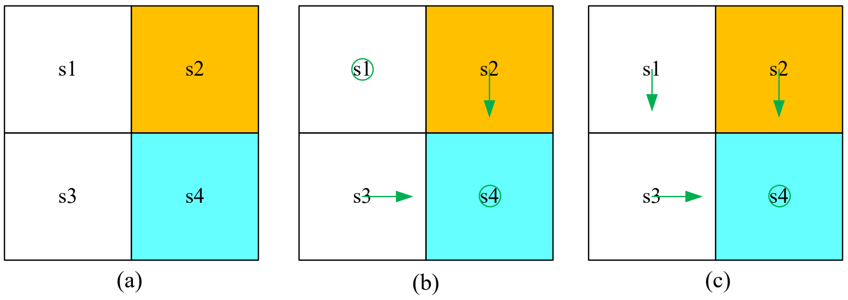


Figure 4.1: An example to demonstrate the implementation of the value iteration algorithm.

The expression of the q-value for each state-action pair is shown in Table 4.1, where

q-value	a_1	a_2	a_3	a_4	a_5
s_1	$-1+\gamma v(s_1)$	$-1+\gamma v(s_2)$	$0+\gamma v(s_3)$	$-1+\gamma v(s_1)$	$0+\gamma v(s_1)$
s_2	$-1+\gamma v(s_2)$	$-1+\gamma v(s_2)$	$1+\gamma v(s_4)$	$0+\gamma v(s_1)$	$-1+\gamma v(s_2)$
s_3	$0+\gamma v(s_1)$	$1+\gamma v(s_4)$	$-1+\gamma v(s_3)$	$-1+\gamma v(s_3)$	$0+\gamma v(s_3)$
s_4	$-1+\gamma v(s_2)$	$-1+\gamma v(s_4)$	$-1+\gamma v(s_4)$	$0+\gamma v(s_3)$	$1+\gamma v(s_4)$

Table 4.1: The expression of $q(s,a)$ for the example shown in Figure 4.1.

q-value	a_1	a_2	a_3	a_4	a_5
s_1	-1	-1	0	-1	0
s_2	-1	-1	1	0	-1
s_3	0	1	-1	-1	0
s_4	-1	-1	-1	0	1

Table 4.2: The value of $q(a,s)$ at $k=0$.

q-table	a_1	a_2	a_3	a_4	a_5
s_1	$-1+\gamma 0$	$-1+\gamma 1$	$0+\gamma 1$	$-1+\gamma 0$	$0+\gamma 0$
s_2	$-1+\gamma 1$	$-1+\gamma 1$	$1+\gamma 1$	$0+\gamma 0$	$-1+\gamma 1$
s_3	$0+\gamma 0$	$1+\gamma 1$	$-1+\gamma 1$	$-1+\gamma 1$	$0+\gamma 1$
s_4	$-1+\gamma 1$	$-1+\gamma 1$	$-1+\gamma 1$	$0+\gamma 1$	$1+\gamma 1$

Table 4.3: The value of $q(a,s)$ at $k=1$.

each row is indexed by a state and each column by an action.

– When $k=0$: set the initial values as $v_0(s_1)=v_0(s_2)=v_0(s_3)=v_0(s_4)=0$.

  q-value calculation: Substituting $v_0(s_i)$ into Table 4.1 gives the q-values as shown in Table 4.2.

  Policy update: Select the actions with the greatest q-value:

$$\pi_1(a_5|s_1)=1,\quad \pi_1(a_3|s_2)=1,\quad \pi_1(a_2|s_3)=1,\quad \pi_1(a_5|s_4)=1.$$

  This policy is visualized in Figure 4.1(b). It is clear that this policy is not good, because it selects to stay unchanged at s_1. It is worth mentioning that, since the q-values for (s_1,a_5) and (s_1,a_3) are the same for s_1, we randomly selected an action.

  Value update: Update the v-value as the greatest q-value for each state:

$$v_1(s_1)=0,\quad v_1(s_2)=1,\quad v_1(s_3)=1,\quad v_1(s_4)=1.$$

– When $k=1$:

  q-value calculation: Substituting $v_1(s_i)$ into Table 4.1 gives the q-values as shown in Table 4.3.

Policy update: The policy is updated to select the greatest q-values:

$$\pi_2(a_3|s_1) = 1, \quad \pi_2(a_3|s_2) = 1, \quad \pi_2(a_2|s_3) = 1, \quad \pi_2(a_5|s_4) = 1.$$

This policy is visualized in Figure 4.1(c).

Value update: Update the v-value as the greatest q-value for each state:

$$v_2(s_1) = \gamma 1, v_2(s_2) = 1 + \gamma 1, v_2(s_3) = 1 + \gamma 1, v_2(s_4) = 1 + \gamma 1.$$

– $k = 2, 3, 4, \ldots$

In fact, policy π_2 as illustrated in Figure 4.1(c) is already optimal. In practice, we can run a few more steps until the value of v_k converges (that is v_k only varies a sufficiently small amount between consequent steps).

4.2 Policy iteration

This section presents another important algorithm, policy iteration. Different from value iteration, policy iteration is not an algorithm directly solving the Bellman optimality equation. However, it has an intimate relationship to value iteration. Moreover, its idea is widely used by many other RL algorithms such as Monte Carlo learning as we will see in the next chapter.

4.2.1 Algorithm analysis

Policy iteration is also an iterative algorithm. In each iteration, it has two steps.

– The first step is *policy evaluation.* As its name suggests, this step aims to evaluate a given policy by calculating the corresponding state value. Mathematically, it is to solve the Bellman equation of π_k:

$$v_{\pi_k} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}. \tag{4.1}$$

This is the matrix-vector form of the Bellman equation, where r_{π_k} and P_{π_k} are known. Here, v_{π_k} is the state value to be solved.

– The second step is *policy improvement.* As its name suggests, this step is to improve the policy. How to do that? Once v_{π_k} is calculated in the first step, a new and improved policy could be obtained as

$$\pi_{k+1} = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_{\pi_k}).$$

Here, the maximization is componentwise.

Three questions naturally follow the above description of the algorithm.

– In the policy evaluation step, how to solve the state value v_{π_k}?

– In the policy improvement step, why is the new policy π_{k+1} better than π_k?

– Why can this algorithm finally reach an optimal policy?

We next answer the three questions one by one.

How to calculate v_{π_k}?

We have already studied the methods to solve the Bellman equation in (4.1) in Chapter 2. We revisit the two methods briefly as follows. The first is a closed-form solution: $v_{\pi_k} = (I-\gamma P_{\pi_k})^{-1} r_{\pi_k}$. Although the closed-form solution is useful for theoretical analysis, it is not practical since it involves calculating a matrix inverse which requires sophisticated numerical algorithms. The second method is an iterative algorithm that can be implemented easily:

$$v_{\pi_k}^{(j+1)} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}^{(j)}, \quad j = 0, 1, 2, ... \tag{4.2}$$

Starting from any initial guess $v_{\pi_k}^{(0)}$, it is ensured that $v_{\pi_k}^{(j)} \to v_{\pi_k}$ as $j \to \infty$.

While the policy evaluation step involves an iterative algorithm as in (4.2), it is interesting to note that policy iteration is an iterative algorithm with another iterative algorithm embedded in the policy evaluation step. This embedded iterative algorithm (4.2) requires an *infinite* number of steps to converge to the true state value v_{π_k} in theory. This is, however, impossible to achieve in practice. In practice, the iteration would stop when a criterion is satisfied. For example, $\|v_{\pi_{k+1}} - v_{\pi_k}\|$ is less than a prespecified small value or j exceeds certain threshold. If we can not calculate the precise value of v_{π_k} by running an infinite number of steps, the imprecise value will be substituted to the policy improvement step. Would it cause problems? The answer is no. The reason will be clear when we introduce the truncated policy iteration algorithm later in this chapter.

Why π_{k+1} is better than π_k?

Why can the policy improvement step improve the policy? The answer is as follows.

Lemma 4.1 (Policy improvement). *If* $\pi_{k+1} = \arg\max_\pi (r_\pi + \gamma P_\pi v_{\pi_k})$, *then* $v_{\pi_{k+1}} \geq v_{\pi_k}$.

Here, $\geq$ is componentwise. That is, $v_{\pi_{k+1}} \geq v_{\pi_k}$ means $v_{\pi_{k+1}}(s) \geq v_{\pi_k}(s)$ for all s. The proof of the lemma is given in the shaded box.

Proof of Lemma 4.1

First of all, since $v_{\pi_{k+1}}$ and v_{π_k} are state values, they satisfy the Bellman equation:

$$\begin{aligned} v_{\pi_{k+1}} &= r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_{k+1}}, \\ v_{\pi_k} &= r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}. \end{aligned}$$

Since $\pi_{k+1} = \arg\max_\pi (r_\pi + \gamma P_\pi v_{\pi_k})$, we know

$$r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_k} \geq r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}.$$

It then follows that

$$\begin{aligned} v_{\pi_k} - v_{\pi_{k+1}} &= r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k} - (r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_{k+1}}) \\ &\leq r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_k} - (r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_{k+1}}) \\ &\leq \gamma P_{\pi_{k+1}} (v_{\pi_k} - v_{\pi_{k+1}}). \end{aligned}$$

Therefore,

$$\begin{aligned} v_{\pi_k} - v_{\pi_{k+1}} \leq \gamma^2 P^2_{\pi_{k+1}} (v_{\pi_k} - v_{\pi_{k+1}}) \leq \ldots &\leq \gamma^n P^n_{\pi_{k+1}} (v_{\pi_k} - v_{\pi_{k+1}}) \\ &\leq \lim_{n\to\infty} \gamma^n P^n_{\pi_{k+1}} (v_{\pi_k} - v_{\pi_{k+1}}) = 0. \end{aligned}$$

The limit is due to $\gamma^n \to 0$ as $n \to \infty$ whereas $P^n_{\pi_{k+1}}$ is always a nonnegative stochastic matrix for any n. Here, a stochastic matrix refers to a nonnegative matrix whose row sum is equal to one for all rows.

Why can policy iteration eventually find an optimal policy?

The policy iteration algorithm generates a sequence: $v_{\pi_0}, v_{\pi_1}, v_{\pi_2}, \ldots, v_{\pi_k}, \ldots$. Suppose v^* is the optimal state value. Then, $v_{\pi_k} \leq v^*$ for all k. Since the policies are continuously improved according to Lemma 4.1, we know that

$$v_{\pi_0} \leq v_{\pi_1} \leq v_{\pi_2} \leq \cdots \leq v_{\pi_k} \leq \cdots \leq v^*.$$

Since v_{π_k} is nondecreasing and always bounded from above by v^*, it follows from the monotone convergence theorem that v_{π_k} will converge to a constant value v_∞ as $k \to \infty$. However, we are not sure whether v_∞ is v^* at this moment. To see that, we give a rigorous analysis in the following result.

Theorem 4.1 (Convergence of policy iteration). *The state value sequence $\{v_{\pi_k}\}_{k=0}^\infty$ generated by the policy iteration algorithm converges to the optimal state value v^*. As a result, the policy sequence $\{\pi_k\}_{k=0}^\infty$ converges to an optimal policy.*

Proof of Theorem 4.1

The idea of the proof is to show that policy iteration converge faster than value iteration. Since value iteration has been proven to be convergent, the convergence of policy iteration immediately follows.

In particular, to prove the convergence of $\{v_{\pi_k}\}_{k=0}^{\infty}$, we introduce another sequence $\{v_k\}_{k=0}^{\infty}$ generated by

$$v_{k+1} = f(v_k) = \max_{\pi}(r_\pi + \gamma P_\pi v_k).$$

This iterative algorithm is exactly the value iteration algorithm. We already know that v_k converges to v^* given any initial value v_0.

We next show that $v_k \leq v_{\pi_k} \leq v^*$ for all k by induction.

For $k = 0$, we can always able to select v_0 such that $v_{\pi_0} \geq v_0$ for any π_0.

For $k \geq 1$, suppose $v_{\pi_k} \geq v_k$.

For $k+1$,

$$\begin{aligned}
v_{\pi_{k+1}} - v_{k+1} &= (r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_{k+1}}) - \max_{\pi}(r_\pi + \gamma P_\pi v_k) \\
&\geq (r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_k}) - \max_{\pi}(r_\pi + \gamma P_\pi v_k) \\
&\qquad\qquad \big(\text{because } v_{\pi_{k+1}} \geq v_{\pi_k} \text{ by Lemma 4.1 and } P_{\pi_{k+1}} \geq 0\big) \\
&= (r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_k}) - (r_{\pi'_k} + \gamma P_{\pi'_k} v_k) \\
&\qquad\qquad \big(\text{suppose } \pi'_k = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_k)\big) \\
&\geq (r_{\pi'_k} + \gamma P_{\pi'_k} v_{\pi_k}) - (r_{\pi'_k} + \gamma P_{\pi'_k} v_k) \\
&\qquad\qquad \big(\text{because } \pi_{k+1} = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_{\pi_k})\big) \\
&= \gamma P_{\pi'_k}(v_{\pi_k} - v_k).
\end{aligned}$$

Since $v_{\pi_k} - v_k \geq 0$ and $P_{\pi'_k}$ is nonnegative, we have $P_{\pi'_k}(v_{\pi_k} - v_k) \geq 0$ and hence $v_{\pi_{k+1}} - v_{k+1} \geq 0$.

Therefore, we show by induction that $v_{\pi_k} \geq v_k$ for any $k \geq 0$. Since v_k converges to v^*, v_{π_k} also converges to v^*.

The above proof not only shows the convergence of policy iteration but also reveals the relationship between policy iteration and value iteration. Loosely speaking, if the two algorithms start from the same initial guess of the state value, policy iteration will converge faster than value iteration due to the additional infinite numbers of iterations embedded in the policy evaluation step. This point will be more clear when we introduce the truncated policy iteration algorithm later.

4.2.2 Elementwise form and implementation

In order to implement the policy iteration algorithm, it is necessary to study its elementwise form.

- First, the policy evaluation step is to solve v_{π_k} from the Bellman equation $v_{\pi_k} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}$ by using the iterative algorithm in (4.2). The elementwise form of this algorithm is

$$v_{\pi_k}^{(j+1)}(s) = \sum_a \pi_k(a|s) \left(\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) v_{\pi_k}^{(j)}(s') \right), \quad s \in \mathcal{S},$$

  where $j = 0, 1, 2, \ldots$.

- Second, the policy improvement step is to solve $\pi_{k+1} = \arg\max_\pi (r_\pi + \gamma P_\pi v_{\pi_k})$. The elementwise form of this equation is

$$\pi_{k+1}(s) = \arg\max_\pi \sum_a \pi(a|s) \underbrace{\left(\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) v_{\pi_k}(s') \right)}_{q_{\pi_k}(s,a)}, \quad s \in \mathcal{S}.$$

  Here, $q_{\pi_k}(s,a)$ is the action value under policy π_k. Let $a_k^*(s) = \arg\max_a q_{\pi_k}(a,s)$. Then, the greedy optimal policy is

$$\pi_{k+1}(a|s) = \begin{cases} 1 & a = a_k^*(s), \\ 0 & a \neq a_k^*(s). \end{cases}$$

The implementation details are summarized in the pseudocode.

4.2.3 Illustrative examples

A simple example

Consider the example in Figure 4.2. In this example, each state is associated with three possible actions: a_ℓ, a_0, a_r, which represent to move leftwards, stay unchanged, and move rightwards, respectively. The reward setting is $r_{\text{boundary}} = -1$ and $r_{\text{target}} = 1$. The discount rate is $\gamma = 0.9$.

We next show the implementation details of the policy iteration algorithm step by step. When $k = 0$, we start with the initial policy shown in Figure 4.2(a). This policy satisfies $\pi_0(a_\ell|s_1) = 1$ and $\pi_0(a_\ell|s_2) = 1$. This policy is not good because it does not move toward the target area. We next follow the policy iteration algorithm to see if an optimal policy can be obtained.

Pseudocode: Policy iteration algorithm

```
Initialization: The probability model $p(r|s,a)$ and $p(s'|s,a)$ for all $(s,a)$ are known. Initial guess $\pi_0$.
Aim: Search for the optimal state value and an optimal policy.
While the policy has not converged, for the $k$th iteration, do
    Policy evaluation:
    Initialization: an arbitrary initial guess $v_{\pi_k}^{(0)}$
    While $v_{\pi_k}^{(j)}$ has not converged, for the $j$th iteration, do
        For every state $s \in \mathcal{S}$, do
            $v_{\pi_k}^{(j+1)}(s) = \sum_a \pi_k(a|s)\left[\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) v_{\pi_k}^{(j)}(s')\right]$
    Policy improvement:
    For every state $s \in \mathcal{S}$, do
        For every action $a \in \mathcal{A}(s)$, do
            $q_{\pi_k}(s,a) = \sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) v_{\pi_k}(s')$
        $a_k^*(s) = \arg\max_a q_{\pi_k}(s,a)$
        $\pi_{k+1}(a|s) = 1$ if $a = a_k^*$, and $\pi_{k+1}(a|s) = 0$ otherwise
```

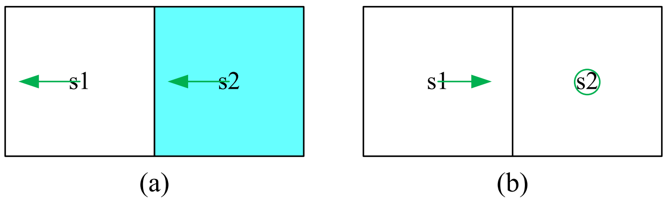


Figure 4.2: An example to illustrate the implementation of the policy iteration algorithm.

– First, in the policy evaluation step, we need to solve the Bellman equation

$$v_{\pi_0}(s_1) = -1 + \gamma v_{\pi_0}(s_1),$$
$$v_{\pi_0}(s_2) = 0 + \gamma v_{\pi_0}(s_1).$$

Since the equation is simple, it can be solved manually that

$$v_{\pi_0}(s_1) = -10, \quad v_{\pi_0}(s_2) = -9.$$

In practice, it is usually solved by the iterative algorithm in (4.2). For example, select the initial of the state values as $v_{\pi_0}^{(0)}(s_1) = v_{\pi_0}^{(0)}(s_2) = 0$. It follows from (4.2) that

$$\begin{cases} v_{\pi_0}^{(1)}(s_1) = -1 + \gamma v_{\pi_0}^{(0)}(s_1) = -1, \\ v_{\pi_0}^{(1)}(s_2) = 0 + \gamma v_{\pi_0}^{(0)}(s_1) = 0, \end{cases}$$
$$\begin{cases} v_{\pi_0}^{(2)}(s_1) = -1 + \gamma v_{\pi_0}^{(1)}(s_1) = -1.9, \\ v_{\pi_0}^{(2)}(s_2) = 0 + \gamma v_{\pi_0}^{(1)}(s_1) = -0.9, \end{cases}$$
$$\begin{cases} v_{\pi_0}^{(3)}(s_1) = -1 + \gamma v_{\pi_0}^{(2)}(s_1) = -2.71, \\ v_{\pi_0}^{(3)}(s_2) = 0 + \gamma v_{\pi_0}^{(2)}(s_1) = -1.71, \end{cases}$$
$$\vdots$$

With more iterations, we can see the trend that $v_{\pi_0}^{(j)}(s_1) \to v_{\pi_0}(s_1) = -10$ and $v_{\pi_0}^{(j)}(s_2) \to v_{\pi_0}(s_2) = -9$ as j increases.

– Second, in the policy improvement step, the key is to calculate $q_{\pi_0}(s,a)$ for each state-action pair. The following q-table can be used to demonstrate such a process:

$q_{\pi_k}(s,a)$	a_ℓ	a_0	a_r
s_1	$-1 + \gamma v_{\pi_k}(s_1)$	$0 + \gamma v_{\pi_k}(s_1)$	$1 + \gamma v_{\pi_k}(s_2)$
s_2	$0 + \gamma v_{\pi_k}(s_1)$	$1 + \gamma v_{\pi_k}(s_2)$	$-1 + \gamma v_{\pi_k}(s_2)$

Table 4.4: The expression of $q_{\pi_k}(s,a)$ for the example in Figure 4.2.

Substituting $v_{\pi_0}(s_1) = -10, v_{\pi_0}(s_2) = -9$ as obtained in the policy evaluation step into Table 4.4 gives By seeking the greatest value of q_{π_0}, the improved policy is:

$q_{\pi_0}(s,a)$	a_ℓ	a_0	a_r
s_1	-10	-9	-7.1
s_2	-9	-7.1	-9.1

Table 4.5: The value of $q_{\pi_k}(s,a)$ when $k=0$.

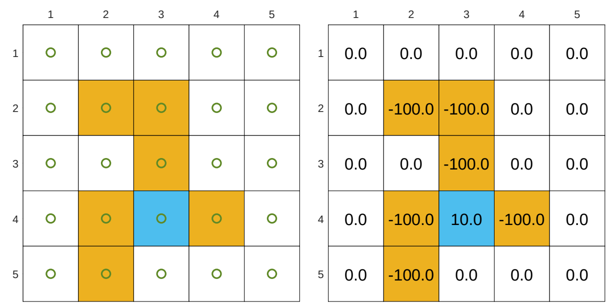
$$\pi_1(a_r|s_1) = 1, \quad \pi_1(a_0|s_2) = 1.$$

This policy is illustrated in Figure 4.2(b). It is intuitively clear that this policy is optimal.

The above process shows that one single iteration can successfully find the optimal policy for this simple example. Of course, more iterations are required for more complex examples.

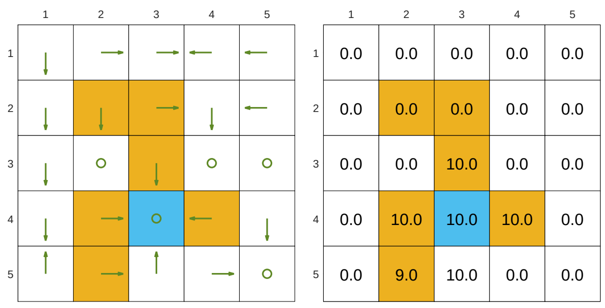
A more complicated example

We next examine the properties of the policy iteration algorithm by considering more complicated examples as shown in Figure 4.3. The reward setting is $r_{\text{boundary}} = -1$,



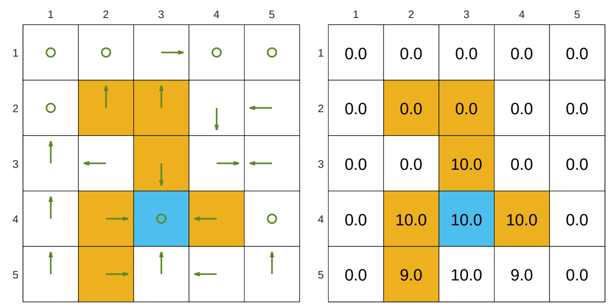
	1	2	3	4	5
1	0.0	0.0	0.0	0.0	0.0
2	0.0	-100.0	-100.0	0.0	0.0
3	0.0	0.0	-100.0	0.0	0.0
4	0.0	-100.0	10.0	-100.0	0.0
5	0.0	-100.0	0.0	0.0	0.0

(a) π_0 and v_{π_0}



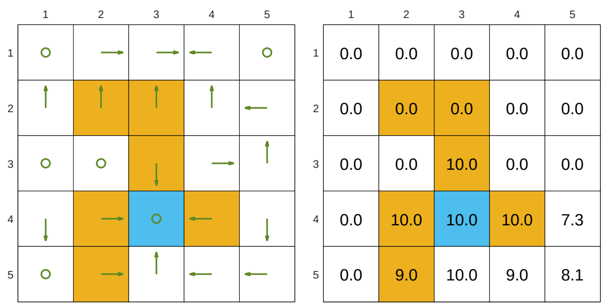
	1	2	3	4	5
1	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	10.0	0.0	0.0
4	0.0	10.0	10.0	10.0	0.0
5	0.0	9.0	10.0	0.0	0.0

(b) π_1 and v_{π_1}



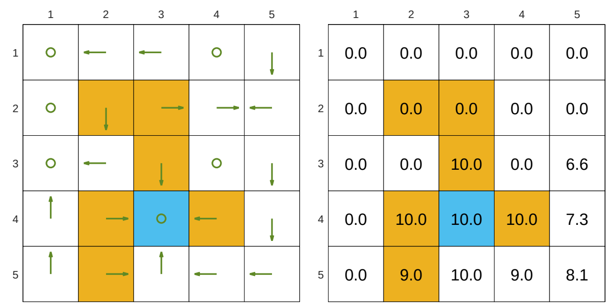
	1	2	3	4	5
1	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	10.0	0.0	0.0
4	0.0	10.0	10.0	10.0	0.0
5	0.0	9.0	10.0	9.0	0.0

(c) π_2 and v_{π_2}



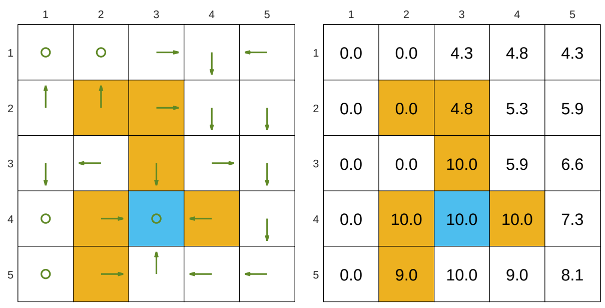
	1	2	3	4	5
1	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	10.0	0.0	0.0
4	0.0	10.0	10.0	10.0	7.3
5	0.0	9.0	10.0	9.0	8.1

(d) π_3 and v_{π_3}



	1	2	3	4	5
1	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	10.0	0.0	6.6
4	0.0	10.0	10.0	10.0	7.3
5	0.0	9.0	10.0	9.0	8.1

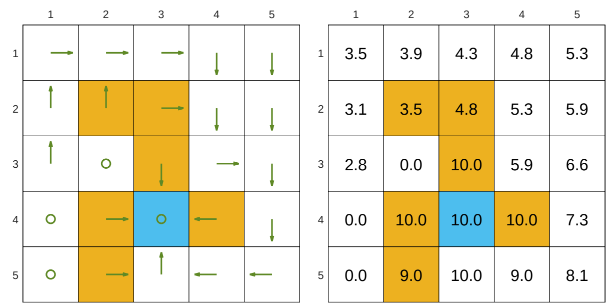
(e) π_4 and v_{π_4}



	1	2	3	4	5
1	0.0	0.0	4.3	4.8	4.3
2	0.0	0.0	4.8	5.3	5.9
3	0.0	0.0	10.0	5.9	6.6
4	0.0	10.0	10.0	10.0	7.3
5	0.0	9.0	10.0	9.0	8.1

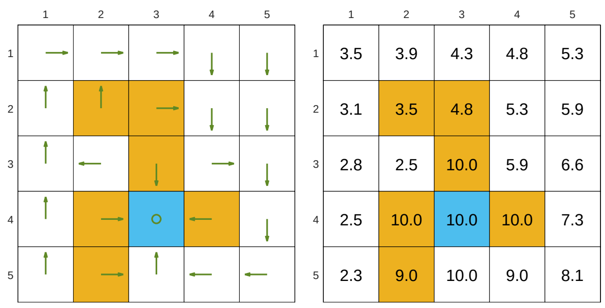
(f) π_5 and v_{π_5}

$\vdots$ $\vdots$



	1	2	3	4	5
1	3.5	3.9	4.3	4.8	5.3
2	3.1	3.5	4.8	5.3	5.9
3	2.8	0.0	10.0	5.9	6.6
4	0.0	10.0	10.0	10.0	7.3
5	0.0	9.0	10.0	9.0	8.1

(g) π_9 and v_{π_9}



	1	2	3	4	5
1	3.5	3.9	4.3	4.8	5.3
2	3.1	3.5	4.8	5.3	5.9
3	2.8	2.5	10.0	5.9	6.6
4	2.5	10.0	10.0	10.0	7.3
5	2.3	9.0	10.0	9.0	8.1

(h) π_{10} and $v_{\pi_{10}}$

Figure 4.3: The evolution process of the policies generated by the policy iteration algorithm.

$r_{\text{forbidden}} = -10$, $r_{\text{target}} = 1$. The discount rate is $\gamma = 0.9$. The policy iteration algorithm can converge to the optimal policy (Figure 4.3(h)) starting from a random initial policy (Figure 4.3(a)).

Two interesting phenomena are observed in the process of policy searching.

– First, if we observe how the policy evolves, an interesting pattern is that the states that are close to the target area find the optimal policies earlier than those states far away. Only if the close states can find trajectories to the target and then update their state values, those farther states can find trajectories passing through the close ones to reach the target.

– Second, if we observe the spatial pattern of the state values of the final policy, the state values exhibit an interesting pattern: the states that are located closer to the target have greater state values. The reason is that the agent starting from a farther state has to travel for many steps to get a positive reward when reaching the target. Such a reward would be severely discounted and hence small. The "closeness" is of course evaluated based on the final policy. That is, if a state has to travel many steps to reach the target, then it is not close to the target, even though its Euclidean distance to the target is short.

4.3 Truncated policy iteration

We next introduce a more general algorithm called *truncated policy iteration*. This algorithm is more general in the sense that value iteration and policy iteration can be viewed as two extremes of it.

4.3.1 Compare value iteration and policy iteration

First of all, we list the steps of policy iteration and value iteration, respectively.

– Policy iteration: starting from any policy π_0,

  ◦ Step 1: policy evaluation (PE). Given π_k, solve v_{π_k} from

$$v_{\pi_k} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}.$$

  ◦ Step 2: policy improvement (PI). Given v_{π_k}, solve π_{k+1} from

$$\pi_{k+1} = \arg\max_{\pi}(r_\pi + \gamma P_\pi v_{\pi_k}).$$

– Value iteration: starting from any v_0,

	Policy iteration algorithm	Value iteration algorithm	Comments
1) Policy:	π_0	N/A	
2) Value:	$v_{\pi_0} = r_{\pi_0} + \gamma P_{\pi_0} v_{\pi_0}$	$v_0 \doteq v_{\pi_0}$	
3) Policy:	$\pi_1 = \arg\max_\pi(r_\pi + \gamma P_\pi v_{\pi_0})$	$\pi_1 = \arg\max_\pi(r_\pi + \gamma P_\pi v_0)$	The two policies are the same
4) Value:	$v_{\pi_1} = r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}$	$v_1 = r_{\pi_1} + \gamma P_{\pi_1} v_0$	$v_{\pi_1} \geq v_1$ since $v_{\pi_1} \geq v_{\pi_0}$
5) Policy:	$\pi_2 = \arg\max_\pi(r_\pi + \gamma P_\pi v_{\pi_1})$	$\pi_2' = \arg\max_\pi(r_\pi + \gamma P_\pi v_1)$	π_2 is better than π_2' ($v_{\pi_2} \geq v_{\pi_2'}$)
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 4.6: Compare the implementation steps of policy iteration and value iteration.

- Step 1: policy update (PU). Given v_k, solve π_{k+1} from

$$\pi_{k+1} = \arg\max_\pi(r_\pi + \gamma P_\pi v_k).$$

- Step 2: value update (VU). Given π_{k+1}, solve v_{k+1} from

$$v_{k+1} = r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_k.$$

We can compare the two algorithms more closely by writing out their iteration processes:

$$\begin{aligned}
&\text{Policy iteration: } \pi_0 \xrightarrow{PE} v_{\pi_0} \xrightarrow{PI} \pi_1 \xrightarrow{PE} v_{\pi_1} \xrightarrow{PI} \pi_2 \xrightarrow{PE} v_{\pi_2} \xrightarrow{PI} \dots \\
&\text{Value iteration: } \qquad v_0 \xrightarrow{PU} \pi_1' \xrightarrow{VU} v_1 \xrightarrow{PU} \pi_2' \xrightarrow{VU} v_2 \xrightarrow{PU} \dots
\end{aligned}$$

The iteration processes of the two algorithms are very similar.

To illustrate their difference, we let the two algorithms start from the *same initial condition*: $v_0 = v_{\pi_0}$. The evolvement processes of the two algorithms are shown in Table 4.6. In the first three steps, the two algorithms generate the same results since $v_0 = v_{\pi_0}$. They become different in the fourth step. In the fourth step, policy iteration solves $v_{\pi_1} = r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}$ and requires an infinite number of calculations, whereas value iteration executes $v_1 = r_{\pi_1} + \gamma P_{\pi_1} v_0$, which is a one-step calculation. If we explicitly write out the iterative process solving $v_{\pi_1} = r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}$ in the fourth step, everything will be crystal:

$$\begin{aligned}
& v_{\pi_1}^{(0)} = v_0 \\
\text{value iteration} \leftarrow v_1 \longleftarrow & v_{\pi_1}^{(1)} = r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}^{(0)} \\
& v_{\pi_1}^{(2)} = r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}^{(1)} \\
& \vdots \\
\text{truncated policy iteration} \leftarrow \bar{v}_1 \longleftarrow & v_{\pi_1}^{(j)} = r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}^{(j-1)} \\
& \vdots \\
\text{policy iteration} \leftarrow v_{\pi_1} \longleftarrow & v_{\pi_1}^{(\infty)} = r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}^{(\infty)}
\end{aligned}$$

where the initial value $v_{\pi_1}^{(0)}$ is set to be $v_{\pi_1}^{(0)} = v_0 = v_{\pi_0}$. It can be seen clearly from the above iterative process that

– if the iteration is run *only once*, then $v_{\pi_1}^{(1)}$ is actually v_1 as calculated in the value iteration algorithm;

– if the iteration is run *an infinite number of times*, then $v_{\pi_1}^{(\infty)}$ is actually v_{π_1} as calculated in the policy iteration algorithm.

– if the iteration is run *a finite number of times* denoted as j_{truncate}, then such an algorithm is called *truncated policy iteration.* It is called this name simply because the rest iterations from j_{truncate} to ∞ are truncated.

As a result, value iteration and policy iteration can be viewed as two extreme cases of truncated policy iteration, which stop at $j_{\text{truncate}} = 1$ and $j_{\text{truncate}} = \infty$, respectively.

Finally, the above comparison is based on the condition that $v_{\pi_1}^{(0)} = v_0 = v_{\pi_0}$. Without this condition, the two algorithms cannot be compared because they start from different initial conditions.

4.3.2 Truncated policy iteration algorithm

In a nutshell, truncated policy iteration is the same as policy iteration except that it merely runs a finite number of iterations in the policy evaluation step. The implementation details are summarized in the pseudocode. It must be noted that v_k, $v_k^{(j)}$, and q_k in the algorithm are no longer state values or action values. That is simply because, if we only run a finite number of iterations in the policy evaluation step, we can only get an approximation of the true state values.

If v_k is not v_{π_k} due to truncation, will the truncated policy iteration algorithm still be able to find optimal policies? The answer is yes. Intuitively, truncated policy iteration is in between value iteration and policy iteration. On the one hand, it converges faster than the value iteration algorithm, because it computes more than one iteration in the

Pseudocode: Truncated policy iteration algorithm

```
Initialization: The probability model $p(r|s,a)$ and $p(s'|s,a)$ for all $(s,a)$ are known.
Initial guess $\pi_0$.
Aim: Search for the optimal state value and an optimal policy.

While the policy has not converged, for the $k$th iteration, do
    Policy evaluation:
    Initialization: select the initial guess as $v_k^{(0)} = v_{k-1}$. The maximum iteration is set
    to be $j_{\text{truncate}}$.
    While $j < j_{\text{truncate}}$, do
        For every state $s \in \mathcal{S}$, do
            $v_k^{(j+1)}(s) = \sum_a \pi_k(a|s) \left[\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) v_k^{(j)}(s')\right]$
    Set $v_k = v_k^{(j_{\text{truncate}})}$
    Policy improvement:
    For every state $s \in \mathcal{S}$, do
        For every action $a \in \mathcal{A}(s)$, do
            $q_k(s,a) = \sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) v_k(s')$
        $a_k^*(s) = \arg\max_a q_k(s,a)$
        $\pi_{k+1}(a|s) = 1$ if $a = a_k^*$, and $\pi_{k+1}(a|s) = 0$ otherwise
```

policy evaluation step. On the other hand, it converges slower than the policy iteration algorithm, because it only computes a finite number of iterations. This intuition is illustrated by Figure 4.4. Such intuition is also supported by the following mathematical analysis.

Proposition 4.1 (Value improvement). *Consider the iterative algorithm in the policy evaluation step:*

$$v_{\pi_k}^{(j+1)} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}^{(j)}, \quad j = 0, 1, 2, ...$$

If the initial guess is selected as $v_{\pi_k}^{(0)} = v_{\pi_{k-1}}$, it holds that

$$v_{\pi_k}^{(j+1)} \geq v_{\pi_k}^{(j)}$$

for every $j = 0, 1, 2, \ldots$.

Proof of Proposition 4.1

First, since $v_{\pi_k}^{(j)} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}^{(j-1)}$ and $v_{\pi_k}^{(j+1)} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}^{(j)}$, we have

$$v_{\pi_k}^{(j+1)} - v_{\pi_k}^{(j)} = \gamma P_{\pi_k}(v_{\pi_k}^{(j)} - v_{\pi_k}^{(j-1)}) = \cdots = \gamma^j P_{\pi_k}^j (v_{\pi_k}^{(1)} - v_{\pi_k}^{(0)}). \tag{4.3}$$

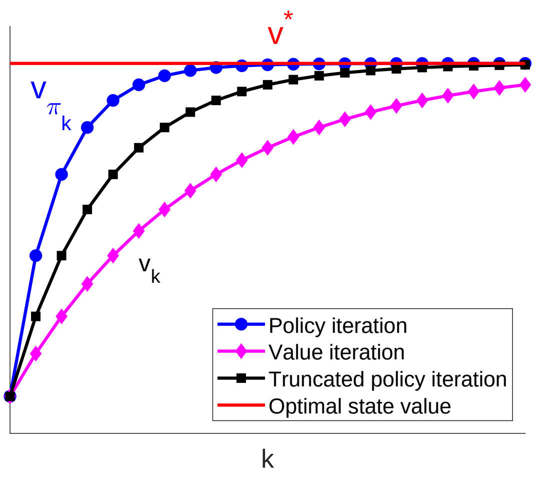


Figure 4.4: Illustration of the relationship among value iteration, policy iteration, and truncated policy iteration.

Second, since $v_{\pi_k}^{(0)} = v_{\pi_{k-1}}$, we have

$$v_{\pi_k}^{(1)} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}^{(0)} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_{k-1}} \geq r_{\pi_{k-1}} + \gamma P_{\pi_{k-1}} v_{\pi_{k-1}} = v_{\pi_{k-1}} = v_{\pi_k}^{(0)}.$$

where the inequality is due to $\pi_k = \arg\max_\pi (r_\pi + \gamma P_\pi v_{\pi_{k-1}})$. Therefore, $v_{\pi_k}^{(1)} \geq v_{\pi_k}^{(0)}$, substituting which into (4.3) gives $v_{\pi_k}^{(j+1)} \geq v_{\pi_k}^{(j)}$.

It is notable that Proposition 4.1 is valid based on the assumption that $v_{\pi_k}^{(0)} = v_{\pi_{k-1}}$. However, $v_{\pi_{k-1}}$ is unavailable in practice whereas only v_{k-1} is available. Nevertheless, Proposition 4.1 still sheds light on the convergence of truncated policy iteration. A more in-depth treatment on this topic can be found in [2, Section 6.5]

Up to now, the advantages of truncated policy iteration are clear. Compared to policy iteration which requires an infinite number of iterations in the policy evaluation step, the truncated one merely requires a finite number of iterations and is more computationally efficient. Compared to value iteration, the truncated policy iteration algorithm can speed up the convergence rate by running a few more iterations in the policy evaluation step, which will be verified by the example studied in the following.

4.3.3 Illustrative examples

We next use examples to demonstrate the impact of the selection of j_{truncate} on the convergence rate of the algorithm. The setup is the same as the example in Figure 4.3. Define $\|v_k - v^*\|$ as the state value error at time k. Starting from the same initial guess of the state value, the evolution of the state value errors are shown in Figure 4.5. The stop criterion is $\|v_k - v^*\| < 0.01$.

Here, j_{truncate} is selected as $1, 3, 6, 100$. When $j_{\text{truncate}} = 1$, the truncated policy iteration algorithm is exactly the value iteration algorithm; when $j_{\text{truncate}} = 100$, it is

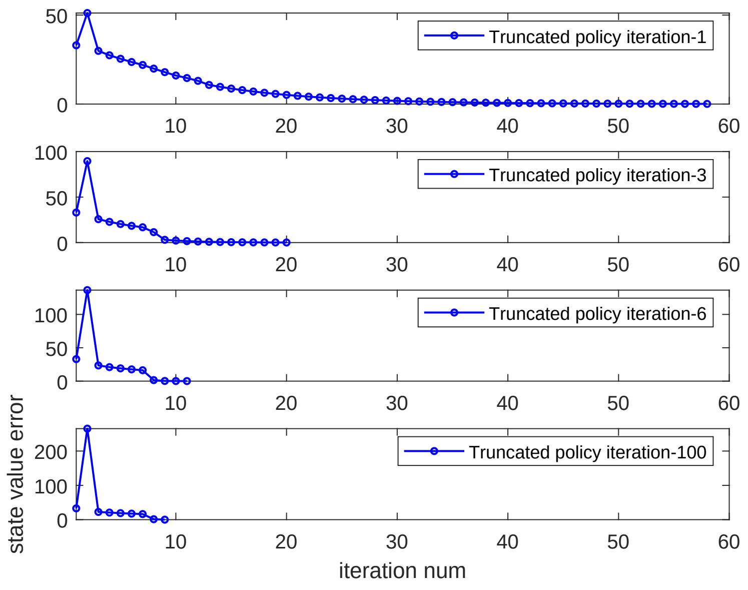


Figure 4.5: Convergence processes of truncated policy iteration. Here, j_{truncate} is selected as $1, 3, 6, 100$. That means the policy evaluation step merely runs j_{truncate} iterations.

approximately the policy iteration algorithm. It is clear that the greater the value of j_{truncate} is, the faster the value estimate converges, which is consistent with the previous theoretical analysis. However, the benefit of increasing j_{truncate} drops quickly when j_{truncate} is large. For example, the overall convergence rate is not enhanced significantly when j_{truncate} increases from 6 to 100. Therefore, it usually can generate satisfactory performance in practice if we simply run a few iterations in the policy evaluation step.

4.4 Summary

This chapter introduced three model-based RL algorithms.

– Value iteration: Value iteration is nothing but the iterative algorithm suggested by the contraction mapping theorem for solving the Bellman optimality equation. When put in the context of RL, it can be decomposed into two steps, value update and policy update. It is notable that the intermediate values generated by value iteration are not state values of any policies in general.

– Policy iteration: Policy iteration is a slightly more complicated algorithm than value iteration. It contains two steps: policy evaluation and policy improvement. The policy evaluation step requires solving a Bellman equation by using an iterative algorithm. This iterative algorithm further requires an infinite number of iterations to solve the true state value. As a result, the values calculated by the policy evaluation step are true state values.

– Truncated policy iteration: Truncated policy iteration is in between value iteration and policy iteration. That is because it runs a finite number of iterations in the policy evaluation step rather than merely one iteration as the value iteration algorithm or infinite iterations as the policy iteration algorithm. As a result, value iteration and policy iteration can be viewed as two extremes of truncated policy iteration. The intermediate values generated by the truncated policy iteration algorithm are not state values.

All the three algorithms share the same common point. That is, every iteration has two steps. One step is to update the value estimate and the other is to update the policy. Such kind of value-policy interaction in each iteration, which is known as *generalized policy iteration* [3], is a basic idea for many RL algorithms.

Finally, the algorithms introduced in this chapter are model-based. They are also the simplest and first RL algorithms ever introduced in this book. Starting from the next chapter, we will study model-free RL algorithms. Readers will see that model-free RL algorithms such as Monte Carlo based methods are straightforward to understand if the model-based ones have been well understood.

4.5 Q&A

– Q: What steps are there in the value iteration algorithm?

  A: In each iteration of the value iteration algorithm, there are two steps: policy update and value update.

– Q: Why is it guaranteed that value iteration can find optimal policies?

  A: That is because value iteration is exactly the algorithm suggested by the contraction mapping theorem when we solve the Bellman optimality equation in the last chapter. The convergence of this algorithm is guaranteed by the contraction mapping theorem.

– Q: Are the intermediate value generated by the value iteration algorithm state values?

  A: No, because these values are not guaranteed to satisfy the Bellman equation of any policy.

– Q: What steps are there in the policy iteration algorithm?

  A: In each iteration of the policy iteration algorithm, there are two steps: policy evaluation and policy improvement. In the policy evaluation step, the algorithm aims to solve the Bellman equation to obtain the state value of the current policy. In the policy improvement step, the algorithm aims to update the policy so that the new policy has greater state values.

– Q: Is there another iteration algorithm embedded in the policy iteration algorithm?

  A: Yes. In the policy evaluation step of the policy iteration algorithm, an iterative algorithm is required to solve the Bellman equation.

- Q: Are the intermediate values generated by the policy iteration algorithm state values?
  A: Yes. Since these values are obtained by solving the Bellman equation, they are state values of the intermediate policies generated during the iteration process.
- Q: Is it guaranteed that the policy iteration algorithm can find optimal policies?

  A: Yes. We have given a rigorous proof of its convergence in this chapter.
- Q: What is the relationship between truncated policy iteration and policy iteration?

  A: As its name suggested, the truncated policy iteration algorithm only runs a finite number of iterations in the policy evaluation step, whereas the policy iteration algorithm requires running an infinite number of iterations in theory.
- Q: What is the relationship between truncated policy iteration and value iteration?

  A: Value iteration can be viewed as an extreme of truncated policy iteration, where a single iteration is run in the policy evaluation step.
- Q: Are the intermediate values generated by the truncated policy iteration algorithm state values?

  A: No. That is because we need to run an infinite number of iterations in the policy evaluation step (in theory) in order to get true state values. Therefore, with a finite number of iterations, we can only get approximated values instead of true state values.
- Q: How many iterations should we run in the policy evaluation step in the truncated policy iteration algorithm?

  A: The general guideline is to run a few but not too many. As demonstrated in Figure 4.5, a few iterations in the policy evaluation step can help speed up the overall convergence rate, but the benefit of running more iterations is not significant.
- Q: While truncated policy iteration cannot accurately evaluate a policy in every iteration, is it still convergent?

  A: Yes. We have discussed the intuition and mathematics behind this conclusion in this chapter. Since value iteration and policy iteration are two extremes of the truncated one and both the two extremes are convergent, it is intuitively clear that the truncated one is also convergent.
- Q: What is generalized policy iteration?

  A: Generalized policy iteration is not a specific algorithm. Instead, it refers to the general idea or framework of switching between policy-evaluation and policy-improvement processes for the purpose of optimal policy searching. It is rooted in the policy iteration algorithm. Different from the policy iteration algorithm, generalized policy iteration does not specify the details of the policy-evaluation and policy-improvement. Many reinforcement learning algorithms falls into the scope of generalized policy iteration.

Chapter 5

Monte Carlo Learning

This chapter introduces the reinforcement learning (RL) algorithms based on Monte Carlo (MC) estimation. They are the first class of model-free RL algorithms ever introduced in this book. When system models are not available, MC methods learn optimal policies from the interaction experience between the agent and the environment.

The basic idea of MC learning is very simple. In particular, the simplest MC learning algorithm can be easily obtained by replacing the model-based policy evaluation step in the policy iteration algorithm introduced in the last chapter with a model-free one. This simplest algorithm can be further extended in various ways to obtain more complex MC learning algorithms that can use samples more efficiently.

5.1 Motivating example: Monte Carlo estimation

Monte Carlo estimation refers to a broad class of techniques that use stochastic samples to solve approximation problems. The basic idea of MC estimation is demonstrated by the following *mean estimation* problem.

Consider a random variable X that can take values in a finite set of real numbers denoted as $\mathcal{X}$. Suppose our task is to calculate the mean or expectation of X: $\mathbb{E}[X]$. There are two approaches to calculating $\mathbb{E}[X]$.

– The first approach is *model-based*. In particular, the model refers to the probability $p(X = x)$ for X taking any $x \in \mathcal{X}$. If the model is known, then the mean can be calculated based on the definition of expectation:

$$\mathbb{E}[X] = \sum_{x \in \mathcal{X}} p(x)x.$$

For example, suppose we flip a coin repeatedly. The random variable X has two possible values: $X = 1$ (head of the coin) and $X = -1$ (tail of the coin). If we know the probability distribution of X exactly, for example, $p(X = 1) = 0.5$ and

$p(X = -1) = 0.5$, then the mean can be calculated directly as

$$\mathbb{E}[X] = 0.5 \cdot 1 + 0.5 \cdot (-1) = 0.$$

– The second approach is *model-free*. In practice, the probability distribution of X may not be known precisely. In this case, how to estimate the mean? The basic idea is to sample X many times and get a sampling sequence $\{x_1, x_2, \ldots, x_n\}$. Then, the mean can be approximated as

$$\mathbb{E}[X] \approx \bar{x} = \frac{1}{n}\sum_{j=1}^{n} x_j.$$

When n is small, the approximation may not be accurate. However, as n increases, the approximation becomes more and more accurate. When $n \to \infty$, we have $\bar{x} \to \mathbb{E}[X]$ (Figure 5.1). This is guaranteed by the *Law of Large Numbers*: The average of a large number of samples would be close to the expected value and will be closer to the expected value when more samples are used (the proof is given in the following shaded box).

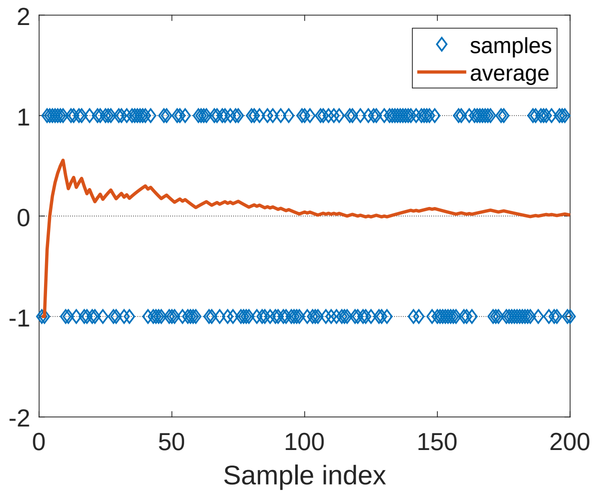


Figure 5.1: An example to demonstrate the Law of Large Numbers. Here, the samples are drawn from $\{+1, -1\}$ following a uniform distribution. The average gradually converges to zero, the true expectation value, as the sample number increases.

The above two approaches clearly demonstrate the fundamental philosophy of model-based and model-free RL: When the system model is available, the expectation can be calculated based on the model. When the model is unavailable, the expectation can be estimated approximately using stochastic samples. Readers may wonder why we suddenly start talking about the mean estimation problem in this chapter. That is simply because state value and action value are both defined as expectations (or means).

It is worth mentioning that the samples used for mean estimation must be *independent and identically distributed* (iid). The reason is obvious. If the sampling values are correlated to each other, then it is impossible to approximate the expected value. An extreme case is that all the sampling values are the same as the first one. In this case, the average of the samples is always equal to the first sample no matter how many samples we use.

Law of Large Numbers

For a random variable X. Suppose $\{x_i\}_{i=1}^n$ are some iid samples. Let $\bar{x} = \frac{1}{n}\sum_{i=1}^n x_i$ be the average of the samples. Then,

$$\begin{aligned}\mathbb{E}[\bar{x}] &= \mathbb{E}[X],\\ \mathrm{var}[\bar{x}] &= \frac{1}{n}\mathrm{var}[X].\end{aligned}$$

The above two equations indicate that $\bar{x}$ is an unbiased estimate of $\mathbb{E}[X]$ and its variance decreases to zero as n increases to infinity.

The proof is given below.

First, $\mathbb{E}[\bar{x}] = \mathbb{E}\big[\sum_{i=1}^n x_i/n\big] = \sum_{i=1}^n \mathbb{E}[x_i]/n = \mathbb{E}[X]$, where the last equability is because the samples are *identically distributed* (that is, $\mathbb{E}[x_i] = \mathbb{E}[X]$).

Second, $\mathrm{var}(\bar{x}) = \mathrm{var}\big[\sum_{i=1}^n x_i/n\big] = \sum_{i=1}^n \mathrm{var}[x_i]/n^2 = (n \cdot \mathrm{var}[X])/n^2 = \mathrm{var}[X]/n$, where the second equality is because the samples are *independent* and the third equability is because the samples are *identically distributed* (that is, $\mathrm{var}[x_i] = \mathrm{var}[X]$).

5.2 The simplest MC learning algorithm

We now introduce the simplest MC learning algorithm. This algorithm is obtained by replacing the *model-based policy evaluation step* in the policy iteration algorithm as introduced in the last chapter with a *model-free MC estimation step.* This algorithm is too simple to use in practice. However, it is important for us to understand the core idea of model-free RL. This algorithm will be extended to obtain more complex and practical MC learning algorithms later in this chapter.

5.2.1 Converting policy iteration to be model-free

Policy iteration is a model-based algorithm introduced in the last chapter. We next show that the model-based part of this algorithm can be replaced by MC estimation so that the algorithm can become model-free.

The policy iteration algorithm contains two steps in each iteration. The first is *policy evaluation*, which is to compute v_{π_k} by solving $v_{\pi_k} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}$. The second step is *policy improvement*, which is to compute the greedy policy $\pi_{k+1} = \arg\max_\pi (r_\pi + \gamma P_\pi v_{\pi_k})$.

The elementwise form of the policy improvement step is

$$\begin{aligned}\pi_{k+1}(s) &= \arg\max_{\pi} \sum_a \pi(a|s) \left[\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) v_{\pi_k}(s') \right] \\ &= \arg\max_{\pi} \sum_a \pi(a|s) q_{\pi_k}(s,a), \quad s \in \mathcal{S}.\end{aligned}$$

It can be seen from the above equation that $q_{\pi_k}(s,a)$ is the core. Once $q_{\pi_k}(s,a)$ is obtained, $\pi_{k+1}(s)$ can be easily obtained.

How to obtain action values? There are two approaches as demonstrated in the coin-flipping example in the last section.

– The first approach is *model-based.* This is the approach adopted by the policy iteration algorithm. In particular, we can first calculate the state value v_{π_k} by solving the Bellman equation. Then, we can calculate the action values by using

$$q_{\pi_k}(s,a) = \sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) v_{\pi_k}(s'). \tag{5.1}$$

This approach require to know the system model $\{p(r|s,a), p(s'|s,a)\}$.

– The second approach is *model-free.* Recall that the definition of action value is

$$\begin{aligned}q_{\pi_k}(s,a) &= \mathbb{E}[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots |S_t = s, A_t = a],\end{aligned}$$

which is the expectation of the return obtained starting from (s,a). Since $q_{\pi_k}(s,a)$ is an expectation, it can be estimated by MC estimation as demonstrated in the last section. Specifically, starting from (s,a), the agent can interact with the environment following policy π_k and then obtain a number of episodes. The return of each episode is a random sample of $R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$. Suppose there are n episodes and denote the return of the ith episode as $g^{(i)}(s,a)$. Then, $q_{\pi_k}(s,a)$ can be approximated as

$$q_{\pi_k}(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a] \approx \frac{1}{n} \sum_{i=1}^{n} g^{(i)}(s,a).$$

This approximation is the MC estimation. We know that, if the number of episodes n is sufficiently large, the approximation will be sufficiently accurate.

5.2.2 MC Basic algorithm

We are now ready to present the first MC learning algorithm.

In particular, given an initial policy π_0, there are two steps in each iteration. Suppose the current iteration is the kth.

Pseudocode: MC Basic algorithm (a model-free variant of policy iteration)

```
Initialization: Initial guess $\pi_0$.
Aim: Search for an optimal policy.

While the value estimate has not converged, for the $k$th iteration, do
    For every state $s \in \mathcal{S}$, do
        For every action $a \in \mathcal{A}(s)$, do
            Collect sufficiently many episodes starting from $(s, a)$ following $\pi_k$
            MC-based policy evaluation step:
            $q_{\pi_k}(s, a) =$ average return of all the episodes starting from $(s, a)$
        Policy improvement step:
        $a_k^*(s) = \arg\max_a q_{\pi_k}(s, a)$
        $\pi_{k+1}(a|s) = 1$ if $a = a_k^*$, and $\pi_{k+1}(a|s) = 0$ otherwise
```

– *Step 1: policy evaluation.* This step is to obtain $q_{\pi_k}(s, a)$ for all (s, a). Specifically, for each action-state pair (s, a), collect sufficiently many episodes. The average of their returns is used to approximate $q_{\pi_k}(s, a)$.

– *Step 2: policy improvement.* This step is to solve $\pi_{k+1}(s) = \arg\max_\pi \sum_a \pi(a|s) q_{\pi_k}(s, a)$ for all $s \in \mathcal{S}$. The greedy optimal policy is $\pi_{k+1}(a_k^*|s) = 1$ where $a_k^* = \arg\max_a q_{\pi_k}(s, a)$.

This algorithm is the simplest MC learning algorithm. Though simple, it is important for understanding more complicated algorithms. In this book, this algorithm is called *MC Basic*. The pseudocode is given in the following box.

MC Basic is a variant of the policy iteration algorithm. The difference is that MC Basic calculates action values directly from experience samples, whereas policy iteration calculates state values first and then action values based on the system model.

Why does MC Basic estimate action values instead of state values? That is because state values cannot be used to improve policies directly. Even if we are given state values, we still need to calculate action values from these state values using (5.1). However, such a calculation requires the system model. Therefore, when models are not available, we should directly estimate action values.

Since policy iteration is convergent, MC Basic is also convergent given as long as the experience samples are sufficient. Finally, though simple, MC Basic is not practical in practice due to its low sample efficiency. It is, however, important for us the grasp the core idea of model-free RL. Later in this chapter, we will extend MC Basic to more complex and practical MC learning algorithms.

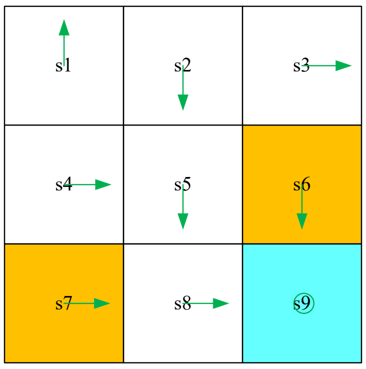


Figure 5.2: An example to illustrate the MC Basic algorithm.

5.2.3 Illustrative examples

A simple example: A step-by-step implementation

We next use an example to illustrate the implementation details of the MC Basic algorithm. The reward setting is $r_{\text{boundary}} = r_{\text{forbidden}} = -1$, and $r_{\text{target}} = 1$. The discount rate is $\gamma = 0.9$. The initial policy π_0 as shown in Figure 5.2 is not optimal. Specifically, the policy for s_1 or s_3 is not optimal.

Although we need to calculate the action values of all states, we merely demonstrate those of s_1 due to space limitation. Starting from s_1, there are five possible actions. For each action, we need to run sufficiently many episodes to well approximate the action value. However, since this example is deterministic in terms of both policy and model, the estimation of each action value merely requires a single episode, because running multiple times would generate exactly the same trajectory.

For iteration $k = 0$, we can obtain the following five episodes starting from s_1:

– Starting from (s_1, a_1), the episode is $s_1 \xrightarrow{a_1} s_1 \xrightarrow{a_1} s_1 \xrightarrow{a_1} \ldots$. Hence, the action value is

$$q_{\pi_0}(s_1, a_1) = -1 + \gamma(-1) + \gamma^2(-1) + \cdots = \frac{-1}{1-\gamma}.$$

– Starting from (s_1, a_2), the episode is $s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_5 \xrightarrow{a_3} \ldots$. Hence, the action value is

$$q_{\pi_0}(s_1, a_2) = 0 + \gamma 0 + \gamma^2 0 + \gamma^3(1) + \gamma^4(1) + \cdots = \frac{\gamma^3}{1-\gamma}.$$

– Starting from (s_1, a_3), the episode is $s_1 \xrightarrow{a_3} s_4 \xrightarrow{a_2} s_5 \xrightarrow{a_3} \ldots$. Hence, the action value is

$$q_{\pi_0}(s_1, a_3) = 0 + \gamma 0 + \gamma^2 0 + \gamma^3(1) + \gamma^4(1) + \cdots = \frac{\gamma^3}{1-\gamma}.$$

– Starting from (s_1, a_4), the episode is $s_1 \xrightarrow{a_4} s_1 \xrightarrow{a_1} s_1 \xrightarrow{a_1} \dots$. Hence, the action value is

$$q_{\pi_0}(s_1, a_4) = -1 + \gamma(-1) + \gamma^2(-1) + \cdots = \frac{-1}{1-\gamma}.$$

– Starting from (s_1, a_5), the episode is $s_1 \xrightarrow{a_5} s_1 \xrightarrow{a_1} s_1 \xrightarrow{a_1} \dots$. Hence, the action value is

$$q_{\pi_0}(s_1, a_5) = 0 + \gamma(-1) + \gamma^2(-1) + \cdots = \frac{-\gamma}{1-\gamma}.$$

By observing the action values, we see that

$$q_{\pi_0}(s_1, a_2) = q_{\pi_0}(s_1, a_3) = \frac{\gamma^3}{1-\gamma} > 0$$

are the maximum. As a result, the policy can be improved as

$$\pi_1(a_2|s_1) = 1 \quad \text{or} \quad \pi_1(a_3|s_1) = 1.$$

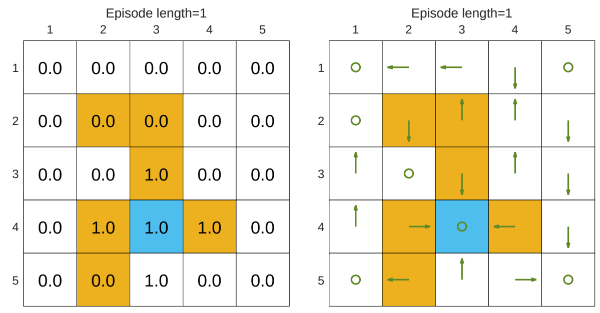
It can be seen that the improved policy, which takes either a_2 or a_3 at s_1, is optimal. Therefore, we can successfully obtain the optimal policy by using merely one iteration for this simple example. In this simple example, the initial policy for all the states except s_1 and s_3 is already optimal. Therefore, the policy can become optimal after merely a single iteration. When the policy is non-optimal for other states, more iterations are required.

A comprehensive example: Episode length and sparse reward

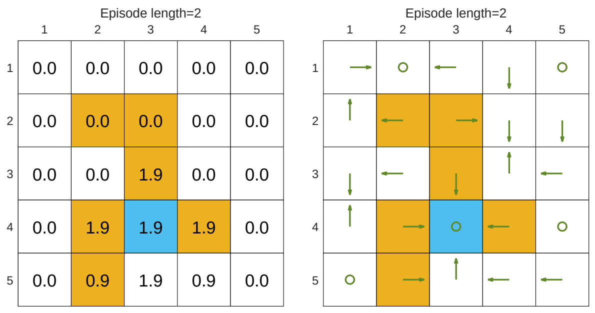
We next present a more comprehensive example to discuss some interesting properties of MC learning. The example is a 5-by-5 grid world. The reward setting is $r_{\text{boundary}} = -1$, $r_{\text{forbidden}} = -10$, and $r_{\text{target}} = 1$. The discount rate is $\gamma = 0.9$.

First of all, we show by example that the length of each episode plays an important role. All the plots in Figure 5.3 show the final results given by the MC Basic algorithm. The difference is that different results are based on different episode lengths. It is notable that the episode length has a great impact on the final policies. When the length of each episode is too short, neither the policy nor the value estimate is optimal. See, for example, Figure 5.3(a)-(d). When the episode length increases, the policy and value estimate gradually approach the optimal ones. See, for example, Figure 5.3(h). In the extreme case where the episode length is 1, only the states that are adjacent to the target have nonzero values. All the others have zero values because each episode is too short to reach the target or get positive rewards.

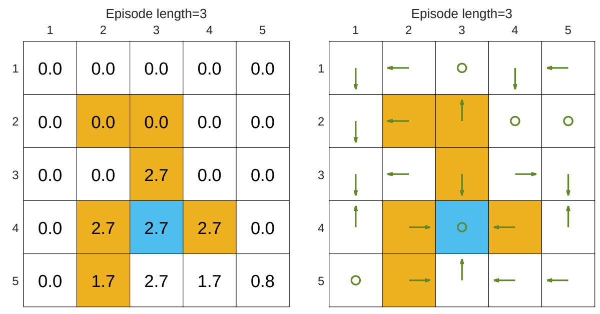
As the episode length increases, an interesting spatial pattern emerges. That is, the states that are closer to the target possess nonzero earlier than those farther away. The reason is as follows. Starting from a state, the agent has to travel at least a certain number of steps to reach the target state and then receive positive rewards. If the length



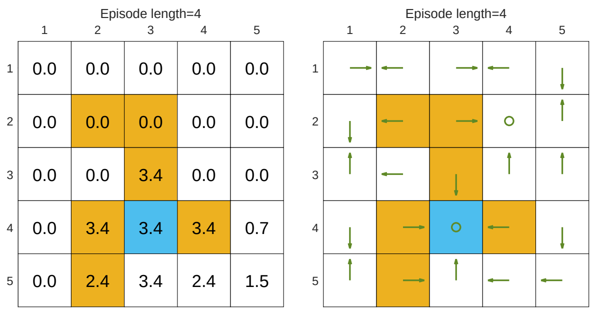
(a) Final value and policy with episode length=1



(b) Final value and policy with episode length=2

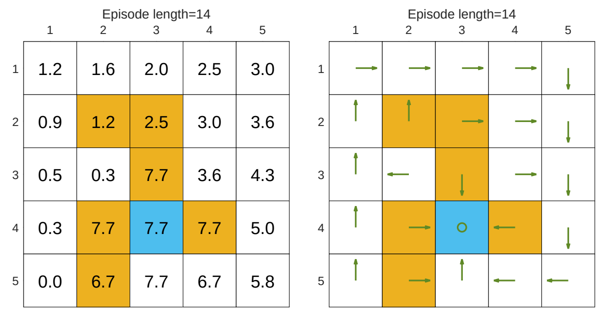


(c) Final value and policy with episode length=3

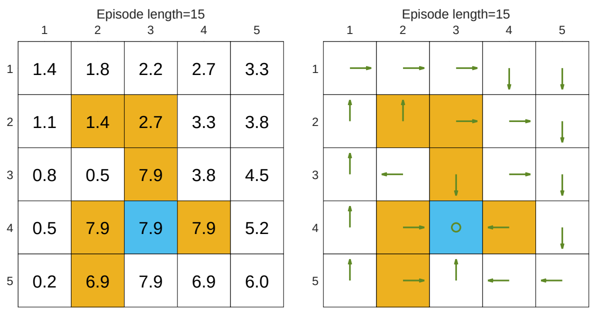


(d) Final value and policy with episode length=4

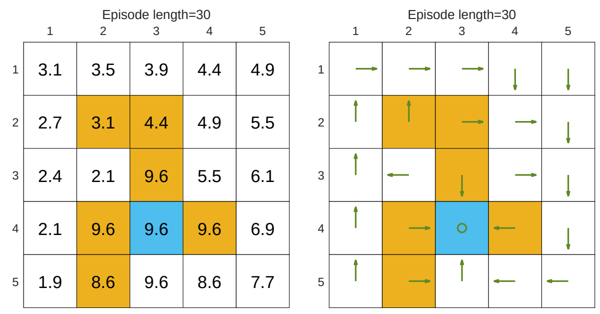
⋮ ⋮ ⋮



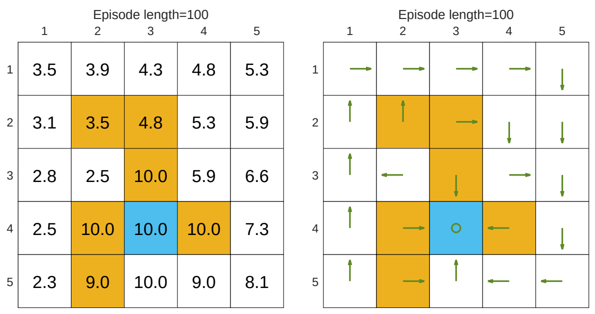
(e) Final value and policy with episode length=14



(f) Final value and policy with episode length=15



(g) Final value and policy with episode length=30



(h) Final value and policy with episode length=100

Figure 5.3: The policies and state values obtained by the MC Basic algorithm given different episode lengths. As can be seen, only if the length of each episode is sufficiently long, the state values can be accurately estimated.

of an episode is less than this desired number of steps, it is certain that the agent cannot reach the target state. Hence, the return would be zero and so is the estimated state value. In this example, the episode length must be no less than 15, which is the minimum number of steps required to reach the target starting from the bottom-left state.

While the above analysis suggests that each episode must be sufficiently long, the episodes are not necessarily infinitely long. As shown in this example, when the length is 30, the algorithm can find an optimal policy although the value estimate is not optimal yet.

The above analysis is related to an important reward design problem: *sparse reward*, which refers to the scenario that, only if the agent reaches the target state, can a positive reward be obtained. In other words, no positive rewards can be obtained unless the target is reached. Sparse reward requires long episodes that must reach the target. This requirement is challenging to meet when the state space is large. As a result, sparse reward downgrades learning efficiency. One simple technique to solve this problem is to design *non-sparse rewards*. For instance, in this grid world example, we can redesign the reward setting so that the agent can obtain a small amount of positive rewards when reaching those states located near the target. In this way, an "attractive field" can be formed around the target so that the agent can find the target easier.

5.3 MC Exploring Starts

We next extend the simple MC Basic algorithm to obtain another MC learning algorithm that is slightly more complicated but more sample-efficient.

5.3.1 Using samples more efficiently

Following a policy π, suppose we have an episode of samples as follows:

$$s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_4} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_5 \xrightarrow{a_1} \dots$$

where the subscripts refer to the state or action indexes rather than time steps. Every time a state-action pair appears in the episode, it is called a *visit* of that state-action pair. There are different strategies to utilize the visits.

The first strategy is to use the *initial visit*. That is the episode is only used to estimate the action value of the starting state-action pair $q_\pi(s_1, a_2)$. The initial-visit strategy is adopted by the MC Basic algorithm but *not sample-efficient*. That is because the episode also visits many other state-action pairs such as (s_2, a_4), (s_2, a_3), and (s_5, a_1). These visits can also be used to estimate the corresponding action values. In particular, we can decompose this episode into multiple sub-episodes:

$$\begin{aligned}
s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_4} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_5 \xrightarrow{a_1} \dots \quad & \text{[original episode]} \\
s_2 \xrightarrow{a_4} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_5 \xrightarrow{a_1} \dots \quad & \text{[episode starting from } (s_2, a_4)] \\
s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_5 \xrightarrow{a_1} \dots \quad & \text{[episode starting from } (s_1, a_2)] \\
s_2 \xrightarrow{a_3} s_5 \xrightarrow{a_1} \dots \quad & \text{[episode starting from } (s_2, a_3)] \\
s_5 \xrightarrow{a_1} \dots \quad & \text{[episode starting from } (s_5, a_1)]
\end{aligned}$$

That is, the trajectory after the visit of a state-action pair can be viewed as a new episode. The new episodes can be used to estimate more action values such as $q_\pi(s_2, a_4)$, $q_\pi(s_2, a_3)$, and $q_\pi(s_5, a_1)$. In this way, the samples in the episode are utilized more efficiently.

A state-action pair may be visited multiple times in an episode. For example, (s_1, a_2) is visited twice in the above episode. If we only count the first-time visit, such kind of strategy is called *first-visit*. If every time a state-action pair is visited and the rest of the episode is used to estimate its action value, such a strategy is called *every-visit*.

In terms of the efficiency of using samples, the every-visit strategy is the best. If an episode is sufficiently long so that it can visit all the state-action pairs many times, then this single episode is sufficient to estimate all the action values by using the every-visit strategy. However, the samples obtained by the every-visit strategy are relevant, because the trajectory starting from the second visit is merely a subset of the trajectory starting from the first. Nevertheless, if the two visits are far away from each other, which means there is a significant non-overlap portion, the relevance would not be strong. Moreover, the relevance can be further suppressed due to the discount rate. Therefore, when there are few episodes and each episode is very long, the every-visit strategy is a good option.

5.3.2 Updating estimate more efficiently

Another aspect of MC learning is when to update the policy. There are two strategies.

The first strategy is, in the policy evaluation step, to collect all the episodes starting from the same state-action pair and then approximate the action value using the average return of these episodes. This strategy is adopted in the MC Basic algorithm. The drawback of this strategy is that the agent has not to wait until all episodes have been collected. The second strategy, which can overcome this drawback, is to use the return of a single episode to approximate the corresponding action value. In this way, we can improve the policy in an episode-by-episode fashion.

While the return of a single episode cannot accurately approximate the corresponding action value, one may wonder whether the second strategy is good or not. In fact, such kind of *inaccurate approximation* falls into the idea of generalized policy iteration introduced in the last chapter.

Pseudocode: MC Exploring Starts (a sample-efficient variant of MC Basic)

```
Initialization: Initial guess $\pi_0$.
Aim: Search for an optimal policy.

For each episode, do
    Episode generation: Randomly select a starting state-action pair $(s_0, a_0)$ and ensure
    that all pairs can be possibly selected. Following the current policy, generate an
    episode of length $T$: $s_0, a_0, r_1, \dots, s_{T-1}, a_{T-1}, r_T$.
    Policy evaluation and policy improvement:
    Initialization: $g \leftarrow 0$
    For each step of the episode, $t = T-1, T-2, \dots, 0$, do
        $g \leftarrow \gamma g + r_{t+1}$
        Use the first-visit strategy:
        If $(s_t, a_t)$ does not appear in $(s_0, a_0, s_1, a_1, \dots, s_{t-1}, a_{t-1})$, then
            $Returns(s_t, a_t) \leftarrow Returns(s_t, a_t) + g$
            $q(s_t, a_t) = \text{average}(Returns(s_t, a_t))$
            $\pi(a|s_t) = 1$ if $a = \arg\max_a q(s_t, a)$
```

5.3.3 Algorithm description

If the MC Basic algorithm is modified so that the sample usage and estimate update are more efficient, the resulting algorithm is called *MC Exploring Starts* in this book.

The pseudocode of MC Exploring Starts is given in the box below. *Exploring starts* is an important assumption in MC learning. It requires generating sufficiently many episodes starting from *every* state-action pair. Both MC Basic and MC Exploring Starts need this assumption. However, exploring starts may be difficult to achieve in practice. We will study how to remove this requirement in the next section.

In addition, when calculating the discounted return starting from each state-action pair, the procedure starts from the ending states and travels back to the starting state. In this way, the calculation is more efficient. The algorithm uses the first-visit strategy. We can also change to the every-visit strategy without examining whether the state appears for the first time.

5.4 MC learning without exploring starts

We next further extend the MC Exploring Starts algorithm introduced in the last chapter by removing the assumption of exploring starts.

Why is exploring starts important? In theory, exploring starts is necessary to find optimal policies. Only if every action of every state is well explored, can we select the optimal actions correctly. Otherwise, if an action is not explored, this action may happen to be the optimal one and hence be missed. In practice, exploring starts is difficult to achieve. For many applications, especially those involving physical interactions with environments, it is difficult to collect episodes starting from every state-action pair. Therefore, there is a gap between theory and practice. Can we remove the requirement of exploring starts? We next show that we can do it by using soft policies.

5.4.1 Soft policies

A policy is *soft* if the policy has a positive probability to take any action at any state. With a soft policy, a single episode that is sufficiently long can visit *every* state-action pair many times. Thus, the single episode can provide sufficient samples and we do not need to have a large number of episodes starting from different state-action pairs. Then, the requirement of exploring starts can be removed.

The most common soft policy is *ϵ-greedy.* An ϵ-greedy policy is a stochastic policy that has a positive probability to take any action but a higher chance to take the *greedy action.* Here, the greedy action refers to the action with the greatest action value. In particular, suppose $\epsilon \in [0,1]$. The ϵ-greedy policy has the form of

$$\pi(a|s) = \begin{cases} 1 - \dfrac{\epsilon}{|\mathcal{A}(s)|}(|\mathcal{A}(s)| - 1), & \text{for the greedy action,} \\[2ex] \dfrac{\epsilon}{|\mathcal{A}(s)|}, & \text{for the other } |\mathcal{A}(s)| - 1 \text{ actions.} \end{cases}$$

Here, $|\mathcal{A}(s)|$ denotes the number of actions associated with s. It is worth noting that the probability to take the greedy action is always greater than that of any other action, because

$$1 - \frac{\epsilon}{|\mathcal{A}(s)|}(|\mathcal{A}(s)| - 1) = 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} \geq \frac{\epsilon}{|\mathcal{A}(s)|}$$

for any $\epsilon \in [0,1]$. When $\epsilon = 0$, ϵ-greedy becomes greedy.

5.4.2 Algorithm description

How to embed ϵ-greedy policies into MC learning? The answer is to change the policy improvement step from greedy to ϵ-greedy. In particular, the policy improvement step in MC Basic or MC Exploring Starts is to solve

$$\pi_{k+1}(s) = \arg\max_{\pi \in \Pi} \sum_a \pi(a|s) q_{\pi_k}(s,a). \tag{5.2}$$

where Π denotes *the set of all possible policies.* The greedy policy is

$$\pi_{k+1}(a|s) = \begin{cases} 1, & a = a_k^*, \\ 0, & a \neq a_k^*, \end{cases}$$

Pseudocode: MC ϵ-Greedy (a variant of MC Exploring Starts)

```
Initialization: Initial guess $\pi_0$ and the value of $\epsilon \in [0,1]$
Aim: Search for an optimal policy.

For each episode, do
    Episode generation: Randomly select a starting state-action pair $(s_0, a_0)$. Following
    the current policy, generate an episode of length $T$: $s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T$.
    Policy evaluation and policy improvement:
    Initialization: $g \leftarrow 0$
    For each step of the episode, $t = T-1, T-2, \ldots, 0$, do
        $g \leftarrow \gamma g + r_{t+1}$
        Use the first-visit strategy:
        If $(s_t, a_t)$ does not appear in $(s_0, a_0, s_1, a_1, \ldots, s_{t-1}, a_{t-1})$, then
            $Returns(s_t, a_t) \leftarrow Returns(s_t, a_t) + g$
            $q(s_t, a_t) = \text{average}(Returns(s_t, a_t))$
            Let $a^* = \arg\max_a q(s_t, a)$ and
```

$$\pi(a|s_t) = \begin{cases} 1 - \frac{|\mathcal{A}(s_t)|-1}{|\mathcal{A}(s_t)|}\epsilon, & a = a^* \\ \frac{1}{|\mathcal{A}(s_t)|}\epsilon, & a \neq a^* \end{cases}$$

where $a_k^* = \arg\max_a q_{\pi_k}(s, a)$.

Now, the policy improvement step is changed to solve

$$\pi_{k+1}(s) = \arg\max_{\pi \in \Pi_\epsilon} \sum_a \pi(a|s) q_{\pi_k}(s, a), \tag{5.3}$$

where Π_ϵ denotes *the set of all ϵ-greedy policies* with a fixed value of ϵ. The key difference of (5.3) from (5.2) is to search the policy in Π_ϵ, which is a subset of Π. Suppose $a_k^* \doteq \arg\max_a q_{\pi_k}(s, a)$. Then, the ϵ-greedy policy is updated by

$$\pi_{k+1}(a|s) = \begin{cases} 1 - \frac{|\mathcal{A}(s)|-1}{|\mathcal{A}(s)|}\epsilon, & a = a_k^*, \\ \frac{1}{|\mathcal{A}(s)|}\epsilon, & a \neq a_k^*. \end{cases}$$

The pseudocode of the algorithm is given in the following box. This algorithm is referred to as *MC ϵ-Greedy* in this book. The only difference between MC ϵ-Greedy and MC Exploring Starts is that the former uses ϵ-greedy policies and the every-visit strategy. Here, the every-visit strategy must be used. That is because, while an episode can visit every state-action pair many times, if the first visit strategy is used instead, it is a waste of samples and cannot approximate the action values accurately.

5.4.3 Algorithm convergence

If we replace greedy policies with ϵ-greedy policies in the policy improvement step, can we still guarantee to find optimal policies? The answer is both yes and no. By yes, it means that, if given sufficient samples, the algorithm can converge to an ϵ-greedy policy that is optimal among the set Π_ϵ. By no, it means that the finally obtained policy is merely optimal in Π_ϵ but not optimal among all possible policies.

The following result shows that the policy improvement step can still generate better and better ϵ-greedy policies.

Lemma 5.1 (Policy Improvement)**.** *If* $\pi_k \in \Pi_\epsilon$ *and* $\pi_{k+1} = \arg\max_{\pi\in\Pi_\epsilon}(r_\pi + \gamma P_\pi v_{\pi_k})$*, then* $v_{\pi_{k+1}} \geq v_{\pi_k}$ *for any* k*.*

Proof of Lemma 5.1

Since π_k, π_{k+1} are both in Π_ϵ and $\pi_{k+1} = \arg\max_{\pi\in\Pi_\epsilon}(r_\pi + \gamma P_\pi v_{\pi_k})$, we have

$$r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_k} \geq r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}.$$

Since

$$\begin{aligned} v_{\pi_{k+1}} &= r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_{k+1}}, \\ v_{\pi_k} &= r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}, \end{aligned}$$

it follows that

$$\begin{aligned} v_{\pi_{k+1}} - v_{\pi_k} &= (r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_{k+1}}) - (r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}) \\ &\geq (r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_{k+1}}) - (r_{\pi_{k+1}} + \gamma P_{\pi_{k+1}} v_{\pi_k}) \\ &= \gamma P_{\pi_{k+1}} (v_{\pi_{k+1}} - v_{\pi_k}). \end{aligned}$$

As a result,

$$v_{\pi_{k+1}} - v_{\pi_k} \geq \gamma^2 P^2_{\pi_{k+1}} (v_{\pi_{k+1}} - v_{\pi_k}) \geq \cdots \geq \lim_{n\to\infty} \gamma^n P^n_{\pi_{k+1}} (v_{\pi_{k+1}} - v_{\pi_k}) = 0,$$

where the last equality is due to $\lim_{n\to\infty}\gamma^n = 0$ and $P^n_{\pi_{k+1}}$ is a stochastic matrix.

Although Lemma 5.1 shows that the ϵ-policy is continuously improved, it still requires a proof showing that it can reach an optimal ϵ-policy. We need to first show that $f(v) = \max_{\pi\in\Pi_\epsilon}(r_\pi + \gamma P_\pi v_{\pi_k})$ is a contraction mapping and then the proof is analogous to Theorem 3.2. Finally, the above analysis is merely for the replacement of greedy policies with ϵ-greedy ones. In addition to that, the algorithm of MC ϵ-Greedy also adopts the every visit strategy that makes the convergence analysis nontrivial.

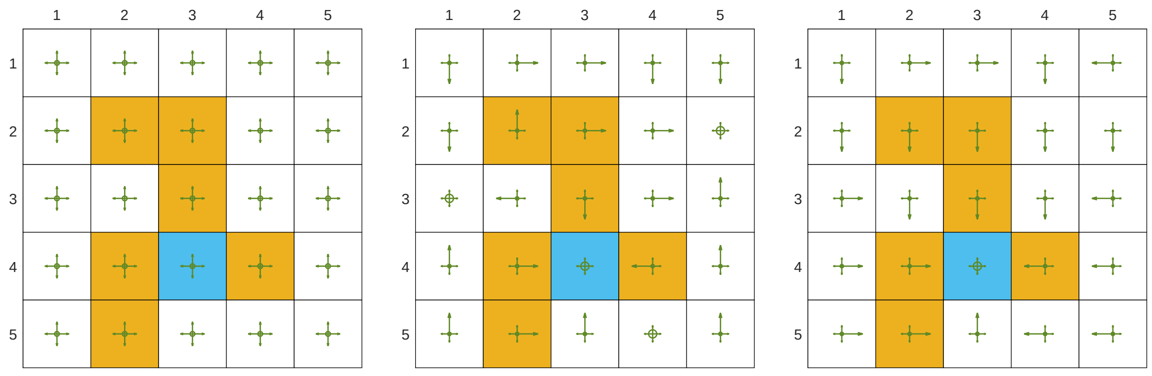

(a) Initial policy (b) After the first iteration (c) After the second iteration

Figure 5.4: The evolution process of the MC ϵ-Greedy algorithm based on single episodes.

5.4.4 Illustrative example

Consider a grid-world example. The aim is to find the optimal policy for every state. In the episode generation step of the MC ϵ-greedy algorithm, a single episode of one million steps is generated. Here, $r_{\text{boundary}} = r_{\text{forbidden}} = -1$, $r_{\text{target}} = 1$, and $\gamma = 0.9$.

The initial policy is a uniform policy that has the same probability of 0.2 to take any action as shown in Figure 5.4. The optimal ϵ-policy with $\epsilon = 0.5$ can be obtained after two iterations. The convergence is fast because the episode is sufficiently long so that all the actions can be visited sufficiently times.

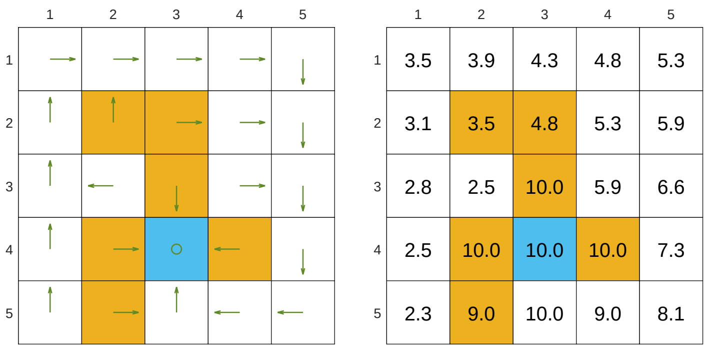
5.5 Exploitation vs exploration

An ϵ-greedy policy aims to well balance *exploitation and exploration.* On the one hand, an ϵ-greedy policy has a higher probability to take the greedy action so that it can exploit the existing value estimates. On the other hand, the ϵ-greedy policy also has a chance to take any other actions so that it can keep exploring.

Exploitation is related to *optimality* because we know the optimal policy should be greedy. The final ϵ-greedy policy given by the MC ϵ-Greedy algorithm is *not optimal* because it is merely optimal in the set Π_ϵ. Therefore, the fundamental idea of ϵ-greedy policies is to enhance exploration by sacrificing optimality.

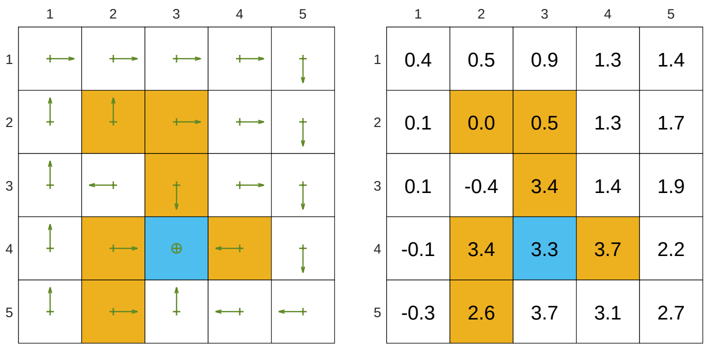
Exploitation and exploration form a fundamental trade-off in RL. That is, if we would like to enhance exploitation and hence optimality, we need to reduce the value of ϵ. However, if we would like to enhance exploration, we need to increase the value of ϵ.

We next discuss this trade-off based on some interesting examples. The RL task here is a 5-by-5 grid world. The reward setting is $r_{\text{boundary}} = -1$, $r_{\text{forbidden}} = -10$, and $r_{\text{target}} = 1$. The discount rate is $\gamma = 0.9$.



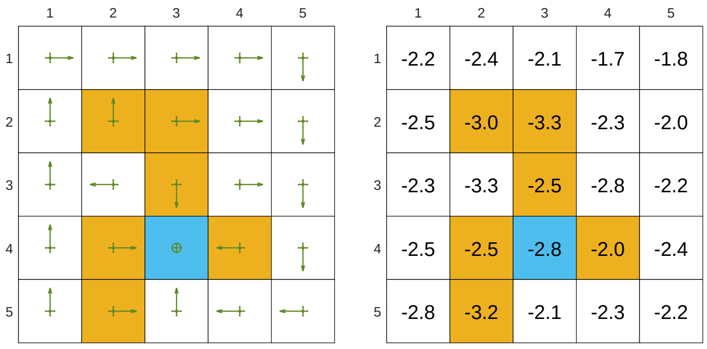
	1	2	3	4	5
1	3.5	3.9	4.3	4.8	5.3
2	3.1	3.5	4.8	5.3	5.9
3	2.8	2.5	10.0	5.9	6.6
4	2.5	10.0	10.0	10.0	7.3
5	2.3	9.0	10.0	9.0	8.1

(a) Given an ϵ-greedy policy with $\epsilon = 0$, the corresponding state values.



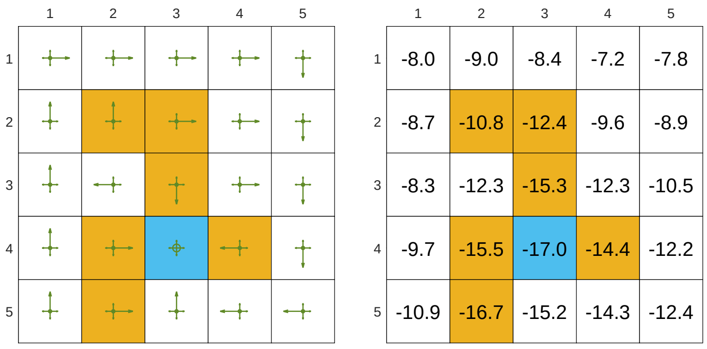
	1	2	3	4	5
1	0.4	0.5	0.9	1.3	1.4
2	0.1	0.0	0.5	1.3	1.7
3	0.1	-0.4	3.4	1.4	1.9
4	-0.1	3.4	3.3	3.7	2.2
5	-0.3	2.6	3.7	3.1	2.7

(b) Given an ϵ-greedy policy with $\epsilon = 0.1$, the corresponding state values.



	1	2	3	4	5
1	-2.2	-2.4	-2.1	-1.7	-1.8
2	-2.5	-3.0	-3.3	-2.3	-2.0
3	-2.3	-3.3	-2.5	-2.8	-2.2
4	-2.5	-2.5	-2.8	-2.0	-2.4
5	-2.8	-3.2	-2.1	-2.3	-2.2

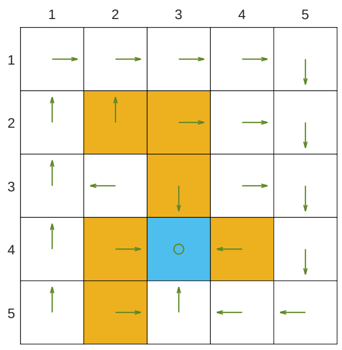
(c) Given an ϵ-greedy policy with $\epsilon = 0.2$, the corresponding state values.



	1	2	3	4	5
1	-8.0	-9.0	-8.4	-7.2	-7.8
2	-8.7	-10.8	-12.4	-9.6	-8.9
3	-8.3	-12.3	-15.3	-12.3	-10.5
4	-9.7	-15.5	-17.0	-14.4	-12.2
5	-10.9	-16.7	-15.2	-14.3	-12.4

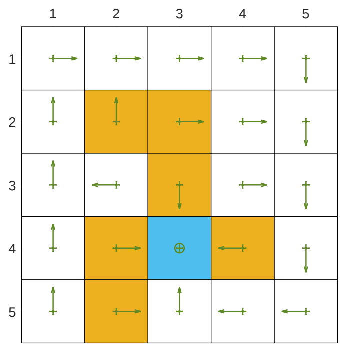
(d) Given an ϵ-greedy policy with $\epsilon = 0.5$, the corresponding state values.

Figure 5.5: The state values of some given ϵ-greedy policies. These ϵ-greedy policies are consistent with each other in the sense that the actions with the greatest probabilities are the same. As can be seen, when the value of ϵ increases, the state values of the ϵ-greedy policies diverge from the optimal ones.



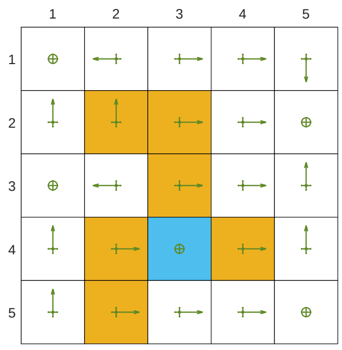
	1	2	3	4	5
1	3.5	3.9	4.3	4.8	5.3
2	3.1	3.5	4.8	5.3	5.9
3	2.8	2.5	10.0	5.9	6.6
4	2.5	10.0	10.0	10.0	7.3
5	2.3	9.0	10.0	9.0	8.1

(a) The optimal ϵ-greedy policy and its state values where $\epsilon = 0$.



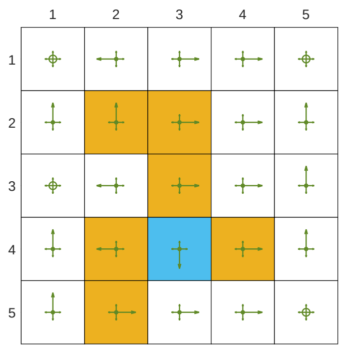
	1	2	3	4	5
1	0.4	0.5	0.9	1.3	1.4
2	0.1	0.0	0.5	1.3	1.7
3	0.1	-0.4	3.4	1.4	1.9
4	-0.1	3.4	3.3	3.7	2.2
5	-0.3	2.6	3.7	3.1	2.7

(b) The optimal ϵ-greedy policy and its state values where $\epsilon = 0.1$.



	1	2	3	4	5
1	-1.1	-1.5	-1.1	-0.6	-0.6
2	-1.5	-2.2	-2.3	-1.0	-0.6
3	-1.2	-2.4	-2.2	-1.5	-0.6
4	-1.6	-2.3	-2.6	-1.4	-1.1
5	-2.0	-3.0	-1.8	-1.4	-1.0

(c) The optimal ϵ-greedy policy and its state values where $\epsilon = 0.2$.



	1	2	3	4	5
1	-4.3	-5.5	-4.5	-2.6	-2.3
2	-5.6	-7.7	-7.7	-4.1	-2.4
3	-5.5	-9.0	-8.0	-5.6	-2.8
4	-6.8	-8.9	-9.4	-5.5	-4.2
5	-7.9	-10.1	-6.7	-5.1	-3.7

(d) The optimal ϵ-greedy policy and its state values where $\epsilon = 0.5$.

Figure 5.6: The optimal ϵ-greedy policies and their corresponding state values given different values of ϵ. Here, these ϵ-greedy policies are optimal among all ϵ-greedy ones (with the same value of ϵ). They are searched by the policy iteration algorithm. As can be seen, when the value of ϵ increases, the optimal ϵ-greedy policies are no longer consistent with the optimal grade one.

Optimality

We now show how the optimality of ϵ-greedy policies becomes worse when ϵ increases.

– First, a greedy optimal policy and the corresponding optimal state values are shown in Figure 5.5(a). They are obtained by the model-based value iteration algorithm.

– Second, the state values of some given ϵ-greedy policies are shown in Figure 5.5(b)-(d). These given ϵ-greedy policies are *consistent* with the greedy policy in Figure 5.5(a) but has different values of ϵ. Here, an ϵ-greedy is said to be consistent with another greedy policy if the actions with the greatest probabilities in the ϵ-greedy policy are the same as those in the greedy one.

  It is clear that, as ϵ increases, the state values of the ϵ-greedy policies decrease, indicating that the optimality of these ϵ-greedy policies becomes worse. It is notable that the value of the target state becomes the lowest instead of the highest when ϵ is large. That is simply because the target area is surrounded by some forbidden areas. When ϵ is large, the agent at the target area may enter these forbidden areas with a higher chance and get more negative rewards.

– Third, while the ϵ-greedy policies in Figure 5.5 are given, we next search optimal ϵ-greedy policies using the policy iteration algorithm with the constraint that all policies must be ϵ-greedy. Figure 5.6 shows the obtained optimal ϵ-greedy policies.

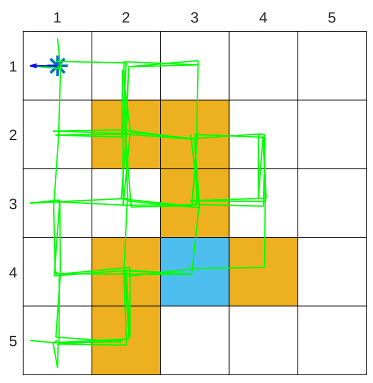
  When $\epsilon = 0$, the policy is greedy and optimal among all policies. As ϵ increases, the obtained ϵ-greedy policies are *not consistent* with the optimal greedy one anymore. Specifically, when ϵ is small such as 0.1, the resulting ϵ-greedy policy is consistent with the optimal greedy one. However, when ϵ increases to, for example, 0.2, the obtained ϵ-greedy policies are not consistent with the optimal greedy one. Therefore, if we would like to obtain ϵ-greedy policies that are consistent with the optimal greedy ones, the value of ϵ should be sufficiently small.

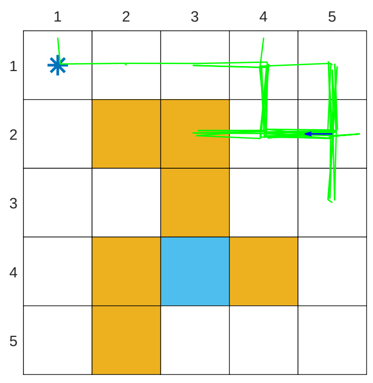
  Why are the ϵ-greedy policies not consistent with the optimal greedy one when ϵ is large? We can answer this question by considering the target state. In the greedy case, the optimal policy at the target state is to stay unchanged to gain positive rewards. However, when ϵ is large, the value of staying at the target state may be negative because there is a high chance to enter the forbidden areas and get negative rewards. Therefore, the optimal policy at the target state in this case is to escape instead of staying unchanged.

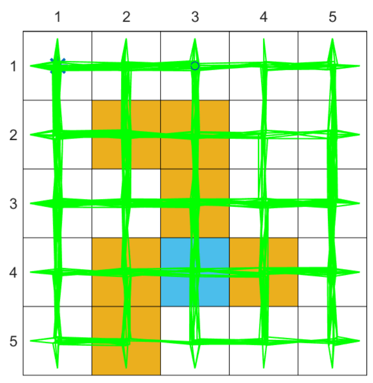
Exploration ability

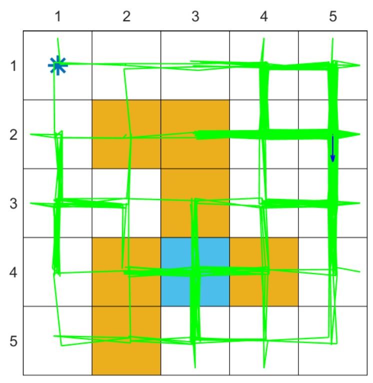
We next illustrate how the exploration ability of an ϵ-greedy policy is enhanced as ϵ increases by using examples.

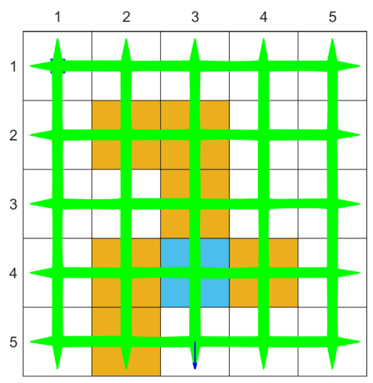
First, consider the figures in the left column of Figure 5.7. Here, $\epsilon = 1$. As a result, the ϵ-policy has the same probability of 0.2 to take any action at any state. Stating from

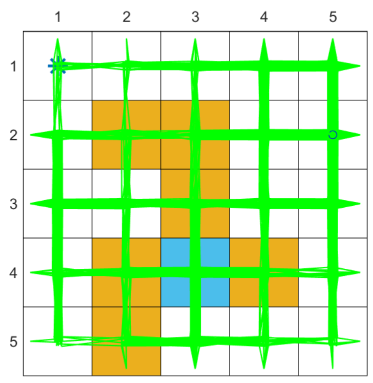

(a) $\epsilon = 1$, trajectory of 100 steps

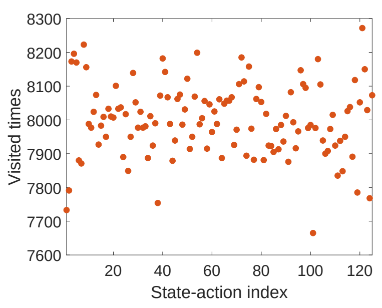

(e) $\epsilon = 0.5$, trajectory of 100 steps


(b) $\epsilon = 1$, trajectory of 1000 steps

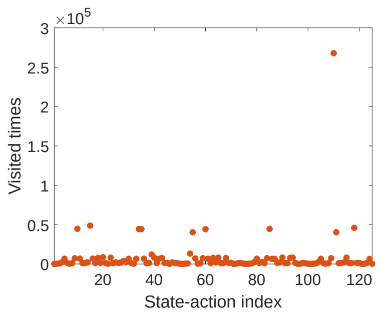

(f) $\epsilon = 0.5$, trajectory of 1000 steps


(c) $\epsilon = 1$, trajectory of 10000 steps


(g) $\epsilon = 0.5$, trajectory of 10000 steps



(d) $\epsilon = 1$, visited times of each action of 1 million steps



(h) $\epsilon = 0.5$, visited times of each action of 1 million steps

Figure 5.7: Exploration ability of ϵ-greedy policies with different values of ϵ.

the state-action pair (s_1, a_1), the trajectories generated following the policy are shown in Figure 5.7(a)-(c). As can be seen, all the state-action pairs can be visited many times when the episode is sufficiently long thanks to the strong exploration ability of the policy. Specifically, the times that the state-action pairs are visited are almost even as shown in Figure 5.7(d).

Second, consider the figures in the right column of Figure 5.7. Here, $\epsilon = 0.5$. As a result, the ϵ-greedy policy, which is plotted in Figure 5.5, has less exploration ability than the case of $\epsilon = 1$. Stating from the state-action pair (s_1, a_1), the trajectories generated following the policy are shown in Figure 5.7(e)-(g). Although every action can still be visited when the episode is sufficiently long, the distribution of the visit times may be extremely uneven. Those actions with greater selecting probabilities are visited much more times. For example, given an episode of one million steps, some actions are visited more than 250,000 times, while most actions are visited merely hundreds or even tens of times as shown in Figure 5.7(h).

According to the above analysis, although ϵ-greedy policies have a certain exploration ability, the exploration ability is limited when ϵ is small. Therefore, sufficient samples are required to ensure all the state-action pairs can be visited. Note that our task here is to find out the optimal policy for *every* state, which demands many visits for every state-action pair. In practice, if we are only interested in finding a good path from a given state to the target state, fewer samples are required. Moreover, one common technique used in practice is to set ϵ to be large initially to enhance exploration and gradually reduce it to ensure optimality of the final policy.

5.6 Summary

This chapter introduced the first class of model-free RL algorithms: MC learning methods. We first introduced the idea of MC estimation and how to estimate the expectation of a random variable using samples. When models are unavailable, the model-based component in the policy iteration algorithm can be replaced by a model-free MC estimation component. This leads to the simplest MC learning algorithm, which is called MC Basic in this book. The complication of MC learning appears when we would like to use samples more efficiently or avoid unrealistic assumptions on exploring starts. In particular, three MC-based algorithms were introduced in this chapter.

– MC Basic: This is the simplest MC learning algorithm. Although this algorithm is too simple to be used in practice, it clearly demonstrates the core idea of MC learning. This algorithm is obtained by simply replacing the model-based policy evaluation step in the policy iteration algorithm with a model-free MC-based estimation component. It is guaranteed that, given sufficient samples, this algorithm can converge to optimal policies and optimal state values.

– MC Exploring Starts: This algorithm is a variant of MC Basic. It can be obtained from the MC Basic algorithm by using first-visit or every-visit strategies to use experience samples more efficiently.

– MC ϵ-Greedy: This algorithm is a variant of the MC Exploring Starts algorithm. Specifically, in the policy improvement step, it searches for the best ϵ-greedy policies instead of greedy policies. In this way, the exploration ability of the policy is enhanced and hence the unrealistic assumption on exploring starts can be removed.

One important tradeoff of ϵ-greedy policies is exploration and exploitation. Specifically, when we increase the value of ϵ, the exploration ability of the ϵ-greedy would increase. However, the exploitation of the greedy action would decrease. On the other hand, if we reduce the value of ϵ so that we can fully exploit the greedy actions, the exploration ability of the policy would be compromised and hence the assumption on exploring starts becomes necessary.

5.7 Q&A

– Q: What is Monte Carlo estimation?

  A: Monte Carlo estimation refers to a broad class of techniques that use stochastic samples to solve approximation problems.

– Q: What is the mean estimation problem?

  A: The mean estimation problem refers to calculating the expectation of a random variable based on stochastic samples.

– Q: What are the two ways to solve the mean estimation problem?

  A: The two ways are model-based and model-free. In particular, if the probability distribution function is known, the expectation can be calculated based on its definition. If the probability distribution is unknown, we can use Monte Carlo estimation to approximate the expectation. Such kind of approximation is accurate when the number of samples is large.

– Q: How is mean estimation related to reinforcement learning?

  A: A core problem in reinforcement learning is calculating state values and action values. Both state value and action value are defined as expected values of returns. Hence, estimating state or action values is essentially a mean estimation problem.

– Q: What is the basic idea to obtain model-free reinforcement learning algorithms?

  A: The basic idea is to convert the model-based policy iteration algorithm to a model-free one. In particular, the policy iteration algorithm aims to calculate values based on the system model. We can also use Monte Carlo estimation to directly approximate action values based on episodes.

– Q: What are initial-visit, first-visit, and every-visit strategies, respectively?

  A: They are different strategies to use the samples in an episode. An episode may visit many state-action pairs. The initial-visit strategy uses the entire episode to estimate the action value of the initial state-action pair. In addition, one state action pair may be visited multiple times in an episode. If we only count the first-time visit, such kind of strategy is called first-visit. If every time a state-action pair is visited and the rest of the episode is used to estimate its action value, such a strategy is called every-visit.

– Q: What is the difference between MC Exploring Starts and MC Basic?

  A: MC Exploring Starts is a sample-efficient version of MC Basic. It introduces the first-visit or every-visit strategies to use episodes of samples more efficiently and, in the meantime, convert the policy update step to episode-by-episode.

– Q: What is exploring starts? Why is it important?

  A: Exploring starts is a requirement that an infinite number of (or sufficiently many) episodes must be generated starting from every state-action pair. In theory, exploring starts is necessary to find optimal policies. Only if every action value for every state is well explored, can we guarantee to select the optimal actions correctly. On the contrary, if an action is not explored, this action may happen to be the optimal one and hence be missed.

– Q: What is the idea of removing the requirement of exploring starts?

  A: The fundamental idea to remove the requirement of exploring starts is to make the policies soft. On the one hand, soft policies are stochastic so that an episode can visit state-action pairs many times. On the other hand, soft policies still exploit greedy actions to search for optimal policies.

– Q: Can an ϵ-greedy policy be optimal?

  A: The answer is both yes and no. By yes, it means that, if given sufficient samples, the MC ϵ-greedy algorithm can converge to an ϵ-greedy policy that is optimal among the set of all ϵ-greedy policies. By no, it means that the finally obtained policy is not optimal among all possible policies.

– Q: What does it mean when we say one ϵ-greedy policy is consistent with another greedy policy?

  A: An ϵ-greedy policy is called consistent with another greedy policy if the actions with the greatest probabilities in the ϵ-greedy policy are the same as those in the greedy one.

– Q: Is it possible to use one episode to visit all state-action pairs?

  A: Yes, if the policy is soft such as ϵ-greedy.

– Q: What is the relationship between MC Basic, MC Exploring Starts, and MC ϵ-greedy?

A: MC Basic is the simplest MC-based reinforcement learning algorithm. It is important because it reflects the core idea of model-free MC-based reinforcement learning. MC Exploring Starts is a variant of MC Basic by adjusting the sample usage strategy. Furthermore, MC ϵ-greedy is a variant of MC Exploring Starts by removing the requirement of exploring starts. Therefore, the basic idea is simple, but the complication appears when we would like to achieve better performance. Nevertheless, it is important to split the core idea from the complication that may be distracting for understanding a problem for beginners.

Chapter 6

Stochastic Approximation

In this next chapter, we will introduce the classical temporal-difference reinforcement learning (RL) algorithms. Before that, we need to press the pause button to get prepared better. That is because the ideas and expressions of temporal-difference algorithms are very different from what we have studied so far in this book. Many readers, who see the temporal-difference algorithms for the first time, often wonder why these algorithms were designed in the first place and why they work effectively. In fact, there is a knowledge gap between the contents in the previous and upcoming chapters. This chapter fills the gap by introducing basic stochastic approximation algorithms. Stochastic approximation refers to a broad class of stochastic iterative algorithms solving root finding or optimization problems. We will see in the next chapter that the temporal-difference algorithms are special stochastic approximation algorithms. As a result, it will be much easier to understand these algorithms.

6.1 Motivating example: mean estimation

Consider a random variable X which takes values in the finite set $\mathcal{X}$. Our aim is to estimate $\mathbb{E}[X]$. Suppose that we collected a sequence of iid samples $\{x_i\}_{i=1}^n$. The expectation of X can be approximated by

$$\mathbb{E}[X] \approx \bar{x} \doteq \frac{1}{n}\sum_{i=1}^{n} x_i. \tag{6.1}$$

The approximation in (6.1) is the basic idea of Monte Carlo estimation. We know that $\bar{x} \to \mathbb{E}[X]$ as $n \to \infty$ according to the Law of Large Numbers as introduced in the last chapter.

The problem that we would like to discuss is how to calculate the mean $\bar{x}$ in (6.1). There are two ways. The first way, which is trivial, is to collect all the samples and then calculate the average. The drawback of such a way is that, if the samples are collected one by one over some time, we have to wait until all the samples are collected. If the sampling time is long, such a wait is a waste of time. The second way can avoid this drawback because it calculates the average in an *incremental* and *iterative* manner. In particular, suppose

$$w_{k+1} = \frac{1}{k}\sum_{i=1}^{k} x_i, \quad k = 1, 2, \ldots$$

and hence

$$w_k = \frac{1}{k-1}\sum_{i=1}^{k-1} x_i, \quad k = 2, 3, \ldots$$

Then, w_{k+1} can be expressed in terms of w_k as

$$w_{k+1} = \frac{1}{k}\sum_{i=1}^{k} x_i = \frac{1}{k}\left(\sum_{i=1}^{k-1} x_i + x_k\right) = \frac{1}{k}((k-1)w_k + x_k) = w_k - \frac{1}{k}(w_k - x_k).$$

Therefore, we obtain the following iterative algorithm:

$$w_{k+1} = w_k - \frac{1}{k}(w_k - x_k). \tag{6.2}$$

We can use this iterative algorithm to calculate the mean $\bar{x}$ incrementally. It can be verified that

$$\begin{aligned}
w_1 &= x_1, \\
w_2 &= w_1 - \frac{1}{1}(w_1 - x_1) = x_1, \\
w_3 &= w_2 - \frac{1}{2}(w_2 - x_2) = x_1 - \frac{1}{2}(x_1 - x_2) = \frac{1}{2}(x_1 + x_2), \\
w_4 &= w_3 - \frac{1}{3}(w_3 - x_3) = \frac{1}{3}(x_1 + x_2 + x_3), \\
&\vdots \\
w_{k+1} &= \frac{1}{k}\sum_{i=1}^{k} x_i. \qquad (6.3)
\end{aligned}$$

An advantage of (6.2) is that a mean estimate can be obtained immediately once a sample is received. Then, the mean estimate can be used for other purposes immediately. Of course, the mean estimate is not accurate in the beginning due to insufficient samples. However, it is better than nothing. As more samples are obtained, the estimation accuracy can be improved gradually according to the Law of Large Numbers.

One can also define $w_{k+1} = \frac{1}{1+k}\sum_{i=1}^{k+1} x_i$ and $w_k = \frac{1}{k}\sum_{i=1}^{k} x_i$. It would not make too much difference. In this case, the corresponding iterative algorithm is $w_{k+1} = w_k - \frac{1}{1+k}(w_k - x_{k+1})$. The details of the derivation is left as an exercise to the reader.

Furthermore, consider an algorithm with a more general expression:

$$w_{k+1} = w_k - \alpha_k (w_k - x_k), \tag{6.4}$$

which is exactly the same as (6.2) except that $1/k$ is replaced by $\alpha_k > 0$. Does this algorithm still converge to the mean $\mathbb{E}[X]$? In fact, the answer is yes if $\{\alpha_k\}$ satisfies some mild conditions as we show in the next section. We will also show that (6.4) is a special stochastic approximation algorithm and also a special stochastic gradient descent algorithm.

The algorithms in (6.2) and (6.4) are the first stochastic iterative algorithms ever introduced in this book. In the next chapter, the reader will see that the temporal-difference algorithms have similar (but more complex) expressions.

6.2 Robbins-Monro Algorithm

Stochastic approximation refers to a broad class of stochastic iterative algorithms solving root finding or optimization problems. Compared to many other root-finding algorithms such as gradient-based methods, stochastic approximation is powerful in the sense that it does not require knowing the expression of the objective function or its derivative.

The Robbins-Monro (RM) algorithm [6] is a pioneering work in the field of stochastic approximation. The famous stochastic gradient descent algorithm is a special form of the RM algorithm. We next introduce its details. Suppose we would like to find the root of the equation

$$g(w) = 0,$$

where $w \in \mathbb{R}$ is the variable to be solved and $g : \mathbb{R} \to \mathbb{R}$ is a function. Many problems can be eventually converted to this root-finding problem. For example, suppose $J(w)$ is an objective function to be minimized. Then, the optimization problem can be converted to $g(w) = \nabla_w J(w) = 0$. Note that an equation like $g(w) = c$ with c as a constant can also be converted to the above equation by rewriting $g(w) - c$ as a new function.

If the expression of g or its derivative is known, there are many numerical algorithms that can solve this problem. However, the problem we are facing is firstly *the expression of the function g is unknown* (for example, the function is represented by an artificial neuron network) and secondly *g cannot be measured or observed accurately.* Suppose we can only obtain a noisy observation of g:

$$\tilde{g}(w, \eta) = g(w) + \eta,$$

where $\eta \in \mathbb{R}$ is the observation error. Here, η may be white noise or structural error [7]. Therefore, the system can be viewed as a *black-box* problem, because only the input w and the noisy output $\tilde{g}(w, \eta)$ are known. Our aim is to solve $g(w) = 0$ from w and $\tilde{g}$.

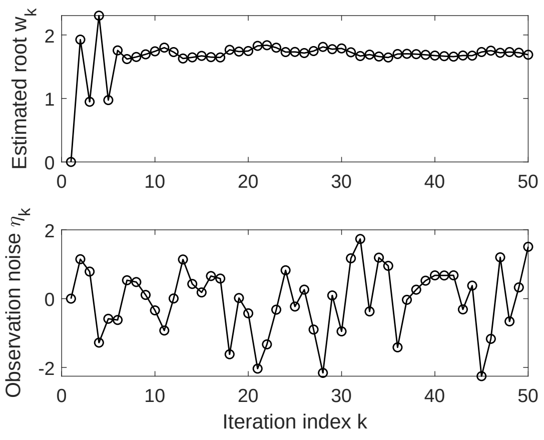


Figure 6.1: An illustrative example of the RM algorithm.

The RM algorithm that can solve the above problem is

$$w_{k+1} = w_k - a_k \tilde{g}(w_k, \eta_k), \qquad k = 1, 2, 3, \ldots \tag{6.5}$$

where w_k is the kth estimate of the root, $\tilde{g}(w_k, \eta_k)$ is the kth noisy observation, and a_k is a positive coefficient. As can be seen, the RM algorithm does not require knowing any information about the function. It only requires knowing the input and output data.

To illustrate the RM algorithm, consider an example where $g(w) = w^3 - 5$. The true root is $5^{1/3} \approx 1.71$. If we can only observe the input w and the output $\tilde{g}(w) = g(w) + \eta$, we can use the RM to find the root. In particular, suppose η_k is iid and obeys a standard normal distribution with a mean of zero and standard deviation of 1. The initial guess is $w_1 = 0$ and a_k is selected to be $a_k = 1/k$. The evolution of w_k is shown in Figure 6.1. As can be seen, even though the observation is corrupted by a noise η_k, the estimate w_k can still converge to the true root.

6.2.1 Convergence properties

Why can the RM algorithm in (6.5) find the root of $g(w) = 0$? We first illustrate the idea by an example and then give a rigorous convergence analysis.

Consider the example shown in Figure 6.2. In this example, $g(w) = \tanh(w-1)$. The true root of $g(w) = 0$ is $w^* = 1$. We apply the RM algorithm with $w_1 = 3$ and $a_k = 1/k$. To better illustrate the reason of convergence, we simply set $\eta_k \equiv 0$. The RM algorithm in this case is $w_{k+1} = w_k - a_k g(w_k)$ since $\tilde{g}(w_k, \eta_k) = g(w_k)$ when $\eta_k = 0$. The resulting $\{w_k\}$ is shown in Figure 6.2. As can be seen, w_k converges to the true root $w^* = 1$.

This simple example can illustrate why the RM algorithm converges.

- When $w_k > w^*$, we have $g(w_k) > 0$. Then, $w_{k+1} = w_k - a_k g(w_k) < w_k$ and hence w_{k+1} is closer to w^* than w_k.
- Similarly, when $w_k < w^*$, we have $g(w_k) < 0$. Then, $w_{k+1} = w_k - a_k g(w_k) > w_k$ and

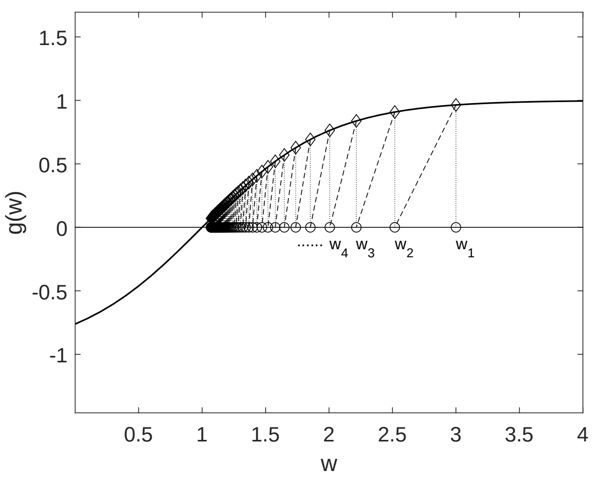


Figure 6.2: An example to illustrate the convergence of the RM algorithm.

w_{k+1} is closer to w^* than w_k.

Therefore, w_{k+1} gets closer to w^* in either case.

As illustrated by the above example, the convergence of the RM algorithm is intuitively straightforward to see. However, it is nontrivial to prove it rigorously in the presence of stochastic observation errors. A rigorous convergence result is given below.

Theorem 6.1 (Robbins-Monro Theorem)**.** *In the Robbins-Monro algorithm, if*

1) $0 < c_1 \le \nabla_w g(w) \le c_2$ *for all* w*;*

2) $\sum_{k=1}^{\infty} a_k = \infty$ *and* $\sum_{k=1}^{\infty} a_k^2 < \infty$*;*

3) $\mathbb{E}[\eta_k|\mathcal{H}_k] = 0$ *and* $\mathbb{E}[\eta_k^2|\mathcal{H}_k] < \infty$*;*

where $\mathcal{H}_k = \{w_k, w_{k-1}, \dots\}$*, then* w_k *converges with probability 1 (w.p.1) to the root* w^* *satisfying* $g(w^*) = 0$*.*

We postpone the proof of this result to the next section after a powerful tool for analyzing the convergence of stochastic sequences is introduced. This theorem relies on the notion of "convergence with probability 1", which is introduced in the appendix of this book.

The interpretations of the three conditions in Theorem 6.1 are given below.

– The first condition requires g to be monotonically increasing (or nondecreasing). This condition ensures that the root of $g(w) = 0$ exists and is unique.

– The second condition of $\{a_k\}$ is interesting. We often see conditions like this in RL algorithms. The condition of $\sum_{k=1}^{\infty} a_k^2 < \infty$ requires that a_k must converge to zero as $k \to \infty$. The condition of $\sum_{k=1}^{\infty} a_k = \infty$ requires that a_k should not converge to zero too fast.

– The third condition is a mild condition. The observation error η_k is not required to be Gaussian. A special yet common case is that $\{\eta_k\}$ is an iid stochastic sequence satisfying

$\mathbb{E}[\eta_k] = 0$ and $\mathbb{E}[\eta_k^2] < \infty$. This special case implies the third condition because η_k is independent to $\mathcal{H}_k$ and hence we have $\mathbb{E}[\eta_k|\mathcal{H}_k] = \mathbb{E}[\eta_k] = 0$ and $\mathbb{E}[\eta_k^2|\mathcal{H}_k] = \mathbb{E}[\eta_k^2]$.

Furthermore, we examine the second condition in Theorem 6.1 more closely.

– Why is the second condition important for the convergence of the algorithm?

  This question of course can be answered when we present the rigorous proof of the theorem later. Here, we would like to give some insightful intuition.

  First, $\sum_{k=1}^{\infty} a_k^2 < \infty$ indicates that $a_k \to 0$ as $k \to \infty$. Why is this condition important? Since

$$w_{k+1} - w_k = -a_k \tilde{g}(w_k, \eta_k),$$

  if $a_k \to 0$, then $a_k \tilde{g}(w_k, \eta_k) \to 0$ and hence $w_{k+1} - w_k \to 0$, indicating that w_{k+1} and w_k get close to each other when $k \to \infty$. Otherwise, if a_k does not converge, then w_k may still fluctuate when $k \to \infty$.

  Second, $\sum_{k=1}^{\infty} a_k = \infty$ indicates that a_k should not converge to zero too fast. Why is this condition important? Summarizing $w_2 = w_1 - a_1 \tilde{g}(w_1, \eta_1)$, $w_3 = w_2 - a_2 \tilde{g}(w_2, \eta_2)$, $\ldots$, $w_{k+1} = w_k - a_k \tilde{g}(w_k, \eta_k)$ leads to

$$w_\infty - w_1 = \sum_{k=1}^{\infty} a_k \tilde{g}(w_k, \eta_k).$$

  Suppose $w_\infty = w^*$. If $\sum_{k=1}^{\infty} a_k < \infty$, then $\sum_{k=1}^{\infty} a_k \tilde{g}(w_k, \eta_k)$ may be bounded. Then, if the initial guess w_1 is chosen arbitrarily far away from w^*, then the above equality would be invalid. Therefore, the condition $\sum_{k=1}^{\infty} a_k = \infty$ can make sure that the algorithm converges given an arbitrary initial guess.

– What kinds of sequences satisfy $\sum_{k=1}^{\infty} a_k = \infty$ and $\sum_{k=1}^{\infty} a_k^2 < \infty$?

  One typical sequence is

$$\alpha_k = \frac{1}{k}.$$

  Why is that? On the one hand, it holds that

$$\lim_{n \to \infty} \left(\sum_{k=1}^{n} \frac{1}{k} - \ln n \right) = \kappa,$$

  where $\kappa \approx 0.577$ is called the Euler-Mascheroni constant (also called Euler's constant) [8]. Since $\ln n \to \infty$ as $n \to \infty$, we have

$$\sum_{k=1}^{\infty} \frac{1}{k} = \infty.$$

  In fact, $H_n = \sum_{k=1}^{n} 1/k$ has a specific name in the number theory: Harmonic number

[9]. On the other hand, it is notable that

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6} < \infty.$$

The limit $\sum_{k=1}^{\infty} 1/k^2$ also has a specific name in the number theory: Basel problem. Therefore, the sequence $\{1/k\}$ satisfies the second condition in Theorem 6.1. Of course, a slight modification, such as $a_k = 1/k + 1$ or $a_k = c_k/k$ where c_k is bounded and maybe varying, also preserves the condition.

While the RM algorithm is guaranteed to converge when the three conditions in Theorem 6.1 are satisfied, it may still be effective to a certain extent even though some of the conditions are not satisfied. For example, in the example in Figure 6.1, $g(x) = x^3 - 5$ does not satisfy the first condition on gradient boundedness. Nevertheless, the RM algorithm can still find the root if the initial guess is adequately (not arbitrarily) selected. More importantly, we will see that a_k is often selected as a sufficiently small constant in many RL algorithms. Although the second condition is not satisfied in this case, the algorithm can still work effectively.

6.2.2 Application to mean estimation

We next apply the Robbins-Monro Theorem to analyze the mean estimation problem discussed in the first section of this chapter. Recall that

$$w_{k+1} = w_k + \alpha_k(x_k - w_k).$$

is the mean estimation algorithm in (6.4). It was mentioned that while α_k satisfies some mild conditions, this algorithm can converge to $\mathbb{E}[X]$. The convergence proof was not given. We now show that it is a special RM algorithm and hence its convergence naturally follows.

Consider

$$g(w) \doteq w - \mathbb{E}[X].$$

Our aim is to solve $g(w) = 0$. If we can do that, then we obtain the value of $\mathbb{E}[X]$. The observation we can get is

$$\tilde{g}(w, x) \doteq w - x,$$

because we can only obtain a sample x of X. Note that

$$\begin{aligned}\tilde{g}(w,\eta) &= w - x \\ &= w - x + \mathbb{E}[X] - \mathbb{E}[X] \\ &= (w - \mathbb{E}[X]) + (\mathbb{E}[X] - x) \doteq g(w) + \eta,\end{aligned}$$

where $\eta \doteq \mathbb{E}[X] - x$. Therefore, the observation $\tilde{g}(w,\eta)$ is the sum of $g(w)$ and an observation error η. The RM algorithm for solving $g(x) = 0$ is

$$w_{k+1} = w_k - \alpha_k \tilde{g}(w_k, \eta_k) = w_k - \alpha_k(w_k - x_k),$$

which is exactly the algorithm in (6.4). It is guaranteed by Theorem 6.1 that w_k converges $\mathbb{E}[X]$ with probability 1 if $\sum_{k=1}^{\infty} \alpha_k = \infty$ and $\sum_{k=1}^{\infty} \alpha_k^2 < \infty$ and $\{x_k\}$ is iid.

It should be noted that there is no requirement for the distribution of X. The convergence is guaranteed as long as the samples of X are iid and the variance is bounded according to Theorem 6.1. Moreover, while we can obtain the analytical expression of w_{k+1} as $w_{k+1} = 1/k \sum_{i=1}^{k} x_i$ when $\alpha_k = 1/k$, we would not be able to write out its analytical expression when α_k is general. The convergence analysis is nontrivial in this case and the Robbins-Monro Theorem provides an elegant tool to analyze the convergence.

6.3 Dvoretzky's convergence theorem

Dvoretzky's Theorem, published in 1956 [10], is a classic result in the area of stochastic approximation. It can be used to prove the convergence of the RM algorithm and many RL algorithms.

This section is a little mathematically intensive. Readers that are interested in the convergence analysis of stochastic algorithms are recommended to study this section. Otherwise, this section can be skipped.

Theorem 6.2 (Dvoretzky's Theorem)**.** *Consider a stochastic process*

$$w_{k+1} = (1 - \alpha_k) w_k + \beta_k \eta_k,$$

where $\{\alpha_k\}_{k=1}^{\infty}, \{\beta_k\}_{k=1}^{\infty}, \{\eta_k\}_{k=1}^{\infty}$ are stochastic sequences. Here $\alpha_k \geq 0, \beta_k \geq 0$ for all k. Then, w_k would converge to zero with probability 1 if the following conditions are satisfied:

1) $\sum_{k=1}^{\infty} \alpha_k = \infty$, $\sum_{k=1}^{\infty} \alpha_k^2 < \infty$; $\sum_{k=1}^{\infty} \beta_k^2 < \infty$ uniformly w.p.1;

2) $\mathbb{E}[\eta_k | \mathcal{H}_k] = 0$ and $\mathbb{E}[\eta_k^2 | \mathcal{H}_k] \leq C$ w.p.1;

where $\mathcal{H}_k = \{w_k, w_{k-1}, \ldots, \eta_{k-1}, \ldots, \alpha_{k-1}, \ldots, \beta_{k-1}, \ldots\}$.

Before presenting the proof of this theorem, we first clarify some problems.

- In the RM algorithm, the coefficient sequence $\{\alpha_k\}$ is deterministic. However, Dvoretzky's Theorem allows $\{\alpha_k\}, \{\beta_k\}$ to be random variables depending on $\mathcal{H}_k$. Thus, it is more general and powerful because we may encounter the case where α_k or β_k is a function of w_k or η_k.
- In the first condition, it is stated as "uniformly w.p.1". That is because α_k and β_k may be random variables and hence the definition of their limits must be in the stochastic sense. In the second condition, it is also stated as "w.p.1". This is because $\mathcal{H}_k$ is a sequence of random variables instead of specific values. The definition of the conditional expectation in this case is in the sense of w.p.1.
- The statement of Theorem 6.2 is slightly different from [11] in the sense that Theorem 6.2 does not require $\sum_{k=1}^{\infty} \beta_k = \infty$ in the first condition. Even in the extreme case where $\beta_k = 0$ for all k, the sequence can still converge.

6.3.1 Proof of Dvoretzky's Theorem

The original proof of Dvoretzky's theorem was given in 1956 [10]. There are quite a few ways to prove it. We next present a proof based on quasimartingales inspired by [12].

Proof of Dvoretzky's Theorem. Let $h_k \doteq w_k^2$. Then

$$\begin{aligned} h_{k+1} - h_k &= w_{k+1}^2 - w_k^2 \\ &= (w_{k+1} - w_k)(w_{k+1} + w_k) \\ &= (-\alpha_k w_k + \beta_k r_k)[(2 - \alpha_k) w_k + \beta_k r_k] \\ &= -\alpha_k (2 - \alpha_k) w_k^2 + \beta_k^2 \eta_k^2 + 2(1 - \alpha_k)\beta_k r_k w_k. \end{aligned}$$

Taking expectation on both sides of the above equation gives

$$\mathbb{E}[h_{k+1} - h_k | \mathcal{H}_k] = \mathbb{E}[-\alpha_k(2 - \alpha_k) w_k^2 | \mathcal{H}_k] + \mathbb{E}[\beta_k^2 \eta_k^2 | \mathcal{H}_k] + \mathbb{E}[2(1 - \alpha_k)\beta_k r_k w_k | \mathcal{H}_k]. \tag{6.6}$$

Since w_k is determined by $\mathcal{H}_k$, it can be taken out from the expectation. Moreover, suppose α_k, β_k is totally determined by $\mathcal{H}_k$. This is valid if, for example, $\{\alpha_k\}$ and $\{\beta_k\}$ are deterministic sequences or functions of w_k. Then, they can also be taken out from the expectation. Therefore, (6.6) becomes

$$\mathbb{E}[h_{k+1} - h_k | \mathcal{H}_k] = -\alpha_k(2 - \alpha_k) w_k^2 + \beta_k^2 \mathbb{E}[\eta_k^2 | \mathcal{H}_k] + 2(1 - \alpha_k)\beta_k \mathbb{E}[\eta_k w_k | \mathcal{H}_k]. \tag{6.7}$$

For the first term, since $\sum_{k=1}^{\infty} \alpha_k^2 < \infty$ implies $\alpha_k \to 0$ w.p.1. As a result, there exists a finite n such that $\alpha_k \leq 1$ w.p.1 for all $k \geq n$. Without loss of generality, we can simply consider the case of $k \geq n$ and hence $\alpha_k \leq 1$ w.p.1. Then, $-\alpha_k(2-\alpha_k)w_k^2 \leq 0$. For the second term, we have $\beta_k^2\mathbb{E}[\eta_k^2|\mathcal{H}_k] \leq \beta_k^2 C$ as assumed. For the third term, we have $2(1-\alpha_k)\beta_k\mathbb{E}[\eta_k w_k|\mathcal{H}_k] = 2(1-\alpha_k)\beta_k w_k\mathbb{E}[\eta_k|\mathcal{H}_k] = 0$ since $\mathbb{E}[\eta_k|\mathcal{H}_k] = 0$. Therefore, (6.7) becomes

$$\mathbb{E}[h_{k+1} - h_k|\mathcal{H}_k] = -\alpha_k(2-\alpha_k)w_k^2 + \beta_k^2\mathbb{E}[\eta_k^2|\mathcal{H}_k] \leq \beta_k^2 C \tag{6.8}$$

and hence

$$\sum_{k=1}^{\infty}\mathbb{E}[h_{k+1} - h_k|\mathcal{H}_k] \leq \sum_{k=1}^{\infty}\beta_k^2 C < \infty.$$

The last inequality is due to the condition $\sum_{k=1}^{\infty}\beta_k^2 < \infty$. Then, based on the theorem of quasimartingales,we conclude that h_k converges w.p.1.

While we now know that h_k is convergent and so is w_k, we next determine what value w_k converges to. It follows from (6.8) that

$$\sum_{k=1}^{\infty}\alpha_k(2-\alpha_k)w_k^2 = \sum_{k=1}^{\infty}\beta_k^2\mathbb{E}[\eta_k^2|\mathcal{H}_k] - \sum_{k=1}^{\infty}\mathbb{E}[h_{k+1} - h_k|\mathcal{H}_k].$$

The first term on the right hand side is bounded as assumed. The second term is bounded because h_k converges and hence $h_{k+1} - h_k$ is summable. Hence, $\sum_{k=1}^{\infty}\alpha_k(2-\alpha_k)w_k^2$ on the left-hand side is also bounded. Since we consider the case of $\alpha_k \leq 1$, we have

$$\infty > \sum_{k=1}^{\infty}\alpha_k(2-\alpha_k)w_k^2 \geq \sum_{k=1}^{\infty}\alpha_k w_k^2 \geq 0.$$

Therefore, $\sum_{k=1}^{\infty}\alpha_k w_k^2$ is bounded. Since $\sum_{k=1}^{\infty}\alpha_k = \infty$, we must have $w_k \to 0$ w.p.1. □

6.3.2 Application to mean estimation

While the mean estimation algorithm, $w_{k+1} = w_k + \alpha_k(x_k - w_k)$, has been analyzed based on the RM Theorem, we next show that its convergence can also be directly proven based on the Dvoretzky's theorem.

Proof. Let $w^* = \mathbb{E}[X]$. The mean estimation algorithm $w_{k+1} = w_k + \alpha_k(x_k - w_k)$ can be rewritten as

$$w_{k+1} - w^* = w_k - w^* + \alpha_k(x_{k+1} - w^* + w^* - w_k)$$

Let $\Delta \doteq w - w^*$. Then, we have

$$\begin{aligned}\Delta_{k+1} &= \Delta_k + \alpha_k(x_{k+1} - w^* - \Delta_k)\\ &= (1-\alpha_k)\Delta_k + \alpha_k \underbrace{(x_k - w^*)}_{\eta_k}.\end{aligned}$$

Since $\{x_k\}$ is iid, then $\mathbb{E}[x_k|\mathcal{H}_k] = \mathbb{E}[x_k] = w^*$. Moreover, $\mathbb{E}[\eta_k] = \mathbb{E}[x_k - w^*] = 0$ and $\mathbb{E}[\eta_k^2] = \mathbb{E}[x_k^2] - \mathbb{E}[X]^2$ is bounded. Following Dvoretzky's theorem, we conclude that Δ_k converges to zero and hence w_k converges to $w^* = \mathbb{E}[X]$ w.p.1. □

6.3.3 Application to the Robbins-Monro theorem

We are now ready to prove the Robbins-Monro theorem by using Dvoretzky's theorem.

Proof of the Robbins-Monro theorem. The RM algorithm aims to find the root of $g(w) = 0$. Suppose the root is w^* such that $g(w^*) = 0$. The RM algorithm is

$$\begin{aligned}w_{k+1} &= w_k - a_k\tilde{g}(w_k, \eta_k)\\ &= w_k - a_k[g(w_k) + \eta_k].\end{aligned}$$

Then, we have

$$w_{k+1} - w^* = w_k - w^* - a_k[g(w_k) - g(w^*) + \eta_k].$$

Due to the mean value theorem (Appendix x), we have $g(w_k) - g(w^*) = \nabla_w g(w_k')(w_k - w^*)$, where $w_k' \in [w_k, w^*]$. Let $\Delta_k \doteq w_k - w^*$. The above equation becomes

$$\begin{aligned}\Delta_{k+1} &= \Delta_k - a_k[\nabla_w g(w_k')(w_k - w^*) + \eta_k]\\ &= \Delta_k - a_k\nabla_w g(w_k')\Delta_k + a_k r_k\\ &= [1 - \underbrace{a_k\nabla_w g(w_k')}_{\alpha_k}]\Delta_k + a_k r_k.\end{aligned}$$

Note that $\nabla_w g(w)$ is always bounded by $0 < c_1 \le \nabla_w g(w) \le c_2$ as assumed. Since $\sum_{k=1}^{\infty} a_k = \infty$ and $\sum_{k=1}^{\infty} a_k^2 < \infty$ as assumed, we know $\sum_{k=1}^{\infty} \alpha_k = \infty$ and $\sum_{k=1}^{\infty} \alpha_k^2 < \infty$. Thus, all the conditions in Dvoretzky's theorem are satisfied and hence Δ_k converges to zero w.p.1. □

The proof of the RM theorem demonstrates the powerfulness of Dvoretzky's theorem. In particular, α_k in the proof is a stochastic sequence depending on w_k' and w_k rather than a deterministic sequence. In this case, Dvoretzky's theorem is still applicable.

6.3.4 An extension of Dvoretzky's Theorem

We next extend Dvoretzky's theorem to a more general theorem that can handle multiple variables. This general theorem, proposed by [11], can be used to analyze the convergence of stochastic iterative algorithms such as Q-learning and temporal-difference RL algorithms.

Theorem 6.3. *Let x be any element of a set $\mathcal{X}$. For the stochastic iterative process*

$$\Delta_{k+1}(x) = (1 - \alpha_k(x))\Delta_k(x) + \beta_k(x)e_k(x), \tag{6.9}$$

$\Delta_k(x)$ converges to zero w.p.1 if

1) The set $\mathcal{X}$ is finite;

2) $\sum_k \alpha_k(x) = \infty$, $\sum_k \alpha^2(x) < \infty$, $\sum_k \beta_k^2(x) < \infty$, and $\mathbb{E}[\beta_k(x)|\mathcal{H}_k] \le \mathbb{E}[\alpha_k(x)|\mathcal{H}_k]$ uniformly w.p.1;

3) $\|\mathbb{E}[e_k(x)|\mathcal{H}_k]\|_\infty \le \gamma\|\Delta_k\|_\infty$, where $\gamma \in (0,1)$;

4) $\mathrm{var}[e_k(x)|\mathcal{H}_k] \le C(1 + \|\Delta_k(x)\|_\infty)^2$, where C is a constant.

Here, $\mathcal{H}_k = \{\Delta_k, \Delta_{k-1}, \dots, e_{k-1}, \dots, \alpha_{k-1}, \dots, \beta_{k-1}, \dots\}$ stands for the history. The notation $\|\cdot\|_\infty$ refers to the maximum norm.

Proof. As an extension, it can be proven based on Dvoretzky's theorem. Details can be found in [11] and are omitted here. □

Some remarks about this theorem are given below.

– We first clarify some notations in the theorem. The variable x can be viewed as an index. In the context of RL, it often represents a state or state-action pair. Although there is a maximum norm in the conditions, every variable in this theorem is scalar. The maximum norm is defined as $\|\mathbb{E}[e_k(x)|\mathcal{H}_k]\|_\infty = \max_x |\mathbb{E}[e_k(x)|\mathcal{H}_k]|$ and $\|\Delta_k(x)\|_\infty = \max_x |\Delta_k(x)|$. If we put them for different states in a vector, this norm is actually the L_∞ norm of a vector:

$$\|\mathbb{E}[e_k(x)|\mathcal{H}_k]\|_\infty \doteq \max_x |\mathbb{E}[e_k(x)|\mathcal{H}_k]| = \left\| \begin{bmatrix} \vdots \\ \mathbb{E}[e_k(x)|\mathcal{H}_k] \\ \vdots \end{bmatrix} \right\|_\infty .$$

where different rows of the vectors correspond to different x.

– This theorem is more general than Dvoretzky's theorem. First, it can handle the case of multiple variables due to the maximum norm. This is important for RL problems that have multiple states. Second, while Dvoretzky's theorem requires that $\mathbb{E}[e_k(x)|\mathcal{H}_k] = 0$ and $\text{var}[e_k(x)|\mathcal{H}_k] \le C$, this theorem only requires that the expectation and variance are bounded by the error Δ_k.

– While (6.9) is merely for a single state, the reason that it can handle multiple states is because of conditions 3 and 4, which are for the entire state space. Moreover, when applying this theorem to prove the convergence of RL algorithms, we need to show that (6.9) is valid for every state.

6.4 Stochastic gradient descent

This section introduces stochastic gradient descent (SGD) algorithms, which are widely used in the field of machine learning. We will show that SGD is a special RM algorithm and the mean estimation algorithm is a special SGD algorithm.

Suppose we would like to solve the following optimization problem:

$$\min_w \quad J(w) = \mathbb{E}[f(w, X)], \tag{6.10}$$

where w is the parameter to be optimized and X is a random variable. The expectation is with respect to X. Here, w and X can be either scalars or vectors. The function $f(\cdot)$ is a scalar.

A straightforward method to solve (6.10) is *gradient descent*. Let the gradient of $\mathbb{E}[f(w, X)]$ be $\nabla_w \mathbb{E}[f(w, X)] = \mathbb{E}[\nabla_w f(w, X)]$. Then, the gradient-descent algorithm is

$$w_{k+1} = w_k - \alpha_k \nabla_w \mathbb{E}[f(w_k, X)] = w_k - \alpha_k \mathbb{E}[\nabla_w f(w_k, X)] \tag{6.11}$$

The gradient-descent algorithm can find the optimal solution under some mild conditions such as the convexity of f. An introduction to gradient-descent algorithms is given in Appendix B.

The problem of the gradient descent algorithm in (6.11) is that the expected value on the right-hand side is difficult to calculate. One potential way to calculate the expected value is based on the probability distribution of X. However, the distribution is often unknown in practice. Another way is to collect a large number of iid samples $\{x_i\}_{i=1}^n$ of X so that the expected value can be approximated as

$$\mathbb{E}[\nabla_w f(w_k, X)] \approx \frac{1}{n} \sum_{i=1}^n \nabla_w f(w_k, x_i).$$

Then, (6.11) becomes

$$w_{k+1} = w_k - \frac{1}{n}\sum_{i=1}^{n} \nabla_w f(w_k, x_i). \tag{6.12}$$

This algorithm is called *batch gradient descent (BGD)* because it uses all the samples as a single batch in every iteration.

The problem of the BGD algorithm in (6.12) is that it requires all the samples in each iteration. In practice, since the samples may be collected incrementally, it is favorable to optimize w instantly every time a sample is collected. To that end, we can use the following algorithm:

$$w_{k+1} = w_k - \alpha_k \nabla_w f(w_k, x_k), \tag{6.13}$$

where x_k is the sample collected at time step k. This is the well-known *stochastic gradient descent* algorithm. The reason that this algorithm is called stochastic is that it relies on stochastic samplings $\{x_k\}$.

Compared to the gradient descent algorithm in (6.11), SGD replaces the true gradient $\mathbb{E}[\nabla_w f(w, X)]$ by the *stochastic gradient* $\nabla_w f(w_k, x_k)$. Since $\nabla_w f(w_k, x_k) \neq \mathbb{E}[\nabla_w f(w, X)]$, whether such a replacement can still ensure $w_k \to w^*$ as $k \to \infty$? The answer is yes. We next show some intuitive explanation and will give the rigourous proof of the convergence in Section 6.4.5. Because

$$\begin{aligned}\nabla_w f(w_k, x_k) &= \mathbb{E}[\nabla_w f(w, X)] + \Big(\nabla_w f(w_k, x_k) - \mathbb{E}[\nabla_w f(w, X)]\Big)\\ &\doteq \mathbb{E}[\nabla_w f(w, X)] + \eta_k,\end{aligned}$$

the SGD algorithm can be rewritten as

$$w_{k+1} = w_k - \alpha_k \mathbb{E}[\nabla_w f(w, X)] - \alpha_k \eta_k.$$

Therefore, SGD is the same as gradient descent except it has a perturbation term $\alpha_k \eta_k$. Since $\{x_k\}$ is iid, we have $\mathbb{E}_{x_k}[\nabla_w f(w_k, x_k)] = \mathbb{E}_X[\nabla_w f(w, X)]$. As a result,

$$\mathbb{E}[\eta_k] = \mathbb{E}\Big[\nabla_w f(w_k, x_k) - \mathbb{E}[\nabla_w f(w, X)]\Big] = \mathbb{E}_{x_k}[\nabla_w f(w_k, x_k)] - \mathbb{E}_X[\nabla_w f(w, X)] = 0.$$

Therefore, the perturbation term η_k has zero mean, which intuitively suggests that it would not jeopardize the convergence. A rigourous proof of the convergence of SGD will be given in Section 6.4.5.

6.4.1 Application to mean estimation

We next apply SGD to analyze the mean estimation problem and show that the mean estimation algorithm in (6.4) is a special SGD algorithm. To that end, formulate the mean estimation problem as an optimization problem:

$$\min_w \quad J(w) = \mathbb{E}\left[\frac{1}{2}\|w - X\|^2\right] \doteq \mathbb{E}[f(w, X)], \tag{6.14}$$

where $f(w, X) = \|w - X\|^2/2$ and $\nabla_w f(w, X) = w - X$. It can be verified that the optimal solution is $w^* = \mathbb{E}[X]$.

The gradient descent algorithm for solving (6.14) is

$$\begin{aligned} w_{k+1} &= w_k - \alpha_k \nabla_w J(w_k) \\ &= w_k - \alpha_k \mathbb{E}[\nabla_w f(w_k, X)] \\ &= w_k - \alpha_k \mathbb{E}[w_k - X]. \end{aligned}$$

The gradient descent algorithm is not applicable here because $\mathbb{E}[w_k - X]$ or $\mathbb{E}[X]$ on the right-hand side is unknown (in fact, it is what we need to solve).

The SGD algorithm for solving (6.14) is

$$w_{k+1} = w_k - \alpha_k \nabla_w f(w_k, x_k) = w_k - \alpha_k (w_k - x_k),$$

where x_k is a sample obtained at time step k. It is noticed this SGD algorithm is the same as the iterative mean estimation algorithm in (6.4). Therefore, (6.4) is an SGD algorithm specifically for solving the mean estimation problem.

6.4.2 Convergence pattern of SGD

SGD uses the stochastic gradient $\nabla_w f(w_k, x_k)$ to approximate the true gradient $\mathbb{E}[\nabla_w f(w_k, X)]$. Since the stochastic gradient is random and hence the approximation is inaccurate, an important question is whether the convergence of SGD is slow or random.

To answer this question, we consider the *relative error* between the stochastic and batch gradients:

$$\delta_k \doteq \frac{|\nabla_w f(w_k, x_k) - \mathbb{E}[\nabla_w f(w_k, X)]|}{|\mathbb{E}[\nabla_w f(w_k, X)]|}.$$

For the sake of simplicity, we assume all the variables are *scalar*. If δ_k is small, we can expect that SGD behaves similarly to the standard gradient descent. Since w^* is assumed to be the optimal solution and hence $\mathbb{E}[\nabla_w f(w^*, X)] = 0$, the relative error can be written as

$$\delta_k = \frac{|\nabla_w f(w_k, x_k) - \mathbb{E}[\nabla_w f(w_k, X)]|}{|\mathbb{E}[\nabla_w f(w_k, X)] - \mathbb{E}[\nabla_w f(w^*, X)]|} = \frac{|\nabla_w f(w_k, x_k) - \mathbb{E}[\nabla_w f(w_k, X)]|}{|\mathbb{E}[\nabla_w^2 f(\tilde{w}_k, X)(w_k - w^*)]|}. \tag{6.15}$$

where the last equality is due to the mean value theorem and $\tilde{w}_k \in [w_k, w^*]$. Suppose f is strictly convex such that $\nabla_w^2 f \geq c > 0$ for all w, X, where c is a positive bound. Then, the denominator of δ_k in (6.15) becomes

$$\begin{aligned}\big|\mathbb{E}[\nabla_w^2 f(\tilde{w}_k, X)(w_k - w^*)]\big| &= \big|\mathbb{E}[\nabla_w^2 f(\tilde{w}_k, X)](w_k - w^*)\big| \\ &= \big|\mathbb{E}[\nabla_w^2 f(\tilde{w}_k, X)]\big|\big|(w_k - w^*)\big| \\ &\geq c|w_k - w^*|.\end{aligned}$$

Substituting the above inequality to (6.15) gives

$$\delta_k \leq \frac{\big|\overbrace{\nabla_w f(w_k, x_k)}^{\text{stochastic gradient}} - \overbrace{\mathbb{E}[\nabla_w f(w_k, X)]}^{\text{true gradient}}\big|}{\underbrace{c|w_k - w^*|}_{\text{distance to the optimal solution}}}.$$

The above equation suggests an interesting convergence pattern of SGD. In particular, the relative error δ_k is inversely proportional to $|w_k - w^*|$. As a result, when $|w_k - w^*|$ is large, δ_k is small. In this case, the SGD algorithm behaves like the gradient descent algorithm and hence w_k approach w^* quickly. However, when w_k is close to w^*, the relative error δ_k may be large and the convergence exhibits more randomness.

A good example to demonstrate the above analysis is the mean estimation problem. Consider the mean estimation problem in (6.14). When w and X are both scalar, we have $f(w, X) = |w - X|^2/2$ and hence

$$\begin{aligned}\nabla_w f(w, x_k) &= w - x_k, \\ \mathbb{E}[\nabla_w f(w, x_k)] &= w - \mathbb{E}[X] = w - w^*.\end{aligned}$$

As a result, the relative error is

$$\delta_k = \frac{|\nabla_w f(w_k, x_k) - \mathbb{E}[\nabla_w f(w_k, X)]|}{|\mathbb{E}[\nabla_w f(w_k, X)]|} = \frac{|(w_k - x_k) - (w_k - \mathbb{E}[X])|}{|w_k - w^*|} = \frac{|\mathbb{E}[X] - x_k|}{|w_k - w^*|}.$$

The expression of the relative error clearly shows that δ_k is inversely proportional to $|w_k - w^*|$. As a result, when w_k is far away from w^*, the relative error is small and SGD behaves like gradient descent. On the other hand, δ_k is proportional to $|\mathbb{E}[X] - x_k|$. As a result, the mean of δ_k is proportional to the variance of X.

The simulation results in Figure 6.3 illustrate the above analysis. In this simulation, the random variable $X \in \mathbb{R}^2$ represents a two-dimensional position in the plane. Its

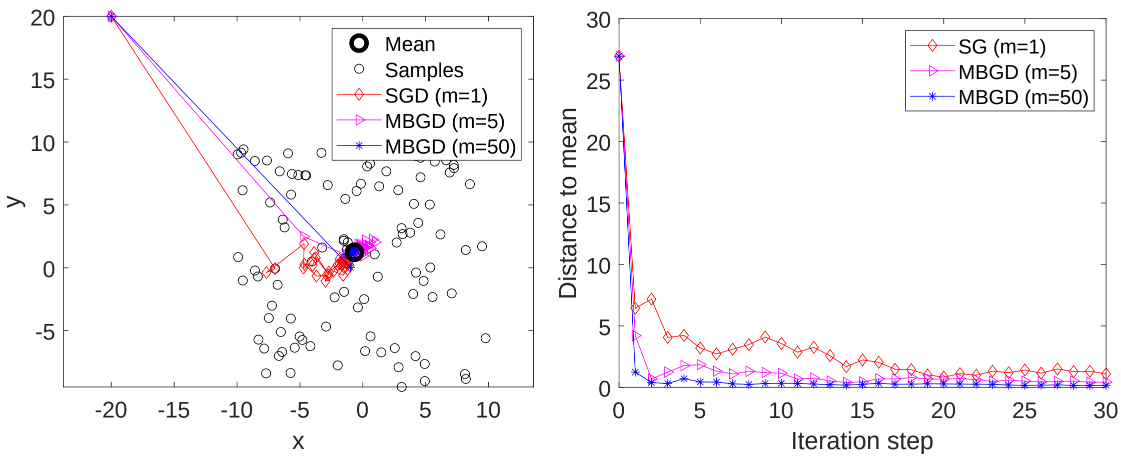


Figure 6.3: An example to demonstrate the convergence process of different gradient descent algorithms. In this example, the random variable $X \in \mathbb{R}^2$ represents a two-dimensional position in the plane. Its distribution is uniform in the square area centered at the origin with the side length as 20. The true mean is $\mathbb{E}[X] = 0$. The mean estimation is based on 100 iid samples.

distribution is uniform in the square area centered at the origin with the side length as 20. The true mean is $\mathbb{E}[X] = 0$. The mean estimation is based on 100 iid samples. As can be seen, although the initial guess of the mean is far away from the true value, the SGD estimate can approach the neighborhood of the true value fast. When the estimate is close to the true value, it exhibits certain randomness but still approaches the true value gradually.

6.4.3 A deterministic formulation of SGD

The formulation of SGD we introduced above involves random variables. One may often encounter a deterministic formulation of SGD without involving any random variables.

Consider a finite set of real numbers $\{x_i\}_{i=1}^n$, where x_i does not have to be a sample of any random variable. The optimization problem to be solved is to minimize the average:

$$\min_w \quad J(w) = \frac{1}{n}\sum_{i=1}^n f(w, x_i),$$

where $f(w, x_i)$ is a parameterized function. Here, w is the parameter to be optimized. The gradient descent algorithm for solving this problem is

$$w_{k+1} = w_k - \alpha_k \nabla_w J(w_k) = w_k - \alpha_k \frac{1}{n}\sum_{i=1}^n \nabla_w f(w_k, x_i). \tag{6.16}$$

Suppose the set is large and we can only fetch a single number every time. In this case, we hope to calculate the optimal solution in an incremental manner. Then, we can use the following iterative algorithm:

$$w_{k+1} = w_k - \alpha_k \nabla_w f(w_k, x_k). \tag{6.17}$$

It must be noted that x_k here is the number fetched at time step k instead of the kth element in the set of $\{x_i\}_{i=1}^n$.

The algorithm in (6.17) is very similar to SGD, but the problem formulation is subtly different because it does not involve any random variables or expected values. Then, many questions arise. For example, is this algorithm SGD? How should we use the finite set of numbers $\{x_i\}_{i=1}^n$? Should we sort these numbers in a certain order and then use them one by one? Or should we randomly sample a number from the set?

A quick answer to the above questions is that, although no random variables are involved in the above formulation, we can introduce a random variable manually and convert the *deterministic formulation* to the *stochastic formulation* of SGD as in (6.10). In particular, suppose X is a random variable defined on the set $\{x_i\}_{i=1}^n$. Suppose its probability distribution is uniform such that $p(X = x_i) = 1/n$. Then, the deterministic optimization problem becomes a stochastic one:

$$\min_w \quad J(w) = \frac{1}{n}\sum_{i=1}^n f(w, x_i) = \mathbb{E}[f(w, X)].$$

The last equality in the above equation is strict instead of approximate. Therefore, the algorithm in (6.17) is SGD and the estimate converges if x_k is *uniformly* and independently sampled from $\{x_i\}_{i=1}^n$. Note that x_k may repeatedly take the same number in $\{x_i\}_{i=1}^n$ since it is sampled randomly.

6.4.4 BGD, SGD, and Mini-batch GD

While we have introduced BGD and SGD, this section introduces mini-batch GD (M-BGD). The three types of gradient descent algorithms are also compared.

Suppose we would like to minimize $J(w) = \mathbb{E}[f(w, X)]$ given a set of random samples $\{x_i\}_{i=1}^n$ of X. The BGD, SGD, MBGD algorithms solving this problem are, respectively,

$$\begin{aligned}
w_{k+1} &= w_k - \alpha_k \frac{1}{n}\sum_{i=1}^n \nabla_w f(w_k, x_i), && \text{(BGD)}\\
w_{k+1} &= w_k - \alpha_k \frac{1}{m}\sum_{j\in\mathcal{I}_k} \nabla_w f(w_k, x_j), && \text{(MBGD)}\\
w_{k+1} &= w_k - \alpha_k \nabla_w f(w_k, \tilde{x}_k). && \text{(SGD)}
\end{aligned}$$

In the BGD algorithm, all the samples are used in every iteration. When n is large, $(1/n)\sum_{i=1}^n \nabla_w f(w_k, x_i)$ is close to the true gradient $\mathbb{E}[\nabla_w f(w_k, X)]$. In the MBGD algorithm, $\mathcal{I}_k$ is a subset of $\{1, \ldots, n\}$ with the size as $|\mathcal{I}_k| = m$. The set $\mathcal{I}_k$ is obtained by

m times idd samplings. In the SGD algorithm, x_k is randomly sampled from $\{x_i\}_{i=1}^n$ at time k.

MBGD can be viewed as an intermediate version between SGD and BGD. Compared to SGD, MBGD has less randomness because it uses more samples instead of just one as in SGD. Compared to BGD, MBGD does not require to use all the samples in every iteration, making it more flexible and efficient. As a result, MBGD well blends the merits of both SGD and BGD while avoiding their shortcomings. If $m = 1$, then MBGD becomes SGD. However, if $m = n$, MBGD does *not* become BGD strictly speaking, because MBGD uses randomly fetched n samples whereas BGD uses all n numbers. In particular, MBGD may use a value in $\{x_i\}_{i=1}^n$ multiple times whereas BGD uses each value once.

The convergence speed of MBGD is faster than SGD in general. That is because SGD uses $\nabla_w f(w_k, x_k)$ to approximate $(1/n)\sum_{i=1}^n \nabla_w f(w_k, x_i)$, whereas MBGD uses $(1/m)\sum_{j\in\mathcal{I}_k} \nabla_w f(w_k, x_j)$, which is more close to the true gradient because the randomness is averaged out. The convergence of the MBGD algorithm can be proved similarly to the SGD case.

A good example to demonstrate the above analysis is the mean estimation problem. In particular, given some numbers $\{x_i\}_{i=1}^n$, our aim is to calculate the mean $\bar{x} = \sum_{i=1}^n x_i/n$. This problem can be equivalently stated as the following optimization problem:

$$\min_w \quad J(w) = \frac{1}{2n}\sum_{i=1}^n \|w - x_i\|^2,$$

whose optimal solution is $w^* = \bar{x}$. The three algorithms for solving this problem are, respectively,

$$w_{k+1} = w_k - \alpha_k \frac{1}{n}\sum_{i=1}^n (w_k - x_i) = w_k - \alpha_k(w_k - \bar{x}), \qquad \text{(BGD)}$$
$$w_{k+1} = w_k - \alpha_k \frac{1}{m}\sum_{j\in\mathcal{I}_k} (w_k - x_j) = w_k - \alpha_k\left(w_k - \bar{x}_k^{(m)}\right), \qquad \text{(MBGD)}$$
$$w_{k+1} = w_k - \alpha_k(w_k - x_k), \qquad \text{(SGD)}$$

where $\bar{x}_k^{(m)} = \sum_{j\in\mathcal{I}_k} x_j/m$. Furthermore, if $\alpha_k = 1/k$, the above equation can be solved as

$$w_{k+1} = \frac{1}{k}\sum_{j=1}^{k} \bar{x} = \bar{x}, \qquad \text{(BGD)}$$

$$w_{k+1} = \frac{1}{k}\sum_{j=1}^{k} \bar{x}_j^{(m)}, \qquad \text{(MBGD)}$$

$$w_{k+1} = \frac{1}{k}\sum_{j=1}^{k} x_j. \qquad \text{(SGD)}$$

The derivation of the above equations is similar to (6.3) and omitted here. It can be seen that the estimate of BGD at each step is exactly the optimal solution $w^* = \bar{x}$. The MBGD approach the mean faster than SGD because $\bar{x}_k^{(m)}$ is already an average.

A simulation example is given in Figure 6.3 to demonstrate the convergence of MBGD. Let $\alpha_k = 1/k$. As can be seen in Figure 6.3, all MBGD algorithms with different mini-batch sizes can converge to the mean. The case with $m = 50$ converges the fastest, while SGD with $m = 1$ is the slowest. This is consistent with the above analysis. Nevertheless, the convergence rate of SGD is still fast especially when w_k is far away from w^*.

6.4.5 Convergence of SGD

We leave the rigorous proof of the convergence of SGD to the last part of this section.

Theorem 6.4 (Convergence of SGD). *In the SGD algorithm in* (6.13), *if*

1) $0 < c_1 \le \nabla_w^2 f(w, X) \le c_2$*;*

2) $\sum_{k=1}^{\infty} a_k = \infty$ *and* $\sum_{k=1}^{\infty} a_k^2 < \infty$*;*

3) $\{x_k\}_{k=1}^{\infty}$ *is iid;*

then w_k *converges to the root of* $\nabla_w \mathbb{E}_X[f(w, X)] = 0$ *with probability 1.*

The idea of the proof is to show that the SGD algorithm is a special RM algorithm. We can also prove in other ways, for example, based on a more fundamental tool of quasimartingales in rigorously probability theory.

Proof of Theorem 6.4

We next show that the SGD algorithm is a special Robbins-Monro algorithm. Then, the convergence of SGD naturally follows from the Robbins-Monro theorem.

The problem to be solved by SGD is to minimize $J(w) = \mathbb{E}_X[f(w, X)]$. This problem can be converted to a root-finding problem: that is to find the root of

$\nabla_w J(w) = \mathbb{E}_X[\nabla_w f(w, X)] = 0$. Let

$$g(w) = \nabla_w J(w) = \mathbb{E}_X[\nabla_w f(w, X)].$$

Then, the aim of SGD is to find the root of $g(w) = 0$. This is exactly the problem solved by the RM algorithm. What we can measure is $\tilde{g}(w, x) = \nabla_w f(w, x)$, where x is a sample of X. Note that $\tilde{g}$ can be rewritten as

$$\begin{aligned}\tilde{g}(w, \eta) &= \nabla_w f(w, x) \\ &= \mathbb{E}_X[\nabla_w f(w, X)] + \underbrace{\nabla_w f(w, x) - \mathbb{E}_X[\nabla_w f(w, X)]}_{\eta(w,x)}.\end{aligned}$$

Then, the RM algorithm for solving $g(w) = 0$ is

$$w_{k+1} = w_k - a_k \tilde{g}(w_k, \eta_k) = w_k - a_k \nabla_w f(w_k, x_k).$$

which is exactly the SGD algorithm. As a result, the SGD algorithm is a special RM algorithm. The convergence of SGD also naturally follows from Theorem 6.1 if the three conditions in Theorem 6.1 are satisfied.

1) Since $\nabla_w g(w) = \nabla_w \mathbb{E}_X[\nabla_w f(w, X)] = \mathbb{E}_X[\nabla_w^2 f(w, X)]$, it follows from $c_1 \leq \nabla_w^2 f(w, X) \leq c_2$ that $c_1 \leq \nabla_w g(w) \leq c_2$. Thus the first condition in Theorem 6.1 is satisfied.

2) The second condition in Theorem 6.1 is the same as the second one in this theorem.

3) The third condition in Theorem 6.1 is $\mathbb{E}[\eta_k|\mathcal{H}_k] = 0$ and $\mathbb{E}[\eta_k^2|\mathcal{H}_k] < \infty$. To see that, since $\{x_k\}$ is idd, we have $\mathbb{E}_{x_k}[\nabla_w f(w, x_k)] = \mathbb{E}_X[\nabla_w f(w, X)]$ for all k. Therefore,

$$\mathbb{E}[\eta_k|\mathcal{H}_k] = \mathbb{E}[\nabla_w f(w_k, x_k) - \mathbb{E}_X[\nabla_w f(w_k, X)]|\mathcal{H}_k].$$

Since $\mathcal{H}_k = \{w_k, w_{k-1}, \dots\}$, the first term is $\mathbb{E}[\nabla_w f(w_k, x_k)|\mathcal{H}_k] = \mathbb{E}_{x_k}[\nabla_w f(w_k, x_k)]$. The second term is $\mathbb{E}[\mathbb{E}_X[\nabla_w f(w_k, X)]|\mathcal{H}_k] = \mathbb{E}_X[\nabla_w f(w_k, X)]$ because $\mathbb{E}_X[\nabla_w f(w_k, X)]$ is a function of w_k. Therefore,

$$\mathbb{E}[\eta_k|\mathcal{H}_k] = \mathbb{E}_{x_k}[\nabla_w f(w_k, x_k)] - \mathbb{E}_X[\nabla_w f(w_k, X)] = 0.$$

Similarly, it can be proved that $\mathbb{E}[\eta_k^2|\mathcal{H}_k] < \infty$ if $|\nabla_w f(w, x)| < \infty$ for all w given any x.

While the three conditions in Theorem 6.1 are satisfied, the convergence of the SGD immediately follows.

6.5 Summary

Instead of introducing new RL algorithms, this chapter introduced the preliminaries to stochastic approximation such as the RM algorithm and the SGD algorithm. Compared to many other root-finding algorithms, the RM algorithm does not require knowing the expression of the objective function or its derivative. It is shown that the SGD algorithm is a special RM algorithm. Moreover, an important problem frequently discussed throughout this chapter is mean estimation. The mean estimation algorithm (6.4) is the first stochastic iterative algorithm we ever introduced in this book. We showed that it is a special SGD algorithm. We will see in the next chapter that the temporal-difference learning algorithms have similar expressions. Finally, the name stochastic approximation was first used by Robbins-Monro's paper in 1951 [6]. A rigorous treatment of stochastic approximation can be found in [7].

6.6 Q&A

– Q: What is stochastic approximation?

  A: Stochastic approximation refers to a broad class of stochastic iterative algorithms solving root finding or optimization problems.

– Q: Why do we need to study stochastic approximation?

  A: That is because the temporal-difference RL algorithms that will be introduced in the next chapter can be viewed as stochastic approximation algorithms. With the knowledge introduced in this chapter, we can be better prepared for the algorithms next chapter. At least, it would not be abrupt for us to see these algorithms for the first time.

– Q: Why do we frequently discuss the mean estimation problem in this chapter?

  A: It is because state value and action value are both defined as means of random variables. In the last chapter, we studied how to approximate means using Monte Carlo methods. In this chapter, we showed that means can be calculated incrementally based on random samples.

– Q: What is the advantage of the RM algorithm compared to other root-finding numerical algorithms?

  A: Compared to many other numerical algorithms for root-finding, the RM algorithm is powerful in the sense that it does not require knowing the expression of the objective function or its derivative. As a result, it is a black-box technique, which only requires knowing the input and output of the objective function. The famous stochastic gradient descent algorithm is a special form of it.

– Q: What is the basic idea of stochastic gradient descent?

  A: Stochastic gradient descent aims to solve optimization problems involving random variables. While the probability distributions of the random variables are not known, s-tochastic gradient descent can solve the optimization problems merely by using samples. Mathematically, it replaces the gradient expressed as an expectation in the gradient descent algorithm with a stochastic gradient.

– Q: Can stochastic gradient descent converge fast?

  A: Stochastic gradient descent has an interesting convergence pattern. That is, if the estimate is far away from the optimal solution, then the convergence is fast. When the estimate is close to the solution, the randomness of the stochastic gradient becomes influential and the convergence rate downgrades.

– Q: What is mini-batch gradient descent? What are its advantages compared to SGD and BGD?

  A: Mini-batch gradient descent can be viewed as an intermediate version between SGD and BGD. Compared to SGD, it has less randomness because it uses more samples instead of just one as in SGD. Compared to BGD, it does not require using all the samples, which is more flexible and efficient.

Chapter 7

Temporal-Difference Learning

This chapter introduces temporal-difference (TD) learning, which is one of the most well-known methods in reinforcement learning (RL). Similar to Monte Carlo learning, TD learning is also model-free, but it has some advantages as we will see. With the preparation in the last chapter, we will see in this chapter that TD learning algorithms can be viewed as special Robbins-Monro (RM) algorithms solving the Bellman or Bellman optimality equation.

7.1 Motivating examples: stochastic algorithms

We next consider some stochastic problems and show how to use the RM algorithm to solve them.

1) First, consider the simple mean estimation problem. That is to calculate

$$w = \mathbb{E}[X],$$

based on some iid samples $\{x\}$ of the random variable X. By writing $g(w) = w - \mathbb{E}[X]$, we can reformulate the problem to a root-finding problem

$$g(w) = 0.$$

Since we can only obtain a sample x of X, the noisy observation we can get is

$$\tilde{g}(w, \eta) = w - x = (w - \mathbb{E}[X]) + (\mathbb{E}[X] - x) \doteq g(w) + \eta.$$

Then, according to the last chapter, we know the RM algorithm for solving $g(w) = 0$ is

$$\begin{aligned} w_{k+1} &= w_k - \alpha_k \tilde{g}(w_k, \eta_k) \\ &= w_k - \alpha_k (w_k - x_k). \end{aligned}$$

It has been proven in the last chapter that w_k converges to $\mathbb{E}[X]$ with probability 1 if $\sum_{k=1}^{\infty} \alpha_k = \infty$ and $\sum_{k=1}^{\infty} \alpha_k^2 < \infty$.

2) Second, we consider a little more complex problem. That is to estimate the mean of a function $v(X)$,

$$w = \mathbb{E}[v(X)],$$

based on some iid random samples $\{x\}$ of X. To solve this problem, we define

$$\begin{aligned} g(w) &= w - \mathbb{E}[v(X)] \\ \tilde{g}(w, \eta) &= w - v(x) = (w - \mathbb{E}[v(X)]) + (\mathbb{E}[v(X)] - v(x)) \doteq g(w) + \eta. \end{aligned}$$

Then, the problem becomes a root-finding problem: $g(w) = 0$. The corresponding RM algorithm is

$$\begin{aligned} w_{k+1} &= w_k - \alpha_k \tilde{g}(w_k, \eta_k) \\ &= w_k - \alpha_k [w_k - v(x_k)]. \end{aligned}$$

3) Third, we consider a more complex problem. That is to calculate

$$w = \mathbb{E}[R + \gamma v(X)],$$

where R, X are random variables, γ is a constant, and $v(\cdot)$ is a function. This problem is more complex than the first two because it involves multiple random variables. We can still apply the RM algorithm in this case. Suppose we can obtain samples $\{x\}$ and $\{r\}$ of X and R. we define

$$\begin{aligned} g(w) &= w - \mathbb{E}[R + \gamma v(X)], \\ \tilde{g}(w, \eta) &= w - [r + \gamma v(x)] \\ &= (w - \mathbb{E}[R + \gamma v(X)]) + (\mathbb{E}[R + \gamma v(X)] - [r + \gamma v(x)]) \\ &\doteq g(w) + \eta. \end{aligned}$$

Then, the problem becomes a root-finding problem: $g(w) = 0$. The corresponding RM algorithm is

$$\begin{aligned} w_{k+1} &= w_k - \alpha_k \tilde{g}(w_k, \eta_k) \\ &= w_k - \alpha_k [w_k - (r_k + \gamma v(x_k))]. \end{aligned}$$

The reader may have noticed that the above three examples become more and more complex. However, they all can be solved by the RM algorithm. In fact, the RM algorithm of the third example already has a similar expression as the TD learning algorithms introduced in the following sections.

7.2 TD learning of state values

TD learning often refers to a broad class of RL algorithms. For example, all the algorithms introduced in this chapter fall into the scope of TD learning. However, in this section TD learning specifically refers to a classic algorithm for estimating state values [3, 13].

7.2.1 Algorithm

Suppose π is a given policy. Our aim is to calculate the state values under π. Recall that the definition of state value is

$$v_\pi(s) = \mathbb{E}\big[R + \gamma G | S = s\big], \quad s \in \mathcal{S} \tag{7.1}$$

where S, R, and G are the random variables representing the current state, immediate reward, and discounted return, respectively. Since

$$\mathbb{E}[G|S = s] = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) v_\pi(s') = \mathbb{E}[v_\pi(S')|S = s],$$

where S' is the random variable representing the next state, we can rewrite (7.1) as

$$v_\pi(s) = \mathbb{E}\big[R + \gamma v_\pi(S')|S = s\big], \quad s \in \mathcal{S}. \tag{7.2}$$

Equation (7.2) is another expression of the Bellman equation. It is sometimes called the *Bellman expectation equation*, an important tool to design and analyze TD algorithms.

To obtain the state values, we need to solve (7.2). What we have is an episode $(s_0, r_1, s_1, \dots, s_t, r_{t+1}, s_{t+1}, \dots)$ generated following the given policy π. Here, t denotes the time step. To solve (7.2) using the episode, the TD learning algorithm is

$$v_{t+1}(s_t) = v_t(s_t) - \alpha_t(s_t)\Big[v_t(s_t) - [r_{t+1} + \gamma v_t(s_{t+1})]\Big], \tag{7.3}$$

$$v_{t+1}(s) = v_t(s), \quad \text{for all } s \neq s_t, \tag{7.4}$$

where $t = 0, 1, 2, \dots$. Here, $v_t(s_t)$ is the estimated state value of $v_\pi(s_t)$; $\alpha_t(s_t)$ is the learning rate of s_t at time t.

It is notable that, at time t, only the value of the visited state s_t is updated whereas the values of the unvisited states $s \neq s_t$ remain unchanged. The update in (7.4) will be omitted in all the algorithms introduced in this chapter without causing any confusion. Nevertheless, this equation should be kept in mind because the algorithm would not be mathematically complete without this equation.

Why does the TD learning algorithm look like this? One may be confused the first time seeing the algorithm. Notably, the TD algorithm in (7.3) has a similar expression to those RM algorithms introduced in the previous section. In fact, the TD algorithm can be derived by applying the RM algorithm to solve the Bellman equation. The details are given in the shaded box below. From this point of view, readers can also better understand the essence of the TD algorithm.

A derivation of the TD algorithm

We next show how the TD algorithm can be obtained by applying the RM algorithm to solve the Bellman equation in (7.2).

In particular, by defining

$$g(v_\pi(s)) = v_\pi(s) - \mathbb{E}\big[R + \gamma v_\pi(S')|s\big],$$

we can rewrite (7.2) as

$$g(v_\pi(s)) = 0.$$

Since we can only obtain the samples r and s' of R and S', the noisy observation we have is

$$\begin{aligned}\tilde{g}(v_\pi(s)) &= v_\pi(s) - \big[r + \gamma v_\pi(s)\big]\\ &= \underbrace{\Big(v_\pi(s) - \mathbb{E}\big[R + \gamma v_\pi(S')|s\big]\Big)}_{g(v_\pi(s))} + \underbrace{\Big(\mathbb{E}\big[R + \gamma v_\pi(S')|s\big] - \big[r + \gamma v_\pi(s)\big]\Big)}_{\eta}.\end{aligned}$$

Therefore, the RM algorithm for solving $g(v_\pi(s)) = 0$ is

$$\begin{aligned}v_{k+1}(s) &= v_k(s) - \alpha_k \tilde{g}(v_k(s))\\ &= v_k(s) - \alpha_k\Big(v_k(s) - \big[r_k + \gamma v_\pi(s'_k)\big]\Big), \qquad k = 1, 2, 3, \dots \end{aligned} \tag{7.5}$$

where $v_k(s)$ is the estimate of $v_\pi(s)$ at the kth step; r_k, s'_k are the samples of R, S' obtained at the kth step.

The RM algorithm in (7.5) has two assumptions that deserve special attention.

1) In order to implement this algorithm, we must repeatedly start from s to obtain the experience set $\{(s, r_k, s'_k)\}$ for $k = 1, 2, 3, \dots$.
2) It is notable that $v_\pi(s'_k)$ on the right-hand side of (7.5) is not $v_k(s'_k)$ because we only estimate $v_\pi(s)$ and assume that $v_\pi(s')$ is already known for any other state s'. Only in this way can the above RM algorithm be strictly valid.

These two assumptions are usually invalid in practice. In particular, we may not be able to sample experience repeatedly starting from s in practice. As a result, we cannot obtain the set $\{(s, r_k, s'_k)\}$. What we usually have are episodes of sequential experiences, where the state s may be visited or sampled once, and then the agent moves to other states. Moreover, we cannot assume that $v_\pi(s')$ is already known for any other state s'. Instead, we need to estimate $v_\pi(s)$ for every $s \in \mathcal{S}$.

To remove the two assumptions in the RM algorithm, we can modify it and then obtain the TD algorithm in (7.3). One modification is that $\{(s, r_k, s'_k)\}$ is changed to $\{(s_t, r_{t+1}, s_{t+1})\}$ so that the algorithm can utilize the sequential samples in an episode rather than fixing on a specific state s. Another modification is that the update of $v_t(s_t)$ depends on $v_t(s_{t+1})$ rather than $v_\pi(s_{t+1})$ because $v_\pi(s_{t+1})$ is also to be estimated. Whether such an update can ensure convergence? The answer is yes and will be analyzed later.

7.2.2 Properties

We next discuss some important properties of the TD algorithm.

First, we examine the expression of the TD algorithm more closely. In particular, (7.3) can be annotated as

$$\underbrace{v_{t+1}(s_t)}_{\text{new estimate}} = \underbrace{v_t(s_t)}_{\text{current estimate}} -\alpha_t(s_t)\Big[\overbrace{v_t(s_t) - [\underbrace{r_{t+1} + \gamma v_t(s_{t+1})}_{\text{TD target } \bar{v}_t}]}^{\text{TD error } \delta_t}\Big], \tag{7.6}$$

where

$$\bar{v}_t \doteq r_{t+1} + \gamma v(s_{t+1})$$

is called the *TD target* and

$$\delta_t \doteq v(s_t) - [r_{t+1} + \gamma v(s_{t+1})] = v(s_t) - \bar{v}_t$$

is called the *TD error*. It is clear that the new estimate $v_{t+1}(s_t)$ is a combination of the current estimate $v_t(s_t)$ and the TD error.

– What is the interpretation of the TD error? The TD error should be zero in the expectation sense. That is simply because

$$\begin{aligned}
\mathbb{E}[\delta_t|S_t = s_t] &= \mathbb{E}\big[v_\pi(S_t) - [R_{t+1} + \gamma v_\pi(S_{t+1})]|S_t = s_t\big] \\
&= v_\pi(s_t) - \mathbb{E}\big[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s_t\big] \\
&= 0.
\end{aligned}$$

Therefore, the TD error reflects the deficiency between the current estimate v_t and the true state value v_π.

In a more abstract level, the TD error can be interpreted as *innovation*, which means new information obtained from the experience (s_t, r_{t+1}, s_{t+1}). The fundamental idea of TD learning is to correct our current understanding of the state value based on the new information obtained. Innovation is a fundamental concept in many estimation problems such as Kalman filtering.

– Why is $\bar{v}_t$ called the TD target? That is because $\bar{v}_t$ is a *target value* that the algorithm attempts to drive $v(s_t)$ to approach. To see that, we consider a simple case where α and $\bar{v}$ are both constant. Then, (7.6) becomes

$$v_{t+1} = v_t - \alpha(v_t - \bar{v}) = (1-\alpha)v_t + \alpha\bar{v}.$$

It can be shown that $v_t \to \bar{v}$ as $t \to \infty$ if $0 < \alpha < 1$. To prove that, let $\delta_t = v_t - \bar{v}$. Then the above equation becomes $\delta_{t+1} = (1-\alpha)\delta_t$. Hence, we have $\delta_{t+1} = (1-\alpha)\delta_t = (1-\alpha)^2\delta_{t-1} = \cdots = (1-\alpha)^t\delta_1$. Since $(1-\alpha)^t \to 0$ as $t \to \infty$, we know $\delta_t \to 0$ and hence $v_t \to \bar{v}$ as $t \to \infty$. Therefore, $\bar{v}$ is the target value that v_t will converge to.

Second, the TD algorithm in (7.3) can only estimate the state value of a given policy. To find optimal policies, we still need to further calculate the action values and then do policy improvement. In fact, when the system model is unavailable, we should directly estimate action values. Nevertheless, the TD algorithm introduced in this section is very basic and important for understanding other algorithms introduced in the rest of the chapter.

Third, while TD learning and MC learning are both model-free, what are the advantages and disadvantages of TD learning compared to MC learning? The answers are summarized in Table (7.1).

7.2.3 Convergence

Although the TD algorithm can be viewed as an RM algorithm solving the Bellman equation, it is still necessary to give a rigorous convergence analysis.

Theorem 7.1 (Convergence of TD Learning)**.** *By the TD algorithm* (7.3)*, $v_t(s)$ converges with probability 1 to $v_\pi(s)$ for all $s \in \mathcal{S}$ as $t \to \infty$ if $\sum_t \alpha_t(s) = \infty$ and $\sum_t \alpha_t^2(s) < \infty$ for all $s \in \mathcal{S}$.*

It should be noted that the condition of $\sum_t \alpha_t(s) = \infty$ and $\sum_t \alpha_t^2(s) < \infty$ should be valid for all $s \in \mathcal{S}$. That requires every state must be visited an infinite (or sufficiently many) number of times. At time step t, if $s = s_t$ which means that s is visited at time t, then $\alpha_t(s) > 0$; otherwise, $\alpha_t(s) = 0$ for all the other $s \neq s_t$.

In addition, the learning rate α is often selected as a small constant. In this case, the condition that $\sum_t \alpha_t^2(s) < \infty$ is invalid anymore. When α is constant, it can still be shown that the algorithm converges in the sense of expectation sense.

TD/Sarsa learning	MC learning
Online: TD learning is online. It can update the state/action values immediately after receiving a reward.	**Offline:** MC learning is offline. It has to wait until an episode has been completely collected. That is because it must calculate the discounted return from a state-action pair to the end of the episode.
Continuing tasks: Since TD learning is online, it can handle both episodic and continuing tasks. Continuing tasks may not have a terminal state.	**Episodic tasks:** Since MC learning is offline, it can only handle episodic tasks where the episodes terminate after a finite number of steps.
Bootstrapping: TD bootstraps because the update of a state/action value relies on the previous estimate of this value. As a result, TD requires an initial guess of the values.	**Non-bootstrapping:** MC is not bootstrapping, because it can directly estimate state/action values without any initial guess.
Low estimation variance: The estimation variance of TD is lower than MC because there are fewer random variables. For instance, when updating an action value, Sarsa merely requires samples of three random variables: $R_{t+1}, S_{t+1}, A_{t+1}$.	**High estimation variance:** The estimation variance of MC is higher since many random variables are involved. For example, to estimate $q_\pi(s_t, a_t)$, we need samples of $R_{t+1}+\gamma R_{t+2}+\gamma^2 R_{t+3}+\ldots$. Suppose the length of each episode is L. Assume each state has the same number of actions as $\lvert\mathcal{A}\rvert$. Then there are $\lvert\mathcal{A}\rvert^L$ possible episodes in total following a soft policy. If we merely use a few episodes to estimate, it is not surprising that the estimation variance is high though the bias is zero.

Table 7.1: Comparison between TD learning and MC learning.

Proof of Theorem 7.1

We prove the convergence based on Theorem 6.3 given in the last chapter. To do that, we need first to construct a process like the one in Theorem 6.3.

Consider an arbitrary state $s \in \mathcal{S}$. Let k be the times s has been visited. Note that k is different from t. Here, t is the time step when sampling the experiences (s_t, r_{t+1}, s_{t+1}). It represents the total times that all the states are visited, whereas k is the times that s is visited. Every time s is visited, $k \to k+1$.

The algorithm (7.3) implies the following process:

$$\begin{aligned} v_{k+1}(s) &= v_k(s) - \alpha_k(s)[v_k(s) - (r_{k+1} + \gamma v_k(s'))] \\ &= (1-\alpha_k(s))v_k(s) + \alpha_k(s)(r_{k+1} + \gamma v_k(s')), \qquad k = 1,2,\ldots \end{aligned} \tag{7.7}$$

where s' is the next state transited from s. We next show that the three conditions in Theorem 6.3 are satisfied.

First, the convergence of (7.7) requires $\sum_k \alpha_k(s) = \infty$ and $\sum_k \alpha_k^2(s) < \infty$ according to Theorem 6.3. This condition is tricky in the context of multiple states. It must be noted that this condition must be satisfied for each state, which means every state must be visited an infinite number of times.

Second, define the estimation error as $\Delta_k(s) = v_k(s) - v_\pi(s)$, where $v_\pi(s)$ is the state value of s under policy π. Then, deducting $v_\pi(s)$ on both sides of (7.7) gives

$$\Delta_{k+1}(s) = (1-\alpha_k(s))\Delta_k(s) + \alpha_k(s)[\underbrace{r_{k+1} + \gamma v_k(s_{k+1}) - v_\pi(s)}_{e_k}], \qquad k = 1,2,\ldots$$

To apply Theorem 6.3, we need to show that $\|\mathbb{E}[e_k(s)|\mathcal{H}_k]\|_\infty \leq \gamma\|\Delta_k(s)\|_\infty$, where $\mathcal{H}_k = \{\Delta_k, \Delta_{k-1}, \ldots, e_{k-1}, \ldots, \alpha_{k-1}, \ldots\}$. To see that, we have

$$\begin{aligned} \|\mathbb{E}[e_k(s)|\mathcal{H}_k]\|_\infty = \left\| \begin{matrix} \vdots \\ \mathbb{E}[e_k(s)|\mathcal{H}_k] \\ \vdots \end{matrix} \right\|_\infty &= \|r_\pi + \gamma P_\pi v_k - v_\pi\|_\infty \\ &= \|r_\pi + \gamma P_\pi v_k - (r_\pi + \gamma P_\pi v_\pi)\|_\infty \\ &= \gamma\, \|P_\pi v_k - P_\pi v_\pi\|_\infty \\ &\leq \gamma\, \|v_k - v_\pi\|_\infty \\ &= \gamma \left\| \begin{matrix} \vdots \\ \Delta_k(s) \\ \vdots \end{matrix} \right\|_\infty = \gamma\|\Delta_k(s)\|_\infty. \end{aligned}$$

The symbol $\|\cdot\|_\infty$ is abused a little here: It stands for the maximum norm calculated over all states and, in the meantime, the L_∞ norm of vectors. Since the discount rate $\gamma \in (0,1)$, condition 3 in the theorem is satisfied.

Third, regarding the variance, we have $\text{Var}[e_k|\mathcal{H}_k] = \text{Var}[r_{k+1} + \gamma v_k(s_{k+1}) - v_\pi(s)|\mathcal{H}_k] = \text{Var}[r_{k+1} + \gamma v_k(s_{k+1})|\mathcal{H}_k]$. Since r_{k+1} is bounded, condition 4 can be proved without difficulties.

The above proof is essentially the same as [11].

7.3 TD learning of action values: Sarsa

The TD algorithm introduced in the last section can only estimate state values. In this section, we introduce, Sarsa, an algorithm that can directly estimate action values. Estimating action values is important because the policy can be improved based on action values.

7.3.1 Algorithm

Given a policy π, our aim is to estimate the action values of π. Suppose we have an episode of experiences generated following π: $(s_0, a_0, r_1, s_1, a_1, \dots, s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, \dots)$. Based on the episode, we can use the following *Sarsa* algorithm to estimate the action values:

$$\begin{aligned} q_{t+1}(s_t, a_t) &= q_t(s_t, a_t) - \alpha_t(s_t, a_t)\Big[q_t(s_t, a_t) - [r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1})]\Big], \qquad (7.8)\\ q_{t+1}(s, a) &= q_t(s, a), \quad \text{for all } (s, a) \neq (s_t, a_t), \end{aligned}$$

where $t = 0, 1, 2, \dots$. Here, $q_t(s_t, a_t)$ is the estimated action value of (s_t, a_t); $\alpha_t(s_t, a_t)$ is the learning rate depending on s_t, a_t. At time step t, only the state-action (s_t, a_t) which is being visited is updated. The estimates of all the other state-action pairs remain the same.

Some properties of Sarsa are discussed as follows.

– Why is this algorithm called Sarsa? That is because each step of the algorithm involves $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. Sarsa is the abbreviation of state-action-reward-state-action. The algorithm of Sarsa was first proposed in [14] and the name of Sarsa was due to [3].

– Why is Sarsa designed in this way? One may have noticed that the expression of Sarsa is very similar to the TD algorithm introduced in the last section. We can obtain Sarsa by replacing the state value estimate $v(s)$ in the TD algorithm with the action value estimate $q(s, a)$. As a result, Sarsa is nothing but an action-value version of the TD algorithm.

– What does the Sarsa algorithm do mathematically? Similar to the analysis of the TD learning in the last section, we know that Sarsa is a stochastic approximation algorithm solving the following equation:

$$q_\pi(s, a) = \mathbb{E}\left[R + \gamma q_\pi(S', A')|s, a\right], \quad \text{for all } s, a. \qquad (7.9)$$

Equation (7.13) is another expression of the Bellman equation expressed in terms of action values. The proof is given in the shaded box below.

Show that (7.13) **is the Bellman equation**

First of all, the Bellman equation expressed in terms of action values is

$$q_\pi(s,a) = \sum_r rp(r|s,a) + \gamma \sum_{s'} \sum_{a'} q_\pi(s',a') p(s'|s,a) \pi(a'|s') \tag{7.10}$$

$$= \sum_r rp(r|s,a) + \gamma \sum_{s'} p(s'|s,a) \sum_{a'} q_\pi(s',a') \pi(a'|s'). \tag{7.11}$$

This expression has been given in Section 2.6.2. It describes the relationship among the values of all the actions. Since

$$\begin{aligned} p(s',a'|s,a) &= p(s'|s,a)p(a'|s',s,a) \\ &= p(s'|s,a)p(a'|s') \quad \text{(due to conditional independence)} \\ &\doteq p(s'|s,a)\pi(a'|s'), \end{aligned}$$

(7.10) can be rewritten as

$$q_\pi(s,a) = \sum_r rp(r|s,a) + \gamma \sum_{s'} \sum_{a'} q_\pi(s',a') p(s',a'|s,a).$$

By the definition of expected value, the above equation is equivalent to (7.13). Hence, (7.13) is the Bellman equation.

– Since Sarsa is the action-value version of the TD algorithm in the last section, its convergence also naturally follows. The convergence analysis is similar to Theorem 7.1 and omitted here. The convergence result is summarized below.

Theorem 7.2 (Convergence of Sarsa learning). *By the Sarsa algorithm in* (7.8), $q_t(s,a)$ *converges with probability 1 to the action value* $q_\pi(s,a)$ *as* $t \to \infty$ *for all* (s,a) *if* $\sum_t \alpha_t(s,a) = \infty$ *and* $\sum_t \alpha_t^2(s,a) < \infty$ *for all* (s,a).

It should be noted that the condition of $\sum_t \alpha_t(s,a) = \infty$ and $\sum_t \alpha_t^2(s,a) < \infty$ should be valid for all (s,a). That requires every state-action pair must be visited an infinite (or sufficiently many) number of times. At time step t, if $(s,a) = (s_t,a_t)$, then $\alpha_t(s,a) > 0$; otherwise, $\alpha_t(s,a) = 0$ for all the other $(s,a) \neq (s_t,a_t)$.

7.3.2 Implementation

The Sarsa algorithm in (7.8) can only estimate the action values of a given policy. The ultimate goal of RL is to find optimal policies. To do that, inspired by the model-based policy iteration algorithm, we can combine Sarsa with a policy improvement step to search for optimal policies. Such a combination for optimal policy search is also often called Sarsa when the context is clear. As shown in the pseudocode, each iteration has two steps. The first is to update the q-value of the visited state-action pair. The second is to update the policy as an ϵ-greedy one. Two points deserve special attention.

In the q-value update step, unlike the model-based policy iteration or value iteration algorithm where the values of all states are updated in each iteration, Sarsa only updates a single state-action pair that is visited at time step t. After that, the policy of s_t is updated immediately. This is based on the idea of generalized policy iteration. That is, although the q-value estimated based on the experience obtained in a single step is not sufficiently accurate, we do not need to wait until a sufficiently accurate q-value can be estimated before updating the policy. Moreover, after the policy is updated, the policy is immediately used to generate the next experience. To ensure all the state-action pairs can be visited, the policy is ϵ-greedy instead of greedy.

A simulation example is shown in Figure 7.1 to demonstrate the Sarsa algorithm. It should be noted that the task here is not to find the optimal policies for all states. Instead, it aims to find a good path from a specific starting state to a target state. This task is often encountered in practice because in many cases both the starting state and target state are fixed and we only need to find a good path connecting them. This task is also relatively simple because we do not need to explore all the states. However, it is still necessary in theory to explore all states to ensure the searched path is optimal. Nevertheless, without exploring all the states, the algorithm can usually generate a good path even though its optimality is not ensured.

The simulation setup and results shown in Figure 7.1 are elaborated below.

- In this example, all the episodes start from the top left state and end in the target state. The reward setting is $r_{\text{target}} = 0$, $r_{\text{forbidden}} = r_{\text{boundary}} = -10$, and $r_{\text{other}} = -1$. The learning rate is $\alpha = 0.1$ and the value of ϵ is 0.1. The initial guess of the action values are selected as $q_0(s, a) = 0$ for all s, a. The initial policy obeys a uniform distribution: $\pi_0(a|s) = 0.2$ for all s, a.
- The left figure in Figure 7.1(a) shows the final policy obtained by Sarsa. As can be seen, this policy can successfully reach the target state from the starting state. However, the policies for other states may not be optimal. That is because the other states are not of interest and every episode ends when the target state is reached. If we need to find the optimal policies for all states, we should ensure all the states are visited sufficiently many times by, for example, starting episodes from different states.
- The right-top subfigure in Figure 7.1(a) shows the total reward collected by each episode. Here, the total reward is the non-discounted sum of all immediate rewards obtained along an episode. As can be seen, the total reward of each episode increases gradually. That is simply because the initial policy is not good and hence negative rewards are frequently obtained. As the policy becomes better, the rewards increase

Pseudocode: Policy searching by Sarsa

Initialization: Initial $q_0(s,a)$ and $\pi_0(a|s)$ for all s,a. Learning rate $\alpha_t(s_t,a_t)$ is selected as a small positive constant for all t. Small $\epsilon > 0$.
Aim: Search a good policy that can lead the agent to a target state from an initial state-action pair (s_0,a_0).

```
For each episode, do
    If the current $s_t$ is not the target state, do
        Collect the experience $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$: In particular, take action $a_t$ following $\pi_t(s_t)$, generate $r_{t+1}, s_{t+1}$, and then take action $a_{t+1}$ following $\pi_t(s_{t+1})$.
        Update q-value:
            $q_{t+1}(s_t,a_t) = q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - [r_{t+1} + \gamma q_t(s_{t+1},a_{t+1})]\Big]$
        Update policy:
            $\pi_{t+1}(a|s_t) = 1 - \frac{\epsilon}{|\mathcal{A}(s)|}(|\mathcal{A}(s)| - 1)$ if $a = \arg\max_a q_{t+1}(s_t,a)$
            $\pi_{t+1}(a|s_t) = \frac{\epsilon}{|\mathcal{A}(s)|}$ otherwise
```

accordingly.

– The right-bottom subfigure in Figure 7.1(a) shows that the length of each episode drops gradually. That is because the initial policy is not good and may take many detours before reaching the target. As the policy becomes better, the length of the trajectory becomes shorter.

7.4 TD learning of action values: Expected Sarsa

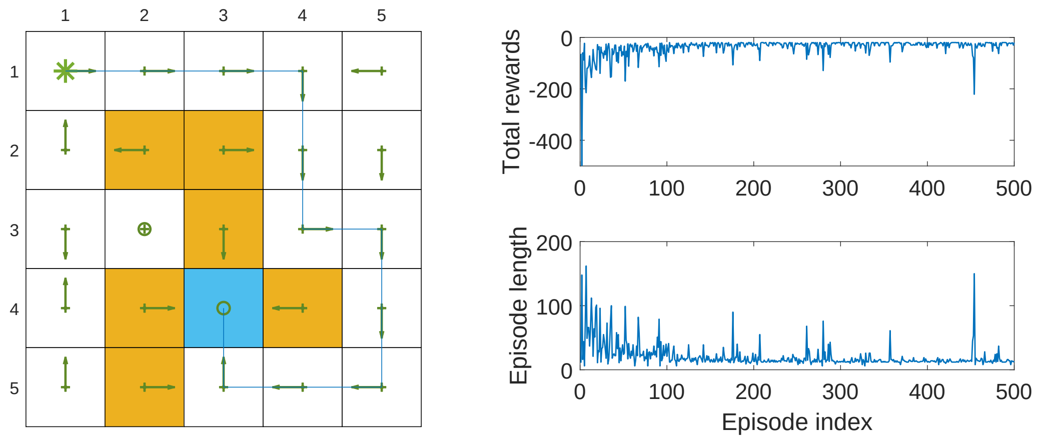
A variant of Sarsa is the *Expected Sarsa* algorithm:

$$\begin{aligned} q_{t+1}(s_t,a_t) &= q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - (r_{t+1} + \gamma \mathbb{E}[q_t(s_{t+1},A)|s_{t+1}])\Big], \\ q_{t+1}(s,a) &= q_t(s,a), \quad \text{for all } (s,a) \neq (s_t,a_t), \end{aligned} \tag{7.12}$$

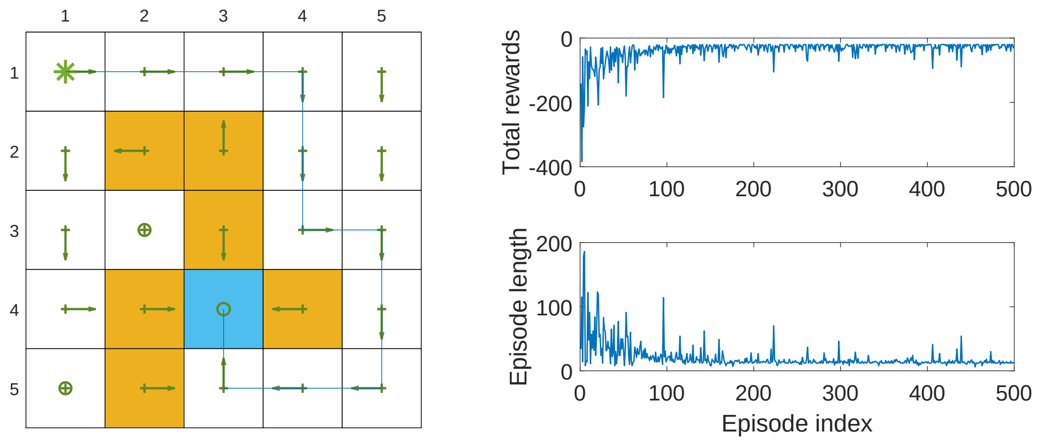
where

$$\mathbb{E}[q_t(s_{t+1},A)|s_{t+1}]) = \sum_a \pi_t(a|s_{t+1}) q_t(s_{t+1},a) \doteq v_t(s_{t+1})$$

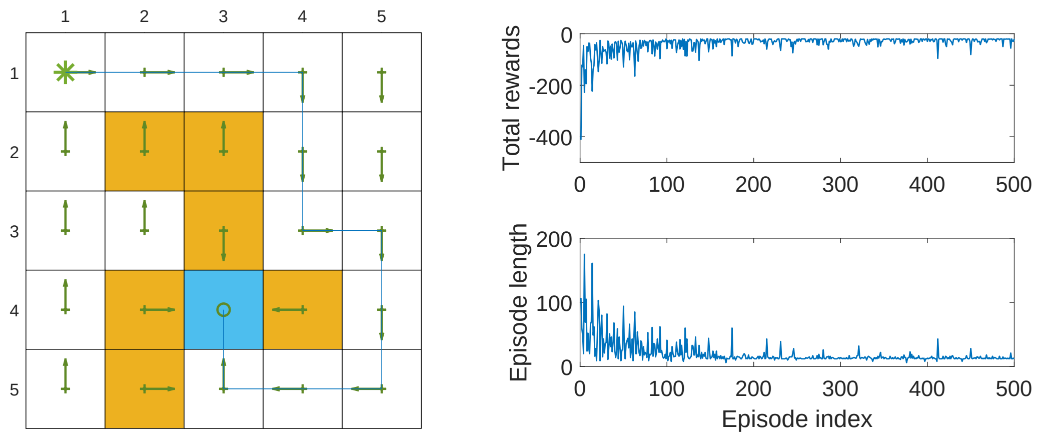
is the expected value of $q_t(s_{t+1},a)$ under policy π_t. The expression of the Expected Sarsa algorithm is very similar to that of Sarsa. They are different only in terms of the TD target. In particular, the TD target in Expected Sarsa is $r_{t+1} + \gamma\mathbb{E}[q_t(s_{t+1},A)|s_{t+1}]$ while that of Sarsa is $r_{t+1} + \gamma q_t(s_{t+1},a_{t+1})$. Since the algorithm involves an expected value, it is named Expected Sarsa. Although calculating the expected value may increase the computational complexity a little, it is beneficial in the sense that it reduces the estimation



(a) Sarsa



(b) Expected Sarsa



(c) Q-learning

Figure 7.1: Examples to demonstrate Sarsa, Expected Sarsa, and Q-learning. In each example, the episodes start from the top-left state and end in the target state. The aim is to find a good path from the starting state to the target state. The reward setting is $r_{\text{target}} = 0$, $r_{\text{forbidden}} = r_{\text{boundary}} = -10$, and $r_{\text{other}} = -1$. The learning rate is $\alpha = 0.1$ and the value of ϵ is 0.1. The left figures above show the final policy obtained by different algorithms. The right figures show the total reward and length of every episode.

variances because it reduces random variables in Sarsa from $\{s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}\}$ to $\{s_t, a_t, r_{t+1}, s_{t+1}\}$. As a result, Expected Sarsa performs generally better than Sarsa. The simulation result of Expected Sarsa is given in Figure 7.1(b).

Similar to the analysis of the TD learning algorithm in Section 7.2.1, it can be seen that Expected Sarsa is a stochastic approximation algorithm for solving the following equation:

$$q_\pi(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma \mathbb{E}[q_\pi(S_{t+1}, A_{t+1})|S_{t+1}]\Big|S_t = s, A_t = a\Big], \quad \text{for all } s, a. \tag{7.13}$$

The above equation may look strange at the first glance. In fact, it is another expression of the Bellman equation. To see that, since

$$\mathbb{E}[q_\pi(S_{t+1}, A_{t+1})|S_{t+1}] = \sum_{A'} q_\pi(S_{t+1}, A')\pi(A'|S_{t+1}) = v_\pi(S_{t+1}),$$

substituting the above equation into (7.13) gives

$$q_\pi(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a\Big],$$

which is clearly the Bellman equation.

The implementation of Expected Sarsa is the same as Sarsa, except for the q-value update step. Details are omitted here.

7.5 TD learning of action values: n-step Sarsa

This section introduces n-step Sarsa, another extension of Sarsa. It will be shown that Sarsa and MC learning are two extreme cases of n-step Sarsa.

First, the definition of action value is

$$q_\pi(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a].$$

The discounted return G_t can be written in different forms as

$$\begin{aligned}
\text{Sarsa} \longleftarrow \quad G_t^{(1)} &= R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}),\\
G_t^{(2)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, A_{t+2}),\\
&\vdots\\
n\text{-step Sarsa} \longleftarrow \quad G_t^{(n)} &= R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^n q_\pi(S_{t+n}, A_{t+n}),\\
&\vdots\\
\text{MC} \longleftarrow \quad G_t^{(\infty)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots
\end{aligned}$$

It should be noted that $G_t = G_t^{(1)} = G_t^{(2)} = G_t^{(n)} = G_t^{(\infty)}$, where the superscripts merely indicate the different decomposition structures of G_t.

Sarsa aims to solve

$$q_\pi(s,a) = \mathbb{E}[G_t^{(1)}|s,a] = \mathbb{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|s,a].$$

MC learning aims to solve

$$q_\pi(s,a) = \mathbb{E}[G_t^{(\infty)}|s,a] = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots |s,a].$$

An intermediate algorithm called *n-step Sarsa* aims to solve

$$q_\pi(s,a) = \mathbb{E}[G_t^{(n)}|s,a] = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \dots + \gamma^n q_\pi(S_{t+n}, A_{t+n})|s,a].$$

The algorithm of n-step Sarsa is

$$\begin{aligned} q_{t+1}(s_t,a_t) &= q_t(s_t,a_t) \\ &\quad - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - [r_{t+1} + \gamma r_{t+2} + \dots + \gamma^n q_t(s_{t+n}, a_{t+n})]\Big]. \end{aligned} \tag{7.14}$$

n-step Sarsa is more general because it becomes the (one-step) Sarsa algorithm when $n = 1$ and the MC learning algorithm when $n = \infty$.

To implement (7.14), we need the experience $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, \dots, r_{t+n}, s_{t+n}, a_{t+n})$. Since $(r_{t+n}, s_{t+n}, a_{t+n})$ has not been collected at time t, we are not able to implement (7.14) directly in practice. However, we can wait until time $t + n$ to update the q-value of (s_t, a_t). To that end, (7.14) can be modified to

$$\begin{aligned} q_{t+n}(s_t,a_t) &= q_{t+n-1}(s_t,a_t) \\ &\quad - \alpha_{t+n-1}(s_t,a_t)\Big[q_{t+n-1}(s_t,a_t) - [r_{t+1} + \gamma r_{t+2} + \dots + \gamma^n q_{t+n-1}(s_{t+n}, a_{t+n})]\Big], \end{aligned}$$

where $q_{t+n}(s_t,a_t)$ is the estimate of $q_\pi(s_t,a_t)$ at time $t+n$.

Since n-step Sarsa includes Sarsa and MC learning as two extreme cases, it is not surprising that the performance of n-step Sarsa is a blend of Sarsa and MC learning as well. Specifically, if n is selected to be a large number, its performance is close to MC learning and hence has a large variance but a small bias. If n is selected to be small, its performance is close to Sarsa and hence has a relatively large bias due to the initial guess and relatively low variance. A complete treatment of multi-step temporal-difference learning can be found in [3, Chapter 7]. Finally, n-step Sarsa is also for policy evaluation. It can be combined with the policy improvement step to search for optimal policies. The implementation is similar to Sarsa and omitted here.

7.6 TD learning of optimal action values: Q-learning

In this section, we introduce the Q-learning algorithm [15, 16], one of the most widely used RL algorithms. It should be noted that Sarsa can only estimate the action values of a given policy. It must be combined with a policy improvement step to find optimal policies and hence their optimal action values. By contrast, Q-learning can directly estimate optimal action values.

7.6.1 Algorithm

We first give the expression of the Q-learning algorithm and then analyze it in detail. The Q-learning algorithm is

$$\begin{aligned} q_{t+1}(s_t,a_t) &= q_t(s_t,a_t) - \alpha_t(s_t,a_t)\left[q_t(s_t,a_t) - [r_{t+1} + \gamma \max_{a\in\mathcal{A}(s_{t+1})} q_t(s_{t+1},a)]\right], \qquad (7.15)\\ q_{t+1}(s,a) &= q_t(s,a), \quad \text{for all } (s,a) \neq (s_t,a_t), \end{aligned}$$

where $t = 0, 1, 2, \ldots$. Here, $q_t(s_t, a_t)$ is the estimated action value of (s_t, a_t) and $\alpha_t(s_t, a_t)$ is the learning rate depending on s_t, a_t.

The expression of Q-learning is very similar to Sarsa. They are different only in terms of the TD target: The TD target in Q-learning is $r_{t+1} + \gamma \max_{a\in\mathcal{A}(s_{t+1})} q_t(s_{t+1}, a)$ while that of Sarsa is $r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1})$.

Why is Q-learning designed in its form and what does it do mathematically? Similar to the analysis of the TD learning algorithm in Section 7.2.1, it can be seen from the expression of Q-learning that it is a stochastic approximation algorithm for solving the action values from the following equation:

$$q(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_a q(S_{t+1},a)\Big|S_t = s, A_t = a\right], \quad \text{for all } s, a. \qquad (7.16)$$

This equation actually is the Bellman optimality equation expressed in terms of action values. Interested readers can check the proof given in the shaded box below. The convergence analysis of Q-learning is similar to Theorem 7.1 and omitted here. Detailed proofs can be found in [11, 16].

Show that (7.16) is the Bellman optimality equation

By the definition of expectation, (7.16) can be rewritten as

$$q(s,a) = \sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) \max_{a\in\mathcal{A}(s')} q(s',a).$$

Taking maximum on both sides of the equation gives

$$\max_{a\in\mathcal{A}(s)} q(s,a) = \max_{a\in\mathcal{A}(s)} \left[\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) \max_{a\in\mathcal{A}(s')} q(s',a)\right].$$

Denote $v(s) \doteq \max_{a\in\mathcal{A}(s)} q(s,a)$. Then the above equation becomes

$$\begin{aligned} v(s) &= \max_{a\in\mathcal{A}(s)} \left[\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a)v(s')\right] \\ &= \max_\pi \sum_{a\in\mathcal{A}(s)} \pi(a|s) \left[\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a)v(s')\right], \end{aligned}$$

which is clearly the Bellman optimality equation.

7.6.2 Off-policy vs on-policy

Before further discussing the properties and implementation of Q-learning, we first introduce two important concepts: *on-policy learning* and *off-policy learning.*

There exist two policies in a TD learning task: *behavior policy* and *target policy.* The behavior policy is used to generate experience samples. The target policy is constantly updated toward an optimal policy. When the behavior policy is the same as the target policy, such a kind of learning is called on-policy. Otherwise, when they are different, the learning is called off-policy. Off-policy learning is more general than on-policy because off-policy becomes on-policy when the behavior policy and target policy are the same.

The advantage of off-policy learning compared to on-policy learning is that it can search for optimal policies based on the experiences generated by any other policies, which may be a policy designed by other non-learning approaches or a policy executed by a human operator. As an important special case, the behavior policy can be selected to be *exploratory.* For example, if we would like to estimate the action values of all state-action pairs, we must generate episodes visiting every state-action pair sufficiently many times. In this case, exploratory behavior policies are favorable. By contract, although Sarsa uses ϵ-greedy policies to maintain certain exploration ability, the value of ϵ is usually small and hence the exploration ability is limited. In this case, a large number of episodes are required to ensure all the state-action pairs are visited, lowering the sample efficiency. To overcome this limitation, we can use policies with a strong exploration ability to generate episodes and then use off-policy learning to learn optimal policies. A representative explanatory policy is the uniform policy that selects any action at a state with the same probability.

Another concept that may be confused with on-policy/off-policy is *online/offline* *learning.* Online learning refers to the case where the value and policy can be updated once an experience sample is obtained. Offline learning refers to the case that the update can only be done after all experience samples have been collected. For example, TD learning is online, whereas MC learning is offline. An on-policy learning algorithm such as Sarsa must work online because the updated policy must be used to generate new experience samples. An off-policy learning algorithm such as Q-learning can work either online or offline. It can either update the value and policy upon receiving an experience sample or update after collecting all experience samples.

7.6.3 Q-learning is off-policy

All the TD learning algorithms introduced in this chapter are on-policy, except Q-learning.

1) Sarsa is on-policy. That is because the target policy is used to generate new experience samples once it has been updated. Hence, the behavior policy is the same as the target policy. The on-policy attribute of Sarsa can be clearly seen from the mathematical expression of the algorithm. In particular, the TD target in Sarsa is

$$\bar{q}_t = r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1}).$$

Here, a_{t+1} is generated following $\pi_t(s_{t+1})$, which is both the target and behavior policy.

2) Expected Sarsa is also on-policy. It can be clearly seen from its TD target

$$\bar{q}_t = r_{t+1} + \gamma \sum_a \pi_t(a|s_{t+1}) q_t(s_{t+1}, a),$$

where $\pi_t(s_{t+1})$ is used in the calculation.

3) MC learning is also on-policy because it aims to estimate the action values of the policy used to generate the episodes.

4) Q-learning is off-policy. That means the behavior policy can be different from the target policy. This property can also be seen from the TD target of Q-learning:

$$\bar{q}_t = r_{t+1} + \gamma \max_a q_t(s_{t+1}, a).$$

The calculation of this TD target does not require any policy.

Whether an algorithm is on-policy or off-policy is determined by the fundamental properties of the algorithm rather than how it is implemented. The fundamental reason why Q-learning is off-policy is that it aims to solve the *Bellman optimality equation.* Since solving the Bellman optimality equation does not depend on any specific policy, it can use experience samples generated by any other policies. By contrast, the fundamental reason why other TD algorithms are on-policy is that they aim to solve the *Bellman equation of* *a given policy*. Since the Bellman equation depends on a given policy, solving it requires the episodes generated by the given policy.

7.6.4 Implementation

Since Q-learning is off-policy, it can be implemented in an off-policy or on-policy fashion. The pseudocode of both the on-policy version and the off-policy version of Q-learning is given in the following boxes.

The on-policy version is the same as Sarsa except for the TD target in the q-value update step. In this case, the behavior policy is the same as the target policy, which is an ϵ-greedy policy.

In the off-policy version, the episode is generated by the behavior policy π_b which can be any policy. If we would like to sample experiences for every state-action pair, π_b should be selected to be exploratory. Here, the target policy π_T is greedy rather than ϵ-greedy. That is because the target policy is not used to generate episodes and hence is not required to be exploratory. Moreover, the off-policy version of Q-learning presented here is offline. That is because all the experience samples are collected first and then processed to estimate an optimal target policy. Of course, it can be modified to become online. Once a sample is collected, the sample can be used to update the q-value and target policy immediately. Nevertheless, the updated target policy is not used to generate new samples.

Pseudocode: Policy searching by Q-learning (on-policy version)

Initialization: Initial $q_0(s,a)$ and $\pi_0(a|s)$ for all s,a. Learning rate $\alpha_t(s_t,a_t)$ is selected as a small positive constant for all t. Small $\epsilon > 0$.
Aim: Search a good policy that can lead the agent to a target state from an initial state-action pair (s_0,a_0).

```
For each episode, do
    If the current $s_t$ is not the target state, do
        Collect the experience $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$: In particular, take action $a_t$ following $\pi_t(s_t)$, generate $r_{t+1}, s_{t+1}$, and then take action $a_{t+1}$ following $\pi_t(s_{t+1})$.
        Update q-value:
            $q_{t+1}(s_t,a_t) = q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - [r_{t+1} + \gamma \max_a q_t(s_{t+1},a)]\Big]$
        Update policy:
            $\pi_{t+1}(a|s_t) = 1 - \frac{\epsilon}{|\mathcal{A}(s)|}(|\mathcal{A}(s)| - 1)$ if $a = \arg\max_a q_{t+1}(s_t,a)$
            $\pi_{t+1}(a|s_t) = \frac{\epsilon}{|\mathcal{A}(s)|}$ otherwise
```

Pseudocode: Optimal policy search by Q-learning (off-policy version)

```
Initialization: Initial guess $q_0(s, a)$ for all $s, a$. Learning rate $\alpha_t(s_t, a_t)$ is selected as a small positive constant for all $t$. Behavior policy $\pi_b$.
Aim: Learn an optimal target policy $\pi_T$ and optimal action values from some episodes generated by $\pi_b$.

For each episode $\{s_0, a_0, r_1, s_1, a_1, r_2, \dots\}$ generated by $\pi_b$, do
    For each step $t = 0, 1, 2, \dots$ of the episode, do
        Update q-value:
            $q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t)\Big[q(s_t, a_t) - [r_{t+1} + \gamma \max_a q_t(s_{t+1}, a)]\Big]$
        Update target policy:
            $\pi_{T,t+1}(a|s_t) = 1$ if $a = \arg\max_a q_{t+1}(s_t, a)$
            $\pi_{T,t+1}(a|s_t) = 0$ otherwise
```

7.6.5 Illustrative examples

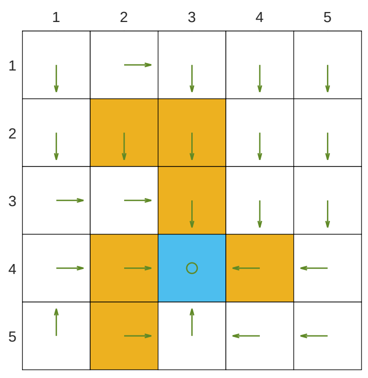
We next present some examples to demonstrate off-policy Q-learning. The task in these examples is to find an optimal policy for all the states. The reward setting is $r_{\text{boundary}} = r_{\text{forbidden}} = -1$, and $r_{\text{target}} = 1$. The discount rate is $\gamma = 0.9$. The learning rate is $\alpha = 0.1$.

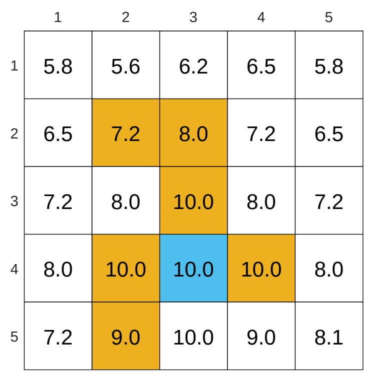
1) We first calculate the ground truth by using the model-based policy iteration algorithm. In particular, an optimal policy and the corresponding optimal state values are shown in Figure 7.2(a) and (b), respectively.

2) We use a uniform behavior policy, where the probability of taking any action at any state is 0.2, to generate a single episode of 100,000 steps. The behavior policy is shown in Figure 7.2(c) and the episode collected is shown in Figure 7.2(d). Due to the good exploration ability of the behavior policy, the episode visits every state-action pair many times.

3) Based on the episode generated by the behavior policy, the final target policy learned by Q-learning is shown in Figure 7.2(e). This policy is optimal because the estimated state value error (root-mean-squared error) converges to zero as shown in Figure 7.2(f). Here, the estimated state value can be calculated from the estimated action values In addition, one may have noticed that the learned optimal policy is not the same as the one in Figure 7.2(a). For example, the policies for (row=3,column=4) are different. That is because there are multiple optimal policies that have the same optimal state values.

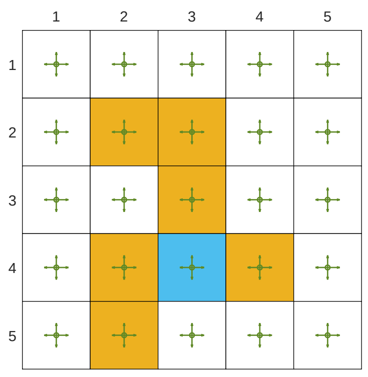
4) Since Q-learning bootstraps, the performance of the algorithm depends on the initial guess of the action values. As shown in Figure 7.2(g), when the initial guess is close to the true value, the estimate converges within about 10,000 steps. Otherwise, the convergence requires more steps (Figure 7.2(h)). Nevertheless, these figures demon-

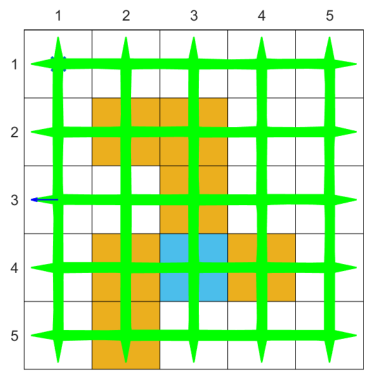

(a) Optimal policy

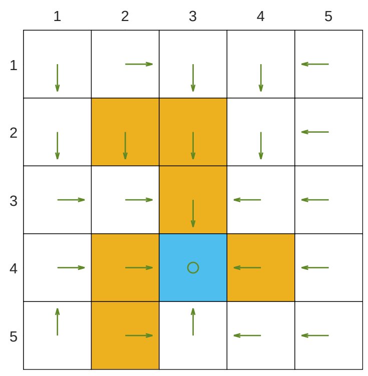


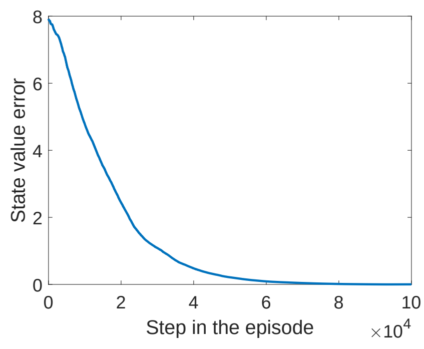
	1	2	3	4	5
1	5.8	5.6	6.2	6.5	5.8
2	6.5	7.2	8.0	7.2	6.5
3	7.2	8.0	10.0	8.0	7.2
4	8.0	10.0	10.0	10.0	8.0
5	7.2	9.0	10.0	9.0	8.1

(b) Optimal state value

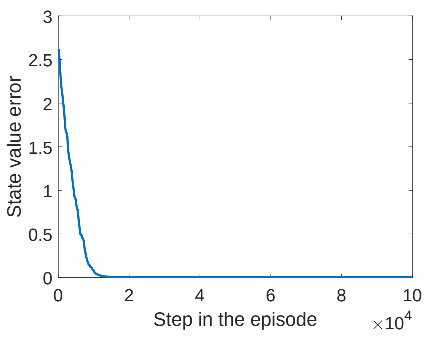

(c) Behavior policy


(d) Generated episode

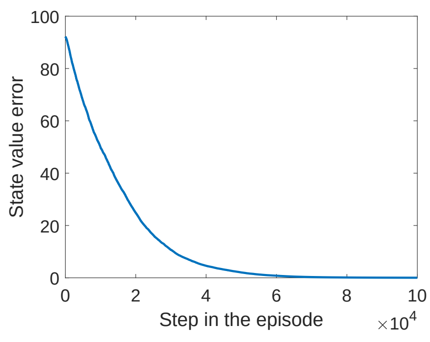

(e) Estimated policy



(f) State value error: $q_0(s,a) = 0$

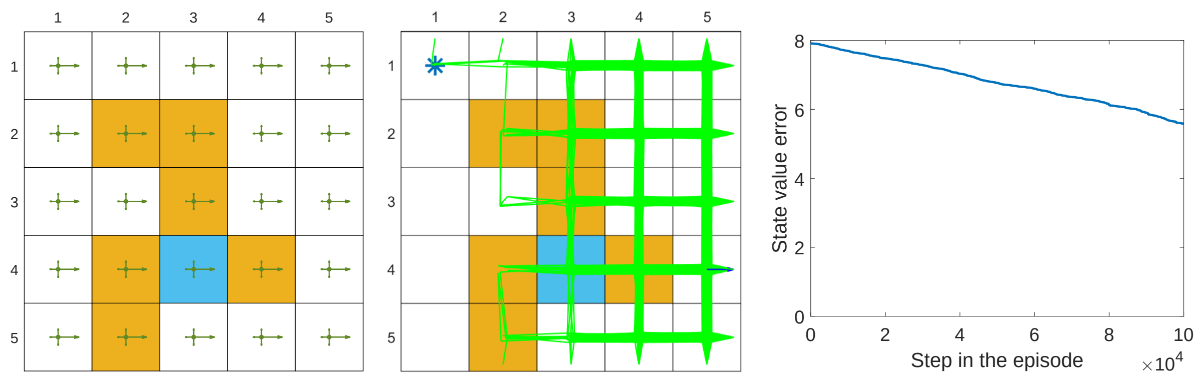


(g) State value error: $q_0(s,a) = 10$

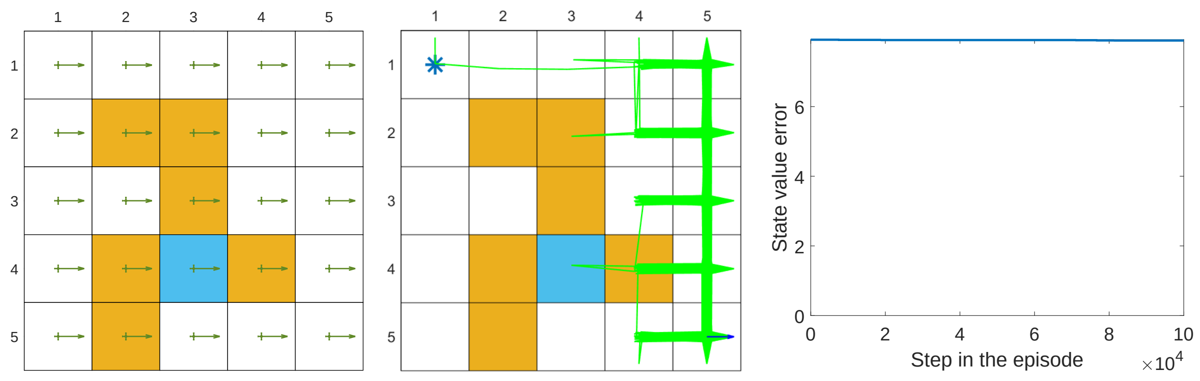


(h) State value error: $q_0(s,a) = 100$

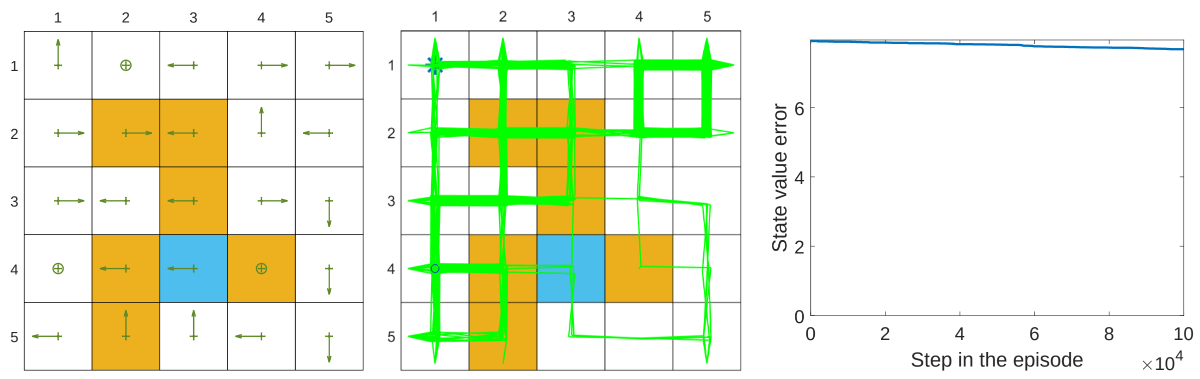
Figure 7.2: Examples to demonstrate off-policy learning by Q-learning. The optimal policy and optimal state values are shown in (a) and (b), respectively. The behavior policy and the generated single episode are shown in (c) and (d), respectively. The estimated policy and the estimation error evolution are shown in (e) and (f), respectively. The cases with different initial q-value are shown in (g) and (h), respectively.



(a) $\epsilon = 0.5$



(b) $\epsilon = 0.1$



(c) $\epsilon = 0.1$

Figure 7.3: Performance of Q-learning given different non-exploratory behavior policies. The figures in the left column show the behavior policies. The figures in the middle column show the generated episodes following the corresponding behavior policies. The episode in every example has 100,000 steps. The figures in the right column show the evolution of the root-mean-squared error of the estimated state values, which are calculated from the estimated action values.

strate that, even though the initial value is not sufficiently inaccurate, Q-learning can still learn satisfactorily fast.

When the behavior policy is not exploratory, the learning performance drops significantly. For example, consider the behavior policies shown in Figure 7.3. They are ϵ-greedy policies with $\epsilon = 0.5$ or 0.1. By the way, the uniform exploratory policy can be viewed as ϵ-greedy with $\epsilon = 1$. It can be clearly seen that, when ϵ decreases from 1 to 0.5 and then to 0.1, the learning rate drops significantly. As a result, although ϵ-greedy policies have certain exploration abilities, their exploration ability becomes weak when ϵ is small.

7.7 A unified viewpoint

Up to now, we have introduced different TD algorithms including Sarsa, Expected Sarsa, n-step Sarsa, and Q-learning. In this section, we introduce a unified framework to incorporate all these algorithms as well as MC learning.

These TD algorithms can all be interpreted as stochastic approximation algorithms solving the Bellman equation or Bellman optimality equation. In particular, all the algorithms can also be expressed in a unified expression:

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t)[q_t(s_t, a_t) - \bar{q}_t], \tag{7.17}$$

where $\bar{q}_t$ is the *TD target*. Different TD algorithms have different $\bar{q}_t$. See Table 7.2 for a list. The MC method can also be expressed as (7.17) by setting $\alpha_t(s_t, a_t) = 1$ and hence $q_{t+1}(s_t, a_t) = \bar{q}_t$.

Similar to the analysis of the TD learning algorithm in Section 7.2.1, it can be seen that (7.17) aims to solve a unified equation: $q(s, a) = \mathbb{E}[\bar{q}_t|s, a]$. This equitation has different expressions for different algorithms. These expressions are summarized in Table 7.2. As can be seen, all of them expect Q-learning aim to solve the Bellman equation with different expressions. Q-learning aims to solve the Bellman optimality equation, which is the fundamental reason why Q-learning is off-policy whereas the others are on-policy.

7.8 Summary

This chapter introduces an important class of RL algorithms called TD learning. The specific algorithms that we introduced include Sarsa, Expected Sarsa, n-step Sarsa, and Q-learning. From a mathematical point of view, all these algorithms are stochastic approximation algorithms solving the Bellman (optimality) equation.

The TD algorithms introduced in this chapter except Q-learning are to evaluate a given policy. That is to estimate the state/action values from experience samples. Together with policy improvement, they can be used to search for optimal policies based on the

Algorithm	**Expression of $\bar{q}_t$ in** (7.17)
Sarsa	$\bar{q}_t = r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1})$
n-step Sarsa	$\bar{q}_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^n q_t(s_{t+n}, a_{t+n})$
Expected Sarsa	$\bar{q}_t = r_{t+1} + \gamma \sum_a \pi_t(a\|s_{t+1}) q_t(s_{t+1}, a)$
Q-learning	$\bar{q}_t = r_{t+1} + \gamma \max_a q_t(s_{t+1}, a)$
Monte Carlo	$\bar{q}_t = r_{t+1} + \gamma r_{t+2} + \dots$

Algorithm	**Equation aimed to solve**
Sarsa	BE: $q_\pi(s,a) = \mathbb{E}\left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \| S_t = s, A_t = a\right]$
n-step Sarsa	BE: $q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^n G_{t+n} \| S_t = s, A_t = a]$
Expected Sarsa	BE: $q_\pi(s,a) = \mathbb{E}\big[R_{t+1} + \gamma \mathbb{E}[q_\pi(S_{t+1}, A_{t+1}) \| S_{t+1}] \big\| S_t = s, A_t = a\big]$
Q-learning	BOE: $q(s,a) = \mathbb{E}\left[R_{t+1} + \max_a q(S_{t+1}, a) \big\| S_t = s, A_t = a\right]$
Monte Carlo	BE: $q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \dots \| S_t = s, A_t = a]$

Table 7.2: A unified point of view of TD algorithms. Here, BE and BOE stand for the Bellman equation and Bellman optimality equation, respectively.

idea of generalized policy iteration. Moreover, these TD algorithms are also on-policy. That is, the target policy is used as the behavior policy to generate experience samples. In addition, compared to the MC learning method, TD learning is online and iterative. That is they can incrementally update the value and policy based on the experience sample obtained in each step.

Compared to other TD algorithms, Q-learning is special in the sense that it is off-policy. In particular, the target policy can be different from the behavior policy in Q-learning. As a result, the behavior policy can be chosen to be an exploratory policy to increase sampling efficiency. The fundamental reason why Q-learning is off-policy is that Q-learning aims to solve the Bellman optimality equation, which is independent of any specific policy. By contrast, the other TD algorithms introduced in this chapter are on-policy because they aim to solve the Bellman equation of a given policy.

It is worth mentioning that there are some methods to convert an on-policy learning algorithm to off-policy. Importance sampling is a widely used method [3, 17]. Moreover, there are various extensions of the TD algorithms introduced in this chapter. For example, the TD(λ) method provides a more general and unified framework for TD learning. The TD algorithm in Section 7.6 is simply TD(0), a special case of TD(λ). Details of TD(λ) can be found in [3, 13].

7.9 Q&A

can off-policy, can the behavior policy be time-varying?

– Q: From a mathematical point of view, what does TD learning do?

  A: TD algorithms are stochastic approximation algorithms for solving state/action values from the Bellman equation.

– Q: What does the term "TD" in TD learning mean?

  A: Every TD algorithm has a TD error, which represents the deficiency between the new sample and the current estimates. This deficiency is between two different time steps and hence called temporal difference.

– Q: What does the term "learning" in TD learning mean?

  A: Up to now, it has become clear that the term learning is mathematically estimating some values from experience samples. In particular, all the RL algorithms introduced so far contain value estimation and policy estimation. Policy estimation depends on value estimation. Value estimation is to estimate state/action values from experience samples, which is referred to as a learning process.

– Q: What are online learning and offline learning?

  A: Offline learning means the learning process can only start after a complete episode has been collected. Online learning means the value and policy can be updated instantly when a sample is received. MC learning is offline. TD learning is online.

– Q: What is the fundamental difference between TD learning and MC learning?

  A: The expression of a TD learning algorithm looks quite different from MC learning. That is because MC learning is offline whereas TD learning is online, iterative, and bootstrapping.

– Q: What is the fundamental common property between TD learning and MC learning?

  A: They both are algorithms estimating state/action values from experience samples.

– Q: While TD learning such as Sarsa aims to estimate action values, how can it be used for optimal policy searching?

  A: To obtain an optimal policy, the value estimation process should interact with the policy update process. That is, after a value is updated, the corresponding policy should be updated. Then, a new sample generated by the updated policy is used to update values again. This is the idea of generalized policy iteration.

– Q: When we use Sarsa for policy searching, why do we update policies as ϵ-greedy?

  A: That is because the policy is also used to generate samples for value estimation and hence we hope it is exploratory.

- Q: While the convergence of TD learning requires that the learning rate converges to zero gradually, why is the learning rate often chosen to be constant in practice?

  A: The fundamental reason is the policy to be evaluated is nonstationary. A TD learning algorithm like Sarsa aims to estimate the action values of a *given fixed policy.* In other words, it aims to evaluate the given policy. If the policy is stationary, the learning rate can decrease to zero gradually to ensure convergence with probability 1 as stated in Theorem 7.1.

  However, when we put Sarsa in the context of optimal policy searching, the value estimation process keeps interacting with the policy update process. Therefore, the policy that Sarsa aims to evaluate keeps *changing.* In this case, we need to use a constant learning rate because, every time we have a new policy to evaluate, the learning rate cannot be too small. The drawback of a constant learning rate is that the value estimate may still fluctuate eventually. However, as long as the constant learning rate is sufficiently small, the fluctuation would not jeopardize the convergence.

- Q: Should we estimate the optimal policies for all states or a subset of states?

  A: It depends on the task. One may have noticed that some tasks in the illustrative examples in this chapter are not to find the optimal policies for all states. Instead, they only need to find a good path from a specific starting state to the target state. Such a task is not data demanding because we do not need to visit every state-action pair sufficiently many times. As shown in the examples in Figure 7.1, even if some states are not visited, we can still obtain a good path.

  It, however, must be noted that the path is only good but not guaranteed to be optimal. That is because not all state-action pairs have been explored and there may be better paths unexplored. Nevertheless, given sufficient data, there is a high probability for us to find a good or *locally* optimal path. By locally optimal, we mean the path is optimal within a neighborhood of the path.

- Q: What are behavior policy and target policy?

  A: There exist two policies in a TD learning task: *behavior policy* and *target policy.* The behavior policy is used to generate experience samples. The target policy is the one constantly updated based on the experience samples.

- Q: What are off-policy learning and on-policy learning?

  A: When the behavior policy is the same as the target policy, such kind of learning is called on-policy. When they are different, the learning is called off-policy.

- Q: What are the advantages of off-policy learning compared to on-policy learning?

  A: Since on-policy learning is a special case of off-policy learning when the behavior and target policies are the same, off-policy learning is more flexible than on-policy learning. For example, off-policy learning can estimate optimal policies based on some experience samples generated by any other behavior policies.

– Q: What is the fundamental reason for Q-learning to be off-policy while all the other TD algorithms in this chapter are on-policy?

  A: The fundamental reason why Q-learning is off-policy is that Q-learning aims to solve the *Bellman optimality equation*. The other TD algorithms are on-policy because they aim to solve the *Bellman equation of a given policy*. Since the Bellman optimality equation does not depend on any specific policy, it can use the experience samples generated by any other policies to estimate the optimal policy. However, the Bellman equation depends on a given policy, solving it of course requires the experience samples generated by the given policy.

– Q: Why does the off-policy version of Q-learning update policies as greedy instead of ϵ-greedy?

  A: Since the target policy is not required to generate experience samples, it is not required to be exploratory.

Chapter 8

Value Function Approximation

So far in this book, state and action values are represented by *tables*. Although such a tabular representation is intuitive, it would encounter some problems when the state or action space is large. For example, it may be impossible to store too many state or action values. Moreover, since the value of a state is updated only if it is visited, the values of unvisited states cannot be estimated. However, when the state space is large, it is difficult to ensure that all the states are visited. It is therefore favorable if the values of visited states can be generalized to unvisited states.

These problems can be solved if we approximate the state and action values using *parameterized functions*. More specifically, we can use $\hat{v}(s, w)$ as a parameterized function to approximate the true state value $v_\pi(s)$ of a policy π. Here, s is the state variable and $w \in \mathbb{R}^m$ is a parameter vector. The dimension of w may be much less than $|\mathcal{S}|$. In this way, we only need store the m-dimensional vector w (as well as the function structure) rather than the $|\mathcal{S}|$-dimensional state values $\{v_\pi(s)\}_{s\in\mathcal{S}}$. On the other hand, when a state s is visited, the parameter w is updated so that the values of some other unvisited states can also be updated. In this way, the learned values can generalize to unvisited states.

Another reason why the content of this chapter is important is that this is the venue where artificial neural networks are introduced into reinforcement learning (RL) as non-linear function approximators. In addition, the idea of *value function approximation* can be extended to *policy function approximation* as introduced in the next chapter.

8.1 Motivating examples: curve fitting

Now we use an example to demonstrate what function approximation is.

Suppose we have some states $s_1, \dots, s_{|\mathcal{S}|}$. Their state values are $v_\pi(s_1), \dots, v_\pi(s_{|\mathcal{S}|})$, where π is a given policy. If we plot the values, we get $|\mathcal{S}|$ dots as shown in Figure 8.1. Suppose $|\mathcal{S}|$ is very large and we hope to use a simple curve to approximate these dots to save storage. The simplest way is to use a straight line to fit the dots. Suppose the

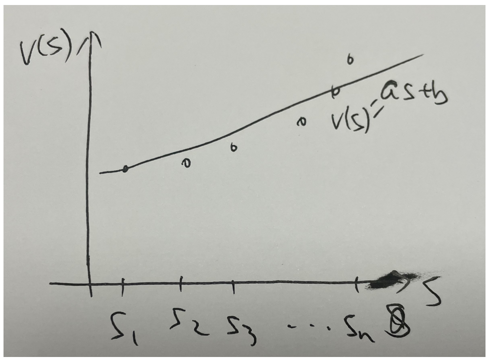


Figure 8.1: An illustration of function approximation of samples.

equation of the straight line is

$$v_\pi(s) \approx as + b,$$

where $a, b \in \mathbb{R}$. The equation can be rewritten as

$$v_\pi(s) \approx as + b = \underbrace{[s, 1]}_{\phi^T(s)} \underbrace{\begin{bmatrix} a \\ b \end{bmatrix}}_{w} = \phi^T(s)w, \tag{8.1}$$

where w is the *parameter vector* and $\phi(s)$ the *feature vector* of s. It is notable that $v_\pi(s)$ is *linear* in w.

The benefit of the approximation approach is obvious. While the tabular representation needs to store $|\mathcal{S}|$ state values, now we only need to store two parameters a and b. Every time we would like to access the value of a state, we only need to calculate the inner product between the parameter vector w and the feature vector $\phi(s)$ as indicated in (8.1). Such a benefit is, however, *not* free. It comes with a cost: the state values can not be represented accurately. This is not surprising because there does not exist a straight line that can perfectly fit all the points in Figure 8.1. That is why this method is called value *approximation*.

In addition to a straight line, we can approximate the points using a second-order polynomial curve as

$$v_\pi(s) \approx as^2 + bs + 1 = \underbrace{[s^2, s, 1]}_{\phi^T(s)} \underbrace{\begin{bmatrix} a \\ b \\ c \end{bmatrix}}_{w} = \phi^T(s)w.$$

In this case, the dimensions of w and $\phi(s)$ increase, but the values may be fitted more accurately. It must be noted that, although $v_\pi(s)$ is a *nonlinear* function in s, it is *linear* in w.

Of course, we can use even higher-order polynomial curves to fit the dots. It also needs more parameters. From a fundamental point of view, when we use a low-dimensional vector to represent a high-dimensional data set, some information will certainly be lost. Therefore, value function approximation enhances storage and computational efficiency by sacrificing accuracy.

In practice, which feature vector should we use to fit the data points? Should we fit the points as a first-order straight line or a second-order curve? The answer is nontrivial. That is because the selection of feature vector relies on certain domain knowledge. The better we understand a problem, the better feature vectors we can select. If we do not have domain knowledge, a popular solution is to use neural networks to let the network learn features and then fit the points automatically.

Another important problem is how to find the optimal parameter vector. This is a regression problem. We can find the optimal w by optimizing the following objective function:

$$J_1 = \sum_{i=1}^{|\mathcal{S}|} \|\phi^T(s_i)w - v_\pi(s_i)\|^2 = \left\| \begin{bmatrix} \phi^T(s_1) \\ \vdots \\ \phi^T(s_{|\mathcal{S}|}) \end{bmatrix} w - \begin{bmatrix} v_\pi(s_1) \\ \vdots \\ v_\pi(s_{|\mathcal{S}|}) \end{bmatrix} \right\|^2 \doteq \|\Phi w - v_\pi\|^2,$$

where

$$\Phi \doteq \begin{bmatrix} \phi^T(s_1) \\ \vdots \\ \phi^T(s_{|\mathcal{S}|}) \end{bmatrix} \in \mathbb{R}^{|\mathcal{S}|\times 2}, \qquad v_\pi \doteq \begin{bmatrix} v_\pi(s_1) \\ \vdots \\ v_\pi(s_{|\mathcal{S}|}) \end{bmatrix} \in \mathbb{R}^{|\mathcal{S}|}.$$

This is a simple least-squares problem. The optimal value of w that can minimize J_1 can be obtained as

$$w^* = (\Phi^T\Phi)^{-1}\Phi v_\pi.$$

The curve fitting example presented in this section illustrates the basic idea of value function approximation. This idea will be formally and thoroughly studied in the rest of this chapter.

8.2 Objective function

Let $v_\pi(s)$ and $\hat{v}(s,w)$ be the true state value and approximated state value of $s \in \mathcal{S}$. Our goal is to find an *optimal* w so that $\hat{v}(s,w)$ can best approximate $v_\pi(s)$ for every s. To find the optimal w, we need *two steps*. The first step, as discussed in this section, is to define an objective function. The second step, as discussed in the next section, is to derive algorithms optimizing the objective function.

The objective function considered in value function approximation is

$$J(w) = \mathbb{E}[(v_\pi(S) - \hat{v}(S,w))^2]. \tag{8.2}$$

Our goal is to find the best w that can minimize $J(w)$. Here, the expectation is with respect to the random variable $S \in \mathcal{S}$. Thus, the objective function can be interpreted as the average approximation error over all states.

While S in (8.2) is a random variable, what is the probability distribution of S? This problem is often confusing to beginners because we have not discussed the probability distribution of states so far in this book. There are several ways to define the probability distribution of S.

- The first way is to use an *even distribution*. That is to treat all the states to be *equally important* by setting the probability of each state as $1/|\mathcal{S}|$. In this case, the objective function in (8.2) becomes

  $$J(w) = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} (v_\pi(s) - \hat{v}(s, w))^2.$$

  However, the importance of some states may be less than others. For example, some states may be rarely visited by a policy. Therefore, this way does not consider the real dynamics of the Markov process under the given policy.

- The second way, which is the focus of this chapter, is to use the *stationary distribution*. Stationary distribution is an important concept that will be frequently used in this book. In short, it describes the *long-run behavior* of a Markov process. Interested readers may see the details given in the shaded box.

  Let $\{d_\pi(s)\}_{s \in \mathcal{S}}$ denote the stationary distribution of the Markov process under policy π. By definition, $d_\pi(s) \geq 0$ and $\sum_{s \in \mathcal{S}} d_\pi(s) = 1$. The objective function in (8.2) can be rewritten as

  $$J(w) = \sum_{s \in \mathcal{S}} d_\pi(s)(v_\pi(s) - \hat{v}(s, w))^2.$$

  This function is a weighted squared error. Since more frequently visited states have higher values of $d_\pi(s)$, their weights in the objective function are also higher than those rarely visited states.

  The value of $d_\pi(s)$ is nontrivial to obtain because it requires knowing the state transition probability matrix P_π (see the shaded box below). Fortunately, we do not need to calculate the specific value of $d_\pi(s)$ to minimize this objective function as shown in the next section.

Stationary distribution of a Markov process

Stationary distribution is also called *steady-state distribution*, or *limiting distribution*. It describes the long-run behavior of a Markov process. It is useful not only for value approximation as introduced in this chapter but also for policy approximation as introduced in the next chapter.

The key tool to analyze stationary distribution is $P_\pi \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}$, which is the probability transition matrix under the given policy π. If the states are indexed as $1,\dots,|\mathcal{S}|$, then $[P_\pi]_{ij}$ is the probability jumping from state i to state j.

1) What is the interpretation of P_π^k?

First of all, it is necessary to understand some properties of P_π^k $(k = 1, 2, 3, \dots)$.

The probability of the state transiting from i to j using exactly k steps is denoted as

$$p_{ij}^{(k)} = \text{Prob}(S_{t_k} = j|S_{t_0} = i),$$

where t_0 and t_k are the initial and kth time steps, respectively. Apparently,

$$[P_\pi]_{ij} = p_{ij}^{(1)},$$

which means that $[P_\pi]_{ij}$ is the probability transiting from i to j using *a single step*. Now consider P_π^2. It can be verified that

$$[P_\pi^2]_{ij} = [P_\pi P_\pi]_{ij} = \sum_{q=1}^{|\mathcal{S}|}[P_\pi]_{iq}[P_\pi]_{qj}.$$

Since $[P_\pi]_{iq}[P_\pi]_{qj}$ is the joint probability transiting from i to q and then from q to j, we know that $[P_\pi^2]_{ij}$ is the probability transiting from i to j using *exactly two steps*. That is

$$[P_\pi^2]_{ij} = p_{ij}^{(2)}.$$

Similarly, we know that

$$[P_\pi^k]_{ij} = p_{ij}^{(k)},$$

which means that $[P_\pi^k]_{ij}$ is the probability transiting from i to j using *exactly k steps*.

2) Stationary probability distribution

Now we come back to the probability distribution of the states.

Let $d_0 \in \mathbb{R}^{|\mathcal{S}|}$ be the initial probability distribution vector of the states. For example, if s is always selected as the starting state, then $d_0(s) = 1$ and the other entries of d_0 are 0. Let d_k be the probability distribution vector after exactly k steps starting from d_0. The relationship between d_k and d_0 is

$$d_k^T = d_0^T P_\pi^k.$$

Why multiply d_0 to the left of P_π^k? As aforementioned, the ith column of P_π^k is the probability reaching each state after k steps starting from state i.

When we consider the long-run behavior of the Markov process, it holds under certain conditions that

$$P_\pi^k \to W = \mathbf{1}_n d_\pi^T, \quad \text{as} \quad k \to \infty, \tag{8.3}$$

where W is a constant matrix with all the rows equal to a vector denoted as d_π^T. This is an interesting result. The conditions under which (8.3) is valid will be discussed shortly. If (8.3) is valid, then we have

$$d_k^T = d_0^T P_\pi^k \to d_0^T W = d_0^T \mathbf{1}_n d_\pi^T = d_\pi^T \quad \text{as} \quad k \to \infty.$$

That means the state distribution converges to a constant value d_π, which is called the *stationary distribution*. The stationary distribution depends on the dynamics of the Markov process and hence the policy π. Interestingly, the stationary distribution is independent of the initial distribution d_0. That is, no matter which state the agent starts from, the state where the agent is located after a sufficiently long period can always be described by the stationary distribution.

What is the value of d_π? Since $d_k^T = d_{k-1}^T P_\pi$ and $d_k, d_{k-1} \to d_\pi$, we know

$$d_\pi^T = d_\pi^T P_\pi.$$

As a result, d_π is the left eigenvector of P_π associated with the eigenvalue of 1.

3) Under what conditions do stationary distributions exist?

A general class of Markov processes that have unique stationary distributions is *regular* Markov processes. To define "regular", we need first introduce some basic concepts.

- State j is said to be *accessible* from state i if $[P_\pi]_{ij}^k > 0$ for certain finite integer k, which means the agent starting from i can always reach j after a finite number of transitions.
- If two states i and j are mutually accessible, then the two states are said to *communicate*.
- A Markov process is called *irreducible* if all the states communicate with each other. In other words, the agent starting from an arbitrary state can always reach any other states within a finite number of steps.

– A Markov process is called *regular* if it is irreducible and, in the meantime, there exists a state i such that $[P_\pi]_{ii} > 0$. Here, $[P_\pi]_{ii} > 0$ means the agent may transit to i starting from i with one step.

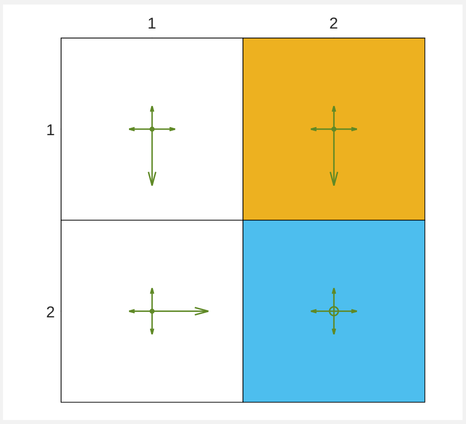
For a regular Markov process, there exists a unique stationary distribution d_π such that $d_\pi^T = d_\pi^T P_\pi$. More details can be found in [18, Chapter IV].

4) What kind of policies can lead to stationary distributions?

Once the policy is determined, a Markov decision process becomes a Markov process, whose long-run behavior is jointly determined by the given policy and the system model. Then, an important question is what kind of policies can lead to regular Markov processes? In general, the answer is *exploration policies*. For example, ϵ-greedy policies introduced in the last chapter are exploratory and can often lead to regular Markov processes. That is because the ϵ-greedy policies are exploratory in the sense that each action has a positive probability to be taken. As a result, the states can communicate with each other under an ϵ-greedy when the system model allows.

5) Illustrative examples

We next show some examples to illustrate stationary distribution. We estimate the stationary distribution by using the fraction of the times each state is visited. In particular, suppose the number of times that the agent visits s in an episode generated following π is $n_\pi(s)$. Then $d_\pi(s)$ can be approximated by $d_\pi(s) \approx n_\pi(s)/\sum_{s'\in\mathcal{S}} n_\pi(s')$.



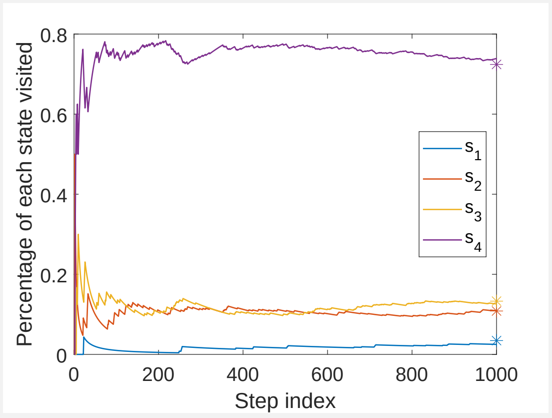


Figure 8.2: Long-run behavior of an ϵ-greedy policy with $\epsilon = 0.5$. The asterisks in the right figure represent the theoretical values of the elements in d_π.

First, consider the example in Figure 8.2. The policy in this example is ϵ-greedy with $\epsilon = 0.5$. The states are indexed as s_1, s_2, s_3, s_4, which respectively correspond to the top-left, top-right, bottom-left, bottom-right cells in the grid. It can be verified that the Markov process induced by the policy is regular. That is due to the following reasons. First, since all the states communicate, the resulting Markov process is irreducible. Second, since every state can transit to itself, the resulting Markov process is regular. Mathematically, the matrix P_π^T is

$$P_\pi^T = \begin{bmatrix} 0.3 & 0.1 & 0.1 & 0 \\ 0.1 & 0.3 & 0 & 0.1 \\ 0.6 & 0 & 0.3 & 0.1 \\ 0 & 0.6 & 0.6 & 0.8 \end{bmatrix}.$$

The eigenvalues of P_π^T can be calculated as $\{-0.0449, 0.3, 0.4449, 1\}$. The unit-length (right) eigenvector of P_π^T corresponding to the eigenvalue of 1 is $[0.0463, 0.1455, 0.1785, 0.9720]^T$. After scaling this vector so that the sum of all its elements is equal to 1, we obtain the stationary distribution vector as

$$d_\pi = \begin{bmatrix} 0.0345 \\ 0.1084 \\ 0.1330 \\ 0.7241 \end{bmatrix}.$$

The ith element of d_π corresponds to the probability for the agent to be located on s_i in the long run.

We next verify the above theoretical value of d_π by executing the policy for sufficiently many steps in simulation and observing the long-run behavior. Specifically, we randomly select a starting state. For example, suppose the starting state is s_1. Following the policy, we run 1,000 steps. The number of each state visited during the process is shown in Figure 8.2. As can be seen, after hundreds of steps, the times each state visited converge to the theoretical values of d_π.

Second, consider the example in Figure 8.3. The policy in this example is ϵ-greedy with $\epsilon = 0.3$. It can also be verified that the Markov process induced by the policy is regular. The corresponding stationary distribution is $d_\pi = [0.0099, 0.0633, 0.0717, 0.8551]^T$. As shown in the right subfigure, the long-run behavior gradually converges to d_π. Compared to the first example, it can be seen in the second example that the probability of visiting s_4 is higher in the long run. That is simply because the policy is less exploratory due to a small value of ϵ and the target state s_4 is visited more frequently.

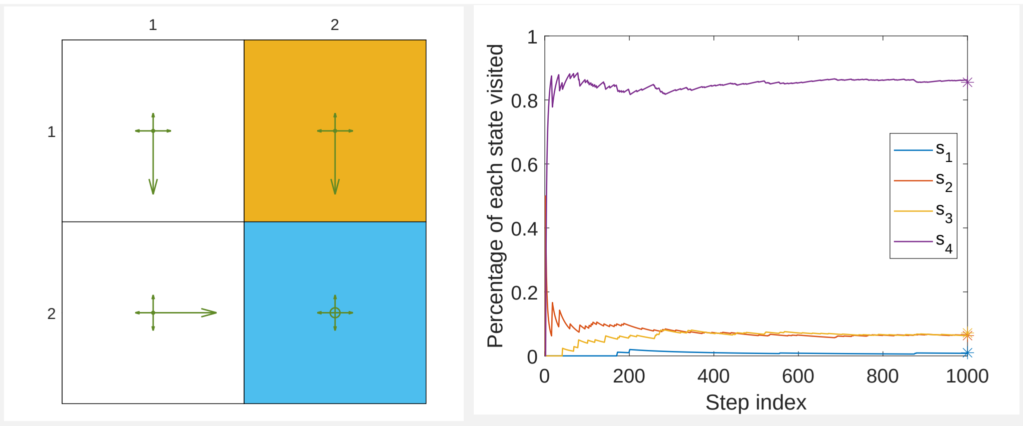


Figure 8.3: Long-run behavior of an ϵ-greedy policy with $\epsilon = 0.3$. The asterisks in the right figure represent the theoretical values of the elements in d_π.

Third, consider the example in Figure 8.4. The policy in this example is greedy with $\epsilon = 0$. The resulting Markov process is not irreducible because no state can communicate with each other. However, it can be verified that P_π^T in this case still has a single eigenvalue equal to 1 and the associated eigenvector is $d_\pi = [0, 0, 0, 1]^T$. The long-run behavior illustrated in Figure 8.4 is consistent with d_π. Compared to the previous two examples, the probability to visit the target state in the long run is as high as one since the policy here is greedy. This example also demonstrates that the condition that the Markov process should be regular is merely sufficient but not necessary to ensure a unique stationary distribution.

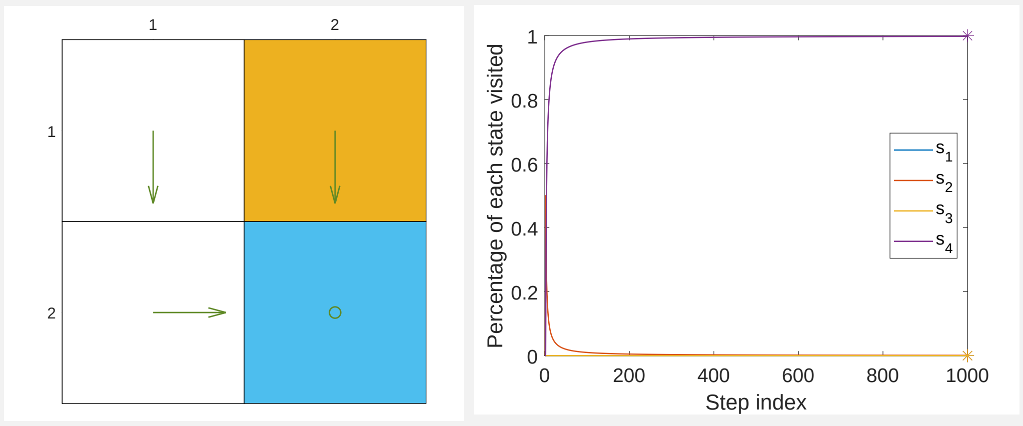


Figure 8.4: Long-run behavior of a greedy policy. The asterisks in the right figure represent the theoretical values of the elements in d_π.

8.3 Optimization algorithms

To minimize the objective function $J(w)$ in (8.2), we can use the gradient-descent algorithm:

$$\begin{aligned}w_{k+1} &= w_k - \alpha_k \nabla_w \mathbb{E}[(v_\pi(S) - \hat{v}(S,w))^2] \\ &= w_k - \alpha_k \mathbb{E}[\nabla_w (v_\pi(S) - \hat{v}(S,w))^2] \\ &= w_k - 2\alpha_k \mathbb{E}[(v_\pi(S) - \hat{v}(S,w))(-\nabla_w \hat{v}(S,w))] \\ &= w_k + 2\alpha_k \mathbb{E}[(v_\pi(S) - \hat{v}(S,w))\nabla_w \hat{v}(S,w)], \end{aligned} \tag{8.4}$$

where the coefficient 2 before α_k can be dropped without loss of generality. The above gradient-descent algorithm requires calculating the expectation. By the spirit of stochastic gradient descendent, we can remove the expectation operation from (8.4) to obtain the following algorithm:

$$w_{t+1} = w_t + \alpha_t (v_\pi(s_t) - \hat{v}(s_t, w_t))\nabla_w \hat{v}(s_t, w_t), \tag{8.5}$$

where s_t is a sample of S.

It must be noted that (8.5) is *not* implementable because it requires the true state value v_π, which is the unknown to be estimated. We can replace $v_\pi(s_t)$ with an approximation so that the algorithm is implementable. In particular, suppose we have an episode $(s_0, r_1, s_1, r_2, \dots)$.

– First, let g_t be the discounted return calculated starting from s_t in the episode. Then, g_t can be used as an approximation of $v_\pi(s_t)$. The algorithm in (8.5) hence becomes

$$w_{t+1} = w_t + \alpha_t (g_t - \hat{v}(s_t, w_t))\nabla_w \hat{v}(s_t, w_t).$$

This is the algorithm of Monte Carlo learning with function approximation.

– Second, by the spirit of TD learning, $r_{t+1} + \gamma \hat{v}(s_{t+1}, w_t)$ can be viewed as an approximation of $v_\pi(s_t)$. Then, the algorithm in (8.5) becomes

$$w_{t+1} = w_t + \alpha_t \left[r_{t+1} + \gamma \hat{v}(s_{t+1}, w_t) - \hat{v}(s_t, w_t) \right] \nabla_w \hat{v}(s_t, w_t). \tag{8.6}$$

This is the algorithm of TD learning with function approximation.

The TD algorithm in (8.6) is one of the focuses of this chapter. In the next two sections, we will continue to study this algorithm from the perspectives of function structure and convergence analysis. Notably, (8.6) can only approximate the *state values* of a given policy. We will extend this algorithm to Sarsa and Q-learning with function approximation that can approximate *action values* and then used to search for optimal policies in Sections 8.7 and 8.8.

Pseudocode: TD learning with function approximation

```
Initialization: A function $\hat{v}(s,w)$ that is a differentiable in $w$. Initial parameter $w_0$.
Aim: Approximate the true state values of a given policy $\pi$.

For each episode $\{(s_t, r_{t+1}, s_{t+1})\}$ generated following the policy $\pi$, do
    For each sample $(s_t, r_{t+1}, s_{t+1})$, do
        In the general case, $w_{t+1} = w_t + \alpha_t \left[r_{t+1} + \gamma \hat{v}(s_{t+1}, w_t) - \hat{v}(s_t, w_t)\right] \nabla_w \hat{v}(s_t, w_t)$
        In the linear case, $w_{t+1} = w_t + \alpha_t \big[r_{t+1} + \gamma \phi^T(s_{t+1}) w_t - \phi^T(s_t) w_t\big] \phi(s_t)$
```

8.4 Linear function approximation

An important problem about the TD algorithm in (8.6) is how to select the function $\hat{v}(s,w)$. There are two potential approaches. The first approach, which is popular nowadays, is to use a neural network as a *nonlinear* approximator, where the input is the state, the output is $\hat{v}(s,w)$, and the network parameter is w. The second approach, which is the focus of this chapter, is to use a *linear* function

$$\hat{v}(s,w) = \phi^T(s)w,$$

where $\phi(s) \in \mathbb{R}^m$ is the feature vector associated with s. The dimensions of the two vectors are the same as m, which is usually much smaller than $|\mathcal{S}|$. In the linear case, we have the gradient

$$\nabla_w \hat{v}(s,w) = \phi(s).$$

Substituting the gradient into (8.6) yields

$$w_{t+1} = w_t + \alpha_t \big[r_{t+1} + \gamma \phi^T(s_{t+1}) w_t - \phi^T(s_t) w_t\big] \phi(s_t), \tag{8.7}$$

which is the algorithm of TD learning with linear function approximation. It is called *TD-Linear* in short. Here we discuss some important problems with it.

– Why do we need to study linear function approximators while nonlinear neural networks are widely used nowadays? That is because the theoretical properties of the TD algorithm in the linear case can be much better understood than in the nonlinear case. For example, the TD algorithm guarantees to converge when linear functions are used, but may diverge when inappropriate nonlinear functions are used [19]. Fully understanding the linear case is necessary for us to appropriately use nonlinear approximators.

– Is linear function approximation powerful? Linear function approximation does have some limitations due to its simple linear form. However, it is still powerful. One piece of evidence is that the tabular representation is merely a special case of linear function approximation. To see that, consider the special feature vector for state s:

$$\phi(s) = e_{i(s)} \in \mathbb{R}^{|\mathcal{S}|},$$

where the subscript $i(s)$ is the index of state s, and the vector $e_{i(s)}$ is the vector with the $i(s)$th entry as 1 and the others as 0. In this case,

$$\hat{v}(s, w) = e_{i(s)}^T w = [w]_{i(s)},$$

where $[w]_{i(s)}$ is the $i(s)$th entry of w. In this case, w is exactly the true state value to be estimated and the TD algorithm becomes the normal tabular TD algorithm. Specifically, (8.7) in this case becomes

$$w_{t+1} = w_t + \alpha_t \left(r_{t+1} + \gamma [w_t]_{i(s_{t+1})} - [w_t]_{i(s_t)} \right) e_{i(s_t)}.$$

The above equation merely updates the $i(s_t)$th entry of w_t. Multiplying $e_{i(s_t)}^T$ on both sides of the equation gives

$$[w_{t+1}]_{i(s_t)} = [w_t]_{i(s_t)} + \alpha_t \left(r_{t+1} + \gamma [w_t]_{i(s_{t+1})} - [w_t]_{i(s_t)} \right).$$

By rewriting $[w_t]_{i(s_t)} = w_t(s_t)$, the above equation becomes

$$w_{t+1}(s_t) = w_t(s_t) + \alpha_t \left(r_{t+1} + \gamma w_t(s_{t+1}) - w_t(s_t) \right),$$

which is exactly the tabular TD algorithm.

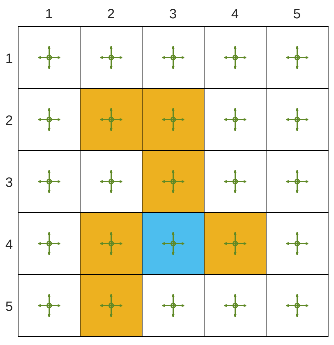
It is notable that, to achieve the lookup table representation, the dimension of w must be $|\mathcal{S}|$. When the dimension is reduced, some information about the state values will be lost and the linear form may not approximate the tabular case accurately.

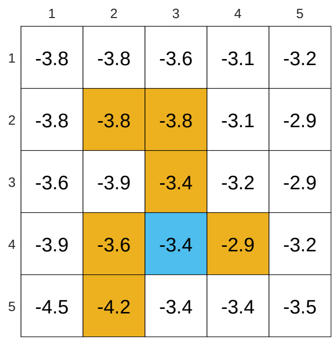
8.5 Illustrative examples

We next present some examples to demonstrate how to use the TD-Linear algorithm in (8.7) to estimate the state values of a given policy. In the meantime, we illustrate how to select feature vectors.

Consider the grid-world example shown in Figure 8.5. The given policy takes any action at any state with the probability of 0.2. Our aim is to estimate the state values under this policy. There are 25 state values in total. We next show that we can use less than 25 parameters to approximate these state values. Set $r_{\text{forbidden}} = r_{\text{boundary}} = -1$, $r_{\text{target}} = 1$, and $\gamma = 0.9$. The true state values are given in Figure 8.5(b). The true state values are further visualized as a three-dimensional surface in Figure 8.5(c).

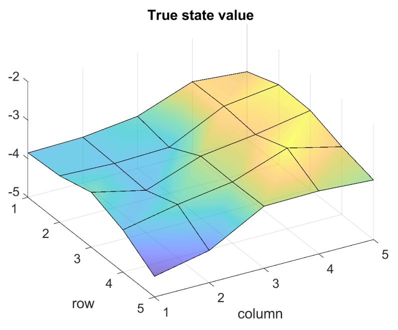
To estimate the state values, 500 episodes were generated following the given policy.


(a)



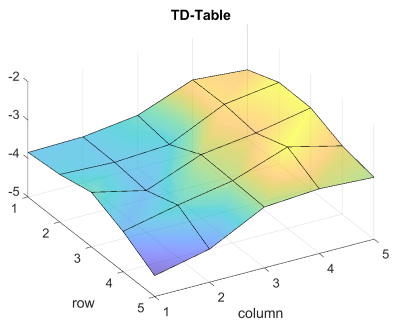
	1	2	3	4	5
1	-3.8	-3.8	-3.6	-3.1	-3.2
2	-3.8	-3.8	-3.8	-3.1	-2.9
3	-3.6	-3.9	-3.4	-3.2	-2.9
4	-3.9	-3.6	-3.4	-2.9	-3.2
5	-4.5	-4.2	-3.4	-3.4	-3.5

(b)

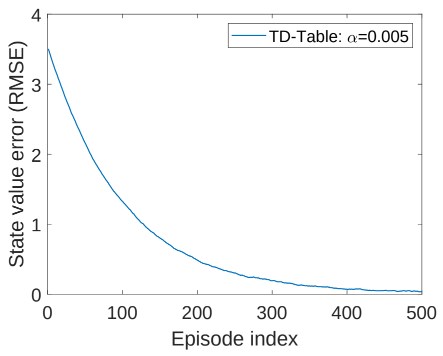


(c)

Figure 8.5: A policy and its true state values. (a) The policy to be evaluated. (b) The true state values are represented in a table. (c) The true state values are visualized as a 3D surface.



(a)



(b)

Figure 8.6: TD-Table estimation results. (a) The estimated state values are visualized as a 3D surface. (b) The evolution of the state estimation error, which is the root-mean-squared error (RMSE).

Each episode has 500 steps and starts from a randomly selected state-action pair following a uniform distribution. For comparison, we first estimate the state values using the tabular TD algorithm introduced in Chapter 7 (TD-Table in short). The results are shown in Figure 8.6. As can be seen, the TD-Table algorithm can accurately estimate the state values.

To implement the TD-Linear algorithm, it is necessary to select the feature vector $\phi(s)$ first. Our goal here is to use $\phi^T(s)w$ to approximate the three-dimensional (3D) surface representing the true state values as shown in Figure 8.5(c). When implementing the TD-Linear algorithm, we initialize w in the way that each element is randomly drawn from a standard normal distribution where the mean is 0 and the standard deviation is 1.

- The first type of feature vector is based on polynomials. In the grid-world example, a state s corresponds to a 2D location. Let x and y denote the column and row indexes of s. In the implementation, we normalize x and y so that their values are within the interval of [-1,+1]. With abused notations, the normalized values are also represented by x and y. Then, the simplest feature vector is

$$\phi(s) = \begin{bmatrix} x \\ y \end{bmatrix} \in \mathbb{R}^2.$$

In this case, we have

$$\hat{v}(s,w) = \phi^T(s)w = [x,y]\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = w_1x + w_2y.$$

When w is fixed, $\hat{v}(s,w) = w_1x + w_2y$ represents a 2D plane that passes through the origin. While the surface of the state values may not pass through the origin, we must introduce a bias to the 2D plane to better approximate the state values. To do that, we can consider the following 3D feature vector:

$$\phi(s) = \begin{bmatrix} 1 \\ x \\ y \end{bmatrix} \in \mathbb{R}^3. \tag{8.8}$$

In this case, the approximated state value is

$$\hat{v}(s,w) = \phi^T(s)w = [1,x,y]\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = w_1 + w_2x + w_3y.$$

When w is fixed, $\hat{v}(s,w)$ corresponds to a general plane that may or may not pass through the origin. Notably, $\phi(s)$ can also be defined as $\phi(s) = [x,y,1]^T$, where the order of the elements does not matter.

The estimation result when we use the feature vector in (8.8) is given in Figure 8.7(a). As can be seen, all the estimated state values form a 2D plane. Although the estimation error decreases as more episodes are used, the error cannot converge to zero due to the limited approximation ability of $\hat{v}(s,w)$ in this case.

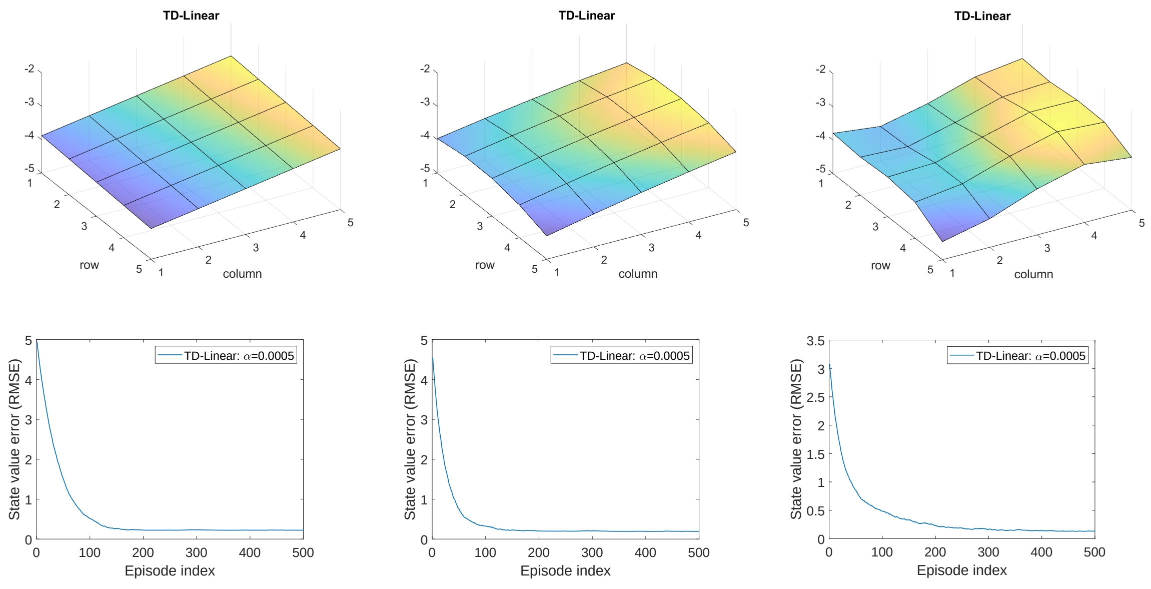
To enhance the approximation ability, we can increase the dimension of the feature vector. To that end, we can consider

$$\phi(s) = [1,x,y,x^2,y^2,xy]^T \in \mathbb{R}^6. \tag{8.9}$$

In this case, $\hat{v}(s,w) = \phi^T(s)w = w_1 + w_2x + w_3y + w_4x^2 + w_5y^2 + w_6xy$ corresponds to a quadratic 3D surface. We can further increase the dimension of the feature vector to use

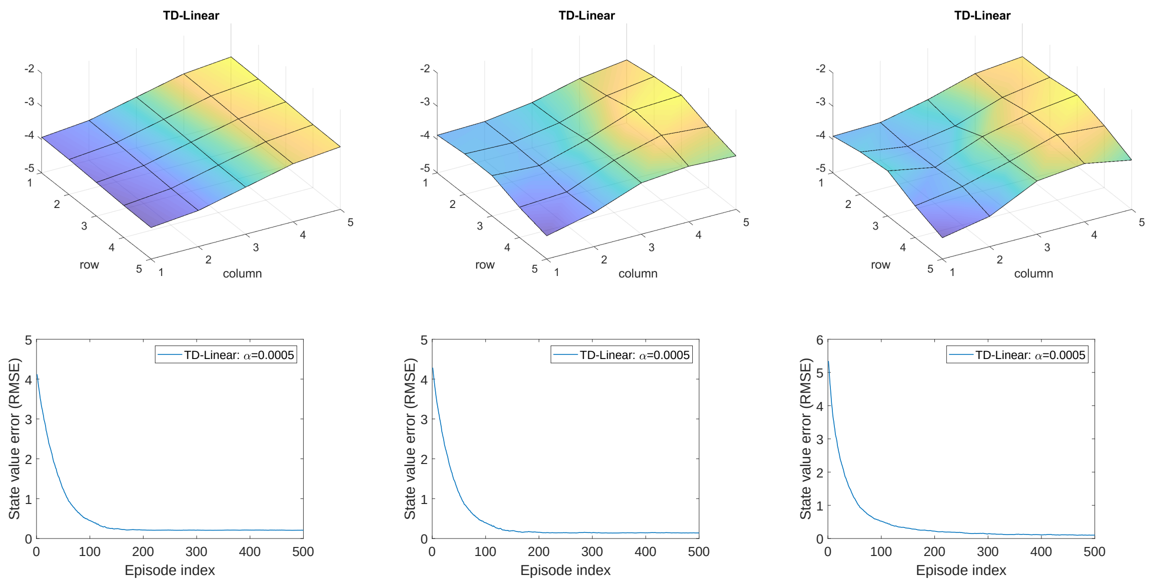
$$\phi(s) = [1,x,y,x^2,y^2,xy,x^3,y^3,x^2y,xy^2]^T \in \mathbb{R}^{10}. \tag{8.10}$$

The estimation results when we use the feature vectors in (8.9) and (8.10) are given in Figure 8.7(b)-(c). As can be seen, the higher the dimension of the feature vector is, the more accurate the state values can be approximated. However, for all three cases, the estimation error cannot converge to zero because these linear approximators still have limited approximation ability.



(a) $\phi(s) \in \mathbb{R}^3$ (b) $\phi(s) \in \mathbb{R}^6$ (c) $\phi(s) \in \mathbb{R}^{10}$

Figure 8.7: TD-Linear estimation results with polynomial features given in (8.8), (8.9), and (8.10).



(a) $q = 1$ and $\phi(s) \in \mathbb{R}^4$ (b) $q = 2$ and $\phi(s) \in \mathbb{R}^9$ (c) $q = 3$ and $\phi(s) \in \mathbb{R}^{16}$

Figure 8.8: TD-Linear estimation results with the Fourier features given in (8.11).

– In addition to polynomial feature vectors, there are many other types of features such as Fourier basis and tile coding. An excellent introduction can be found in [3, Chapter 9]. We next demonstrate how to apply Fourier features to the grid-world example. Fourier features show good performance when applied in Sarsa for searching optimal policies, which will be shown in Section 8.7 later. We normalize x and y of each state to the interval of $[0, 1]$. The feature vector is

$$\phi(s) = \begin{bmatrix} \vdots \\ \cos\big(\pi(c_1 x + c_2 y)\big) \\ \vdots \end{bmatrix} \in \mathbb{R}^{(q+1)^2}. \tag{8.11}$$

Here, π denotes the circumference ratio, which is $3.14\dots$, instead of a policy. More importantly, c_1, c_2 can take any integer in $\{0, 1, \dots, q\}$, where q is a user-selected integer. As a result, there are $(q+1)^2$ possible ways for c_1, c_2 taking values. Hence, the dimension of $\phi(s)$ is $(q+1)^2$. For example, consider the case of $q = 1$. Then, c_1, c_2 take values in $\{0, 1\}$. The feature vector in this case is

$$\phi(s) = \begin{bmatrix} \cos\big(\pi(0x + 0y)\big) \\ \cos\big(\pi(0x + 1y)\big) \\ \cos\big(\pi(1x + 0y)\big) \\ \cos\big(\pi(1x + 1y)\big) \end{bmatrix} = \begin{bmatrix} 1 \\ \cos \pi y \\ \cos \pi x \\ \cos \pi(x + y) \end{bmatrix} \in \mathbb{R}^4.$$

The estimation results when we use the Fourier features with $q = 1, 2, 3$ are shown in Figure 8.8. The dimensions of the feature vectors in the three cases are $4, 9, 16$, respectively. As can be seen, the higher the dimension of the feature vector is, the more accurately the state values can be approximated.

8.6 Theoretical analysis of TD learning

Up to now, we finished the story for introducing TD learning with function approximation. This story started from the objective function in (8.2). To optimize this objective function, we introduced the stochastic algorithm in (8.5). The true value function, which is unknown, in the algorithm is replaced by an approximation, leading to the TD algorithm in (8.6).

Although this story is helpful to understand the basic idea of value function approximation, it is not mathematically rigorous. For example, the algorithm in (8.6) actually does not minimizes the objective function in (8.2) as will be shown in this section.

This section presents a theoretical analysis of the TD algorithm in (8.6) to reveal why the algorithm works effectively and what mathematical problems it solves. Since general nonlinear approximators are difficult to analyze, this section only focuses on the linear cases.

8.6.1 Convergence analysis

To study the convergence of (8.6), we first consider the following deterministic algorithm:

$$w_{t+1} = w_t + \alpha_t \mathbb{E}\Big[\big(r_{t+1} + \gamma \phi^T(s_{t+1}) w_t - \phi^T(s_t) w_t\big)\phi(s_t)\Big], \tag{8.12}$$

where the expectation is with respect to the random variables s_t, s_{t+1}, r_{t+1}. The distribution of s_t is assumed to the stationary distribution d_π. The algorithm in (8.12) is deterministic because the random variables s_t, s_{t+1}, r_{t+1} will all disappear after calculating the expectation.

Why would we consider this deterministic algorithm? First, the convergence of this deterministic algorithm is relatively easier (though nontrivial) to analyze as shown later. Second and more importantly, the convergence of this deterministic algorithm implies the convergence of the stochastic TD algorithm in (8.6). That is because (8.6) can be viewed as a stochastic gradient-descent (SGD) implementation of (8.12). Therefore, we only focus on the convergence of the deterministic algorithm in the rest of this section.

The expression of (8.12) may look complex at first glance. It can be written in a concise expression. To do that, define

$$\Phi = \begin{bmatrix} \vdots \\ \phi^T(s) \\ \vdots \end{bmatrix} \in \mathbb{R}^{|S|\times m}, \quad D = \begin{bmatrix} \ddots & & \\ & d_\pi(s) & \\ & & \ddots \end{bmatrix} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}, \tag{8.13}$$

where Φ is the matrix containing all the feature vectors and D is the diagonal matrix with the stationary distribution in the diagonal entries. The two matrices will be frequently used in this chapter.

Lemma 8.1. *The expectation in* (8.12) *can be written as*

$$\mathbb{E}\Big[\big(r_{t+1} + \gamma \phi^T(s_{t+1}) w_t - \phi^T(s_t) w_t\big)\phi(s_t)\Big] = b - A w_t,$$

where

$$\begin{aligned} A &= \Phi^T D (I - \gamma P_\pi)\Phi \in \mathbb{R}^{m\times m}, \\ b &= \Phi^T D r_\pi \in \mathbb{R}^m. \end{aligned} \tag{8.14}$$

Here, P_π, r_π are the two in the Bellman equation $v_\pi = r_\pi + \gamma P_\pi v_\pi$, and I is the identity matrix of appropriate dimension.

The proof can be found in a shaded box given later.

With the expression in Lemma 8.1, the deterministic algorithm in (8.12) can be rewritten as

$$w_{t+1} = w_t + \alpha_t(b - Aw_t), \tag{8.15}$$

which is a simple iterative algorithm. The convergence properties of this algorithm are analyzed below.

First, what is the eventually converged value of w_t? Hypothetically, if w_t converges to a constant value w^* as $t \to \infty$, then (8.15) implies $w^* = w^* + \alpha_\infty(b - Aw^*)$, which suggests that $b - Aw^* = 0$ and hence

$$w^* = A^{-1}b.$$

Several remarks about this steady-state value are given below.

– Is A invertible? The answer is yes. In fact, A is not only invertible but also positive definite. That is, for any nonzero vector x of appropriate dimension, $x^TAx > 0$. This property is proven in detail in the shaded box given later.

– What is the meaning of $w^* = A^{-1}b$? It is actually the optimal solution for minimizing the *projected Bellman error*, which will be introduced in the next section. One interesting conclusion is that, when the dimension of w is the same as $|\mathcal{S}|$ and $\phi(s_i) = [0, \dots, 1, \dots, 0]^T$ where the entry corresponding to s_i is 1, we have

$$w^* = A^{-1}b = v_\pi.$$

To see that, we have $\Phi = I$ in this case and hence

$$\begin{aligned} A &= \Phi^T D(I - \gamma P_\pi)\Phi = D(I - \gamma P_\pi), \\ b &= \Phi^T D r_\pi = D r_\pi. \end{aligned}$$

Thus

$$w^* = A^{-1}b = (I - \gamma P_\pi)^{-1} D^{-1} D r_\pi = (I - \gamma P_\pi)^{-1} r_\pi = v_\pi.$$

Although it is a special case, the fact that w^* becomes v_π in this case gives us more confidence about the algorithm.

Second, we next that $w_t \to w^* = A^{-1}b$ as $t \to \infty$. In fact, since $w_{t+1} = w_t + \alpha_t(b - Aw_t)$ is a relatively simple algorithm, it can be proved in many ways.

– The first way is to consider $g(w) \doteq b - Aw$. Since w^* is the root of $g(w) = 0$, it is actually a root-finding problem. Such a problem can be solved by the Robbins-Monro (RM) algorithm introduced in Chapter 6. The algorithm $w_{t+1} = w_t + \alpha_t(b - Aw_t)$ is in fact an RM algorithm. The convergence of RM algorithms can shed light on the convergence of $w_{t+1} = w_t + \alpha_t(b - Aw_t)$. That is w_t converges to w^* when $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$.

– The second way is to study the convergence error $\delta_t = w_t - w^*$. We only need to show that δ_t converges to zero. To do that, substituting $w_t = \delta_t + w^*$ into $w_{t+1} = w_t + \alpha_t(b - Aw_t)$ gives

$$\delta_{t+1} = \delta_t - \alpha_t A\delta_t = (I - \alpha_t A)\delta_t.$$

and hence

$$\delta_{t+1} = (I - \alpha_t A)\cdots(I - \alpha_0 A)\delta_0.$$

Consider the simple case where $\alpha_t = \alpha$ for all t. Then, we have

$$\|\delta_{t+1}\| \le \|I - \alpha A\|^{t+1}\|\delta_0\|,$$

where $\|\cdot\|$ denotes the 2-norm of a vector or matrix. When $\alpha > 0$ is sufficiently small, the spectral radius of $I - \alpha A$ is strictly less than one and hence δ_t converges to zero.

Proof of Lemma 8.1

By using conditional expectation, we have

$$\begin{aligned}
&\mathbb{E}\Big[r_{t+1}\phi(s_t) + \phi(s_t)\big(\gamma\phi^T(s_{t+1}) - \phi^T(s_t)\big)w_t\Big] \\
&= \sum_{s\in\mathcal{S}} d_\pi(s)\mathbb{E}\Big[r_{t+1}\phi(s_t) + \phi(s_t)\big(\gamma\phi^T(s_{t+1}) - \phi^T(s_t)\big)w_t \big| s_t = s\Big] \\
&= \sum_{s\in\mathcal{S}} d_\pi(s)\mathbb{E}\Big[r_{t+1}\phi(s_t)\big|s_t = s\Big] + \sum_{s\in\mathcal{S}} d_\pi(s)\mathbb{E}\Big[\phi(s_t)\big(\gamma\phi^T(s_{t+1}) - \phi^T(s_t)\big)w_t\big|s_t = s\Big],
\end{aligned} \tag{8.16}$$

where d_π is the stationary distribution under policy π. Therefore, there is an implicit assumption that s_t obeys the stationary distribution, which requires that 1) the process has run for a sufficiently long time so that stationary distribution has been achieved, and 2) the experience sample used for training is on-policy, which means the sampling must follow the given policy π.

First, consider the first term in (8.16). Since

$$\mathbb{E}\Big[r_{t+1}\phi(s_t)\big|s_t = s\Big] = \phi(s)\mathbb{E}\Big[r_{t+1}\big|s_t = s\Big] = \phi(s)r_\pi(s),$$

where $r_\pi(s) = \sum_a \pi(a|s)\sum_r rp(r|s,a)$, the first term in (8.16) can be written as

$$\sum_{s\in\mathcal{S}} d_\pi(s)\mathbb{E}\Big[r_{t+1}\phi(s_t)\big|s_t = s\Big] = \sum_{s\in\mathcal{S}} d_\pi(s)\phi(s)r_\pi(s) = \Phi^T D r_\pi,$$

where $r_\pi = [\cdots, r_\pi(s), \cdots]^T \in \mathbb{R}^{|\mathcal{S}|}$.

Second, consider the second term in (8.16). Since

$$
\begin{aligned}
&\mathbb{E}\Big[\phi(s_t)\big(\gamma\phi^T(s_{t+1})-\phi^T(s_t)\big)w_t\big|s_t=s\Big]\\
&=-\mathbb{E}\Big[\phi(s_t)\phi^T(s_t)w_t\big|s_t=s\Big]+\mathbb{E}\Big[\gamma\phi(s_t)\phi^T(s_{t+1})w_t\big|s_t=s\Big]\\
&=-\phi(s)\phi^T(s)w_t+\gamma\phi(s)\mathbb{E}\Big[\phi^T(s_{t+1})\big|s_t=s\Big]w_t\\
&=-\phi(s)\phi^T(s)w_t+\gamma\phi(s)\sum_{s'\in\mathcal{S}}p(s'|s)\phi^T(s')w_t,
\end{aligned}
$$

the second term in (8.16) becomes

$$
\begin{aligned}
&\sum_{s\in\mathcal{S}}d_\pi(s)\mathbb{E}\Big[\phi(s_t)\big(\gamma\phi^T(s_{t+1})-\phi^T(s_t)\big)w_t\big|s_t=s\Big]\\
&=\sum_{s\in\mathcal{S}}d_\pi(s)\Big[-\phi(s)\phi^T(s)w_t+\gamma\phi(s)\sum_{s'\in\mathcal{S}}p(s'|s)\phi^T(s')w_t\Big]\\
&=\sum_{s\in\mathcal{S}}d_\pi(s)\phi(s)\Big[-\phi(s)+\gamma\sum_{s'\in\mathcal{S}}p(s'|s)\phi(s')\Big]^T w_t\\
&=(-\Phi^TD\Phi+\gamma\Phi^TDP_\pi\Phi)w_t\\
&=-\Phi^TD(I-\gamma P_\pi)\Phi w_t.
\end{aligned}
$$

Combining the above two terms we have

$$
\mathbb{E}\Big[\big(r_{t+1}+\gamma\phi^T(s_{t+1})w_t-\phi^T(s_t)w_t\big)\phi(s_t)\Big]=\Phi^TDr_\pi-\Phi^TD(I-\gamma P_\pi)\Phi w_t\doteq b-Aw_t \tag{8.17}
$$

where $b=\Phi^TDr_\pi$ and $A=\Phi^TD(I-\gamma P_\pi)\Phi$.

Prove that $A=\Phi^TD(I-\gamma P_\pi)\Phi$ is positive definite.

The matrix A is positive definite if $x^TAx>0$ for any nonzero vector x of appropriate dimension. If A is positive (or negative) definite, then it is denoted as $A\succ 0$ (or $A\prec 0$). Here, $\succ$ and $\prec$ should be differentiated from $>$ and $<$, which indicates an elementwise comparison. Note that A may not be symmetric since P_π may not be symmetric. While positive definite matrices often refer to symmetric matrices, non-symmetric ones can also be positive definite.

To show $A\succ 0$, we first show that

$$
D(I-\gamma P_\pi)\doteq M\succ 0.
$$

It is clear that $M \succ 0$ implies $A \succ 0$ since Φ is a tall matrix with full column rank (if the feature vectors are independent). Note that

$$M = \frac{M + M^T}{2} + \frac{M - M^T}{2}.$$

Since $M - M^T$ is skew-symmetric and hence $x^T(M - M^T)x = 0$ for any x, we know that $M \succ 0$ if and only if $M + M^T \succ 0$.

To show $M + M^T \succ 0$, we apply the fact that strictly diagonal dominant matrices are positive definite [20].

First, we show

$$(M + M^T)\mathbf{1} > 0, \tag{8.18}$$

where $\mathbf{1} = [1, \dots, 1]^T \in \mathbb{R}^{|\mathcal{S}|}$. The proof of (8.18) is given below. Since $P_\pi \mathbf{1} = \mathbf{1}$, we have $M\mathbf{1} = D(I - \gamma P_\pi)\mathbf{1} = D(\mathbf{1} - \gamma \mathbf{1}) = (1 - \gamma)d_\pi$, where $d_\pi = [d_\pi(s_1), \dots, d_\pi(s_n)]^T$. On the other hand, $M^T\mathbf{1} = (I - \gamma P_\pi^T)D\mathbf{1} = (I - \gamma P_\pi^T)d_\pi = (1 - \gamma)d_\pi$, where the last equality is because $P_\pi^T d_\pi = d_\pi$. In summary,

$$(M + M^T)\mathbf{1} = 2(1 - \gamma)d_\pi.$$

Since all the entries of d_π are positive by definition, we have $(M + M^T)\mathbf{1} > 0$.

Second, the element-wise form of (8.18) is

$$\sum_{j=1}^{|\mathcal{S}|}[M + M^T]_{ij} > 0, \quad \text{for all } i$$

which can be further written as

$$[M + M^T]_{ii} + \sum_{j \neq i}[M + M^T]_{ij} > 0,$$

It can be verified that the diagonal entries of M are positive and the off-diagonal entries of M are non-positive. Therefore, the above inequality can be rewritten as

$$\left|[M + M^T]_{ii}\right| > \sum_{j \neq i}\left|[M + M^T]_{ij}\right|, \quad i = 1, \dots, |\mathcal{S}|.$$

The above inequality means that the absolute value of the ith diagonal entry of $M + M^T$ is greater than the sum of the absolute values of the off-diagonal entries in the same row. Thus, $M + M^T$ is strictly diagonal dominant and the proof is complete.

8.6.2 TD learning minimizes the projected Bellman error

While we have shown in the last subsection that the TD algorithm will converge to $w^* = A^{-1}b$, it is still unclear what the interpretation of this solution is. In this section, we show that w^* is the optimal solution minimizing the *projected Bellman error*.

First of all, we revisit the idea of value function approximation as an optimization problem. The objective function introduced in (8.2) is

$$J_E(w) = \mathbb{E}[(v_\pi(S) - \hat{v}(S, w))^2].$$

In a matrix-vector form, $J_E(w)$ can be expressed as

$$J_E(w) = \|\hat{v}(w) - v_\pi\|_D^2,$$

where v_π is the true state value vector, $\hat{v}(w)$ the approximated one, and $\|\cdot\|_D^2$ is the 2-norm weighted by a diagonal matrix D. In particular, $\|x\|_D^2 = x^T D x = \|D^{1/2}x\|_2^2$. Here, D is the matrix with the equilibrium state distribution on the diagonal.

This is one of the first objective functions that we can imagine when talking about function approximation. However, it relies on the true state, which is unknown. To obtain an implementable algorithm, we have to consider other objective functions such as the *Bellman error* and *projected Bellman error* [21–25].

The idea of Bellman error minimization is as follows. Since v_π satisfies the Bellman equation $v_\pi = r_\pi + \gamma P_\pi v_\pi$, it is expected that the approximated one $\hat{v}(w)$ should satisfy this equation as much as possible. Thus, the Bellman error is

$$J_{BE}(w) = \|\hat{v}(w) - (r_\pi + \gamma P_\pi \hat{v}(w))\|_D^2 \doteq \|\hat{v}(w) - T_\pi(\hat{v}(w))\|_D^2,$$

Here, $T_\pi(\cdot)$ is the Bellman operator. In particular, for any vector $x \in \mathbb{R}^{|\mathcal{S}|}$, the Bellman operator is defined as

$$T_\pi(x) \doteq r_\pi + \gamma P_\pi x.$$

Minimizing the Bellman error is a standard least-squares problem. The details of the solution are omitted here.

Notably, $J_{BE}(w)$ may not be minimized to *zero* due to the limited ability of the approximator. By contrast, an objective function that can be minimized to zero is the projected Bellman error:

$$J_{PBE}(w) = \|\hat{v}(w) - MT_\pi(\hat{v}(w))\|_D^2,$$

where $M \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}$ is the orthogonal projection matrix that geometrically projects any vector onto the space of all possible approximations.

In fact, TD learning does *not* optimize this objective function. Instead, it minimizes a *projected Bellman error* as shown below.

While the orthogonal projection may be complex for general nonlinear cases, it is easy to analyze in the linear case where $\hat{v}(w) = \Phi w$. Here, Φ is defined in (8.13). The range space of Φ is the set of all possible linear approximations. Then,

$$M = \Phi(\Phi^T D \Phi)^{-1}\Phi^T D \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}$$

is the projection matrix that geometrically projects any vector onto the range space Φ. Since $\hat{v}(w)$ is in the range of Φ, $J_{PBE}(w)$ can be minimized to zero. It can be proven that the solution minimizing $J_{PBE}(w)$ is $w^* = A^{-1}b$. That is

$$w^* = A^{-1}b = \arg\min_{w} J_{PBE}(w),$$

The proof is given in the shaded box below. Therefore, the TD-Linear algorithm actually minimizes the projected Bellman error rather than (8.2).

Show that $w^* = A^{-1}b$ minimizes $J_{PBE}(w)$

We next show that $w^* = A^{-1}b$ is the solution minimizing $J_{PBE}(w)$. Since $J_{PBE}(w) = 0 \Leftrightarrow \hat{v}(w) - MT_\pi(\hat{v}(w)) = 0$, we only need to study the root of

$$\hat{v}(w) = MT_\pi(\hat{v}(w)).$$

In the linear case, substituting $\hat{v}(w) = \Phi w$ and the expression of M into the above equation gives

$$\Phi w = \Phi(\Phi^T D \Phi)^{-1}\Phi^T D(r_\pi + \gamma P_\pi \Phi w). \tag{8.19}$$

Since Φ has full column rank, we have $\Phi x = \Phi y \Leftrightarrow x = y$ for any x, y. Therefore, (8.19) implies

$$\begin{aligned}
&w = (\Phi^T D \Phi)^{-1}\Phi^T D(r_\pi + \gamma P_\pi \Phi w) \\
&\Longleftrightarrow \Phi^T D(r_\pi + \gamma P_\pi \Phi w) = (\Phi^T D \Phi) w \\
&\Longleftrightarrow \Phi^T D r_\pi + \gamma \Phi^T D P_\pi \Phi w = (\Phi^T D \Phi) w \\
&\Longleftrightarrow \Phi^T D r_\pi = \Phi^T D(I - \gamma P_\pi)\Phi w \\
&\Longleftrightarrow w = (\Phi^T D(I - \gamma P_\pi)\Phi)^{-1}\Phi^T D r_\pi = A^{-1}b = w^*,
\end{aligned}$$

where A, b are given in (8.14).

How far is the solution minimizing the projected Bellman error from the true state value v_π? If we use a linear approximator, the approximated value that minimizes the projected Bellman error is Φw^*. Its difference from the true state value v_π is

$$\|\Phi w^* - v_\pi\|_D \leq \frac{1}{1-\gamma} \min_w \|\hat{v}(w) - v_\pi\|_D = \frac{1}{1-\gamma} \min_w \sqrt{J_E(w)}. \tag{8.20}$$

That is, the difference between Φw^* and v_π is bounded from above by the minimum value of $J_E(w)$. This bound provides more confidence about the TD algorithm with linear value approximation. However, this bound is loose, especially when γ is close to one. It is thus mainly of the theoretical value.

Proof of (8.20)

Note that

$$\begin{aligned}
\|\Phi w^* - v_\pi\|_D &= \|\Phi w^* - Mv_\pi + Mv_\pi - v_\pi\|_D \\
&\leq \|\Phi w^* - Mv_\pi\|_D + \|Mv_\pi - v_\pi\|_D \\
&= \|MT_\pi(\Phi w^*) - MT_\pi(v_\pi)\|_D + \|Mv_\pi - v_\pi\|_D ,
\end{aligned}$$

where the last equality is due to $\Phi w^* = MT_\pi(\Phi w^*)$ and $v_\pi = T_\pi(v_\pi)$. Since

$$MT_\pi(\Phi w^*) - MT_\pi(v_\pi) = M(r_\pi + \gamma P_\pi \Phi w^*) - M(r_\pi + \gamma P_\pi v_\pi) = \gamma M P_\pi(\Phi w^* - v_\pi),$$

we have

$$\begin{aligned}
\|\Phi w^* - v_\pi\|_D &\leq \|\gamma M P_\pi(\Phi w^* - v_\pi)\|_D + \|Mv_\pi - v_\pi\|_D \\
&\leq \gamma \|M\|_D \|P_\pi(\Phi w^* - v_\pi)\|_D + \|Mv_\pi - v_\pi\|_D \\
&= \gamma \|P_\pi(\Phi w^* - v_\pi)\|_D + \|Mv_\pi - v_\pi\|_D \qquad \text{(because } \|M\|_D = 1) \\
&\leq \gamma \|\Phi w^* - v_\pi\|_D + \|Mv_\pi - v_\pi\|_D. \qquad \text{(because } \|P_\pi x\|_D \leq \|x\|_D \text{ for all } x)
\end{aligned}$$

The proof of the facts that $\|M\|_D = 1$ and $\|P_\pi x\|_D \leq \|x\|_D$ are postponed to the end of the shaded box. Recognizing the above inequality gives

$$\begin{aligned}
\|\Phi w^* - v_\pi\|_D &\leq \frac{1}{1-\gamma} \|Mv_\pi - v_\pi\|_D \\
&= \frac{1}{1-\gamma} \min_w \|\hat{v}(w) - v_\pi\|_D,
\end{aligned}$$

where the last equality is because $\|Mv_\pi - v_\pi\|_D$ is the error between v_π and its orthogonal projection into the space of all possible approximations. Therefore, it is the minimum value of the error between v_π and any $\hat{v}(w)$.

We next prove some useful facts, which have already been used in the above proof.

– Properties of matrix weighted norms. By definition, $\|x\|_D = \sqrt{x^T D x} = \|D^{1/2} x\|_2$.

The induced matrix norm is $\|A\|_D = \max_{x\neq 0} \|Ax\|_D/\|x\|_D = \|D^{1/2}AD^{-1/2}\|_2$. For matrices A, B of appropriate dimensions, we have $\|ABx\|_D \leq \|A\|_D\|B\|_D\|x\|_D$. To see that, $\|ABx\|_D = \|D^{1/2}ABx\|_2 = \|D^{1/2}AD^{-1/2}D^{1/2}BD^{-1/2}D^{1/2}x\|_2 \leq \|D^{1/2}AD^{-1/2}\|_2\|D^{1/2}BD^{-1/2}\|_2\|D^{1/2}x\|_2 = \|A\|_D\|B\|_D\|x\|_D$.

– Proof of $\|M\|_D = 1$. That is because $\|M\|_D = \|\Phi(\Phi^T D\Phi)^{-1}\Phi^T D\|_D = \|D^{1/2}\Phi(\Phi^T D\Phi)^{-1}\Phi^T DD^{-1/2}\|_2 = 1$, where the last equality is because the matrix in the 2-norm is an orthogonal projection matrix and the 2-norm of any orthogonal projection matrices is equal to one.

– Proof of $\|P_\pi x\|_D \leq \|x\|_D$ any $x \in \mathbb{R}^{|\mathcal{S}|}$. First,

$$\|P_\pi x\|_D^2 = x^T P_\pi^T D P_\pi x = \sum_{i,j} x_i [P_\pi^T D P_\pi]_{ij} x_j = \sum_{i,j} x_i \left(\sum_k [P_\pi^T]_{ik}[D]_{kk}[P_\pi]_{kj}\right) x_j.$$

Reorganizing the above equation gives

$$\begin{aligned}
\|P_\pi x\|_D^2 &= \sum_k [D]_{kk}\Big(\sum_i [P_\pi]_{ki}x_i\Big)^2 \\
&\leq \sum_k [D]_{kk}\Big(\sum_i [P_\pi]_{ki}x_i^2\Big) \qquad \text{(due to Jensen's inequality)} \\
&= \sum_i \Big(\sum_k [D]_{kk}[P_\pi]_{ki}\Big)x_i^2 \\
&= \sum_i [D]_{ii}x_i^2 \qquad (\text{due to } d_\pi^T P_\pi = d_\pi^T) \\
&= \|x\|_D^2.
\end{aligned}$$

8.6.3 Least-squares TD

This section introduces an algorithm called *least-squares TD* (LSTD) [26]. It is another algorithm that minimizes the projected Bellman error.

Recall that the optimal parameter to minimize the projected Bellman error is $w^* = A^{-1}b$, where $A = \Phi^T D(I - \gamma P_\pi)\Phi$ and $b = \Phi^T D r_\pi$. In fact, it follows from (8.17) that A and b can also be written as

$$\begin{aligned}
A &= \mathbb{E}\Big[\phi(s_t)\big(\phi(s_t) - \gamma\phi(s_{t+1})\big)^T\Big], \\
b &= \mathbb{E}\Big[r_{t+1}\phi(s_t)\Big].
\end{aligned}$$

The above two equations show that A and b are expectations of some random variables. The *idea* of LSTD is simple: if we can use random samples to directly obtain the estimates of A and b, denoted as $\hat{A}$ and $\hat{b}$, then the optimal parameter can be directly estimated as

$w^* \approx \hat{A}^{-1}\hat{b}$.

In particular, suppose that $(s_0, r_1, s_1, \dots, r_t, s_t, \dots)$ is a trajectory obtained by following the policy. Let $\hat{A}_t$ and $\hat{b}_t$ be the estimates of A and b, respectively, at time t. They are calculated as

$$\hat{A}_t = \sum_{k=0}^{t-1} \phi(s_k)\big(\phi(s_k) - \gamma\phi(s_{k+1})\big)^T,$$
$$\hat{b}_t = \sum_{k=0}^{t-1} r_{k+1}\phi(s_k).$$

Then, the estimated parameter is

$$w_t = \hat{A}_t^{-1}\hat{b}_t.$$

Since $\hat{A}_t$ is required to be invertible, $\hat{A}_t$ is usually biased by a small constant matrix δI where I is the identity matrix and δ is a small positive number.

The *advantage* of LSTD is that it uses experience samples more efficiently and converges faster than the purely TD method. Why is that? It is simply because this algorithm is specifically designed based on the expression of the optimal solution. The better we understand a problem, the better algorithms we usually can design.

The *disadvantages* of LSTD are as follows. First, it can only estimate the state values and it is merely applicable to the linear case. While the TD algorithm can be extended to estimating action values as shown in the next section, LSTD can not. Moreover, while the TD algorithm allows nonlinear approximators, LSTD does not. That is because this algorithm is designed specifically based on the expression of w^*. Second, it is the computational cost of LSTD is higher than TD at each update step since LSTD updates an $m \times m$ matrix whereas TD updates an m-dimensional vector. More importantly, at every step, LSTD needs to compute the inverse of $\hat{A}_t$, whose computational complexity is $O(m^3)$. The common method to resolve this problem is to update the inverse of $\hat{A}_t$ directly rather than updating $\hat{A}_t$. In particular, $\hat{A}_{t+1}$ can be calculated recursively as

$$\begin{aligned}
\hat{A}_{t+1} &= \sum_{k=0}^{t} \phi(s_k)\big(\phi(s_k) - \gamma\phi(s_{k+1})\big)^T \\
&= \sum_{k=0}^{t-1} \phi(s_k)\big(\phi(s_k) - \gamma\phi(s_{k+1})\big)^T + \phi(s_t)\big(\phi(s_t) - \gamma\phi(s_{t+1})\big)^T \\
&= \hat{A}_t + \phi(s_t)\big(\phi(s_t) - \gamma\phi(s_{t+1})\big)^T.
\end{aligned}$$

The above expression decomposes $\hat{A}_{t+1}$ into the sum of two matrices. Its inverse can be calculated as [27]

$$\begin{aligned}\hat{A}_{t+1}^{-1} &= \Big(\hat{A}_t + \phi(s_t)\big(\phi(s_t) - \gamma\phi(s_{t+1})\big)^T\Big)^{-1} \\ &= \hat{A}_t^{-1} + \frac{\hat{A}_t^{-1}\phi(s_t)\big(\phi(s_t) - \gamma\phi(s_{t+1})\big)^T\hat{A}_t^{-1}}{1 + \big(\phi(s_t) - \gamma\phi(s_{t+1})\big)^T\hat{A}_t^{-1}\phi(s_t)}.\end{aligned}$$

Therefore, we can directly store and update $\hat{A}_t^{-1}$ to avoid the need to calculate the matrix inverse. The initial value of such recursive algorithm can be selected as $\hat{A}_0^{-1} = \sigma I$, where σ is a positive number. This recursive algorithm does not require setting the step size. However, it requires setting the initial value of $\hat{A}_0^{-1}$. A good tutorial on recursive least squares can be found in [28].

8.7 Sarsa with function approximation

So far in this chapter, we merely considered the problem of state value estimation. To search for optimal policies, we need to estimate action values. This section introduces how to estimate action values using Sarsa in the presence of value function approximation.

There are two steps in every iteration of Sarsa with function approximation. The first is policy evaluation, which is to estimate the action values. The second is policy improvement, which is to find the soft greedy policy based on the current action values. Its difference from tabular Sarsa lies only in the first step for action value estimation. In particular, the action value $q_\pi(s,a)$ is described by a function $\hat{q}(s,a,w)$ parameterized by w. Replacing $\hat{v}(s,w)$ by $\hat{q}(s,a,w)$ in (8.6) gives the following update rule of Sarsa learning with furcation approximation:

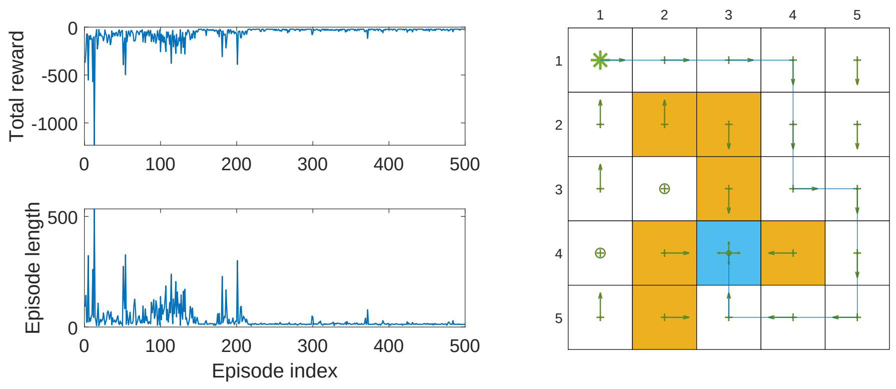
$$w_{t+1} = w_t + \alpha_t\Big[r_{t+1} + \gamma\hat{q}(s_{t+1},a_{t+1},w_t) - \hat{q}(s_t,a_t,w_t)\Big]\nabla_w\hat{q}(s_t,a_t,w_t). \tag{8.21}$$

The analysis of (8.21) is also similar to (8.6) and omitted here. When linear functions are used, we have

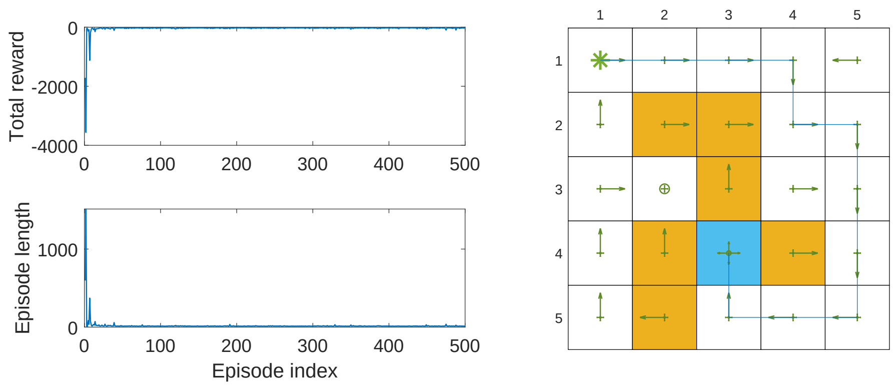
$$\hat{q}(s,a,w) = w^T\phi(s,a),$$

where $\phi(s,a)$ is a feature vector. In this case, $\nabla_w\hat{q}(s,a,w) = \phi(s,a)$. The implementation details can be found in the pseudocode of Sarsa with function approximation. It should be noted that accurately estimating the action values of a given policy needs to run (8.21) sufficiently many times. However, (8.21) is merely executed once before switching to the policy improvement step. This follows the exactly same idea as the tabular Sarsa algorithm. Moreover, the pseudocode aims to solve a task where the policy should lead the agent to the target state starting from a specific state. As a result, the algorithm may not be able to find the optimal policy for every state. Of course, if more experience data is available, the optimal policy for every state can also be obtained.

An illustrative example is given in Figure 8.9(a). In this example, the task is to find a good policy that can lead the agent to the target starting from the top-left state. As can be seen, both the total reward and length of the trajectory gradually converge to steady values. In this example, the linear Fourier function bases of order 5 are used.



(a) Sarsa with linear function approximation.



(b) Q-learning with linear function approximation

Figure 8.9: Sarsa and Q-learning with linear function approximation. Here, $\gamma = 0.9$, $\epsilon = 0.1$, $r_{\text{boundary}} = r_{\text{forbidden}} = -10$, $r_{\text{target}} = 1$, $\alpha = 0.001$.

8.8 Q-learning with function approximation

Similar to Sarsa, tabular Q-learning can also be extended to the case of value function approximation. The update rule is

$$w_{t+1} = w_t + \alpha_t \Big[r_{t+1} + \gamma \max_{a \in \mathcal{A}(s_{t+1})} \hat{q}(s_{t+1}, a, w_t) - \hat{q}(s_t, a_t, w_t) \Big] \nabla_w \hat{q}(s_t, a_t, w_t), \qquad (8.22)$$

which is the same as Sarsa in (8.21) except that $\hat{q}(s_{t+1}, a_{t+1}, w_t)$ in (8.21) is replaced by $\max_{a \in \mathcal{A}(s_{t+1})} \hat{q}(s_{t+1}, a, w_t)$.

Similar to the tabular case, (8.22) can be implemented in either on-policy or off-policy fashion (see the pseudocode). An example demonstrating the on-policy version is given in Figure 8.9(b). In this example, the task is to find a good policy that can lead the agent

Pseudocode: Sarsa with function approximation

```
Initialization: Initial parameter vector $w_0$. Initial policy $\pi_0$.
Aim: Search a policy that can lead the agent to the target from an initial state-action pair $(s_0, a_0)$.

For each episode, do
    If the current $s_t$ is not the target state, do
        Take action $a_t$ following $\pi_t(s_t)$, generate $r_{t+1}, s_{t+1}$, and then take action $a_{t+1}$ following $\pi_t(s_{t+1})$
        Update parameter:
        $w_{t+1} = w_t + \alpha_t\Big[r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, w_t) - \hat{q}(s_t, a_t, w_t)\Big]\nabla_w \hat{q}(s_t, a_t, w_t)$
        Update policy:
        $\pi_{t+1}(a|s_t) = 1 - \frac{\varepsilon}{|\mathcal{A}(s)|}(|\mathcal{A}(s)| - 1)$ if $a = \arg\max_{a \in \mathcal{A}(s_t)} \hat{q}(s_t, a, w_{t+1})$
        $\pi_{t+1}(a|s_t) = \frac{\varepsilon}{|\mathcal{A}(s)|}$ otherwise
```

Pseudocode: Q-learning with function approximation (on-policy version)

```
Initialization: Initial parameter vector $w_0$. Initial policy $\pi_0$. Small $\varepsilon > 0$.
Aim: Search a good policy that can lead the agent to the target from an initial state-action pair $(s_0, a_0)$.

For each episode, do
    If the current $s_t$ is not the target state, do
        Take action $a_t$ following $\pi_t(s_t)$, and generate $r_{t+1}, s_{t+1}$
        Update parameter:
        $w_{t+1} = w_t + \alpha_t\Big[r_{t+1} + \gamma \max_{a \in \mathcal{A}(s_{t+1})} \hat{q}(s_{t+1}, a, w_t) - \hat{q}(s_t, a_t, w_t)\Big]\nabla_w \hat{q}(s_t, a_t, w_t)$
        Update policy:
        $\pi_{t+1}(a|s_t) = 1 - \frac{\varepsilon}{|\mathcal{A}(s)|}(|\mathcal{A}(s)| - 1)$ if $a = \arg\max_{a \in \mathcal{A}(s_t)} \hat{q}(s_t, a, w_{t+1})$
        $\pi_{t+1}(a|s_t) = \frac{\varepsilon}{|\mathcal{A}(s)|}$ otherwise
```

to the target state from the top-left state. As can be seen, Q-learning with linear function approximation can successfully complete the task. Here, linear Fourier basis functions of order 5 are used. The off-policy version will be demonstrated when we introduce deep Q-learning.

8.9 Deep Q-learning

We can introduce deep neural networks into Q-learning to obtain *deep Q-learning* or *deep Q-network* (DQN) [29–31]. Deep Q-learning is one of the earliest algorithms that introduce deep neural networks into RL. Of course, the neural network does not have to be deep. For many tasks including our grid-world examples, a shallow network with one hidden layer is already sufficient. We, however, always refer to the algorithm as deep Q-learning regardless of the depth of the network.

The idea of deep Q-learning is similar to (8.22). However, the mathematical formulation and implementation are substantially different.

8.9.1 Algorithm description

Mathematically, deep Q-learning aims to minimize the objective function:

$$J = \mathbb{E}\left[\left(R + \gamma \max_{a\in\mathcal{A}(S')} \hat{q}(S', a, w) - \hat{q}(S, A, w)\right)^2\right], \tag{8.23}$$

where (S, A, R, S) are random variables representing a state, an action taken at that state, the immediate reward, and the next state. This objective function can be viewed as the *Bellman optimality error*. That is because

$$q(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a\in\mathcal{A}(S_{t+1})} q(S_{t+1}, a)\middle|S_t = s, A_t = a\right], \quad \text{for all } s, a$$

is the Bellman optimality equation. The proof was given in the last chapter when we introduce the Tabular Q-learning algorithm. Therefore, the value of $R+\gamma \max_{a\in\mathcal{A}(S')} \hat{q}(S', a, w)-\hat{q}(S, A, w)$ should be zero in the expectation sense if $\hat{q}$ can accurately approximate the optimal action values.

To minimize the objective function in (8.23), we need to calculate its gradient with respect to w. It is noted that the parameter w not only appears in $\hat{q}(S, A, w)$ but also in $y \doteq R + \gamma \max_{a\in\mathcal{A}(S')} \hat{q}(S', a, w)$. For the sake of simplicity, we can assume that w in y is fixed (at least for a while) when we calculate the gradient. To do that, we can introduce two networks. One is a *main network* representing $\hat{q}(s, a, w)$ and the other is a *target* *network* $\hat{q}(s, a, w_T)$. The objective function in this case degenerates to

$$J = \mathbb{E}\left[\left(R + \gamma \max_{a\in\mathcal{A}(S')} \hat{q}(S', a, w_T) - \hat{q}(S, A, w)\right)^2\right],$$

where w_T is the target network parameter. When w_T is fixed, the gradient of J is

$$\nabla_w J = \mathbb{E}\left[\left(R + \gamma \max_{a\in\mathcal{A}(S')} \hat{q}(S', a, w_T) - \hat{q}(S, A, w)\right)\nabla_w \hat{q}(S, A, w)\right]. \tag{8.24}$$

The basic idea of deep Q-learning is to use the gradient in (8.24) to minimize the objective function in (8.23). However, such an optimization process evolves some important techniques that deserve special attention.

1) One technique is *experience replay* [29, 30, 32]. That is, after we have collected some experience samples, we do not use these samples in the order they were collected. Instead, we store them in a data set, called *replay buffer*, and draw a batch of samples randomly to train the neural network. In particular, let (s, a, r, s') be an experience sample and $\mathcal{B} \doteq \{(s, a, r, s')\}$ be the replay buffer. Every time we train the neural network, we can draw a mini-batch of random samples from the reply buffer. The draw of samples, or called *experience replay*, should follow a *uniform distribution*.

   Why is experience replay necessary in deep Q-learning and why does the reply must follow a uniform distribution? The answers lie in the objective function in (8.23).

   First, to well define the objective function, we must specify the probability distributions for S, A, R, S'. The distributions of R and S' are determined by the system model. The tricky part is the distributions of S and A. Suppose the experience samples are collected by following a behavior policy π_b. Then, the distribution of A is $A \sim \pi_b(S)$. However, if we treat the state-action pair (S, A) as a single instead of two random variables, then we can *remove the dependence* of the sample (S, A, R, S) on π_b. That is because, once (S, A) is given, R and S are purely determined by the system model. The distribution of the state-action pair (S, A) is assumed to be *uniform*.

   Second, although the state-action samples are assumed to be uniformly distributed, they are *not* in practice because they are generated consequently by the behavior policy. To break the correlation between consequent samples to satisfy the assumption of uniform distribution, we can use the experience replay technique by uniformly drawing samples from the reply buffer. This is the mathematical reason why experience replay is necessary and why the experience reply must be uniform. A benefit of random sampling is that each experience sample may be used multiple times, which can increase the data efficiency. This is especially important when we have a limited amount of data.

2) Another technique is to use two networks, a main network and a target network. The

Pseudocode: Deep Q-learning (off-policy version)

```
Initialization: A main network and a target network with the same initial parameter.
Aim: Learn an optimal target network to approximate the optimal action values from the
experience samples generated by a behavior policy $\pi_b$.

Store the experience samples generated by $\pi_b$ in a reply buffer $\mathcal{B} = \{(s, a, r, s')\}$
    For each iteration, do
        Uniformly draw a mini-batch of samples from $\mathcal{B}$
        For each sample $(s, a, r, s')$, calculate the target value as $y_T = r + \gamma \max_{a \in \mathcal{A}(s')} \hat{q}(s', a, w_T)$, where $w_T$ is the parameter of the target network
        Update the main network to minimize $(y_T - \hat{q}(s, a, w))^2$ using the mini-batch $\{(s, a, y_T)\}$
    Set $w_T = w$ every $C$ iterations
```

reason has been explained when we calculate the gradient in (8.24). The implementation details are clarified below. Let w and w_T denote the parameters of the main and target networks, respectively. They are set to be the same initially.

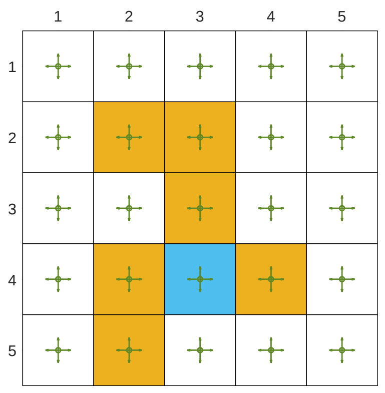
In every iteration, we draw a mini-batch of samples $\{(s, a, r, s')\}$ from the reply buffer. The inputs of the networks include state s and action a. The target output is $y_T \doteq r + \gamma \max_{a \in \mathcal{A}(s')} \hat{q}(s', a, w_T)$. Then, we directly minimize the TD error or called loss function $(y - \hat{q}(s, a, w))^2$ over the mini-batch $\{(s, a, y_T)\}$ instead of a single sample to improve efficiency and stability. Note that, when updating the main network parameter, the target output is calculated based on the target network.

The parameter of the main network is updated in every iteration. By contrast, the target network is set to be the same as the main network every a certain number of iterations to meet the assumption that w_T is fixed when calculating the gradient in (8.24). In addition, we do not bother to explicitly use the gradient in (8.24) to update the main network. Instead, we rely on the built-in gradient calculation methods of neural network software packages. Finally, more theoretical discussion on deep Q-learning can be found in [31].

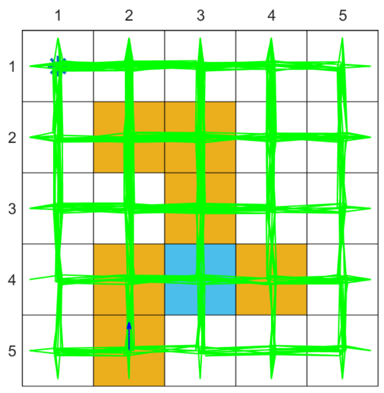
8.9.2 Illustrative examples

An example is given in Figure 8.10 to illustrate deep Q-learning (see the implementation in the pseudocode). This example aims to learn optimal action values for every state-action pair. Once the optimal action values are obtained, the optimal greedy policy can be obtained immediately. One single episode is used to train the network. This episode is generated by a behavior policy shown in Figure 8.10(a). This behavior policy is exploratory in the sense that it takes any action at any state with the same probability.

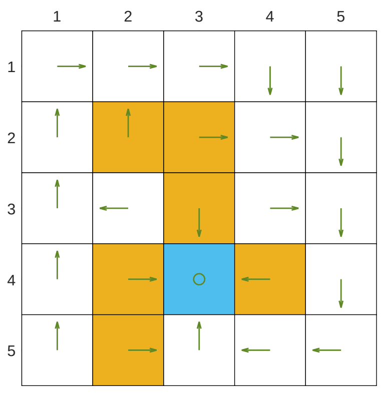
If we use the tabular Q-learning algorithm, we need a very long episode of, for example, 100,000 steps. If we use deep Q-learning, we only need a short episode of, for example, 1,000 steps. This short episode is visualized in Figure 8.10(b). Although there are only 1,000 steps, almost all the state action pairs are visited in this episode due to the strong exploration ability of the behavior policy. Nevertheless, some state-action pairs are visited only a few times.



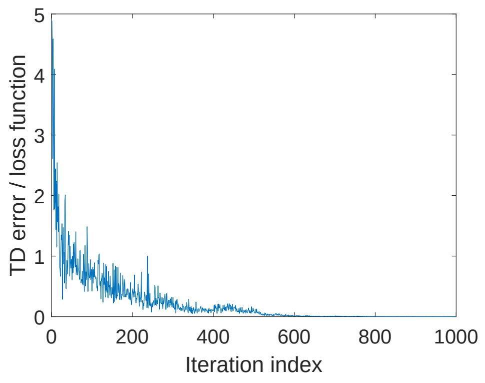
(a) The behavior policy.



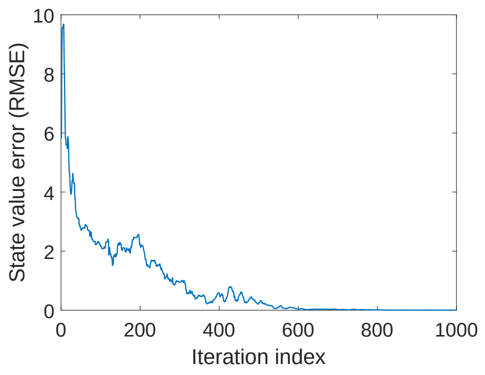
(b) An episode of 1,000 steps.



(c) The finally searched policy.



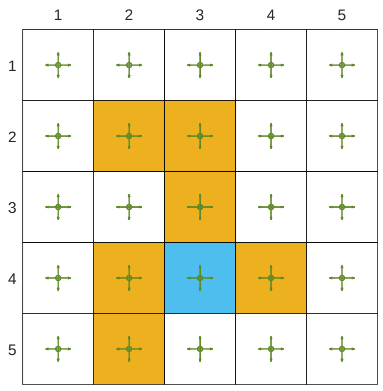
(d) The TD error converges to zero.



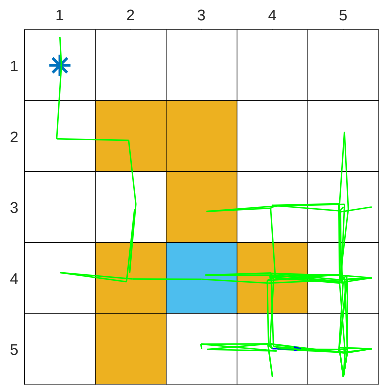
(e) The state estimation error converges to zero.

Figure 8.10: Optimal policy searching by deep Q-learning. Here, $\gamma = 0.9$, $r_{\text{boundary}} = r_{\text{forbidden}} = -10$, $r_{\text{target}} = 1$. The batch size is 100.

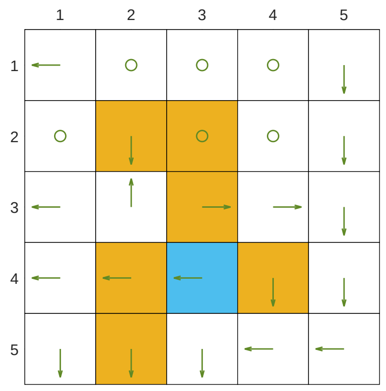
In this example, a shallow neural network with one single hidden layer is used as a nonlinear approximator of $\hat{q}(s, a, w)$. The hidden layer has 100 neurons. The neural network has three inputs and one output. The first two inputs are the normalized row and normalized column of a state. The third input is the normalized action value. Here, the normalization means converting any value to the interval of [0,1]. The output of the network is the predicted action value. The reason that we input the row and column of a state into the network rather than its index is that we know that a state corresponds to a two-dimensional location in the grid. The more information about the state we use when designing the network, the better the network can perform. Of course, the neural network can also be designed in other ways. For example, it can have two inputs and five outputs, where the two inputs are the normalized row and column of a state and the outputs are the five approximated action values for the input state.



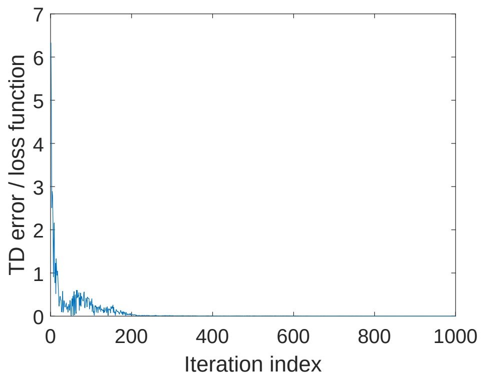
(a) The behavior policy.



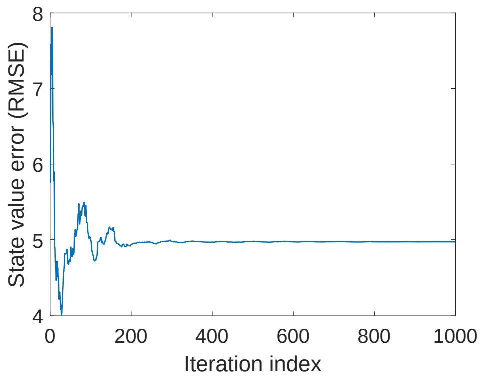
(b) An episode of 100 steps.



(c) The finally searched policy.



(d) The TD error converges to zero.



(e) The state estimation error converges to zero.

Figure 8.11: Optimal policy searching by deep Q-learning. Here, $\gamma = 0.9$, $r_{\text{boundary}} = r_{\text{forbidden}} = -10$, $r_{\text{target}} = 1$. The batch size is 50.

The mini-batch size is 100 samples. The training process is show in Figure 8.10(d)-(e). As can be seen, the loss function, defined as the average TD error of each batch, converges to zero, meaning the network is well-trained eventually. The state value estimation error also converges to zero, indicating that the action values are well approximated. The finally obtained policy is optimal.

This example validates the high efficiency of deep Q-learning. Although we only have a short episode of 1,000 steps, the sample efficiency is high. One reason is that the experience samples can be repeatedly used. Another reason is the method based on function approximation has a strong generalization ability.

We can challenge the algorithm by deliberately considering even fewer experience samples. For example, Figure 8.11 shows the results when we are given an episode of merely 100 steps. In this case, although the network can still be well trained in the sense that the loss function converges to zero, the state estimation error cannot converge to zero. It is not surprising because 100 experience samples are too few to obtain optimal action values.

8.10 Summary

This chapter introduces the method of RL based on value function approximation. The key to understanding this method is to understand the objective functions and optimization algorithm. Unlike the tabular cases, here a policy is optimal if it can minimize certain scalar objective functions. The simplest objective function is the squared error between true state values and the estimated ones. There are also other objective functions such as the Bellman error and the projected Bellman error. We showed that the TD-Linear algorithm actually minimizes the projected Bellman error. Several optimization algorithms such as Sarsa and Q-learning with value approximation were introduced and demonstrated by examples.

Value function approximation is important because it allows artificial neural networks to enter the field of RL. Although neural networks are widely used nowadays as nonlinear function approximators, this chapter mainly focused on the case of linear functions to study fundamental theoretical properties. Fully understanding the linear case is necessary for us to appropriately use nonlinear approximators. Interested readers may refer to [19] for a thorough analysis of algorithms of TD learning with function approximation.

An important concept named stationary distribution emerges in this chapter when we try to define objective functions for optimization. This distribution plays an important role in the convergence analysis. The convergence analysis reveals the importance of on-policy sampling. In particular, it is required that the states are sampled following the stationary distribution of the given policy. If this condition is not satisfied, convergence may not be guaranteed [19]. The stationary distribution also plays an important role in the next chapter where we will use functions to approximate policies instead of values. Stationary distribution is a fundamental property of Markov processes. An excellent introduction to this topic can be found in [18, Chapter IV]. The contents of this chapter heavily rely on matrix analysis. Many results were simply used without detailed explanation. Excellent references on matrix analysis and linear algebra include [20, 33].

8.11 Q&A

- Q: What is the fundamental difference between the tabular representation of values and function approximation?

  A: The fundamental difference is how the optimal policies are defined. For the tabular representation, a policy is optimal if it has greater state values than any other policy. For the value function approximation method, the optimal policy is to minimize a scalar objective function.

- Q: Why do we need to study stationary distribution?

  A: That is because it is a necessary concept to define a valid objective function such as (8.2). In particular, the random variable in the objective function is the state S, which must obey a probability distribution. This probability distribution is usually set as the stationary distribution of the given policy because it can reflect the frequencies the states are visited under the given policy.

– Q: Why do we need to study linear function approximators while nonlinear approximators such as artificial neural networks are widely used nowadays?

  A: Linear function approximation is the simplest case. The theoretical properties in this case can be thoroughly analyzed. By contrast, the nonlinear cases are difficult to analyze. Analyzing the linear cases can certainly help us better understand and use nonlinear approximators.

– Q: Why does deep Q-learning require experience replay?

  A: The fundamental reason is due to the definition of the scalar objective function, which assumes the state-action samples obey a uniform distribution. By contrast, the tabular Q-learning algorithm does not require experience replay. It is, therefore, important to understand the mathematics behind an algorithm in order to use it properly.

Chapter 9

Policy Gradient Methods

The idea of function approximation can be applied to represent not only state/action values but also policies. Up to now in this book, policies have been represented by tables: the action probabilities of all states are stored in a table $\pi(a|s)$, each entry of which is indexed by a state and an action. In this chapter, we show that policies can be represented by parameterized functions denoted as $\pi(a|s, \theta)$, where $\theta \in \mathbb{R}^m$ is a parameter vector. The function representation is also sometimes written as $\pi(a, s, \theta)$, $\pi_\theta(a|s)$, or $\pi_\theta(a, s)$.

When policies are represented as a function, optimal policies can be found by optimizing certain scalar metrics. Such kind of method is called *policy gradient*. Policy gradient is a big step forward in this book because it is *policy-based*. By contrast, all the previous chapters in this book consider *value-based* methods that must estimate state/action values to obtain optimal policies.

The advantages of the policy gradient methods are numerous. For example, when the state or action space is large, the tabular representation will be of low efficiency in terms of storage and policy searching. As a comparison, the dimension of the parameter in a function representation may be significantly less than the number of states. It can also handle continuous state and action spaces.

9.1 Basic idea of policy gradient

To use a parameterized function to represent a policy, we need to first answer some basic questions.

First, how to define optimal policies? When represented as a table, a policy π is defined as optimal if it can maximize *every state value*. When represented by a function, a policy π is fully determined by θ together with the function structure. The policy is defined as optimal if it can maximize certain *scalar metrics*. This is one important difference between tabular and function representations.

Second, how to update policies? When represented by a table, a policy π can be updated by directly changing the entries in the table. However, when represented by a parameterized function, a policy π cannot be updated in this way anymore. Instead, it can only be improved by updating *the parameter* θ. This is an important difference in terms of ways of updating policies.

The basic idea of the policy gradient method is summarized below. Suppose $J(\theta)$ is a scalar metric to define optimal policies. Optimal policies can be obtained by optimizing the metric based on gradient-based algorithms:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t),$$

where $\nabla_\theta J$ is the gradient of J with respect to θ, t is the time step, and α is the optimization rate. A review of gradient-based algorithms is given in Appendix B.

Although the idea of policy gradient is straightforward, the complication emerges when we try to answer the following questions.

– What appropriate metrics should be used? (Section 9.2).

– How to calculate the gradients of the metrics? (Section 9.3)

– How to use experience samples to calculate the gradients? (Section 9.4)

These questions will be answered in detail in the rest of this chapter.

9.2 Metrics to define optimal policies

If a policy is represented by a function, what metrics should we use to define optimality? We next present some popular metrics.

1) The first metric is the *average state value* or simply called *average value*. In particular, let

$$\begin{aligned} v_\pi &= [\dots, v_\pi(s), \dots]^T \in \mathbb{R}^{|\mathcal{S}|} \\ d_\pi &= [\dots, d_\pi(s), \dots]^T \in \mathbb{R}^{|\mathcal{S}|} \end{aligned}$$

be the vector of state values and a vector of weights, respectively. Here, $d_\pi(s) \geq 0$ is the weight for state s and satisfies $\sum_s d_\pi(s) = 1$. The metric of average value is defined as

$$\begin{aligned} \bar{v}_\pi &\doteq d_\pi^T v_\pi \\ &= \sum_s d_\pi(s) v_\pi(s) \\ &= \mathbb{E}[v_\pi(S)], \end{aligned}$$

where $S \sim d_\pi$. As its name suggests, $\bar{v}_\pi$ is simply a weighted average of the state values.

How to select the weights $d_\pi(s)$? One trivial way is to treat all the states equally important and hence select $d_\pi(s) = 1/|\mathcal{S}|$ for every s. Another way, which is often used, is to select $d_\pi(s)$ as the *stationary distribution* satisfying

$$d_\pi^T P_\pi = d_\pi^T,$$

where P_π is the state transition probability matrix. The interpretation of selecting d_π is as follows. The stationary distribution reflects the long-term behavior of the Markov decision process under the given policy. If one state is frequently visited in the long run, it is more important and deserves more weight; if a state is hardly visited, then we give it less weight. Details of stationary distribution can be found in Section 8.2 in the last chapter.

2) The second metric is the *average one-step reward* or simply called *average reward* [2,34,35]. In particular, let

$$r_\pi = [\dots, r_\pi(s), \dots]^T \in \mathbb{R}^{|\mathcal{S}|}$$

be the vector of one-step immediate rewards. Here,

$$r_\pi(s) = \sum_a \pi(a|s) r(s,a)$$

is the average of the one-step immediate reward that can be obtained starting from state s, and $r(s,a) = \mathbb{E}[R|s,a] = \sum_r r p(r|s,a)$ is the average of the one-step immediate reward that can be obtained after taking action a at state s. Then, the metric is defined as

$$\begin{aligned}\bar{r}_\pi &\doteq d_\pi^T r_\pi \\ &= \sum_s d_\pi(s) r_\pi(s) \\ &= \mathbb{E}[r_\pi(S)], \end{aligned} \tag{9.1}$$

where $S \sim d_\pi$. As its name suggests, $\bar{r}_\pi$ is simply a weighted average of the one-step immediate rewards. Here, the weight d_π is the stationary distribution.

3) The third metric is the state value of a specific stating state $v_\pi(s_0)$ [3,34]. For some tasks, we can only start from a specific state s_0. In this case, we only care about the long-term return starting from s_0. The third metric can also be viewed as a weighted average of the state values. To see that,

$$v_\pi(s_0) = \sum_{s \in \mathcal{S}} d_0(s) v_\pi(s),$$

where

$$d_0(s = s_0) = 1, \quad d_0(s \neq s_0) = 0.$$

While we have introduced the definitions of the metrics above, we next give some important remarks about these metrics.

– All these metrics are functions of π. Since π is parameterized by θ, these metrics are functions of θ. In other words, different values of θ can generate different metric values. Therefore, we can search for the optimal values of θ to maximize these metrics. This is the basic idea of policy gradient methods.

– Intuitively, $\bar{r}_\pi$ is more short-sighted because it merely considers the immediate rewards, whereas $\bar{v}_\pi$ considers the total reward overall steps. However, the two metrics are equivalent to each other. In the discounted case where $\gamma < 1$, it will be shown in Lemma 9.1 that

$$\bar{r}_\pi = (1-\gamma)\bar{v}_\pi.$$

– The average reward $\bar{r}_\pi$ has another important definition that is equivalent to 9.1. In particular, suppose an agent follows a given policy and generate a trajectory with the rewards as $(R_{t+1}, R_{t+2}, \dots)$. Then, the average single-step reward along this trajectory is defined as

$$\bar{r}_\pi \doteq \lim_{n\to\infty} \frac{1}{n}\mathbb{E}\Big[R_{t+1} + R_{t+2} + \cdots + R_{t+n}\Big|S_t = s_0\Big] = \lim_{n\to\infty} \frac{1}{n}\mathbb{E}\left[\sum_{k=1}^{n} R_{t+k}\Big|S_t = s_0\right], \tag{9.2}$$

where s_0 is the starting state of the trajectory. An important property is that

$$\lim_{n\to\infty} \frac{1}{n}\mathbb{E}\left[\sum_{k=1}^{n} R_{t+k}\Big|S_t = s_0\right] = \lim_{n\to\infty} \frac{1}{n}\mathbb{E}\left[\sum_{k=1}^{n} R_{t+k}\right] = \sum_s d_\pi(s) r_\pi(s). \tag{9.3}$$

The above equation implies two facts. The first is that the two definitions in (9.1) and (9.2) are equivalent. The second is that the definition in (9.2) is independent to the starting state s_0. The proof of the above equation is given in the following shaded box.

Equivalence between the two definitions of $\bar{r}_\pi$ in (9.1) and (9.2)

$$\begin{aligned}
\bar{r}_\pi &= \lim_{n\to\infty} \frac{1}{n}\mathbb{E}\left[\sum_{k=1}^{n} R_{t+k}|S_t = s_0\right] \\
&= \lim_{n\to\infty} \frac{1}{n}\sum_{k=1}^{n}\mathbb{E}\left[R_{t+k}|S_t = s_0\right] \\
&= \lim_{k\to\infty} \mathbb{E}\left[R_{t+k}|S_t = s_0\right].
\end{aligned}$$

The last equability in the above equation is due to the property of the Cesaro mean. In particular, if $\{a_k\}_{k=1}^{\infty}$ is a convergent sequence such that $\lim_{k\to\infty} a_k$ exists, then $\{1/n\sum_{k=1}^{n} a_k\}_{n=1}^{\infty}$ is also a convergent sequence such that $\lim_{n\to\infty} 1/n\sum_{k=1}^{n} a_k = \lim_{k\to\infty} a_k$.

We next examine $\mathbb{E}\left[R_{t+k}|S_t = s_0\right]$. By the law of total expectation, we have

$$\begin{aligned}
\mathbb{E}\left[R_{t+k}|S_t = s_0\right] &= \sum_s \mathbb{E}\left[R_{t+k}|S_{t+k-1} = s, S_t = s_0\right] p^{(k-1)}(s|s_0) \\
&= \sum_s \mathbb{E}\left[R_{t+k}|S_{t+k-1} = s\right] p^{(k-1)}(s|s_0) \\
&= \sum_s r_\pi(s) p^{(k-1)}(s|s_0),
\end{aligned}$$

where $p^{(k-1)}(s|s_0)$ denotes the probability for transiting from s_0 to s using exactly $k-1$ steps. The second equality in the above equation is due to the Markov memoryless property: that is, the reward obtained the next time depends only on the current state instead of the previous states. The third equality is because $\mathbb{E}\left[R_{t+k}|S_{t+k-1} = s\right]$ is the expected value of the immediate rewards obtained starting from s.

Note that $\lim_{k\to\infty} p^{(k-1)}(s|s_0) = d_\pi(s)$ due to the property of stationary distribution. As a result, the initial state s_0 does not matter. Then, we have

$$\lim_{k\to\infty} \mathbb{E}\left[R_{t+k}|S_t = s_0\right] = \lim_{k\to\infty} \sum_s r_\pi(s) p^{(k-1)}(s|s_0) = \sum_s r_\pi(s) d_\pi(s).$$

The proof is complete.

– One complication of the policy gradient method is that the metrics can be defined in either the discounted case where $\gamma \in [0, 1)$ or the undiscounted case where $\gamma = 1$. We only consider the discounted case so far in this book. The undiscounted case is new and will be analyzed in detail in Section 9.3.2.

The definitions of state values and hence $\bar{v}_\pi$ and $v_\pi(s_0)$ are different when $\gamma = 1$ and $\gamma < 1$. By contrast, the definition of $\bar{r}_\pi$ does not rely on γ. Nevertheless, calculating the gradient of $\bar{r}_\pi$ still needs to distinguish the discounted and undiscounted cases. We will see that the three metrics have different properties in the two cases. This might be one of the most confusing problems for beginners studying policy gradient and deserves special attention.

9.3 Gradients of the metrics

Given the metrics introduced in the last section, we can use the gradient-based method to maximize them. This section calculates the gradients of the metrics. The calculation is one of the most complicated parts of policy gradient methods. To introduce the results clearly, we first present the most important result and then show the proof and related results later.

Theorem 9.1 (Policy gradient theorem). *The gradient of the average reward metric is*

$$\nabla_\theta \bar{r}_\pi(\theta) \simeq \sum_s d_\pi(s) \sum_a \nabla_\theta \pi(a|s,\theta) q_\pi(s,a), \tag{9.4}$$

where $\nabla_\theta \pi$ is the gradient of π with respect to θ. Here, $\simeq$ refers to either strict equality or approximated equality. In particular, it is a strict equation in the undiscounted case where $\gamma = 1$ and an approximated equation in the discounted case where $0 < \gamma < 1$. The approximation is more accurate in the discounted case when γ is closer to 1. Moreover, (9.4) *has a more compact and useful form expressed in terms of expectation:*

$$\nabla_\theta \bar{r}_\pi(\theta) \simeq \mathbb{E}\Big[\nabla_\theta \ln \pi(A|S,\theta) q_\pi(S,A)\Big], \tag{9.5}$$

where ln *is the natural logarithm and $S \sim d_\pi, A \sim \pi(S)$.*

Some important remarks about Theorem 9.1 are given below.

1) This is the well-known *policy gradient theorem* [34–36], a fundamental result for policy gradient methods. This theorem incorporate both the discounted and undiscounted cases, and is a combination of Theorem 9.3 and Theorem 9.5 shown later in the following subsections. Since the two subsections are mathematically intensive, readers are advised to read selectively according to their interests.
2) The gradients of the other two metrics $v_\pi(s_0)$ and $\bar{v}_\pi$ are not given in this theorem. Their gradients in the discounted case are given in Theorem 9.2 and Theorem 9.3 as shown later. We will see that their gradients have similar expressions as (9.5).
3) The expression in (9.5) is more favorable compared to (9.4) because it is expressed in terms of an expectation that can be approximated by experience samples. Why can (9.4) be expressed as (9.5)? The proof is trivial and given below. By the definition of

expectation, (9.4) can be rewritten as

$$\begin{aligned}\nabla_\theta \bar{r}_\pi &\simeq \sum_s d_\pi(s) \sum_a \nabla_\theta \pi(a|s,\theta) q_\pi(s,a) \\ &= \mathbb{E}\left[\sum_a \nabla_\theta \pi(a|S,\theta) q_\pi(S,a)\right],\end{aligned} \tag{9.6}$$

where the state $S \sim d_\pi$. Furthermore, consider the function $\ln \pi$ where ln is the natural logarithm. Its gradient can be easily obtained as

$$\nabla_\theta \ln \pi(a|s,\theta) = \frac{\nabla_\theta \pi(a|s,\theta)}{\pi(a|s,\theta)}$$

and hence

$$\nabla_\theta \pi(a|s,\theta) = \pi(a|s,\theta) \nabla_\theta \ln \pi(a|s,\theta). \tag{9.7}$$

Substituting (9.7) into (9.6) gives

$$\begin{aligned}\nabla_\theta \bar{r}_\pi &\simeq \mathbb{E}\left[\sum_a \pi(a|S,\theta) \nabla_\theta \ln \pi(a|S,\theta) q_\pi(S,a)\right] \\ &= \mathbb{E}\Big[\nabla_\theta \ln \pi(A|S,\theta) q_\pi(S,A)\Big],\end{aligned}$$

where $S \sim d_\pi$ and $A \sim \pi(S)$.

4) It is notable that $\pi(a|s,\theta)$ must be *positive* for all s, a to ensure that $\ln \pi(a|s,\theta)$ is valid. This can be archived by using *softmax functions*:

$$\pi(a|s,\theta) = \frac{e^{h(s,a,\theta)}}{\sum_{a' \in \mathcal{A}} e^{h(s,a',\theta)}},$$

where $h(s,a,\theta)$ is another function indicating the value or preference of selecting a at s. The overall softmax function approximation can be realized by a neural network whose input is s and parameter is θ. The network has $|\mathcal{A}|$ outputs, each of which corresponds to $\pi(a|s,\theta)$ for an action a. The activation function of the output layer should be softmax.

Since $\pi(a|s,\theta) > 0$ for all a, the parameterized policy is *stochastic* and hence *exploratory.* The policy does not directly tell which action to take. Instead, the action should be sampled according to the probability distribution of the policy. We will see in the next chapter that there also exist *deterministic* policy gradient methods.

9.3.1 Gradients in the discounted case

In this section, we derive the gradients of the metrics in the discounted case where $\gamma \in [0,1)$.

We are already familiar with the discounted case since all the previous chapters consider this case. The state value and action value in the discounted case are defined as

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s],\\
q_\pi(s,a) &\doteq \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s, A_t = a],
\end{aligned}
$$

where $\gamma \in (0,1)$ is the discount rate. We know $v_\pi(s) = \sum_a \pi(a|s,\theta) q_\pi(s,a)$ and the state value satisfies the Bellman equation.

Before deriving the gradients of $\bar{r}_\pi$ and $\bar{v}_\pi$, we introduce some useful preliminary results. First, the next lemma shows that $\bar{v}_\pi(\theta)$ and $\bar{r}_\pi(\theta)$ are equivalent metrics.

Lemma 9.1 (Equivalence between $\bar{v}_\pi(\theta)$ and $\bar{r}_\pi(\theta)$)**.** *In the discounted case where* $\gamma \in [0,1)$*, it holds that*

$$
\bar{r}_\pi = (1-\gamma)\bar{v}_\pi. \tag{9.8}
$$

Proof. Note that $\bar{v}_\pi(\theta) = d_\pi^T v_\pi$ and $\bar{r}_\pi(\theta) = d_\pi^T r_\pi$, where v_π and r_π satisfy the Bellman equation $v_\pi = r_\pi + \gamma P_\pi v_\pi$. Multiplying d_π^T on both sides of the Bellman equation gives

$$
\bar{v}_\pi = \bar{r}_\pi + \gamma d_\pi^T P_\pi v_\pi = \bar{r}_\pi + \gamma d_\pi^T v_\pi = \bar{r}_\pi + \gamma \bar{v}_\pi,
$$

which implies (9.8). □

Second, the following lemma gives the gradient of $v_\pi(s)$ for any s.

Lemma 9.2 (Gradient of $v_\pi(s)$)**.** *In the discounted case, it holds for any* $s \in \mathcal{S}$ *that*

$$
\nabla_\theta v_\pi(s) = \sum_{s'} \Pr_\pi(s'|s) \sum_a \nabla_\theta \pi(a|s',\theta) q_\pi(s',a), \tag{9.9}
$$

where

$$
\Pr_\pi(s'|s) \doteq \sum_{k=0}^{\infty} \gamma^k \Pr(s \to s', k, \pi) = \left[(I_n - \gamma P_\pi)^{-1}\right]_{ss'}
$$

is the discounted total probability transiting from s *to* s' *under policy* π*. Here,* $\Pr(s \to s', k, \pi)$ *denotes the probability for transiting from* s *to* s' *using exactly* k *steps under* π*.*

Proof of Lemma 9.2

First, for any $s \in \mathcal{S}$, it holds that

$$\begin{aligned}\nabla_\theta v_\pi(s) &= \nabla_\theta \left[\sum_a \pi(a|s,\theta) q_\pi(s,a)\right] \\ &= \sum_a \left[\nabla_\theta \pi(a|s,\theta) q_\pi(s,a) + \pi(a|s,\theta)\nabla_\theta q_\pi(s,a)\right], \end{aligned} \tag{9.10}$$

where $q_\pi(s,a)$ is the action value given by

$$q_\pi(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) v_\pi(s').$$

Here, $r(s,a) = \sum_r rp(r|s,a)$ is independent of θ. Therefore,

$$\nabla_\theta q_\pi = 0 + \gamma \sum_{s'} p(s'|s,a)\nabla_\theta v_\pi(s'),$$

substituting which into (9.10) gives

$$\nabla_\theta v_\pi(s) = \sum_a \left[\nabla_\theta \pi(a|s,\theta) q_\pi(s,a) + \pi(a|s,\theta)\gamma \sum_{s'} p(s'|s,a)\nabla_\theta v_\pi(s')\right]. \tag{9.11}$$

It is notable that $\nabla_\theta v_\pi$ appears on both sides of the above equation. To calculate $\nabla_\theta v_\pi$, one way is to use the *unrolling technique* [34]. Here, we use the *matrix-vector form*, which we believe is more straightforward to understand. In particular, let

$$u(s) \doteq \sum_a \nabla_\theta \pi(a|s,\theta) q_\pi(s,a).$$

Equation (9.11) can be written in a matrix-vector form as

$$\underbrace{\begin{bmatrix} \vdots \\ \nabla_\theta v_\pi(s) \\ \vdots \end{bmatrix}}_{\nabla_\theta v_\pi \in \mathbb{R}^{n|\mathcal{S}|}} = \underbrace{\begin{bmatrix} \vdots \\ u(s) \\ \vdots \end{bmatrix}}_{u \in \mathbb{R}^{n|\mathcal{S}|}} + \gamma (P_\pi \otimes I_m) \underbrace{\begin{bmatrix} \vdots \\ \nabla_\theta v_\pi(s') \\ \vdots \end{bmatrix}}_{\nabla_\theta v_\pi \in \mathbb{R}^{n|\mathcal{S}|}},$$

which can be written in short as

$$\nabla_\theta v_\pi = u + \gamma (P_\pi \otimes I_m)\nabla_\theta v_\pi.$$

Here, m is the dimension of the parameter vector θ and $\otimes$ is the Kronecker product. The reason that the Kronecker product emerges in the equation is that $\nabla_\theta v_\pi(s)$ is already a vector. The above equation is a linear equation of $\nabla_\theta v_\pi$, which can be solved as

$$\begin{aligned}\nabla_\theta v_\pi &= (I_{nm} - \gamma P_\pi \otimes I_m)^{-1} u \\ &= (I_n \otimes I_m - \gamma P_\pi \otimes I_m)^{-1} u \\ &= \left[(I_n - \gamma P_\pi)^{-1} \otimes I_m\right] u \qquad (9.12)\end{aligned}$$

The elementwise form of the solution is

$$\begin{aligned}\nabla_\theta v_\pi(s) &= \sum_{s'} \left[(I_n - \gamma P_\pi)^{-1}\right]_{ss'} u(s') \\ &= \sum_{s'} \left[(I_n - \gamma P_\pi)^{-1}\right]_{ss'} \sum_a \nabla_\theta \pi(a|s', \theta) q_\pi(s', a) \qquad (9.13)\end{aligned}$$

where $[\cdot]_{ss'}$ is the entry on the sth row and s'th column. The quantity $[(I_n - \gamma P_\pi)^{-1}]_{ss'}$ has a clear probability interpretation. In particular, since $(I_n - \gamma P_\pi)^{-1} = I + \gamma P_\pi + \gamma^2 P_\pi^2 + \cdots$, we have

$$\left[(I_n - \gamma P_\pi)^{-1}\right]_{ss'} = [I]_{ss'} + \gamma [P_\pi]_{ss'} + \gamma^2 [P_\pi^2]_{ss'} + \cdots = \sum_{k=0}^{\infty} \gamma^k [P_\pi^k]_{ss'}.$$

Note that $[P_\pi^k]_{ss'}$ is the probability transiting from s to s' using exactly k steps. Therefore, $[(I_n - \gamma P_\pi)^{-1}]_{ss'}$ is the (discounted) total probability transiting from s to s' using any steps. By denoting $[(I_n - \gamma P_\pi)^{-1}]_{ss'} \doteq \Pr_\pi(s'|s)$, equation (9.13) becomes (9.10).

With the results in Lemma 9.2, we are ready to derive the gradient of $v_\pi(s_0)$ with a specific s_0.

Theorem 9.2 (Gradient of $v_\pi(s_0)$ in the discounted case)**.** *In the discounted case where $\gamma \in [0, 1)$, the gradient of $v_\pi(s_0)$ is*

$$\nabla_\theta v_\pi(s_0) = \mathbb{E}\big[\nabla_\theta \ln \pi(A|S, \theta) q_\pi(S, A)\big], \qquad (9.14)$$

where $S \sim \rho_\pi$ and $A \sim \pi(s, \theta)$. Here, the state distribution ρ_π is

$$\rho_\pi(s) = \Pr_\pi(s|s_0) = \sum_{k=0}^{\infty} \gamma^k \Pr(s_0 \to s, k, \pi) = \left[(I_n - \gamma P_\pi)^{-1}\right]_{s_0 s}, \qquad (9.15)$$

which is the discounted total probability transiting from s_0 to s under policy π.

By comparing with Theorem 9.1, we notice that the expression of $\nabla_\theta v_\pi(s_0)$ for any s_0 is the same as $\nabla_\theta \bar{r}_\pi$. However, the difference is the probability distribution of S. Here, S obeys ρ_π which is different from d_π. The proof of the theorem is given below.

Proof of Theorem 9.2

Proof. First of all, note that $v_\pi(s_0) = \sum_{s\in\mathcal{S}} d_0(s)v_\pi(s)$ where $d_0(s = s_0) = 1$ and $d_0(s \neq s_0) = 0$. Therefore,

$$\nabla_\theta v_\pi(s_0) = \nabla_\theta \sum_s d_0(s)v_\pi(s) = \sum_s d_0(s)\nabla_\theta v_\pi(s).$$

The last equality is because $d_0(s)$ is independent of π. Substituting the expression of $\nabla_\theta v_\pi(s)$ as in Lemma 9.2 to the above equation yields

$$\begin{aligned}
\nabla_\theta v_\pi(s_0) = \sum_s d_0(s)\nabla_\theta v_\pi(s) &= \sum_s d_0(s)\sum_{s'}\Pr_\pi(s'|s)\sum_a \nabla_\theta\pi(a|s',\theta)q_\pi(s',a)\\
&= \sum_{s'}\left(\sum_s d_0(s)\Pr_\pi(s'|s)\right)\sum_a \nabla_\theta\pi(a|s',\theta)q_\pi(s',a)\\
&\doteq \sum_{s'}\rho_\pi(s')\sum_a \nabla_\theta\pi(a|s',\theta)q_\pi(s',a)\\
&= \sum_{s'}\rho_\pi(s')\sum_a \pi(a|s',\theta)\nabla_\theta\ln\pi(a|s',\theta)q_\pi(s',a)\\
&= \mathbb{E}\left[\nabla_\theta\ln\pi(A|S',\theta)q_\pi(S',A)\right],
\end{aligned}$$

where $S' \sim \rho_\pi$ and $A \sim \pi(s,\theta)$. Furthermore, since $d_0(s \neq s_0) = 0$, we have

$$\rho_\pi(s') = \sum_s d_0(s)\Pr_\pi(s'|s) = \Pr_\pi(s'|s_0),$$

which is the discounted total probability transiting from s_0 to s' under policy π. The proof is complete.

The above proof can also be shortened by directly rewriting (9.10) as (9.14) by considering the specific state s_0. The reason that we prove in the above way is to show a more general proof that can handle any fixed state distribution $d_0(s)$. $\square$

With the results in Lemma 9.1 and Lemma 9.2, we are ready to derive the gradients of $\bar{v}_\pi$ and $\bar{r}_\pi$.

Theorem 9.3 (Gradient of $\bar{v}_\pi$ and $\bar{r}_\pi$ in the discounted case)**.** *In the discounted case where $\gamma \in [0, 1)$, the gradients of $\bar{v}_\pi$ and $\bar{r}_\pi$ are, respectively,*

$$\nabla_\theta \bar{v}_\pi \approx \frac{1}{1-\gamma}\sum_s d_\pi(s)\sum_a \nabla_\theta\pi(a|s,\theta)q_\pi(s,a)$$

and

$$\nabla_\theta \bar{r}_\pi \approx \sum_s d_\pi(s) \sum_a \nabla_\theta \pi(a|s,\theta) q_\pi(s,a),$$

where the approximations are more accurate when γ is closer to 1.

It is interesting to note that the gradient of $\bar{r}_\pi$ has the same expression as that in the undiscounted case. This is an important result and is summarized as the policy gradient theorem. Nevertheless, it should be noted that the definition of the action value $q_\pi(s,a)$ is different from the undiscounted case.

Proof of Theorem 9.3

It follows from the definition of $\bar{v}_\pi$ that

$$\begin{aligned}\nabla_\theta \bar{v}_\pi &= \nabla_\theta \sum_s d_\pi(s) v_\pi(s) \\ &= \sum_s \nabla_\theta d_\pi(s) v_\pi(s) + \sum_s d_\pi(s) \nabla_\theta v_\pi(s). \end{aligned} \tag{9.16}$$

This equation contains two terms. On the one hand, substituting the expression of $\nabla_\theta v_\pi$ as in (9.12) into the second term gives

$$\begin{aligned}\sum_s d_\pi(s) \nabla_\theta v_\pi(s) &= (d_\pi^T \otimes I_m) \nabla_\theta v_\pi \\ &= (d_\pi^T \otimes I_m) \left[(I_n - \gamma P_\pi)^{-1} \otimes I_m\right] u \\ &= \left[d_\pi^T (I_n - \gamma P_\pi)^{-1}\right] \otimes I_m u. \end{aligned} \tag{9.17}$$

It is noted that

$$d_\pi^T (I_n - \gamma P_\pi)^{-1} = \frac{1}{1-\gamma} d_\pi^T,$$

which can be easily verified by multiplying $(I_n - \gamma P_\pi)$ on both sides of the equation. Therefore, (9.17) becomes

$$\begin{aligned}\sum_s d_\pi(s) \nabla_\theta v_\pi(s) &= \frac{1}{1-\gamma} d_\pi^T \otimes I_m u \\ &= \frac{1}{1-\gamma} \sum_s d_\pi(s) \sum_a \nabla_\theta \pi(a|s,\theta) q_\pi(s,a). \end{aligned}$$

On the other hand, the first term of (9.16) involves $\nabla_\theta d_\pi$. However, since the second term contains $\frac{1}{1-\gamma}$, the second term becomes dominant and the first term becomes negligible when $\gamma \to 1$. Therefore,

$$\nabla_\theta \bar{v}_\pi \approx \frac{1}{1-\gamma} \sum_s d_\pi(s) \sum_a \nabla_\theta \pi(a|s,\theta) q_\pi(s,a).$$

Furthermore, it follows from $\bar{r}_\pi = (1-\gamma)\bar{v}_\pi$ that

$$\nabla_\theta \bar{r}_\pi = (1-\gamma)\nabla_\theta \bar{v}_\pi \approx \sum_s d_\pi(s) \sum_a \nabla_\theta \pi(a|s,\theta) q_\pi(s,a).$$

The above approximation requires that the first term does not go to infinity when $\gamma \to 1$. A deeper analysis of this problem is omitted here and can be found in [36, Section 4].

9.3.2 Gradients in the undiscounted case

We now show how to calculate the gradients of the metrics in the undiscounted case where $\gamma = 1$. This is the first time we introduce the undiscounted case in this book. We first define and analyze state values in the undiscounted case, and then derive the metric gradients.

State value and the Poisson equation

If $\gamma = 1$, direct summation of the rewards, $\mathbb{E}[R_{t+1} + R_{t+2} + R_{t+3} + \dots | S_t = s]$, over infinitely long trajectories may diverge. Hence, the state value and action value under policy π are defined in a special way as [34]

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}[(R_{t+1} - \bar{r}_\pi) + (R_{t+2} - \bar{r}_\pi) + (R_{t+3} - \bar{r}_\pi) + \dots | S_t = s], \\ q_\pi(s,a) &\doteq \mathbb{E}[(R_{t+1} - \bar{r}_\pi) + (R_{t+2} - \bar{r}_\pi) + (R_{t+3} - \bar{r}_\pi) + \dots | S_t = s, A_t = a], \end{aligned}$$

where $\bar{r}_\pi$ is the average reward, which is fixed when π is given. Here, $v_\pi(s)$ has different names in the literature such as differential reward [35] or bias [2, Section 8.2.1]. It follows from the above definitions that $v_\pi(s) = \sum_a \pi(a|s,\theta) q_\pi(s,a)$. More importantly, the state values satisfy the following Bellman-like equation:

$$v_\pi(s) = \sum_a \pi(a|s,\theta) \left[\sum_r p(r|s,a)(r - \bar{r}_\pi) + \sum_{s'} p(s'|s,a) v_\pi(s') \right].$$

Hence, the action value is $q_\pi(s,a) = \sum_r p(r|s,a)(r - \bar{r}_\pi) + \sum_{s'} p(s'|s,a) v_\pi(s')$. The matrix-vector form of the above equation is

$$v_\pi = r_\pi - \bar{r}_\pi \mathbf{1}_n + P_\pi v_\pi, \tag{9.18}$$

where $\mathbf{1}_n = [1, \dots, 1]^T \in \mathbb{R}^n$ and $n = |\mathcal{S}|$. Equation (9.18) is similar to the Bellman equation and it has a specific name called *Poisson equation* [35,37]. How to calculate v_π from the Poisson equation? The solution is given in the following theorem.

Theorem 9.4 (Solution of the Poisson equation)**.** *Let*

$$v_\pi^* = (I_n - P_\pi + \mathbf{1}_n d_\pi^T)^{-1} r_\pi. \tag{9.19}$$

Then, v_π^* is a solution to the Poisson equation and any solution has the form of

$$v_\pi = v_\pi^* + c_\pi \mathbf{1}_n,$$

where $c_\pi \in \mathbb{R}$ is a constant depending on π

This theorem indicates that the solution of v_π to the Poisson equation is not unique. Interested readers can find the proof of the theorem given as follows.

Proof of Theorem 9.4

1) Why is the solution of v_π not unique?

Substituting $\bar{r}_\pi = d_\pi^T r_\pi$ into (9.18) gives

$$v_\pi = r_\pi - \mathbf{1}_n d_\pi^T r_\pi + P_\pi v_\pi \tag{9.20}$$

and consequently

$$(I_n - P_\pi) v_\pi = (I_n - \mathbf{1}_n d_\pi^T) r_\pi. \tag{9.21}$$

It is noted that $I_n - P_\pi$ is singular because $(I_n - P_\pi)\mathbf{1}_n = 0$ for any π. Therefore, the solution to (9.21) is not unique: if v_π^* is a solution, then $v_\pi^* + x$ is also a solution for any $x \in \text{Null}(I_n - P_\pi)$. When P_π is irreducible, $\text{Null}(I_n - P_\pi) = \text{span}\{\mathbf{1}_n\}$. Hence, any solution to the Poisson equation has the expression of $v_\pi^* + c\mathbf{1}_n$ where $c \in \mathbb{R}$.

2) While the solution to (9.20) **is not unique, can we at least find one specific solution?**

The answer is yes. We next show that v_π^* in (9.19) is a solution to (9.20). For the sake of simplicity, let

$$A \doteq I_n - P_\pi + \mathbf{1}_n d_\pi^T.$$

Then, $v_\pi^* = A^{-1} r_\pi$, substituting which into (9.20) gives

$$A^{-1} r_\pi = r_\pi - \mathbf{1}_n d_\pi^T r_\pi + P_\pi A^{-1} r_\pi.$$

Recognizing the above equation gives $(-A^{-1} + I_n - \mathbf{1}_n d_\pi^T + P_\pi A^{-1}) r_\pi = 0$ and consequently

$$(-I_n + A - \mathbf{1}_n d_\pi^T A + P_\pi)A^{-1} r_\pi = 0.$$

The term in the brackets in the above equation is zero because $-I_n + A - \mathbf{1}_n d_\pi^T A + P_\pi = -I_n + (I_n - P_\pi + \mathbf{1}_n d_\pi^T) - \mathbf{1}_n d_\pi^T (I_n - P_\pi + \mathbf{1}_n d_\pi^T) + P_\pi = 0$. Therefore, v_π^* is a solution.

3) Why is $A = I_n - P_\pi + \mathbf{1}_n d_\pi^T$ invertible?

Since v_π^* involves A^{-1}, it is necessary to show that A is invertible. The analysis is summarized as a lemma below.

Lemma 9.3. *The matrix $I_n - P_\pi + \mathbf{1}_n d_\pi^T$ is invertible and its inverse is*

$$\left[I_n - (P_\pi - \mathbf{1}_n d_\pi^T)\right]^{-1} = \sum_{k=1}^{\infty} (P_\pi^k - \mathbf{1}_n d_\pi^T) + I_n.$$

Proof. First of all, we state some preliminary results without giving proof. Let $\rho(M)$ be the spectral radius of a matrix M. Then, $I - M$ is invertible if $\rho(M) < 1$. Moreover, $\rho(M) < 1$ if and only if $\lim_{k\to\infty} M^k = 0$.

Based on the above facts, we next show that $\lim_{k\to\infty}(P_\pi - \mathbf{1}_n d_\pi^T)^k \to 0$ and then the invertibility of $I_n - (P_\pi - \mathbf{1}_n d_\pi^T)$ immediately follows. To do that, it is noted that

$$(P_\pi - \mathbf{1}_n d_\pi^T)^k = P_\pi^k - \mathbf{1}_n d_\pi^T, \quad k \geq 1,$$

which can be proved by induction. For instance, when $k = 1$, the equation is valid. When $k = 2$, we have

$$\begin{aligned}(P_\pi - \mathbf{1}_n d_\pi^T)^2 &= (P_\pi - \mathbf{1}_n d_\pi^T)(P_\pi - \mathbf{1}_n d_\pi^T) \\ &= P_\pi^2 - P_\pi \mathbf{1}_n d_\pi^T - \mathbf{1}_n d_\pi^T P_\pi + \mathbf{1}_n d_\pi^T \mathbf{1}_n d_\pi^T \\ &= P_\pi^2 - \mathbf{1}_n d_\pi^T,\end{aligned}$$

where the last equality is due to $P_\pi \mathbf{1}_n = \mathbf{1}_n$, $d_\pi^T P_\pi = d_\pi^T$, and $d_\pi^T \mathbf{1}_n = 1$. The case of $k \geq 3$ can be proven similarly and omitted here.

Since d_π is the stationary distribution of the state, it holds that $\lim_{k\to\infty} P_\pi^k = d_\pi^T \mathbf{1}_n$ (the properties of the stationary distribution can be found in Section 8.2). Therefore,

$$\lim_{k\to\infty} (P_\pi - \mathbf{1}_n d_\pi^T)^k = \lim_{k\to\infty} P_\pi^k - d_\pi^T \mathbf{1}_n = 0.$$

As a result, $I_n - (P_\pi - \mathbf{1}_n d_\pi^T)$ is invertible. Furthermore, its inverse is given by

$$\begin{aligned}(I_n - (P_\pi - \mathbf{1}_n d_\pi^T))^{-1} &= \sum_{k=0}^{\infty}(P_\pi - \mathbf{1}_n d_\pi^T)^k \\ &= I_n + \sum_{k=1}^{\infty}(P_\pi - \mathbf{1}_n d_\pi^T)^k \\ &= I_n + \sum_{k=1}^{\infty}(P_\pi^k - \mathbf{1}_n d_\pi^T) \\ &= \sum_{k=0}^{\infty}(P_\pi^k - \mathbf{1}_n d_\pi^T) + \mathbf{1}_n d_\pi^T.\end{aligned}$$

□

The proof of Lemma 9.3 is inspired by [36]. However, [36] inaccurately states that $(I_n - P_\pi + \mathbf{1}_n d_\pi^T)^{-1} = \sum_{k=0}^{\infty}(P_\pi^k - \mathbf{1}_n d_\pi^T)$ (see the statement above equation (16) in [36]). The inaccuracy can also be verified by the fact that $\sum_{k=0}^{\infty}(P_\pi^k - \mathbf{1}_n d_\pi^T)\mathbf{1}_n = 0$ and hence $\sum_{k=0}^{\infty}(P_\pi^k - \mathbf{1}_n d_\pi^T)$ is singular, leading to a conflict with the nonsingularity of A. Lemma 9.3 corrects this inaccuracy.

Derivation of gradients

Theorem 9.4 indicates that the value of v_π in the undiscounted case is not unique. If we would like to get a unique value of v_π, we can add more constraints. For example, by assuming there exists a recurrent state, the state value of this recurrent state is zero [35, Section II] and hence the constant c_π can be determined. There are also other ways. See, for example, equations (8.6.5)-(8.6.7) in [2].

Although the value of v_π is not unique, the solution of $\bar{r}_\pi$ is still unique. In particular, it follows from the Poisson equation that

$$\begin{aligned}\bar{r}_\pi \mathbf{1}_n &= r_\pi + (P_\pi - I_n)v_\pi \\ &= r_\pi + (P_\pi - I_n)(v_\pi^* + c\mathbf{1}_n) \\ &= r_\pi + (P_\pi - I_n)v_\pi^*.\end{aligned}$$

Since v_π is not unique and so is $\bar{v}_\pi$, we only calculate the gradient of $\bar{r}_\pi$ in the undiscounted case.

Theorem 9.5 (Gradient of $\bar{r}_\pi$ in the undiscounted case)**.** *In the undiscounted case, the gradient of the average reward $\bar{r}_\pi$ is*

$$\nabla_\theta \bar{r}_\pi = \sum_s d_\pi(s) \sum_a \nabla_\theta \pi(a|s,\theta) q_\pi(s,a).$$

Proof of Theorem 9.5

First of all, it follows from $v_\pi(s) = \sum_a \pi(a|s,\theta)q_\pi(s,a)$ that

$$\begin{aligned}\nabla_\theta v_\pi(s) &= \nabla_\theta \left[\sum_a \pi(a|s,\theta)q_\pi(s,a)\right]\\ &= \sum_a \left[\nabla_\theta \pi(a|s,\theta)q_\pi(s,a) + \pi(a|s,\theta)\nabla_\theta q_\pi(s,a)\right], \end{aligned} \tag{9.22}$$

where $q_\pi(s,a)$ is the action value satisfying

$$\begin{aligned}q_\pi(s,a) &= \sum_r p(r|s,a)(r-\bar{r}_\pi) + \sum_{s'} p(s'|s,a)v_\pi(s')\\ &= r(s,a) - \bar{r}_\pi + \sum_{s'} p(s'|s,a)v_\pi(s').\end{aligned}$$

Here, $r(s,a) = \sum_r rp(r|s,a)$ is independent of θ. Therefore,

$$\nabla_\theta q_\pi = 0 - \nabla_\theta \bar{r}_\pi + \sum_{s'} p(s'|s,a)\nabla_\theta v_\pi(s'),$$

substituting which into (9.22) gives

$$\begin{aligned}\nabla_\theta v_\pi(s) &= \sum_a \left[\nabla_\theta \pi(a|s,\theta)q_\pi(s,a) + \pi(a|s,\theta)\left(-\nabla_\theta \bar{r}_\pi + \sum_{s'} p(s'|s,a)\nabla_\theta v_\pi(s')\right)\right]\\ &= \sum_a \nabla_\theta \pi(a|s,\theta)q_\pi(s,a) - \nabla_\theta \bar{r}_\pi + \sum_a \pi(a|s,\theta)\sum_{s'} p(s'|s,a)\nabla_\theta v_\pi(s').\end{aligned} \tag{9.23}$$

Let

$$u(s) \doteq \sum_a \nabla_\theta \pi(a|s,\theta)q_\pi(s,a).$$

Equation (9.23) can be written in a matrix-vector form as

$$\underbrace{\begin{bmatrix}\vdots\\ \nabla_\theta v_\pi(s)\\ \vdots\end{bmatrix}}_{\nabla_\theta v_\pi \in \mathbb{R}^{m|\mathcal{S}|}} = \underbrace{\begin{bmatrix}\vdots\\ u(s)\\ \vdots\end{bmatrix}}_{u \in \mathbb{R}^{m|\mathcal{S}|}} - \mathbf{1}_n \otimes \nabla_\theta \bar{r}_\pi + (P_\pi \otimes I_m)\underbrace{\begin{bmatrix}\vdots\\ \nabla_\theta v_\pi(s')\\ \vdots\end{bmatrix}}_{\nabla_\theta v_\pi \in \mathbb{R}^{m|\mathcal{S}|}},$$

where m is the dimension of the parameter vector θ and hence equals the dimension of $\nabla_\theta \bar{r}_\pi$ and $v_\pi(s)$. Here, $\otimes$ is the Kronecker product. The above equation can be written in short as

$$\nabla_\theta v_\pi = u - \mathbf{1}_n \otimes \nabla_\theta \bar{r}_\pi + (P_\pi \otimes I_m)\nabla_\theta v_\pi$$

and hence

$$\mathbf{1}_n \otimes \nabla_\theta \bar{r}_\pi = u + (P_\pi \otimes I_m)\nabla_\theta v_\pi - \nabla_\theta v_\pi.$$

Multiplying $d_\pi^T \otimes I_m$ on both sides of the equation gives

$$\begin{aligned}(d_\pi^T \mathbf{1}_n) \otimes \nabla_\theta \bar{r}_\pi &= d_\pi^T \otimes I_m u + (d_\pi^T P_\pi) \otimes I_m \nabla_\theta v_\pi - d_\pi^T \otimes I_m \nabla_\theta v_\pi \\ &= d_\pi^T \otimes I_m u,\end{aligned}$$

which is

$$\begin{aligned}\nabla_\theta \bar{r}_\pi &= d_\pi^T \otimes I_m u \\ &= \sum_s d_\pi(s) u(s) \\ &= \sum_s d_\pi(s) \sum_a \nabla_\theta \pi(a|s,\theta) q_\pi(s,a).\end{aligned}$$

9.4 Policy gradient by Monte Carlo estimation

With the gradients of the metrics presented in the previous section, we show in this section how to use the gradient-based method to optimize the metrics and hence find optimal policies.

Consider $J(\theta) = \bar{r}_\pi(\theta)$ or $v_\pi(s_0)$. The gradient-ascent algorithm maximizing $J(\theta)$ is

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha \nabla_\theta J(\theta) \\ &= \theta_t + \alpha \mathbb{E}\Big[\nabla_\theta \ln \pi(A|S,\theta_t) q_\pi(S,A)\Big],\end{aligned} \tag{9.24}$$

where $\alpha > 0$ is a constant learning rate. The gradient-ascent algorithm can find a local minima where $\nabla_\theta \bar{r}_\pi(\theta_t) = 0$. Since the expected value on the right-hand side is unknown, we can replace the expected value with a sample:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \ln \pi(a_t|s_t,\theta_t) q_t(s_t,a_t), \tag{9.25}$$

where $q_t(s_t,a_t)$ is an approximation of $q_\pi(s_t,a_t)$. If $q_\pi(s_t,a_t)$ is approximated by Monte Carlo estimation, the algorithm is called *REINFORCE* [38] or *Monte Carlo policy gradient*, which is one of earliest and simplest policy gradient algorithms. Many other policy gradient algorithms such as the actor-critic methods introduced in the next chapter can be obtained by extending REINFORCE.

We examine the interpretation of (9.25) more closely. Since

$$\nabla_\theta \ln \pi(a_t|s_t,\theta_t) = \frac{\nabla_\theta \pi(a_t|s_t,\theta_t)}{\pi(a_t|s_t,\theta_t)},$$

(9.25) can be rewritten as

$$\theta_{t+1} = \theta_t + \alpha \left(\frac{q_t(s_t,a_t)}{\pi(a_t|s_t,\theta_t)} \right) \nabla_\theta \pi(a_t|s_t,\theta_t).$$

The coefficient $\frac{q_t(s_t,a_t)}{\pi(a_t|s_t,\theta_t)}$ can well *balance exploration and exploitation*. First, the coefficient is *proportional* to $q_t(s_t,a_t)$. As a result, if the action value of (s_t,a_t) is great, then the algorithm intends to update the parameter θ so that the probability of taking that action can be enhanced. Therefore, the algorithm attempts to exploit actions with greater values. Second, the coefficient is *inversely proportional* to $\pi(a_t|s_t,\theta_t)$. As a result, if the probability of taking (s_t,a_t) is small, then the algorithm would update the parameters θ so that the probability of taking that action can increase. This reflects that the algorithm attempts to explore actions that have a low probability to take.

Since (9.25) uses samples to approximate the expectation in (9.24), it is important to understand the correct ways to do sampling.

– How to sample S? In theory, S in $\mathbb{E}[\nabla_\theta \ln \pi(A|S,\theta_t) q_\pi(S,A)]$ obeys either the stationary distribution d_π or the discounted total probability distribution ρ_π as in (9.15). Either d_π or ρ_π represents the long-run behavior under π. Therefore, the ideal sampling way is to execute $\pi(\theta_t)$ for sufficiently many steps and then randomly select a state as s_t.
– How to sample A? In theory, the sampling of A in $\mathbb{E}[\nabla_\theta \ln \pi(A|S,\theta_t) q_\pi(S,A)]$ must follow the distribution of $\pi(A|S,\theta)$. Therefore, the ideal way to sample A is to select a_t following $\pi(\theta_t)$ at s_t. Therefore, the algorithm is on-policy.

The ideal ways for sampling S and A are usually not used in practice due to their low sample efficiency. For example, a more sample-efficient implementation of (9.25) is given in the pseudocode. In this implementation, an episode is generated first following $\pi(\theta_t)$. Then, every state-action pair in the episode is used to update $\{\theta_{t+1}, \theta_{t+2}, \dots\}$. Therefore, it uses the every-visit strategy and is hence more efficient in terms of sample usage. However, this implementation does not follow the ideal sampling ways. For example, the updating of $\{\theta_{t+2}, \theta_{t+3}, \dots\}$ uses the samples generated by $\pi(\theta_t)$, whereas the ideal way for updating $\{\theta_{t+2}, \theta_{t+3}, \dots\}$ is to use the samples generated by $\{\pi(\theta_{t+1}), \pi(\theta_{t+2}), \dots\}$, respectively. As a result, this implementation is off-policy. However, this implementation is more sample efficient because it fully utilizes the samples in an episode. Therefore, it is a trade-off between data efficiency and theoretical correctness. Such kind of trade-off is usually acceptable as long as the estimate of q_π is not so inaccurate. With this

Pseudocode: Policy Gradient by Monte Carlo (REINFORCE)

```
Initialization: A parameterized function $\pi(a|s,\theta)$, $\gamma \in [0,1)$, and $\alpha > 0$.
Aim: Search for an optimal policy maximizing $J(\theta)$.

For each episode, do
    Select $s_0$ and generate an episode following $\pi(\theta_t)$. Suppose the episode is $\{s_0, a_0, r_1, \dots, s_{T-1}, a_{T-1}, r_T\}$.
    For $t = 0, 1, \dots, T-1$:
        Action value estimate: use $q_t(s_t,a_t) = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$ to approximate $q_\pi(s_t,a_t)$
        Policy parameter update: $\theta_{t+1} = \theta_t + \alpha \nabla_\theta \ln \pi(a_t|s_t,\theta_t) q_t(s_t,a_t)$
```

point in mind, we know that the algorithm would not work well given a single sufficiently long episode. Instead, we should generate episodes as often as possible using the latest updated policies so that q_π can be estimated more accurately.

9.5 Summary

In this chapter, we introduced policy gradient methods, which provide an efficient and powerful way to handle large or even continuous state and action spaces. Policy gradient methods are *policy-based*. It is a big step forward because all the methods in the previous chapters are *value-based*. The basic idea of policy gradient, as shown in this chapter, is very simple: that is to select an appropriate scalar metric and then optimize it by a gradient-ascent algorithm. While the expectation is the gradient-ascent algorithm is unknown, we can approximate it by using experience samples.

The most complicated part of policy gradient methods is the derivation of the gradients of the metrics. That is because the mathematical derivation is nontrivial and, in the meantime, we have to distinguish different metrics and the discounted and undiscounted cases. Fortunately, as suggested by the policy gradient theorem, the gradients of the metrics share a unified and elegant expression.

The policy gradient algorithm REINFORCE introduced in this chapter is based on Monte Carlo estimation. It is the most basic policy gradient algorithm and important to understand more advanced algorithms. In the next chapter, the algorithm will be extended to another class of policy gradient methods called actor-critic.

9.6 Q&A

– Q: What is the basic idea of policy gradient?

A: The basic idea of policy gradient is simple: that is to find an optimal policy by maximizing a metric using the gradient-ascent methods. The complication emerges when we want to select appropriate metrics to define optimal policies, calculate the gradients of the metric, and use samples to approximate the gradient.

– Q: How many definitions are there for the metric of average reward?

  A: There are two definitions. The first definition as shown in (9.1) is a weighted average of the expected reward that can be obtained starting from each state. The second definition as shown in (9.2) is the expected undiscounted total reward that can be obtained starting from a state. The two definitions are equivalent as shown in (9.3).

– Q: Why do we need to care about undiscounted cases?

  A: Before this chapter, we only considered the discounted case. In this chapter, we introduced the undiscounted case. The reason that we care about the undiscounted case is that it can handle continuous tasks without starting states.

– Q: While the definition and the gradient of the average reward $\bar{r}_\pi$ do not involve the discount rate, why do we need to distinguish the discounted and undiscounted cases?

  A: The reason is that, when we attempt to calculate the gradient of $\bar{r}_\pi$, we need to consider state values whose definition involves the discount rate. In particular, we need to respectively study the Bellman equation in the discounted case and the Poisson equation in the undiscounted case.

Appendix A

Preliminaries to Probability Theory

Reinforcement learning (RL) heavily relies on probability theory. We next summarize some concepts and results frequently used in this book.

- *Random variable*: The term "variable" indicates that a random variable can take values in a set of numbers or events. The term "random" indicates that taking a value must follow a probability distribution.

  A random variable is usually denoted as a capital letter. Its value is usually denoted as a lowercase letter. For example, X is a random variable, and x is a value that X can take.

  In this book, we mainly consider the case where a random variable can only take a finite number of values. A random variable can be a scalar or vector.

  Like normal variables, random variables have normal mathematical operations or functions such as summation, product, and absolute value. For example, if X, Y are two random variables, we can calculate $X + Y$, $X + 1$, and XY.

- *Stochastic sequence* is a sequence of random variables.

  One scenario that we often encounter is that we collect a stochastic sampling sequence $\{x_i\}_{i=1}^n$ of a random variable X. For example, consider the task of tossing a die n times. Let x_i be a random variable representing the value obtained in the ith toss, then $\{x_1, x_2, \ldots, x_n\}$ is a stochastic process.

  It may be confusing to beginners why x_i is a random variable instead of a deterministic value. In fact, if the sampling sequence is {1,6,3,5,...}, then this sequence is not a stochastic sequence because all the elements are already determined. However, if we use a variable x_i to represent the value that can be possibly sampled, it is a random variable since x_i can take any values in $\{1, \ldots, 6\}$. Although x_i is a lowercase letter here, it still represents the random variable.

- *Probability*: The notation $p(X = x)$ or $p_X(x)$ describes the probability of the random variable X taking the value x. When the context is clear, $p(X = x)$ is often written as $p(x)$ in short.

– *Joint probability*: The notation $p(X = x, Y = y)$ or $p(x, y)$ describes the probability of the random variable X taking the value of x and, in the meantime, Y taking the value of y. One useful identity is

$$\sum_y p(x, y) = p(x).$$

– *Conditional probability*: The notation $p(X = x|A = a)$ describes the probability of the random variable X taking the value of x given that the random variable A has already taken the value of a. We often write $p(X = x|A = a)$ in short as $p(x|a)$.

It holds that

$$p(x, a) = p(x|a)p(a)$$

and

$$p(x|a) = \frac{p(x, a)}{p(a)}.$$

Since $p(x) = \sum_a p(x, a)$, we have

$$p(x) = \sum_a p(x, a) = \sum_a p(x|a)p(a),$$

which is called the *law of total probability*.

– *Independence*: Two random variables are independent of each other if the sampling value of one random variable does not affect the other. Mathematically, X and Y are independent of each other if

$$p(x, y) = p(x)p(y).$$

Another equivalent definition is

$$p(x|y) = p(x).$$

The above two definitions are equivalent because because $p(x, y) = p(x|y)p(y)$, which implies $p(x|y) = p(x)$ if and only if $p(x, y) = p(x)p(y)$.

– *Conditional independence:* Let X, A, B be three random variables. X is called conditionally independent to A given B if

$$p(X = x|A = a, B = b) = p(X = x|B = b).$$

The intuitive interpretation is as follows. X and A are conditionally independent given B if and only if, given knowledge of whether $B = b$ occurs, the knowledge of whether $X = x$ occurs provides no information on the likelihood of $A = a$ occurring, and vice versa.

In the context of RL, consider three consecutive states: s_t, s_{t+1}, s_{t+2}. Since they are obtained consecutively, s_{t+2} is dependent on s_{t+1} and also s_t. However, if s_{t+1} is already given, then s_{t+2} is conditionally independent to s_t. That is

$$p(s_{t+2}|s_{t+1}, s_t) = p(s_{t+2}|s_{t+1}).$$

This is also the memoryless property of Markov processes.

– *Law of total probability*: The law of total probability is mentioned when we introduce conditional probability. Due to its importance, we list it again below:

$$p(x) = \sum_y p(x, y)$$

and

$$p(x|a) = \sum_y p(x, y|a).$$

– *Chain rule* of conditional probability and joint probability. By the definition of conditional probability, we have

$$p(a, b) = p(a|b)p(b).$$

It can be further extended to

$$p(a, b, c) = p(a|b, c)p(b, c) = p(a|b, c)p(b|c)p(c)$$

and hence $p(a, b, c)/p(c) = p(a, b|c) = p(a|b, c)p(b|c)$. The fact that $p(a, b|c) = p(a|b, c)p(b|c)$ implies the following useful fact about conditional probability:

$$p(x|a) = \sum_b p(x, b|a) = \sum_b p(x|b, a)p(b|a).$$

– *Expectation/expected value/mean*: Suppose X is a random variable and the probability of taking value x is $p(x)$. The expectation, expected value, or called mean of X is defined as

$$\mathbb{E}[X] = \sum_x p(x)x.$$

The linearity property of expectation is

$$\begin{aligned}\mathbb{E}[X + Y] &= \mathbb{E}[X] + \mathbb{E}[Y],\\ \mathbb{E}[aX] &= a\mathbb{E}[X].\end{aligned}$$

The second equation above can be proven by definition. The first equation is proven below:

$$\begin{aligned}\mathbb{E}[X+Y] &= \sum_x \sum_y (x+y)p(X=x,Y=y)\\ &= \sum_x x \sum_y p(x,y) + \sum_y y \sum_x p(x,y)\\ &= \sum_x xp(x) + \sum_y yp(y)\\ &= \mathbb{E}[X] + \mathbb{E}[Y].\end{aligned}$$

Due to the linearity of expectation, we have the following useful fact:

$$\mathbb{E}\left[\sum_i a_i X_i\right] = \sum_i a_i \mathbb{E}[X_i].$$

Similarly, it can be proven that

$$\mathbb{E}[AX] = A\mathbb{E}[X],$$

where $A \in \mathbb{R}^{n\times n}$ is a fixed matrix and $X \in \mathbb{R}^n$ is a random vector.

– *Conditional expectation*: The definition of conditional expectation is

$$\mathbb{E}[X|A=a] = \sum_x xp(x|a).$$

Similar to the law of total probability, we have the law of total expectation:

$$\mathbb{E}[X] = \sum_a \mathbb{E}[X|A=a]p(a).$$

The proof is as follows. By the definition of expectation, the right-hand side equals

$$\begin{aligned}\sum_a \mathbb{E}[X|A=a]p(a) &= \sum_a \left[\sum_x p(x|a)x\right] p(a)\\ &= \sum_x \sum_a p(x|a)p(a)x\\ &= \sum_x \left[\sum_a p(x|a)p(a)\right] x\\ &= \sum_x p(x)x\\ &= \mathbb{E}[X].\end{aligned}$$

The law of total expectation is frequently used in RL.

Similarly, conditional expectation satisfies

$$\mathbb{E}[X|A=a]=\sum_b \mathbb{E}[X|A=a,B=b]p(b|a).$$

This equation is useful in the derivation of the Bellman equation. A hint of the proof is the chain rule: $p(x|a,b)p(b|a)=p(x,b|a)$.

Finally, it is worth noting that $\mathbb{E}[X|A=a]$ is different from $\mathbb{E}[X|A]$. The former is a value, whereas the latter is a random variable. In fact, $\mathbb{E}[X|A]$ is a function of the random variable A. To well define $\mathbb{E}[X|A]$, we need the rigorous probability theory.

– *Gradient of expectation*: Let $f(X,\beta)$ be a scalar function of a random variable X and a deterministic parameter vector β. Then,

$$\nabla_\beta \mathbb{E}[f(X,\beta)]=\mathbb{E}[\nabla_\beta f(X,\beta)].$$

Proof: Since $\mathbb{E}[f(X,\beta)]=\sum_x f(x,a)p(x)$, we have $\nabla_\beta \mathbb{E}[f(X,\beta)]=\nabla_\beta \sum_x f(x,a)p(x)=\sum_x \nabla_\beta f(x,a)p(x)=\mathbb{E}[\nabla_\beta f(X,\beta)]$.

– *Variance, Covariance, Covariance matrix*: For a single random variable X, its variance is defined as $\text{var}(X)=\mathbb{E}[(X-\bar{X})^2]$, where $\bar{X}=\mathbb{E}[X]$. For two random variables X,Y, their covariance is defined as $\text{cov}(X,Y)=\mathbb{E}[(X-\bar{X})(Y-\bar{Y})]$. For a random vector $X=[X_1,\dots,X_n]^T$, the covariance matrix of X is defined as $\text{var}(X)\doteq\Sigma=\mathbb{E}[(X-\bar{X})(X-\bar{X})^T]\in\mathbb{R}^{n\times n}$. The ijth entry of Σ is $[\Sigma]_{ij}=\mathbb{E}[[X-\bar{X}]_i[X-\bar{X}]_j]=\mathbb{E}[(X_i-\bar{X}_i)(X_j-\bar{X}_j)]=\text{cov}(X_i,X_j)$. One trivial property is $\text{var}(a)=0$ if a is deterministic. Moreover, it can be verified that $\text{var}(AX+a)=\text{var}(AX)=A\text{var}(X)A^T=A\Sigma A^T$.

Some useful facts are summarized below.

– Fact: $\mathbb{E}[(X-\bar{X})(Y-\bar{Y})]=\mathbb{E}(XY)-\bar{X}\bar{Y}=\mathbb{E}(XY)-\mathbb{E}(X)\mathbb{E}(Y)$.

  Proof: $\mathbb{E}[(X-\bar{X})(Y-\bar{Y})]=\mathbb{E}[XY-X\bar{Y}-\bar{X}Y+\bar{X}\bar{Y}]=\mathbb{E}[XY]-\mathbb{E}[X]\bar{Y}-\bar{X}\mathbb{E}[Y]+\bar{X}\bar{Y}=\mathbb{E}[XY]-\mathbb{E}[X]\mathbb{E}[Y]-\mathbb{E}[X]\mathbb{E}[Y]+\mathbb{E}[X]\mathbb{E}[Y]=\mathbb{E}[XY]-\mathbb{E}[X]\mathbb{E}[Y]$.

– Fact: $\mathbb{E}[XY]=\mathbb{E}[X]\mathbb{E}[Y]$ if X,Y are independent.

  Proof: $\mathbb{E}[XY]=\sum_x\sum_y p(x,y)xy=\sum_x\sum_y p(x)p(y)xy=\sum_x p(x)x\sum_y p(y)y=\mathbb{E}[X]\mathbb{E}[Y]$.

– Fact: $\text{cov}(X,Y)=0$ if X,Y are independent.

  Proof: when X,Y are independent, $\text{cov}(X,Y)=\mathbb{E}[XY]-\mathbb{E}[X]\mathbb{E}[Y]=\mathbb{E}[X]\mathbb{E}[Y]-\mathbb{E}[X]\mathbb{E}[Y]=0$.

Appendix B

Preliminaries to Gradient Descent

B.1 Basics

Mean value theorem

1) The statement of the theorem:

   Suppose x, y are two real variables and $f(\cdot)$ is a differentiable scalar function.

   – Scalar case: If x, y are *scalars* in $\mathbb{R}$ and $f : \mathbb{R} \to \mathbb{R}$, then there exists $z \in [x, y]$ such that
   $$f(y) - f(x) = f'(z)(y - z),$$
   where $f'(z)$ is the derivative of f at z.

   – Vector case: If x, y are *vectors* in $\mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$, then there exists $z = cx + (1 - c)y$ with $c \in [0, 1]$ such that
   $$f(y) - f(x) = \nabla f(z)^T (y - x), \tag{B.1}$$
   where $\nabla f(z)$ is the gradient of f parameterized at z.

2) Generalization:

   Consider $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ as a new function. Applying (B.1) in the mean value theorem to this new function gives
   $$\nabla f(y) - \nabla f(x) = \nabla^2 f(z)(y - x), \tag{B.2}$$
   where $z = cx + (1 - c)y$ with $c \in [0, 1]$, and $\nabla^2 f \in \mathbb{R}^{n \times n}$ is the *Hessian matrix*.

3) A related and useful result:

   Suppose $x, y \in \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$. Then we have
   $$f(y) = f(x) + \nabla f(x)^T (y - x) + \frac{1}{2}(y - x)^T \nabla^2 f(z)(y - x), \tag{B.3}$$

where

$$z = cx + (1-c)y$$

with $c \in [0,1]$. This is the Talyor expansion or quadratic expansion of multivariate functions [39, Section 9.1.2].

Furthermore, suppose that $\nabla^2 f$ satisfies $\nabla^2 f(z) \preceq LI_n$, where L is a positive constant and I_n is the $n \times n$ identity matrix. Then, (B.3) implies

$$f(y) \leq f(x) + \nabla f(x)^T(y-x) + \frac{L}{2}\|y-x\|^2.$$

Similarly, if $\nabla^2 f(z) \succeq \ell I_n$ with ℓ as a positive constant, then (B.3) implies

$$f(y) \geq f(x) + \nabla f(x)^T(y-x) + \frac{\ell}{2}\|y-x\|^2.$$

Convexity

– Definitions:

  – Convex set: Suppose $\mathcal{D} \subseteq \mathbb{R}^n$ is a subset of $\mathbb{R}^n$. This set is *convex*, if for any $x, y \in \mathcal{D}$ and any $c \in [0,1]$, $z \doteq cx + (1-c)y$ is in $\mathcal{D}$.

  – Convex function: Suppose $f : \mathcal{D} \to \mathbb{R}$ where $\mathcal{D}$ is convex. Then, the function $f(x)$ is *convex* if

$$f(cx + (1-x)y) \leq cf(x) + (1-c)f(y)$$

  for any $x, y \in \mathcal{D}$ and $c \in [0,1]$.

– Evaluation conditions:

How to evaluate if a function is convex or not? Two evaluation conditions are given below.

  – First-order condition: Consider a function $f : \mathcal{D} \to \mathbb{R}$ where $\mathcal{D}$ is convex. Then, f is a convex function if

$$f(y) - f(x) \geq \nabla f(x)^T(y-x), \quad \text{for all } x, y \in \mathcal{D}.$$

  When x is scalar, $\nabla f(x)$ is the slope of the tangent line of $f(x)$ at x. If we fix x and consider different y, the geometric interpretation of this inequality is that the point $(y, f(y))$ is always located above the tangent line.

  – Second-order condition: Consider a function $f : \mathcal{D} \to \mathbb{R}$ where $\mathcal{D}$ is convex. Then, f is a convex function if

$$H \doteq \nabla^2 f(x) \geq 0, \quad \text{for all } x, \tag{B.4}$$

where $H = \nabla^2 f(x)$ is the Hessian matrix, which provides a simple way to evaluate convex functions.

– Degree of convexity:

Given a convex function, it is often of interest how strong the convexity of the function is. Hessian matrix is a useful tool to describe the degree of convexity of a function. If H is close to rank deficient at a point, then the function is *flat* around that point and hence of *weak convexity*. Otherwise, if the minimum singular value of H is positive and large, the function is *curly* around that point and hence *strongly convex*. The degree of convexity influences the step size selection in gradient-descent algorithms, as shown later.

The lower and upper bounds of H play important roles in characterizing the function convexity.

– Lower bound of H: A function is called *strictly convex* if $\nabla^2 f(x) \succeq \ell I_n$ where $\ell > 0$ for all x. This definition of strong convexity is to set a positive lower bound for the second-order derivative of f. An equivalent first-order condition is $(x-y)^T(\nabla f(x) - \nabla f(y)) \geq m\|x-y\|^2$ for all x, y. This first-order condition is equivalent to $\nabla^2 f(x) \succeq \ell I_n$. The proof is based on the mean value theorem in (B.2) and omitted here.

– Upper bound of H: If H is bounded from above so that $\nabla^2 f(x) \leq LI_n$, then the change of the first order derivative $\nabla f(x)$ could not be arbitrarily fast; or equivalently, the function could not be arbitrarily convex at some points.

The upper bound can be implied by a Lipschitz condition of $\nabla f(x)$ as shown below.

Lemma B.1. *Suppose f is a convex function. If $\nabla f(x)$ is Lipschitz continuous with constant L:*

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x-y\|, \quad \textit{for all } x, y,$$

then $\nabla^2 f(x) \preceq LI_n$ for all x. Here, $\|\cdot\|$ is a vector norm.

Proof. Following (B.2), we have $\nabla f(x) - \nabla f(y) = \nabla^2 f(z)(x-y)$. It follows from the Lipschitz property that

$$\|\nabla f(x) - \nabla f(y)\| = \|\nabla^2 f(z)(x-y)\| \leq L\|x-y\|.$$

Let $v \doteq x - y$. Then we have

$$\|\nabla^2 f(z)v\| \leq Lv \Rightarrow \left\|\nabla^2 f(z)\frac{v}{\|v\|}\right\| \leq L. \tag{B.5}$$

By definition, the maximum singular value of $\nabla^2 f(z)$ is

$$\sigma_{\max}(\nabla^2 f(z)) \doteq \max_{v,\|v\|=1} \|\nabla^2 f(z)v\|.$$

Since (B.5) is valid for all v, we have $\sigma_{\max}(\nabla^2 f(z)) \leq L$. Since $\nabla^2 f(z)$ is positive semi-definite, we know $\nabla^2 f(z) \preceq LI_n$. Since z is determined by x, y which can be arbitrarily selected, the theorem is proven. $\square$

B.2 Gradient-Descent Algorithms

Consider the following optimization problem:

$$\min_x f(x)$$

where $x \in \mathcal{D} \subseteq \mathbb{R}^n$ and $f : \mathcal{D} \to \mathbb{R}$.

The well-known gradient-descent algorithm is

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k), \quad k = 0, 1, 2, \ldots \tag{B.6}$$

where α_k is a positive coefficient that may be fixed or time-varying. Here, $\alpha_k$is called the *step size* or *learning rate.*

Why can this algorithm solve the optimization problem? We next first given some intuition. The rigorous proofs will be given later.

1) Direction of change: $\nabla f(x_k)$ is a vector, which points to a direction along which $f(x_k)$ increases the fastest. Hence, the term $-\alpha_k \nabla f(x_k)$ changes x_k in the direction along which $f(x_k)$ decreases the fastest.
2) Magnitude of change: The magnitude of the change $-\alpha_k \nabla f(x_k)$ is jointly determined by the step size α_k and the magnitude of $\nabla f(x_k)$.

   – Magnitude of $\nabla f(x_k)$:

     – When x is close to the optimum x^* where $\nabla f(x^*) = 0$, the magnitude of $\nabla f(x_k)$, which is $\|\nabla f(x_k)\|$, is small. In this case, the update of x_k is slow, which is reasonable because we do not want to update x too aggressively to miss the optimum.
     – When x_k is far from the optimum, the magnitude of $\nabla f(x_k)$ may be large, and hence the update of x_k is fast. This is also reasonable because we hope the estimate could get close to the optimum as fast as possible.

   – Step size α_k:

     – If α_k is too small, the magnitude of $-\alpha_k \nabla f(x_k)$ is too small and hence the convergence is slow. If α_k is too large, the update of x_k is aggressive, which either leads to fast convergence or divergence.

– How to select α_k? The selection of α_k should depend on the degree of convexity of $f(x_k)$. Here, the degree of convexity can be described by the Hessian matrix $\nabla^2 f$. If the function is *curly* around the optimum (the degree of convexity is strong), then the step size α_k should be small to guarantee convergence. If the function is *flat* convex around the optimum (the degree of convexity is weak), then the step size could be large so that x_k could approach the optimum fast. The above intuition will be verified in the following convergence analysis.

Convergence analysis

To prove the convergence of the gradient-descent algorithm in (B.6), we need to make some assumptions.

– $f(x)$ is convex and twice differentiable: The mathematical condition given by this assumption is

$$\nabla^2 f(x) \succeq 0.$$

– $\nabla f(x)$ is Lipschitz continuous with constant L: Following Lemma B.1, the mathematical condition given by this assumption is

$$\nabla^2 f(x) \preceq LI_n.$$

This assumption requires that the first derivative of the function could not change arbitrarily fast.

We next give two proofs to show that (B.6) converges.

The First Proof

Recall

$$f(y) = f(x) + \nabla f(x)^T(y-x) + \frac{1}{2}(y-x)^T\nabla^2 f(z)(y-x)$$

as shown in (B.3). Here, z is a convex combination of x, y. Since $\nabla^2 f(z) \preceq LI_n \Rightarrow \|\nabla^2 f(z)\| \leq L$, we have

$$\begin{aligned} f(y) &\leq f(x) + \nabla f(x)^T(y-x) + \frac{1}{2}\|\nabla^2 f(z)\|\|y-x\|^2 \\ &\leq f(x) + \nabla f(x)^T(y-x) + \frac{L}{2}\|y-x\|^2. \end{aligned} \tag{B.7}$$

Replacing y, x in (B.7) by x_{k+1}, x_k, respectively, gives

$$\begin{aligned} f(x_{k+1}) &\leq f(x_k) + \nabla f(x_k)^T(-\alpha_k \nabla f(x_k)) + \frac{L}{2}\|\alpha_k \nabla f(x_k)\|^2 \\ &= f(x_k) - \alpha_k \|\nabla f(x_k)\|^2 + \frac{\alpha_k^2 L}{2}\|\nabla f(x_k)\|^2 \\ &= f(x_k) - \underbrace{\alpha_k \left(1 - \frac{\alpha_k L}{2}\right)}_{\eta_k} \|\nabla f(x_k)\|^2. \end{aligned} \tag{B.8}$$

We next show that if we select

$$0 < \alpha_k < \frac{2}{L}, \tag{B.9}$$

then the sequence $\{f(x_k)\}_{k=1}^{\infty}$ converges to $f(x*)$ where $\nabla f(x^*) = 0$. First, (B.9) implies that $\eta_k > 0$. It then follows from (B.9) that $f(x_{k+1}) \leq f(x_k)$. Second, we know that $f(x_k)$ is bounded from below by $f(x^*)$, which is the minimum value. As a result, the sequence converges as $k \to \infty$ according to the monotone convergence theorem. Suppose the limit of the sequence is f^*. Then, taking limit on both sides of (B.8) gives

$$\begin{aligned} &\lim_{k\to\infty} f(x_{k+1}) \leq \lim_{k\to\infty} f(x_k) - \lim_{k\to\infty} \eta_k \|\nabla f(x_k)\|^2 \\ &\Leftrightarrow f^* \leq f^* - \lim_{k\to\infty} \eta_k \|\nabla f(x_k)\|^2 \\ &\Leftrightarrow 0 \leq - \lim_{k\to\infty} \eta_k \|\nabla f(x_k)\|^2. \end{aligned}$$

Since $\eta_k \|\nabla f(x_k)\|^2 \geq 0$, the above inequality implies that $\lim_{k\to\infty} \eta_k \|\nabla f(x_k)\|^2 = 0$. As a result, x converges to x^* where $\nabla f(x^*) = 0$. The proof is complete.

If we consider the simplest case where $\alpha_k = \alpha$ is constant, then

$$\eta_k = \eta = \alpha \left(1 - \frac{\alpha L}{2}\right).$$

If $0 < \alpha < 2/L$, then $\eta > 0$ and hence $\|\nabla f(x_k)\|^2 \to 0$ as $k \to \infty$. As a result, $\nabla f(x_k)$ converges to zero, indicating that the minimum is achieved.

The above proof is inspired by [40].

The inequality in (B.9) provides valuable insights in how α_k should be selected. First, the step size should be sufficiently small so that it is bounded from above. The upper bound is determined by the convexity of the function. If the function is flat (L is small), the step size could be large; otherwise, if the function is strongly convex (L is large), then the step size must be sufficiently small.

The Second Proof

Another way to prove the convergence of (B.6) is based on the contraction mapping theorem, which is a general method to analyze the convergence of sequences. We can rewrite (B.6) as

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k) \doteq v(x_k).$$

If we can show that $v(x) : \mathbb{R}^n \to \mathbb{R}^n$ is a contraction mapping, then $\{x_k\}$ converges to the fixed-point satisfying $x^* = v(x^*)$. This is the idea of the proof. The details are given below.

Consider the simplest case where $\alpha_k = \alpha$ is constant. For any x, y we have

$$\begin{aligned} v(x) - v(y) &= x - \alpha \nabla f(x) - y + \alpha \nabla f(y) \\ &= x - y - \alpha(\nabla f(x) - \nabla f(y)) \\ &= x - y - \alpha \nabla^2 f(z)(x - y) \\ &= (I_n - \alpha \nabla^2 f(z))(x - y), \end{aligned}$$

where z is a convex combination of x, y according to the mean value theorem in (B.2). As a result,

$$\|v(x) - v(y)\| \leq \|I_n - \alpha \nabla^2 f(z)\| \|x - y\|. \tag{B.10}$$

Assume $\ell I_n \preceq \nabla^2 f(z)$ which means $f(x)$ is strictly convex. Then, we have $\ell I_n \preceq \nabla^2 f(z) \preceq L I_n$ and hence

$$(1 - \alpha L) I_n \preceq I_n - \alpha \nabla^2 f(z) \preceq (1 - \alpha \ell) I_n.$$

If we select

$$0 < \alpha < \frac{1}{L},$$

then

$$0 \prec (1 - \alpha L) I_n \preceq I_n - \alpha \nabla^2 f(z) \preceq (1 - \alpha \ell) I_n \prec I_n.$$

Hence, $I_n - \alpha \nabla^2 f(z)$ is positive definite and in the meantime $\|I_n - \alpha \nabla^2 f(z)\| \leq 1 - \alpha \ell < 1$. As a result, (B.10) becomes

$$\|v(x) - v(y)\| \leq (1 - \alpha \ell) \|x - y\|.$$

Since $0 < \alpha < 1/L$, we have $0 < 1 - \alpha \ell < 1$. As a result, the above inequality indicates that $v(x)$ is a contraction mapping. It then follows from the contraction mapping theorem that x converges to the fixed point x^* satisfying $v(x^*) = x^*$. It can be verified that $\nabla f(x^*) = 0$. The proof is complete.

The second proof requires the function to be *strongly convex* such that $\ell I_n \preceq \nabla^2 f(z)$, whereas the first proof does not. The second proof is inspired by [41, Lemma 3]. More information about convex optimization can be found in [39].

Bibliography

[1] M. Pinsky and S. Karlin, *An introduction to stochastic modeling (3rd Edition).* Academic press, 1998.

[2] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming.* John Wiley & Sons, 2014.

[3] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[4] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-dynamic programming.* Athena Scientific, 1996.

[5] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *International Conference on Machine Learning*, vol. 99, pp. 278–287, 1999.

[6] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[7] H.-F. Chen, *Stochastic approximation and its applications*, vol. 64. Springer Science & Business Media, 2006.

[8] J. Lagarias, "Euler's constant: Euler's work and modern developments," *Bulletin of the American Mathematical Society*, vol. 50, no. 4, pp. 527–628, 2013.

[9] J. H. Conway and R. Guy, *The book of numbers.* Springer Science & Business Media, 1998.

[10] A. Dvoretzky, "On stochastic approximation," in *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability*, University of California Press, 1956.

[11] T. Jaakkola, M. I. Jordan, and S. P. Singh, "On the convergence of stochastic iterative dynamic programming algorithms," *Neural computation*, vol. 6, no. 6, pp. 1185–1201, 1994.

[12] L. Bottou, "Online learning and stochastic approximations," *Online learning in neural networks*, vol. 17, no. 9, p. 142, 1998.

[13] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, 1988.

[14] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems.* Technical Report, Cambridge University, 1994.

[15] C. J. C. H. Watkins, *Learning from delayed rewards.* PhD thesis, King's College, 1989.

[16] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[17] T. C. Hesterberg, *Advances in importance sampling.* PhD Thesis, Stanford University, 1988.

[18] M. Pinsky and S. Karlin, *An introduction to stochastic modeling.* Academic press, 2010.

[19] J. N. Tsitsiklis and B. Van Roy, "An analysis of temporal-difference learning with function approximation," *IEEE Transactions on Automatic Control*, vol. 42, no. 5, pp. 674–690, 1997.

[20] R. A. Horn and C. R. Johnson, *Matrix analysis.* Cambridge university press, 2012.

[21] M. G. Lagoudakis and R. Parr, "Least-squares policy iteration," *The Journal of Machine Learning Research*, vol. 4, pp. 1107–1149, 2003.

[22] R. Munos, "Error bounds for approximate policy iteration," in *Proceedings of the Twentieth International Conference on Machine Learning*, vol. 3, pp. 560–567, 2003.

[23] A. Geramifard, T. J. Walsh, S. Tellex, G. Chowdhary, N. Roy, and J. P. How, "A tutorial on linear function approximators for dynamic programming and reinforcement learning," *Foundations and Trends in Machine Learning*, vol. 6, no. 4, pp. 375–451, 2013.

[24] B. Scherrer, "Should one compute the temporal difference fix point or minimize the bellman residual? the unified oblique projection view,"

[25] D. P. Bertsekas, *Dynamic programming and optimal control: Approximate dynamic programming (Volume II).* Athena Scientific, 2011.

[26] S. J. Bradtke and A. G. Barto, "Linear least-squares algorithms for temporal difference learning," *Machine Learning*, vol. 22, no. 1, pp. 33–57, 1996.

[27] K. S. Miller, “On the inverse of the sum of matrices,” *Mathematics magazine*, vol. 54, no. 2, pp. 67–72, 1981.

[28] S. A. U. Islam and D. S. Bernstein, “Recursive least squares for real-time implementation,” *IEEE Control Systems Magazine*, vol. 39, no. 3, pp. 82–85, 2019.

[29] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” *arXiv preprint arXiv:1312.5602*, 2013.

[30] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[31] J. Fan, Z. Wang, Y. Xie, and Z. Yang, “A theoretical analysis of deep q-learning,” in *Learning for Dynamics and Control*, pp. 486–489, PMLR, 2020.

[32] L.-J. Lin, *Reinforcement learning for robots using neural networks.* 1992. Technical report.

[33] C. D. Meyer, *Matrix analysis and applied linear algebra.* SIAM, 2000.

[34] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, “Policy gradient methods for reinforcement learning with function approximation,” *Advances in neural information processing systems*, vol. 12, 1999.

[35] P. Marbach and J. N. Tsitsiklis, “Simulation-based optimization of markov reward processes,” *IEEE Transactions on Automatic Control*, vol. 46, no. 2, pp. 191–209, 2001.

[36] J. Baxter and P. L. Bartlett, “Infinite-horizon policy-gradient estimation,” *Journal of Artificial Intelligence Research*, vol. 15, pp. 319–350, 2001.

[37] X.-R. Cao, “A basic formula for online policy gradient algorithms,” *IEEE Transactions on Automatic Control*, vol. 50, no. 5, pp. 696–699, 2005.

[38] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Machine learning*, vol. 8, no. 3, pp. 229–256, 1992.

[39] S. Boyd, S. P. Boyd, and L. Vandenberghe, *Convex optimization.* Cambridge University Press, 2004.

[40] S. Bubeck *et al.*, “Convex optimization: Algorithms and complexity,” *Foundations and Trends in Machine Learning*, vol. 8, no. 3-4, pp. 231–357, 2015.

[41] A. Jung, “A fixed-point of view on gradient methods for big data,” *Frontiers in Applied Mathematics and Statistics*, vol. 3, p. 18, 2017.