ME5406

Deep Learning for Robotics

Part I

Tabular Methods

for

Reinforcement Learning

Peter C. Y. Chen PhD

Associate Professor
Department of Mechanical Engineering
Faculty of Engineering
National University of Singapore
Email: mpechenp@nus.edu.sg
https://sites.google.com/view/peter-chen/home

Contents

Preface

These notes are for the part half (i.e., Part I) of the course. It focuses on tabular methods of reinforcement learning, with an introduction on deep reinforcement learning provided near the end. Under this syllabus, the first five chapters in these notes present a set of major tabular methods for both model-based and model-free reinforcement learning, while the last chapter gives an introduction to the concept of deep reinforcement learning in the context of the Deep-Q-Network method.

These notes have been prepared specifically with the goal of enabling students to learn independently the topics covered in this part of the module. In these notes, I have attempted to explain various abstract concepts in an intuitive way (but with some degree of rigor under suitable context), and to present accessible problems and their detailed solutions to illustrate these concepts.

Chapter 1 presents an overview of the topics covered in Part I of the course. It provides a "scaffold" for an intuitive but focused narrative that links up all the topics covered in this part, in order to enable students to "see" how these topics fit within the overall framework of reinforcement learning. In this context, this chapter serves as a road map that can be referred to frequently as students work through the subsequent chapters.

The subsequent chapters follow a natural progression from model-based to model-free approaches. The coverage of these approaches focuses on the key concepts and results. For students who are keen to explore this field of study beyond these key concepts and results, the following two textbooks recommended for this module are a good place to start: (1) *Reinforcement Learning*: An Introduction, Sutton and Barto, 2nd ed., 2018, and (ii) *Artificial Intelligence: A Modern Approach*, Russell and Norvig, 4th ed., 2020.

Peter C. Y. Chen

January 2026

Part A

Model-based Approaches

Chapter 1

Introduction

This chapter presents an informal but still somewhat technical overview of the major concepts in reinforcement learning. The purpose here is to develop an intuitive narrative on the key methods for formulating and solving reinforcement-learning problems. The emphasis is on providing a streamlined exposition (at the expense of rigor and completeness) on these key methods, in order to demonstrate, at a simplistic level, the utility and versatility of the methods of (deep) reinforcement learning.

1.1 What is Reinforcement Learning?

Imagine that a baby is trying to learn how to crawl on the floor to chase a small moving toy car, with her mother observing nearby. The baby initially may move her hands and legs in some seemingly random patterns, and might fall over during the process. After some attempts she seems to be able to find on her own some coordinated motion patterns of her hands and legs. This eventually enables her to crawl fast enough to catch the toy car, which can be thought of as a form of *reward*. On subsequent occasions when the baby sees the toy car moving around, she will try to "recall" what she did correctly last time (i.e., how she crawled fast enough to catch the toy car), and try to execute a similar pattern of hand and leg movement in order to catch it again. As this "try-and-recall" cycle repeats many times, the baby will finally learn the "best" movement pattern and can execute it seemingly effortlessly. This appears to be the natural way for a baby to learn many physical skills during early development.

Now imagine that we are building a four-legged robot and trying to make it crawl forward on the floor in the same way a baby does. The traditional approach to realize such a crawling robot is to first study a baby's learned patterns of limb movement during crawling, and then to program the movement of mechanical legs of the robot in order to

mimic the baby's coordinated patterns of limb movement. This is analogous to getting a baby to crawl by having her mother hold the baby's hands and legs and to move them in some pre-determined sequence.

In the narrow context of this learning-to-crawl scenario, the essence of reinforcement learning resides in the solution to the following problem: *How can we make the robot learn to crawl on its own, in the same way a baby does*?

1.2 A Crawling Robot

A solution (produced by a method of reinforcement learning) to the problem above has been reported in [24]. We will use this example to illustrate the key aspects of this solution approach.

Figure 1.1 shows a two-link robot crawler. The robot uses the contact between the tip of its forearm link and the ground as an anchoring point to pull its body forward, mimicking a crawling motion, as illustrated in the two sub-figures labeled ① and ②. Each joint can be commanded to take one of five discrete pre-set angles; that is each of θ_1 and θ_2 can take on one of these five pre-set angles. This yields a total of 25 "link configurations" for the robot. The photo in Figure 1.1 shows three such configurations.

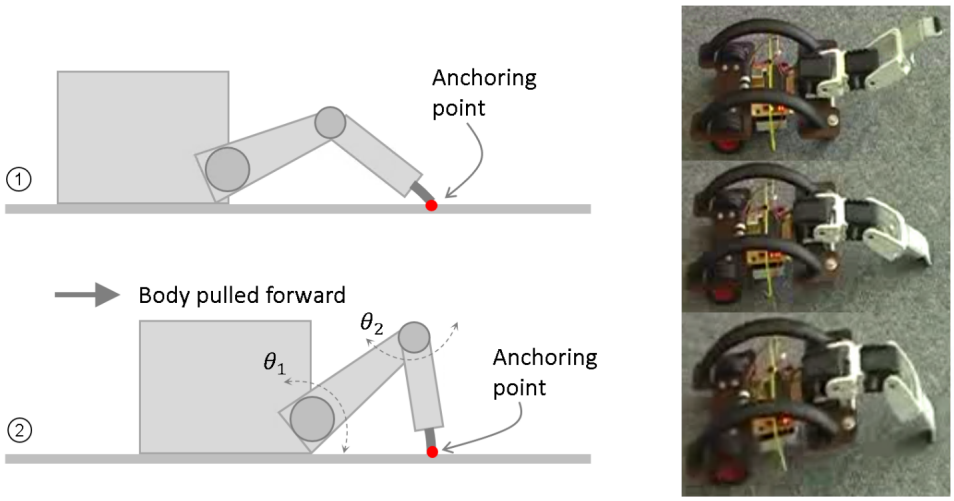


Figure 1.1: A simple crawling robot [24].

If we refer to each configuration, i.e., each pair of (θ_1, θ_2), as a *state*, then the robot will have a state space of 5×5, as is illustrated in Figure 1.2(a). So when a robot moves from one state to another, its configuration changes. We refer to such a change as a *transition*, as indicated by the arrows in Figure 1.2. A diagram that depicts all the states and all the permitted transitions between two states is called a *state transition diagram* (or a state-transition structure). Hence, a state-transition diagram contains all possible transition

sequences, i.e., all possible link movement of the robot. We would naturally expect that only a subset of these sequences generates the most speedy crawling motion, as illustrated in 1.2(b).

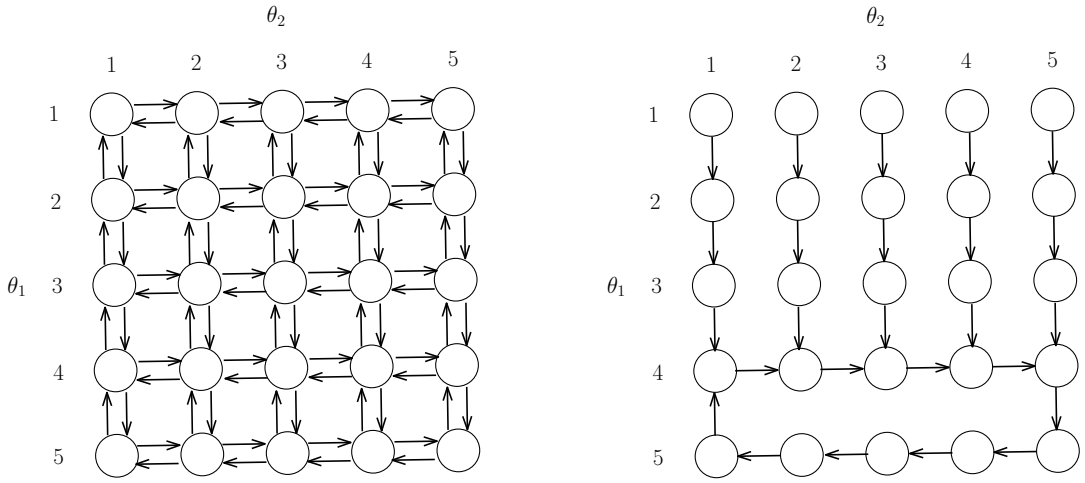


Figure 1.2: State-transition diagram of the crawling robot example. (a) State space of the robot movement, where the numbers 1 to 5 indicate the five pre-set discrete angles of each joint. (b) The transition sequences that produce the most speedy crawling motion.

To control the robot in order to generate the most speedy crawling motion, one can simply program the robot controller to execute the prescribed sequences of link movement, analogous to the mother holding the baby's hands and leg to enable her to crawl. A robot running such hard-coded programs lacks "intelligence" in the sense that it only does what it is instructed to do, while the instructions come from an external source (such as a human programmer). A more sophisticated approach would be to make the robot generate its own instructions (based on the task at hand) and then execute them in order to complete a task.

Now suppose that we just let the robot learn those sequences by "trial-and-error", in the way similar to how the baby learns to crawl on her own. We can allow the robot to randomly move its links to find out the effect on the position of the robot body. On the occasions that these random movements lead to some degree of forward motion, the robot controller will recognize the motion thus generated as a form of reward — the faster the motion, the greater the reward. In other words, the amount of reward is represented by the speed of the crawling motion. The robot controller will then

1. evaluate the relative importance (or 'worth') of each link configuration (i.e., state) in the generation of such forward motion,
2. quantify this worth in terms of a numerical value — which we simple call the Q *values* in this chapter, and
3. store the Q values for subsequent evaluation.

If the robot controller is designed to pursue more rewards by trying more configurations that lead to faster forward motion, then eventually an optimal sequence of link movements — as shown in Figure 1.2(b) — will be found, since the number of link configurations (i.e., states) is fixed at 25 and the angular speed of each joint that actuates a link is physically limited.

In the approach described above, we did not provide the robot controller any specific instructions on which link to move by how much and when. We only program it (i) to try various link movements, (ii) to evaluate the worth of each state, and (iii) to maximize the reward. Eventually the robot "learns" how to achieve the fastest possible crawling motion all by itself. *This is reinforcement learning*.

1.3 Formulation of Reinforcement Learning Problems

This section presents a streamlined and intuitive narrative on

(i) how a reinforcement learning problem can be formulated, and
(ii) a selected set of key approaches currently available to solving it.

Certain gaps are left unfilled and details omitted in order to keep the exposition accessible with minimal requirement of prior knowledge. This section is intended to serve as a precursory preparation for the more technical discussions in the subsequent chapters.

1.3.1 The general setting

agent and evironment

Reinforcement learning concerns the problem of an "autonomous" entity that learns from its interaction with its environment while performing a task. This entity is called the *agent*, while whatever external entities that the agent interacts with is (collectively) called the *environment*.

action

state

policy

reward

The agent-environment interaction is characterized by the *actions* (denoted by a) taken by the agent. The specific condition (or circumstance) under which the agent takes an action is called a *state* (denoted by s). The deterministic relationship between a state and an action is called a *policy* (denoted by π); that is, if the agent at state s takes an action a, then we write $\pi(s) = a$. Taking an action has two consequences that are recognized after a time delay (denoted by Δt). First, the agent moves to a (usually) different state. Second, the agent receives from the environment an evaluation on the effect of taking action a, in the form of a reward (denoted by r) represented by a (usually positive) real number[1]. The reward is determined by the environment alone.

[1]A negative value for the reward is also referred to as a penalty.

Let S_t and A_t be the variables representing the state of, and the action taken by, the agent at time t, respectively. Let R_{t+1} be the variable representing the reward received by the agent after taking action A_t. When executing a task, the agent carries out a cyclic process as follows. Following its policy π, the agent takes action $A_t = a$ at state $S_t = s$ at time t, i.e., $\pi(s) = a$. Due to this action, after one time step, i.e., at time $(t+1)$, the agent finds itself at a new state $S_{t+1} = s'$ and receives a reward $R_{t+1} = r$. Figure 1.3 illustrates this cyclic process, where Δt is the delay of one time step, i.e., $\Delta t = (t+1) - t = 1$.

task execution

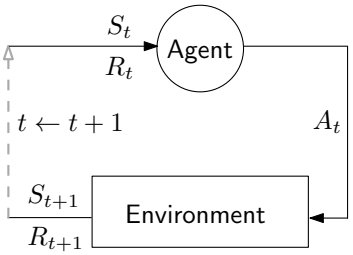


Figure 1.3: Agent-environment interaction.

An agent may carry out a task indefinitely or terminate it at some state. A task that runs indefinitely is referred to as *continuing*, while one that ends at some state is referred to as *episodic*. A cleaning robot (i.e., the agent) continuously roaming one floor of an office building (the environment) and picking up discarded items while recharging itself occasionally is performing a continuing task. A mobile robot (the agent) in a maze (environment) finding a way to the exit is performing an episodic task.

continuing and episodic tasks

The objective of the agent is to perform its task while maximizing the total rewards it receives. The problem of reinforcement learning is to find (as a solution) a policy that enables the agent to reach its objective. A solution to the reinforcement learning problem relies on the definition of the so-called *value functions*. There are two types of such functions, namely, the *state value functions* and the *action value functions*. The first type, denoted by $v(s)$, represents the 'worth' of an agent being at a specific state s, while the second type, denoted by $q(s, a)$, represents the 'worth' of an agent being at a specific state s and taking a specific action a. We will use a simple example to illustrate the utility of the second type of value functions in finding a solution to a reinforcement learning problem.

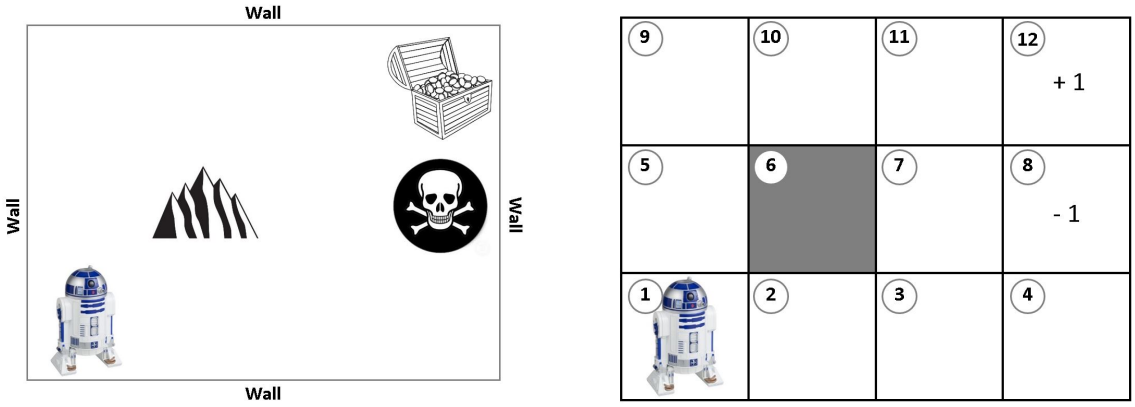
problem of reinforcement learning

value function

1.3.2 A simple example

Consider the treasure-hunting scenario illustrated in Figure 1.4(a). A treasure box is placed at the top right corner of a walled region. A deathtrap has been set to the south of the treasure box. A mobile robot is currently located at the lower left corner of the walled region. A hill blocks the direct access of the robot to the treasure box. The task of the robot is to move through the walled region, with the objective of reaching the treasure box.

To achieve this objective, the robot needs to find a path (starting from its current location) that leads to the treasure box while going around the hill and avoiding the deathtrap.



(a) (b)

Figure 1.4: A simple grid-world: (a) Agent hunts for treasure while avoiding deathtrap. (b) Model of reinforcement learning problem.

A reinforcement learning problem can be formulated for this treasure-hunting task by defining the states, actions, transitions, and rewards. First, the region is divided into sub-regions, such that the hill, the treasure box, and the deathtrap are contained in different regions. Each region can then be naturally considered as a state. Figure 1.4(b) shows the 12 states thus defined.

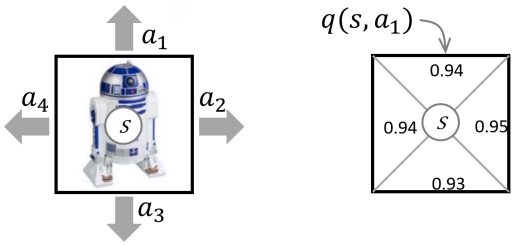
Second, the behavior of the robot when taking an action is defined to reflect the confined geometry of the region and the blocking effect of the hill. Normally the robot at a state s can take one of four actions, i.e., to move up (a_1), right (a_2), down (a_3), or left (a_4) as shown in Figure 1.5(a), and after taking an action the robot will end up in the adjacent state in the direction of that action. This can be expressed by a so-called *deterministic transition function*, denoted by $\bar{f}(s, a)$, whose output is the new state s'. For example, $\bar{f}(1, a_1) = 5$. When the robot is at a state next to a wall or to the hill and takes an action in the direction towards the wall or the hill, it will remain in the same state after taking the action. For instance, when the robot at state 5 takes action a_2 to go right (towards the hill), it will remain in state 5 after taking that action, i.e., $\bar{f}(5, a_2) = 5$. A transition can be probabilistic. For instance, the robot taking action a at state s may end up in the new state s' with a 50% probability; such a transition can be expressed as: $f(s, a, s') = 0.5$, where $f(s, a, s')$ is called a *probabilistic transition function*.

deterministic transition function

probabilistic transition function

Third, to indicate the desire for reaching the treasure box while avoiding the deathtrap, a reward scheme is set up such that the robot will receive a reward of $+1$ and -1 when it enters state 12 and state 8, respectively, as is shown in Figure 1.4(b). This reward scheme can be expressed using the so-called *reward function*, denoted by $\bar{\rho}(s, a, s')$, where s is the current state, a is the action taken by the agent at s, and s' is the new state into which

reward function



(a) (b)

Figure 1.5: Actions and state-action values.

the agent enters after taking a. For instance, when the robot takes action a_2 at state 11 to enter state 12, the reward it receives is: $r = \bar{\rho}(11, a_2, 12) = +1$. There are situations where the reward function can be probabilistic, such as in the case of the so-called multi-armed bandit problem [23].

With the states, actions, the reward function, and the transition function thus established, we have now cast the treasure-hunting reinforcement-learning problem in the framework of the so-called *Markov Decision Processes* (MDP). We will next discuss (in general terms) how a solution can be obtained for this reinforcement learning problem.

Markov Decision Processes

1.3.3 The utility of value functions

Reinforcement-learning approaches to finding a solution to such a problem utilize the notion of *value functions*. There are, as we had mentioned earlier, two types of value functions. One represents the 'worth' of a state s, denoted by $v(s)$; the other represents the 'worth' of a state-action pair (s, a), denoted by $q(s, a)$. To keep the narrative focused, in this section we will talk about $q(s, a)$ only; also we will not go into detail about the formal definition of $q(s, a)$ and how the values of $q(s, a)$ are calculated. It suffices to think of $q(s, a)$ as a quantitative measure of the amount of reward that the robot will collect when it carries out the task. For instance, the state-action values for the state s as shown in Figure 1.5(b) are:

value function $v(s)$ and $q(s, a)$

$$q(s, a_1) = 0.94 \quad q(s, a_2) = 0.95 \quad q(s, a_3) = 0.93 \quad q(s, a_4) = 0.94$$

From these four $q(s, a)$ values, we see that at state s taking the action a_2 (among the four actions available) has the highest 'worth'.

The following example demonstrates the utility of $q(s, a)$. Suppose that the robot is now at state 11. If it takes action a_2 to go right, it will enter state 12 and receive $r = +1$ as the reward. On the other hand, if it takes action a_3 to go downward, it will enter state 7 which is further away from state 12 and is now adjacent to state 8 (i.e., the deathtrap that

has the penalty of $r = -1$ upon entry). Figure 1.6 illustrates this situation. So intuitively, we see that the state-action pair of $(11, a_2)$ is more desirable than the pair of $(11, a_3)$, or quantitatively, $q(11, a_2) > q(11, a_3)$. These two stat-action values enable the robot to decide which action is "better" when it finds itself in state 11. If the robot were to choose between these two actions under two policies $\pi_i(11) = a_2$ and $\pi_j(11) = a_3$, obviously it should[2] choose π_i over π_j. So for the moment we can say that, with respect to state 11, policy π_i is "better" than π_j (expressed as $\pi_i > \pi_j$), with the understanding that this comparison is based on the fact that $q_{\pi_i}(11, a_2) > q_{\pi_j}(11, a_3)$, with $q_{\pi_{(\cdot)}}(s, a)$ represents the value of the state-action function $q(s, a)$ under the policy $\pi_{(\cdot)}$.

comparing policies

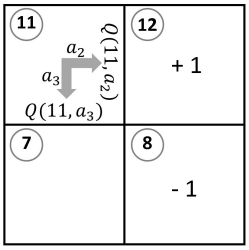


Figure 1.6: Two actions and their state-action values at state 11.

1.3.4 Policy extraction

It turns out that in a reinforcement learning problem (formulated in a state-transition structure with a given reward scheme as described above), for each (s, a) pair there is an optimal $q(s, a)$ value, denoted by $q_*(s, a)$. So the way to solve this reinforcement learning problem involves:

$q_*(s, a)$

1. determining the value of $q_*(s, a)$ for every state-action pair, then
2. picking the action associated with the largest $q_*(s, a)$ value at each state to form an *optimal* policy , denoted by π_*, i.e.,

optimal policy $\pi_*(s)$

$$\pi_*(s) = \text{argmax}_a \, q_*(s, a) \tag{1.1}$$

where the expression $\arg\max_x f(x)$ represents the value of x at which $f(x)$ attains its maximum. For example, with respect to the state s as shown in Figure 1.5(b), $\arg\max_a q(s, a) = a_2$.

Recall that in a reinforcement learning problem, the objective of the agent is to perform its task while maximizing the total rewards it receives. Solving this problem means finding a policy that enables the agent to reach its objective. In the treasure-hunting scenario, the

[2]We say "should" instead of "always" here because we have not considered the issue of exploration, which we will discuss in the subsequent chapters.

task of the robot is to move through the walled region, with the objective of reaching the treasure box. Solving this treasure-hunting problem means finding a path from state 1 to state 12 while avoiding state 8. An optimal policy $\pi_*(s) = \arg\max_a q_*(s,a)$, for all $s \in \{1, 2, \ldots, 12\}$, will yield this path.

Algorithms are available for computing $q_*(s,a)$ for reinforcement learning problems formulated under various settings. We will discuss a set of key algorithms in this module. As an illustration, suppose that the estimated optimal state-action values are as shown in Figure 1.7, where the number at an edge of the square is the state-action value for the action that results in the robot crossing that edge. From these values of $q_*(s,a)$, we can determine the optimal action at a state by selecting the action associated with the largest $q_*(s,a)$ value at that state. For instance, at state 5 the maximum of $q_*(s,a)$ is:

$$\begin{aligned} \max_a q_*(5,a) &= \max\left[q_*(5,a_1), q_*(5,a_2), q_*(5,a_3), q_*(5,a_4)\right] \\ &= \max\left[0.57, 0.51, 0.46, 0.51\right] \\ &= 0.57 = q_*(5,a_1) \end{aligned} \tag{1.2}$$

Hence, the optimal action at state 5 is: $\arg\max_a q_*(5,a) = a_1$; that is, the robot is to move upward. By identifying the optimal action for each and every state, we obtain the following optimal policy (which is indicated by the arrows in Figure 1.7):

$$\begin{array}{lll} \pi_*(1) = a_1 & \pi_*(2) = a_4 & \pi_*(3) = a_1 \\ \pi_*(4) = a_4 & \pi_*(5) = a_1 & \pi_*(7) = a_1 \\ \pi_*(9) = a_2 & \pi_*(10) = a_2 & \pi_*(11) = a_2 \end{array} \tag{1.3}$$

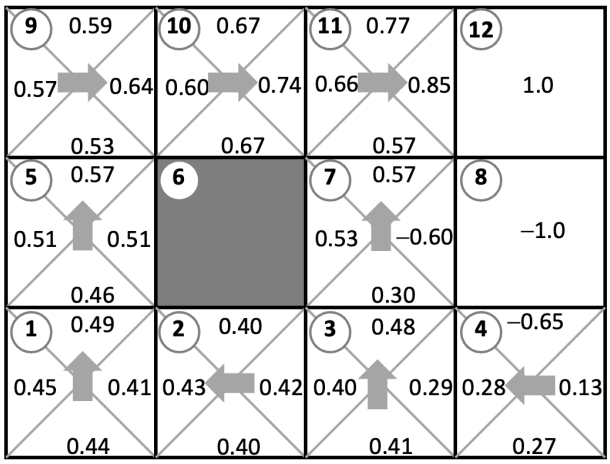


Figure 1.7: Estimated optimal state-action values $q_*(s,a)$ and policy $\pi_*(s)$ for the treasure-hunting problem.

Now from the optimal policy, a path from state 1 to state 12 that avoids state 8 can be identified as (in terms of the states): $1 \to 5 \to 9 \to 10 \to 11 \to 12$. This path is

the solution to the treasure-hunting problem for case where the robot is initially located in state 1. Note that an optimal policy π_* covers the entire state space; that is, it stipulates the optimal path for the robot to reach the treasure box starting from any state other than 6, 8 and 12.

1.4 Solution Approaches

From the discussion of the simple treasure-hunting example in the previous section, we see that, in general, solving a reinforcement learning problem involves the following three main steps:

1 *Model construction*: Formulating the problem in a state-transition structure with a reward pattern. (A complete model is not needed when a model-free solution approach is used.)
2 *Evaluation of value function*: Determining the optimal values of a value function, e.g., $q_*(s, a)$.
3 *Policy extraction*: Obtaining an optimal policy π_* based on the optimal values of the vale function, e.g., $q_*(s, a)$. This optimal policy is a solution to the reinforcement learning problem.

1.4.1 Model of Markov Dynamic Process

The treasure-hunting example illustrates that a reinforcement learning problem can be cast in the framework of Markov Decision Processes with the following elements:

- a set of states (denoted by $\mathcal{S}$)
- a set of actions ($\mathcal{A}$)
- a transition function ($\bar{f}$ or f)
- a reward function ($\bar{\rho}$)

When all these elements are known, they constitute a well-defined model of the interaction between the agent and its environment. Approaches to solving a reinforcement learning problem can thus be classified into two main types, namely, *model-based* and *model-free*, depending on whether a well-defined model is available.

1.4.2 Module-based approaches

If a well-defined model is available, a recursive relationship between the optimal values (either $v_*(s)$ or $q_*(s, a)$) of two states related by the transition function can be analytically established. This recursive relationship is called the *Bellman Optimality Equation*.

Finding a solution to the reinforcement learning problem then means solving the Bellman Optimality Equation to obtain the optimal values for either $v_*(s)$ or $q_*(s,a)$, from which an optimal policy π_* can be extracted — as we did earlier in Section 1.3.4 using $q_*(s,a)$. The difference between using $v_*(s)$ and $q_*(s,a)$ is that we need to know the transition function (i.e., $\bar{f}$ or f) in order to extract an optimal policy from $v_*(s)$. Bellman Equation

There are two main approaches to solving the Bellman Optimality Equation. One is to solve it algebraically, i.e., by solving a set of simultaneous equations. The other is to solve it via some iterative process. Collectively all these approaches are known as *Dynamic Programming*. The approach of solving the Bellman Equation iteratively is a major result in the field of reinforcement learning. Two main techniques under this approach are *policy iteration* and *value iteration*. Dynamic Programming

Policy iteration consists of two sub-processes that interact iteratively in order to yield an optimal policy. The first sub-process is called *policy evaluation*; the second *policy improvement*. The objective of policy evaluation is to determine the state value $v_\pi(s)$ for a given policy π, while the objective of policy improvement is to find a new policy π' such that $\pi' \geq \pi$. policy iteration

An algorithm that implements policy iteration would execute the following steps:

1. Generate a random policy π.
2. [*Policy evaluation*] Go through an iterative process to obtain a close estimate, denoted by $V_\pi(s)$, of the state values $v_\pi(s)$ for the policy π, i.e., $V_\pi(s) \approx v_\pi(s)$.
3. [*Policy improvement*] Find a "better" policy $\pi' > \pi$ based on the values of $V_\pi(s)$.
4. Repeat Step 2 (with $\pi \leftarrow \pi'$) and Step 3 until no $\pi' > \pi$ can be found, in which case the optimal policy is obtained, i.e., $\pi = \pi_*$.

Remark 1.4.1. *In the sequel, we use $v_\pi(s)$ to refer to the "true" values of the state value function under a policy π, and use $V_\pi(s)$ to refer to an estimate of $v_\pi(s)$. The same convention applies to $v_*(s)$ and $V_*(s)$, $q_\pi(s,a)$ and $Q_\pi(s,a)$, and $q_*(s,a)$ and $Q_*(s,a)$.* □ notational convention

Figure 1.8 illustrates the interaction between the two sub-processes of policy evaluation and policy improvement, where the wavy arrow indicates the iterative nature of the policy evaluation sub-process for determining $V_\pi(s)$, with V dnotes some intermediate value during the iterations.

Instead of trying to improve a policy in order to find one that is optimal (as in the case of policy iteration), value iteration aims to finding the optimal values $v_*(s)$ or $q_*(s,a)$ directly. Value iteration also consists of two sub-processes. The first is to obtain $V_*(s)$ or $Q_*(s,a)$. The second sub-process is to extract an optimal policy π_* from $V_*(s)$ or $Q_*(s,a)$. Unlike in policy iteration[3], these two sub-processes do not interact; they are carried out in sequence as illustrated in Figure 1.9. The first sub-process starts with some randomly value iteration

[3]There is another more subtle view about value iteration which puts value iteration under a more general

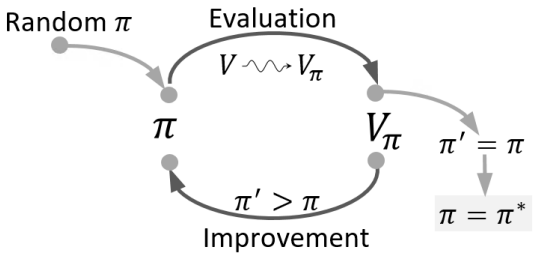


Figure 1.8: Policy iteration.

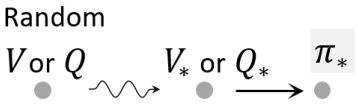


Figure 1.9: Value iteration.

selected values for V or Q and executes an iterative process (as indicated by the wavy arrow in Figure 1.9) to obtain $V_*(s)$ and $Q_*(s, a)$. The second sub-process then extracts an optimal policy π_* from $V_*(s)$ or $Q_*(s, a)$. We have in fact briefly encountered the idea of value iteration in Section 1.3.3, where Figure 1.7 shows the resulting state-action values from the first sub-process, while Equation (1.1) represents the second sub-process.

Solving model-based reinforcement learning problems using Dynamic Programming has two key drawbacks. First, a well-defined model (in the framework of Markov Decision Processes) is required. This requirement is rarely met in practice, because the transition function and the reward function may not be completely determinable in a practical application. Second, the solution process is similar to an exhaustive search, which is computationally intensive. Since practical problems often involve a large number of states and actions, such a solution process can easily become intractable. Model-free approaches overcome these two drawbacks.

drawbacks

1.4.3 Model-free approaches

Model-free approaches do not require the transition function and the reward function to be available at the start of the solution process. They attempt to estimate the state-action value function $q(s, a)$, from which an optimal policy is to be determined. There are two main classes of model-free methods. One is the *Monte Carlo methods*; the other is the *Temporal Difference methods*.

framework called *generalized policy iteration*. Under this framework, value iteration is interpreted as policy iteration with policy evaluation truncated after just one sweep (i.e., one update of every state). To keep our current discussion at the introductory level, we will stick with the 'process-oriented' view about value iteration as is illustrated in Figure 1.9.

Monte-Carlo Methods

Monte Carlo technique

The Monte Carlo methods in reinforcement learning are based on the general *Monte Carlo technique* of finding approximate solutions to mathematical problems using 'experience' gained by random sampling processes. In this module, we will use the term "Monte Carlo technique" to refer to the general mathematical technique, and use the term "Monte Carlo methods" to refer to the methods (based on the Monte Carlo technique) that are developed in the context of reinforcement learning.

Monte Carlo methods

Monte Carlo prediction

The Monte Carlo technique enables the implementation of the policy iteration process (illustrated in Figure 1.8) when a model of agent-environment interaction is not available. In this model-free setting, the policy evaluation sub-process can be implemented by the method of *Monte Carlo prediction* to estimate q_π, while the policy improvement sub-process is the same as that in policy iteration under the model-based setting. Figure 1.10 illustrates this implementation of policy iteration, which is also known as *Monte Carlo control*. Note that Monte Carlo prediction can also be used alone to find the state value $v_\pi(s)$ for a given policy π.

Monte Carlo control

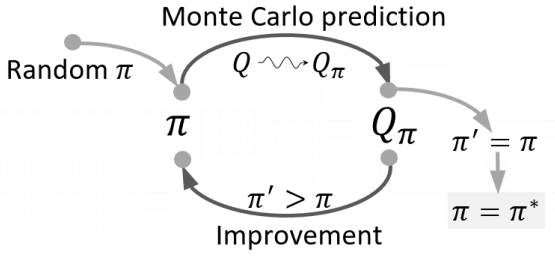


Figure 1.10: Monte Carlo control.

drawback

time step

The main drawback of applying the Monte Carlo technique in solving reinforcement learning problems is that the estimation of either $v_\pi(s)$ or $q_\pi(s,a)$ relies on episodic random sampling, whose outcome can only be calculated upon the completion of an episode (i.e., a sequence of state transitions with each transition considered to take one *time step*). This means that the estimated values $V_\pi(s)$ or $Q_\pi(s,a)$ can be updated only after an episode has ended, i.e., after a number of time steps. This adversely affects the efficiency of Monte Carlo prediction, and hence that of Monte Carlo control. The approach called Temporal Difference overcomes this drawback.

Temporal Difference methods

Similar to Monte Carlo methods, Temporal Difference methods can be used to solve reinforcement learning problems when a model is not available. The key difference between Monte Carlo methods and Temporal Difference methods is in the time frame by which they

update the estimated values $V(s)$ or $Q(s,a)$ when conducting policy evaluation.

In Monte Carlo methods, the estimated values $V(s)$ or $Q(s,a)$ are updated at the completion of a episode during policy evaluation. Temporal Difference methods, on the other hand, can execute the update before the completion of an episode — usually after completing just one or a few state transitions of an episode. This is referred to as *bootstrapping*. If we think of updating the estimates $V(s)$ and $Q(s,a)$ as making the agent "smarter", then bootstrapping enables the agent to become smarter sooner; therein lies the advantage of Temporal Difference methods over Monte Carlo methods. bootstrapping

The core idea of Temporal Difference methods is to update the selected values $V(s)$ or $Q(s,a)$ iteratively using an update rule that involves an error term and can be expressed in the following general form: update rule

$$\text{updated value} \leftarrow \text{current value} + \text{learning rate} \times \underbrace{(\text{estimated target value} - \text{current value})}_{\text{temporal difference error}} \quad (1.4)$$

In this part of the module, we will limit our discussion to three main methods, namely, TD(0), Q-learning, and SARSA. All three methods use the same form of the update rule as given in Equation (1.4) to iteratively find the desired values. They differ mainly in

(i) which type of values are being estimated, i.e., $v(s)$ or $q(s,a)$, and
(ii) how the estimated target value is calculated.

TD(0)

The TD(0) method is designed for policy evaluation (i.e., prediction). For a given policy π, this method randomly selects some initial values for $v(s)$, then executes the update rule as given in Equation (1.4) iteratively until the updated value converges to some close estimate of $v_\pi(s)$. TD(0)

Q-learning

The Q-learning method is designed for finding an optimal policy (i.e., for control). For a given reinforcement learning problem, this method randomly selects some initial values for $q(s,a)$ then executes the update rule as given in Equation (1.4) iteratively until the updated value converges to a close estimate of $q_*(s,a)$, from which an optimal policy can be extracted using Equation (1.1). Q-learning

Q-learning is a popular method in reinforcement learning for two main reasons. First, it is a model-free method. Second, it is simple to implement. In practice, however, the Q-learning method may become computationally intractable for reinforcement learning

problems that have a large number of states and also many possible actions at each state. This is often referred to as the "curse of dimensionality" in the literature. The Deep Q-Network (DQN) method was developed by DeepMind (now owned by Google) to address this issue.

SARSA

The SARSA method is similar to the Q-learning method. The main difference is that it calculates the estimated target value in Equation (1.4) in a different way, which has a significant implication on the behavior of the agent when following the optimal policies obtained by these two methods. SARSA

1.5 Deep Reinforcement Learning

In Q-learning, the main task is to find an estimated optimal state-action values $Q_*(s, a)$. Implementation of Q-learning, using the update rule as given in Equation (1.4), requires storage and retrieval of intermediate values of $Q(s_i, a_j)$ for state-action pairs (s_i, a_j). The usual practice is to use some data structure (e.g., a matrix) for this purpose. We can simply think of the data structure as a "look-up table" for the values of $Q(s_i, a_j)$, or simply a Q-table. When provided with the input s_i and a_j, we just search through this Q-table to find the value of $Q(s_i, a_j)$, as is illustrated in Figure 1.11.

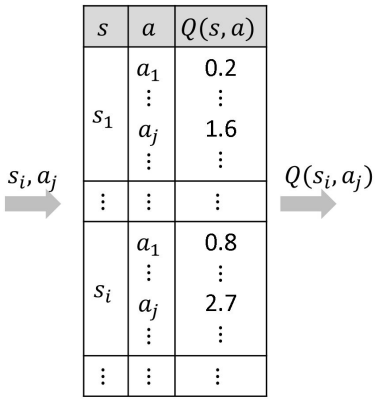


Figure 1.11: An example of a Q-table for obtaining $Q(s_i, a_j)$ for a given input state-action pair (a_i, s_j).

1.5.1 Why we need "deep" reinforcement learning

A major obstacle in using the Q-learning method to solve a practically meaningful problem is that the size of the Q-table can become so huge that it is not possible to "look up"

$Q(s_i, a_j)$ given s_i and a_j.

To get a sense of proportion about this issue, consider the ancient game of Go. In this game, two players take turn to place black and white stones at the vacant intersections of a grid marked on a board. The act of placing a stone is called a *move*, i.e., an action. At any given time during the game, the collective positions of all the stones on the grid constitute a *board position*, which can be thought of as a *state* in a reinforcement learning problem. The number of legal board positions (for the standard size of a 19×19 grid) has been estimated[4] to be approximately 2×10^{170}. The average number of possible moves from a board position is considered to be around 250. So the size of a Q-table for the game of Go would be about $250 \times 2 \times 10^{170}$ [25, 5], making searching the Q-table to find $Q(s_i, a_j)$ computationally intractable.

The Deep Q-Network (DQN) method provides a computational solution for dealing with this issue. It has produced substantial impact on practical implementation of reinforcement-learning solutions for real-life problems [19]. This computation solution is based on a biologically inspired computing system called *(artificial) neural networks*.

1.5.2 A quick introduction to neural networks

A neural network consists of a collection of *processing units* (also referred to as *neurons*). The units are arranged in layers, with each unit in a layer being connected to all the units in the next layer. Each such connection has a strength, represented by a *weight* value denoted by w. Signals flow from layer to layer in a single direction, as is illustrated in Figure 1.12(a). Usually all units (except those in the input layer) implements a nonlinear function, such the $\tanh(\cdot)$ function. This nonlinear function is called an *activation function*. The input to the activation function is the weighted sum of the signals from all units in the preceding layer, as is illustrated in Figure 1.12(b).

processing unit

connection weight

activation function

The layer that receives signals (e.g., x in Figure 1.12) from some external source is called the *input layer*. The layer that sends out signals (e.g., y_{nn} in Figure 1.12) to some external source is called the *output layer*. Any layer located between the input and output layers are called a *hidden layer*. This network structure is referred to as a *multilayer feedforward network*, also known as the *multilayer perceptron* (MLP).

Figure 1.12(a) shows a simple neural network with one input unit, one output unit, and one hidden layer with three units. We can see that the output y_{nn} depends on the input x and the weights $w_{11}^{(1)}$, $w_{21}^{(1)}$, etc. We can represent all these weights collectively in a suitably defined matrix $\mathbf{W}$, and express the relationship between y_{nn}, x and $\mathbf{W}$ in the form

[4]For a comparison, the number of atoms in the known observable universe is estimated to be about 10^{80} [8].

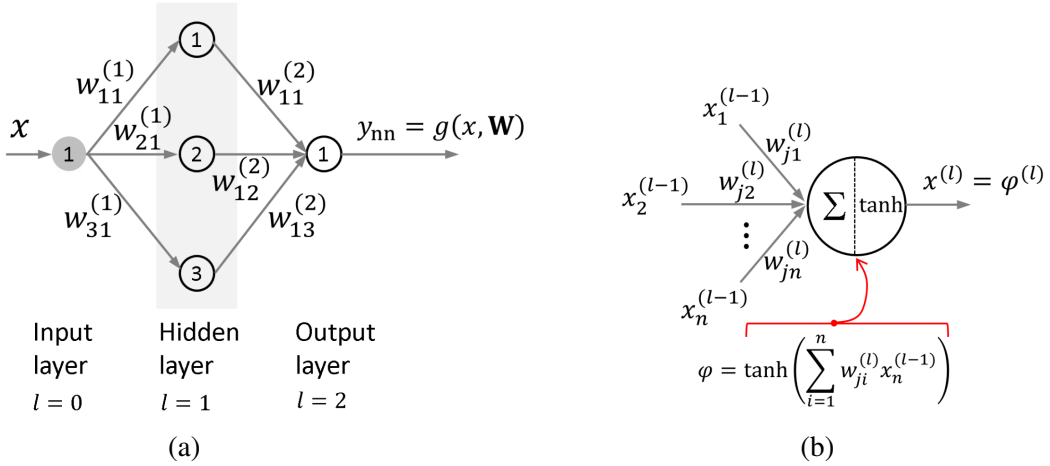


Figure 1.12: (a) A simple neural network with one 3-unit hidden layer. The symbol $w_{ji}^{(l)}$ denotes the weight from the unit i in layer $(l-1)$ to the unit j in layer l. (b) The computation done by a processing unit.

of a vector function $g(x, \mathbf{W})$, i.e.,

$$y_{\text{nn}} = g(x, \mathbf{W}) \tag{1.5}$$

It has been formally established that a suitably structured MLP can approximate any function to any degree of accuracy [7]. We will use the following simple example to illustrate how a MLP can approximate the parabolic function:

$$y = x^2 \tag{1.6}$$

The idea here is to find a set of optimal values $\mathbf{W}^*$ for the weights $\mathbf{W}$, i.e., $\mathbf{W} = \mathbf{W}^*$, such that for any given x, the value of y_{nn} is as close to y as possible, i.e.,

$$y - y_{\text{nn}} = x^2 - g(x, \mathbf{W}^*) \to 0 \tag{1.7}$$

That is, the neural network needs to run a "learning algorithm" to find the optimal weights $\mathbf{W}^*$. Figure 1.13 illustrates this approach of approximating a function using a neural network.

Without going into a detailed analysis, we will just point out here that the accuracy of the approximation (i.e., the difference between y and y_{nn}) in general depends on two factors:

1. the number of hidden units, and
2. the values of the weights $\mathbf{W}$.

To demonstrate the effect of these two factors on the accuracy of the approximation, Figure 1.14 shows sample outputs of two neural network configurations: one with 3 units, the

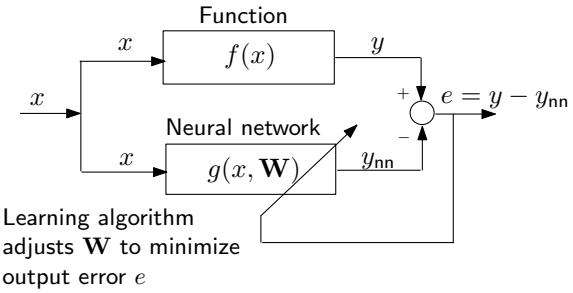
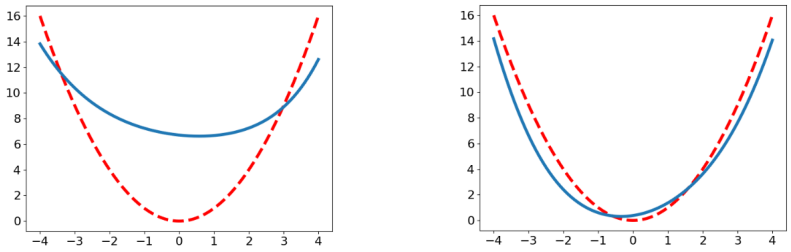


Figure 1.13: Neural network for function approximation.

other with 10 units, in their respective hidden layer. It can be seen that the network with more units in its hidden layer can approximate the parabolic function $y = x^2$ more accurately.



(a) (b)

Figure 1.14: Output of neural network (solid line) for approximating the function $y = x^2$ (dash line). (a) Network with a 3-unit hidden layer. (b) Network with a 10-unit hidden layer.

The questions of how to determine a suitable number of units in the hidden layer and how to find the optimal weight values $\mathbf{W}^*$ for a given function approximation problem is beyond the scope of the current discussion. It suffices to note here that it has been proved that a neural network is inherently capable of approximating any given function to any degree of accuracy if it contains sufficient number of hidden units [7]. It is in this sense that neural networks are also referred to as *universal approximators*.

universal approximator

Neural networks with many hidden layers, where the layers are designed to successively process their respective input to extract (from the input) high-level features, is often described as *deep neural networks*. One example is a network that processes the raw pixel-level data of an image (i.e., the input) to produce a list of labels (i.e., the output, such as "car", "cat", "dog", etc.) for certain regions of that image. What makes a reinforcement learning method "deep" can be attributed to the incorporation of such a deep neural network in the process of finding an optimal policy. Deep Q-Network is one such method.

deep neural network

1.5.3 Deep Q-Network (DQN)

In Q-learning, we "look up" the value $Q(s_i, a_j)$ from the Q-table for a given state-action pair (s_i, a_j). Suppose that there are M actions that the agent can choose from at state s_i. We can think of the Q-table as a vector function, denoted by $f(\cdot)$, that when provided with a state s_i as input produces the vector $\mathbf{q} = [Q(s_i, a_1),\, Q(s_i, a_2),\, \ldots,\, Q(s_i, a_M)]^T$ as output, i.e.,

$$\mathbf{q} = \begin{bmatrix} Q(s_i, a_1) \\ Q(s_i, a_2) \\ \vdots \\ Q(s_i, a_M) \end{bmatrix} = f(s_i) \tag{1.8}$$

The key idea in the DQN method is to use a neural network to approximate $f(s_i)$, as is illustrated in Figure 1.15.

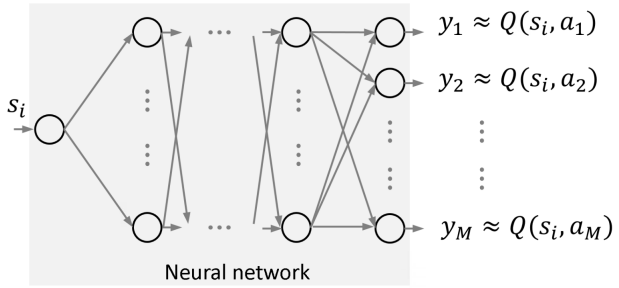


Figure 1.15: The approach used in a DQN for generating the output $Q(s_i, a)$ for all actions at a state s_i that is given as the input to the neural network.

Because neural networks are universal approximators, we can be assured that a suitably designed neural network $g(s_i, \mathbf{W})$ is capable of approximating the function $f(s_i)$ that represents the Q-table as discussed earlier. Hence, we can set up a neural network and try to find a set of optimal weights $\mathbf{W}^*$ such that

$$\mathbf{q} - \mathbf{y}_{\text{nn}} = f(s_i) - g(s_i, \mathbf{W}^*) = \begin{bmatrix} Q(s_i, a_1) \\ Q(s_i, a_2) \\ \vdots \\ Q(s_i, a_M) \end{bmatrix} - \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} \to \mathbf{0} \tag{1.9}$$

We are now in a position to address the question: *how does using a neural network enable us to avoid the intractable computation issue associated with searching the Q-table for the values of $Q(s_i, a_j)$*? The answer is that, for a given input s_i, computing the vector $\mathbf{y}_{\text{nn}}$ using a neural network $g(s_i, \mathbf{W}^*)$, i.e., $\mathbf{y}_{\text{nn}} = g(s_i, \mathbf{W}^*)$, is much less computationally intensive than using the original Q-table, because the size of the neural network (as

reflected by $\mathbf{W}^*$) is much smaller than the size of the Q-table. For example, the size of the network used by DeepMind's *AlphaGO* (as listed in Table 1.1) may seem large but is still manageable using modern computers [6].

Table 1.1: Configuration of value network in AlphaGO.

Input layer	$19 \times 19 \times 49$
Hidden layer	$19 \times 19 \times 192$ (12 layers) 19×19 (1 layer) 256 (1 layer)
Output layer	1

When the state s_i is represented by low-level information (e.g., raw pixel data of an image), a deep neural network, as briefly described in the previous section, can be used to approximate the "Q-table function" $\mathbf{q} = f(s_i)$. *This is the key idea in the Deep Q-Network method*.

1.6 Structure of Part I

An overview of the topics discussed in Part I of this module is depicted in Figure 1.16. The subsequent chapters discuss each topic in greater detail.

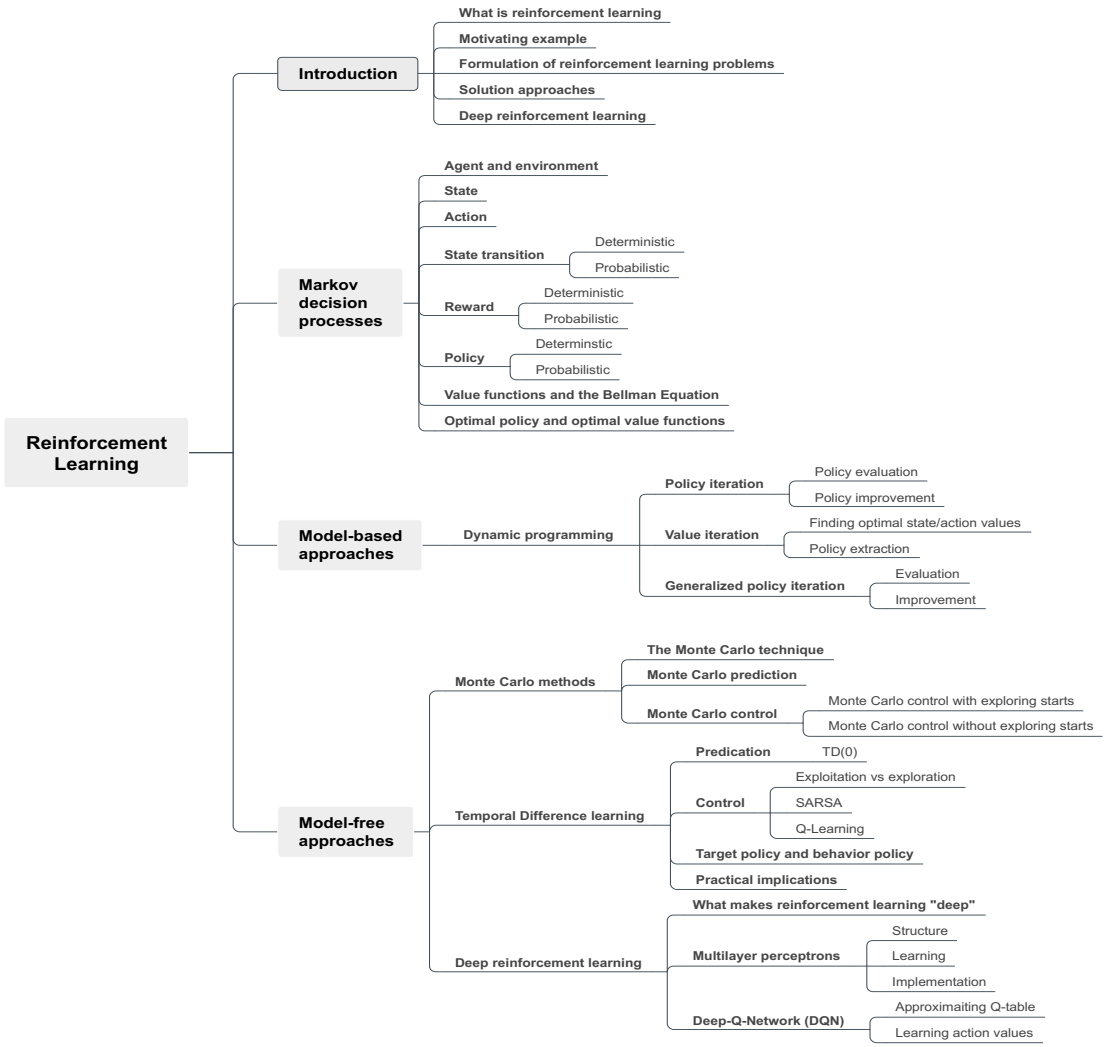


Figure 1.16: Overview of topics.

Chapter 2

Markov Decision Processes

Consider a battery-powered mobile cleaning robot operating in a narrow office corridor. At one end of the corridor is a power outlet where the robot can recharge its battery; at the other end is an empty can that the robot is supposed to pick up, as is illustrated in Figure 2.1. The robot initially has no prior knowledge of the locations of the power outlet and the can. The task of the robot is to find the can without running out of battery power.

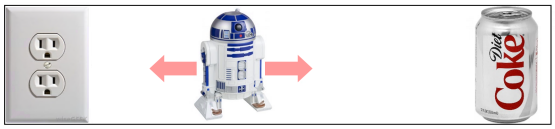


Figure 2.1: A cleaning robot.

Now suppose that the robot is somewhere in the corridor, what should the robot do? To solve this problem, we first need to cast the problem in an appropriate formal setting. To locate the robot more accurately, we can divide the whole corridor area into a number of regions, and called each region a *state*. Let there be 6 states, numbered from 0 to 5, with the location of power outlet fixed in state 0 and that of the can fixed in state 5. The robot is free to move among all the states. So for this problem, the state space is represented by the state set $\{0, 1, 2, 3, 4, 5\}$, as is illustrated in Figure 2.2. state

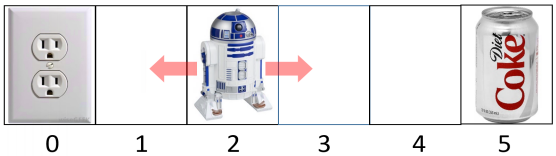


Figure 2.2: States in the cleaning robot example.

We assume that for a narrow corridor the robot can move in two directions only, namely, left and right. We call these *actions*, and denote them by -1 (for moving to action

the left) and $+1$ (for right). So at each state, the robot can take one of the two actions. In other words, the set of actions is $\{-1, +1\}$.

When the robot moves along the corridor, its movement can be described in terms of states and actions, in the sense that an (incremental) movement is realized by the robot at state s taking an action a to enter another state s'. This relationship among s, a, and s' can be expressed as a (deterministic) *transition function* $\bar{f}$ as: $\bar{f}(s, a) = s'$. For example, $\bar{f}(2, +1) = 3$. Once the robot enters either state 0 or state 5, we consider it to have completed its current task; that is, it will not abe able to leave those states on its own.

transition function

To induce the robot to find the can while making sure the batter does not run out, we can provide the robot a "reward" (denoted by r) when it finds the can or the power outlet. Since picking up the can has a higher priority over recharging, we can assign to it a larger reward; for example, a reward of $r = +5$ for finding the can and $r = +1$ for finding the power outlet. In order to collect a reward the robot must enter either state 0 or state 5 by taking an action at state 1 or state 4, such a reward can also be described in terms of states and actions in the form of a *reward function*, denoted by $\rho(s, a, s')$, where s is the current state, a is the action taken by the robot at s, and s' is the new state at which the reward is collected. For example, when the robot at state 4 takes action $+1$ and enters state 5 to collect a reward of 5, we can write: $r = \rho(4, +1, 5) = 5$.

reward function

Figure 2.3 shows the states, actions, and rewards for this simple cleaning robot problem.

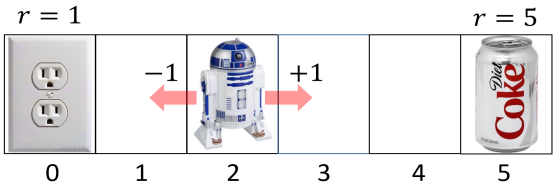


Figure 2.3: States, actions, and rewards in the cleaning robot example.

We now have the following description of the cleaning robot problem:

- A set of states: $\{0, 1, 2, 3, 4, 5\}$
- A set of actions: $\{-1, +1\}$
- A transition function:

$$\begin{array}{lllll} \bar{f}(0, \pm 1) = 0 & \bar{f}(1, +1) = 2 & \bar{f}(1, -1) = 0 & \bar{f}(2, +1) = 3 & \bar{f}(2, -1) = 1 \\ \bar{f}(3, +1) = 4 & \bar{f}(3, -1) = 2 & \bar{f}(4, +1) = 5 & \bar{f}(4, -1) = 3 & \bar{f}(5, \pm 1) = 5 \end{array}$$

- A reward function:

$$\begin{array}{lllll} \rho(0, \pm 1, 0) = 0 & \rho(1, +1, 2) = 0 & \rho(1, -1, 0) = 1 & \rho(2, +1, 3) = 0 & \rho(2, -1, 1) = 0 \\ \rho(3, +1, 4) = 0 & \rho(3, -1, 2) = 0 & \rho(4, +1, 5) = 5 & \rho(4, -1, 3) = 0 & \rho(5, \pm 1, 5) = 0 \end{array}$$

What we have just described is an example of formulating (in an informal way) a reinforcement learning problem using the framework of Markov Decision Processes. We will next discuss the framework of Markov Decision Processes in detail.

2.1 Definitions

Let $\mathbb{R}$ denote the set of real numbers, and $\mathbb{R}^n$ denote an n-dimensional space of real numbers. When we define a function, we can also express it as a "mapping" by specifying its domain and range. For example, a function f that takes as input two elements s and a from two sets $\mathcal{S}$ and $\mathcal{A}$, respectively, and produces as output an element s' that is also in the set $\mathcal{S}$, can be written as

$$f(s,a) = s', \quad s, s' \in \mathcal{S}, \quad a \in \mathcal{A}$$

and, in terms of a mapping, as

$$f : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$$

Definition 2.1.1. *(Argument of the maximum):* Suppose that a given function $f(x)$ has the maximum M. The set of values of x for which f(x) attains M is expressed as

$$\underset{x}{\operatorname{argmax}}\ f(x)$$

Example 2.1.1. For the function $f(x) = \sin x$,

$$\underset{x}{\operatorname{argmax}}\,(\sin x) = \{360°k + 90°\}, \qquad k = 0, 1, 2, \ldots$$

□

Definition 2.1.2. *(Expected value, or expectation, of a random variable):* Suppose that a random variable X can take value x_1 with probability p_1, value x_2 with probability p_2, and so on, up to value x_k with probability p_k. Then the expectation of X is defined as

$$\mathbb{E}[X] = x_1 p_1 + x_2 p_2 + \cdots x_k p_k \tag{2.1}$$

$\mathbb{E}$ is also called the expectation operator. □

The expectation operator is linear with the following properties:

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y] \tag{2.2}$$

$$\mathbb{E}[X + c] = \mathbb{E}[X] + c \tag{2.3}$$

$$\mathbb{E}[cX] = c\mathbb{E}[X] \tag{2.4}$$

where c is a real constant.

Definition 2.1.3. *(L^∞-norm):* The L^∞-norm of a vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$, denoted by $\|\mathbf{x}\|_\infty$, is the element of $\mathbf{x}$ with the largest magnitude, i.e.,

$$\|\mathbf{x}\|_\infty \triangleq \max_i |x_i| \tag{2.5}$$

2.2 Elements of a Markov Decision Process

A Markov Decision Process is a model of a dynamical system[1] comprising of the following elements, namely, *state*, *action*, *transition function*, and *reward*. In the first subsection below, we discuss the definitions of these four elements, then present a way to represent the dynamics of a Markov Decision Process compactly in the form of a four-argument p function.

2.2.1 State, action, transition and reward

In general, a state can be viewed as the situation in which the agent finds itself at any given time. In this module, we only consider discrete states; that is, change in state is to occur only at discrete time instants. When we describe the behavior of the agent as it moves from state to state, it is implicitly understood that such movement is associated with the passage of time. So we sometimes describe such movement with respect to time as follows: The agent at time t takes action A_t at state S_t, which results in the agent reaching state S_{t+1} one time step later at time $t+1$. Here S_t and S_{t+1} are the "time-based" variables denoting the state of the agent at time t and $t+1$, respectively.

S_t, A_t
S_{t+1}

For a given scenario, the behavior of the agent (when it is not under any form of control) can be described by the following three elements:

1. *A set of states*:
$$\mathcal{S} = \{s_1, s_2, \ldots, s_{|S|}\} \tag{2.6}$$
where $|\mathcal{S}|$ denotes the number of elements in $\mathcal{S}$.

2. *A set of actions at state s*:
$$\mathcal{A}(s) = \{a_1, a_2, \ldots, a_{|\mathcal{A}|}\} \tag{2.7}$$

[1]Such a system must exhibit the so-called Markov property, which can be intuitively summarized as "given the present, the future does not depend on the past". A formal definition can be found in many textbook on stochastic processes. Most of physical systems can be considered as having the Markov property. An example of a non-Markov process is that of ion movement across cell membrane [26].

If the number of actions is the same for all states, then we sometimes just simply write $\mathcal{A}$ instead of $\mathcal{A}(s)$.

3. *A transition function*: A transition function can be deterministic or non-deterministic. A deterministic transition function is defined as

$$\bar{f} : \mathcal{S} \times \mathcal{A} \to \mathcal{S} \tag{2.8}$$

In other words, a deterministic transition function $\bar{f}$ specifies the new state s' of the system after taking action a at state s, i.e.,

$$\bar{f}(s, a) = s' \tag{2.9}$$

A non-deterministic transition function, on the other hand, specifies the probability of the system reaching the new state s' after taking action a at state s, i.e.,

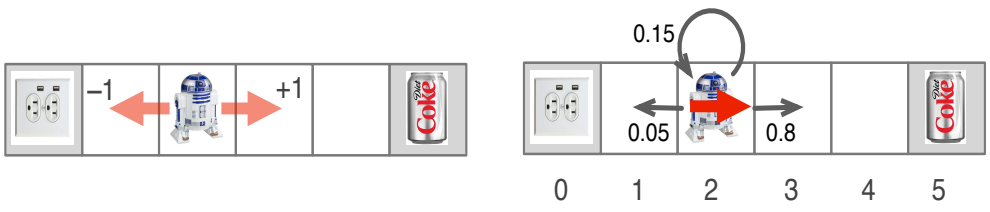
$$f : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1] \tag{2.10}$$

This probability is denoted by the symbol $P^a_{ss'}$, i.e.,

$$f\left(s, a, s'\right) = \mathbb{P}\left(S_{t+1} = s' \middle| S_t = s, A_t = a\right) \equiv P^a_{ss'} \tag{2.11}$$

where $\mathbb{P}(a|b)$ denotes the probability of 'a *being true under condition* b'. Note that $\bar{f}(s, a)$ and $f\left(s, a, s'\right)$ are equivalent if $P^a_{ss'} = 1 \equiv 100\%$.

Example 2.2.1. Consider the mobile robot designed for a cleaning task as shown in Figure 2.4(a). The robot's behavior consisting of picking up a can and recharging its battery. Suppose that the robot can only move to the left (defined as the action $a = -1$) or right ($a = 1$) in a horizontal grid, with each square in the grid representing one location of the robot. In this case, the robot is the agent with the state set $\mathcal{S} = \{0, 1, 2, 3, 4, 5\}$ and the action set $\mathcal{A} = \{-1, 1\}$.



(a) Robot for cleaning task. (b) Probabilities of transitions.

Figure 2.4: A simple cleaning task.

Suppose that due to uncertainties in the robot's operation (such as slippery floor or error in the robot's position-tracking system, etc.), the robot's action may become

non-deterministic. For example, for the case as shown in Figure 2.4(b), the transition function is defined as

$$\begin{cases} f(2,1,3) = 0.8 \\ f(2,1,2) = 0.15 \\ f(2,1,1) = 0.05 \end{cases} \tag{2.12}$$

In this case, even though the robot is "commanded" by its controller to move to the right to enter state 3, it only does so 80% of the time, and actually goes left to enter state 1 with a probability of 5%; for 15% of the time it just remains at state 2. □

4. *Reward function*: When the agent takes an action a to make a transition from state s to another state s', it receives a numerical reward r. This reward is received one time-step later, with the agent in the new state s'. If we use the "time-based" variables to express the reward, then we write

$$R_t = \bar{\rho}\left(S_{t-1}, A_{t-1}, S_t \,|\, S_{t-1} = s, A_{t-1} = a, S_t = s'\right) = \bar{\rho}(s, a, s') = r \tag{2.13}$$

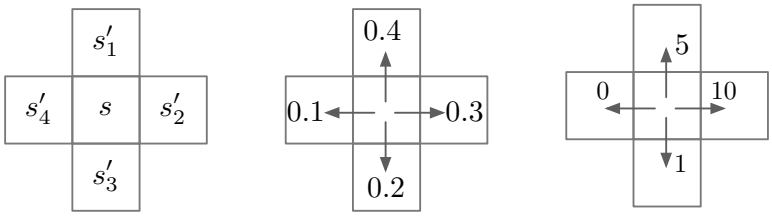
where R_t is the random variable representing the reward and r is a specific value from a real-value set $\mathcal{R} \in \mathbb{R}$, and

$$\bar{\rho} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathcal{R} \tag{2.14}$$

is called a deterministic reward function.

The arguments of the reward function $\bar{\rho}(s, a, s')$ presented above are s, a, and s'. It is also possible to define a reward function with respect to just s (i.e., $\bar{\rho}(s)$) or s and a (i.e., $\bar{\rho}(s, a)$) as its arguments, which may be more convenient in some setting.

Example 2.2.2. Consider the situation in which an agent at state s takes action a in a stochastic setting, as illustrated in Figure 2.5. Because of the non-deterministic transition function $f\,(s, a, s')$, there are four possibilities for the new state s', namely, s'_1, s'_2, s'_3, and s'_4.



(a) States. (b) Transition probabilities. (c) Rewards.

Figure 2.5: Taking a at s with a non-deterministic transition function.

The reward received by the agent for reaching the four different new states are (as

given in the figure):

$$\begin{aligned} R_{t+1}|_{s_{t+1}=s'_1} = \bar{\rho}\,(s,a,s'_1) &= 5 \\ R_{t+1}|_{s_{t+1}=s'_2} = \bar{\rho}\,(s,a,s'_2) &= 10 \\ R_{t+1}|_{s_{t+1}=s'_3} = \bar{\rho}\,(s,a,s'_3) &= 1 \\ R_{t+1}|_{s_{t+1}=s'_4} = \bar{\rho}\,(s,a,s'_4) &= 0 \end{aligned} \tag{2.15}$$

and the transition functions are:

$$\begin{aligned} f\,(s,a,s'_1) &= \mathbb{P}\,(s_{t+1}=s'_1|\,s_t=s,a_t=a) \equiv P^a_{ss'_1} = 0.4 \\ f\,(s,a,s'_2) &= \mathbb{P}\,(s_{t+1}=s'_2|\,s_t=s,a_t=a) \equiv P^a_{ss'_2} = 0.3 \\ f\,(s,a,s'_3) &= \mathbb{P}\,(s_{t+1}=s'_3|\,s_t=s,a_t=a) \equiv P^a_{ss'_3} = 0.2 \\ f\,(s,a,s'_4) &= \mathbb{P}\,(s_{t+1}=s'_4|\,s_t=s,a_t=a) \equiv P^a_{ss'_4} = 0.1 \end{aligned} \tag{2.16}$$

Thus, the expected reward for taking action a at state s, denoted by $r(s,a)$, at time step t is

$$\begin{aligned} r(s,a) &= \mathbb{E}\big[R_{t+1}\big] \\ &= \sum_{s'}\Big(P^a_{ss'},\ R_{t+1}\Big) = \sum_{s'} P^a_{ss'}\,\bar{\rho}\,(s,a,s') \\ &= P^a_{ss'_1}\,\bar{\rho}\,(s,a,s'_1) + P^a_{ss'_2}\,\bar{\rho}\,(s,a,s'_2) \\ &\quad + P^a_{ss'_3}\,\bar{\rho}\,(s,a,s'_3) + P^a_{ss'_4}\,\bar{\rho}\,(s,a,s'_4) \\ &= 0.4\times 5 + 0.3\times 10 + 0.2\times 1 + 0.1\times 0 \\ &= 5.2 \end{aligned} \tag{2.17}$$

□

Remark 2.2.1. *Note the difference, as illustrated in Example 2.2.2, between (i) the expected reward $r(s,a)$ for taking action a at s, and (ii) the reward r for taking a at s* and reaching a specific new state. *In that example, for case (i) the expected reward is 5.2, while for case (ii) the specific value of the reward depends on which new state is reached, i.e., r, will be one of the R_{t+1} values as indicated in Equation (2.15). There is also another notion $r(s)$ discussed below; it denotes the expected reward received by the agent at state s.* □

If the reward received by the agent is non-deterministic, the corresponding reward function gives the probability of the agent receiving a specific values for the reward r. Two forms of non-deterministic function have been in used the literature.

1. State dependent reward function: $\rho(r\,|\,s)$

The probability of the agent receiving a reward r depends only on the state s, i.e.,

$$\rho : \mathcal{R} \times \mathcal{S} \to [0, 1] \tag{2.18}$$

For example, $\rho(1 \mid 5) = 0.2$ means that the probability of the agent receiving a reward of 1 at state 5 is 20%.

The *expected reward* received by the agent at state s, denoted by $r(s)$, is (for m possible values of reward):

$$r(s) = \mathbb{E}\big[R_t \mid S_{t-1} = s\big] = r_1\, \rho(r_1 \mid s) + \cdots + r_m\, \rho(r_m \mid s) = \sum_r r\, \rho(r \mid s) \tag{2.19}$$

Note that r is a value of reward, while $r(\cdot)$ is a function for computing the expected reward.

2. State-and-action dependent reward function: $\rho(r \mid s, a)$

   The probability of the agent receiving a reward r depends on both the state s and the action a taken at s, i.e.,

$$\rho : \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1] \tag{2.20}$$

   In this case, the *expected reward* received by the agent for taking action a at state s, denoted by $r(s, a)$, is (for m possible values of reward):

$$\begin{aligned} r(s, a) &= \mathbb{E}\big[R_t \mid S_{t-1} = s,\, A_{t-1} = a\big] \\ &= r_1\, \rho(r_1 \mid s, a) + \cdots + r_m\, \rho(r_m \mid s, a) \\ &= \sum_r r\, \rho(r \mid s, a) \end{aligned} \tag{2.21}$$

2.2.2 The four-argument p function

We can combine the (probabilistic) transition function $f(s, a, s')$ and the state dependent reward function $\rho(r \mid s)$ to form a four-argument function, called the *p-function*, that gives the probability of the agent receiving a reward r after taking action a at state s and transitioning to state s', i.e.,

p-function

$$p(s', r \mid s, a) = f(s, a, s') \cdot \rho(r \mid s') \tag{2.22}$$

that is,

$$p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1] \tag{2.23}$$

Using this p-function, other quantitative characterizations of a Markov Decision Process

can be computed. A few cases are illustrated below.

- Probabilistic transition function $f(s,a,s')$:

$$f(s,a,s') \equiv p(s'|s,a) = \mathbb{P}\{S_t = s' \,|\, S_{t-1} = s, A_{t-1} = a\} = \sum_{r\in\mathcal{R}} p(s', r \,|\, s, a) \quad (2.24)$$

- Next-state dependent reward function $\rho(r \,|\, s')$:

$$p(s', r \,|\, s, a) = f(s,a,s') \cdot \rho(r \,|\, s') = \sum_{r\in\mathcal{R}} p(s', r \,|\, s, a) \cdot \rho(r \,|\, s')$$

$$\Rightarrow \quad \rho(r \,|\, s') = \frac{p(s', r \,|\, s, a)}{\sum_{r\in\mathcal{R}} p(s', r \,|\, s, a)} \quad (2.25)$$

- State-and-action dependent reward function $\rho(r \,|\, s, a)$:

$$\rho(r \,|\, s, a) = \sum_{s'\in\mathcal{S}} p(s', r \,|\, s, a) \quad (2.26)$$

- Expected reward based on state and action $r(s,a)$:

$$r(s,a) = \mathbb{E}\big[\, R_t | \, S_{t-1} = s,\, A_{t-1} = a \,\big] \;=\; \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} r p(s', r \,|\, s, a) \quad (2.27)$$

- Expected reward for state-action-next-state triples $r(s,a,s')$:

$$\begin{aligned} r(s,a,s') &= \mathbb{E}\big[\, R_t | \, S_{t-1} = s, A_{t-1} = a, S_t = s' \big] \\ &= \sum_{r\in\mathcal{R}} r\, \rho(r \,|\, s') = \sum_{r\in\mathcal{R}} r \cdot \frac{p(s', r \,|\, s, a)}{\sum_{r\in\mathcal{R}} p(s', r \,|\, s, a)} \end{aligned} \quad (2.28)$$

The above cases demonstrate the main the advantage of using the p-function in describing the dynamics of a Markov Decision Process. In the sequel, we will use this function whenever possible.

Remark 2.2.2. *In Equations (2.25) and (2.19), both $\rho(r|s')$ and $\rho(r|s)$ are state-dependent rewards. The use of the symbols 's' and 's' is a just choice based on context.* □

Example 2.2.3. In the discussion on the cleaning robot problem pertaining to Figure 2.2, when the robot takes action $a = +1$ at state s, it reaches state 5 according to the deterministic transition function $\bar{f}(4,+1) = 5$. Now suppose that

1. the reward that the robot will receive is numerically equivalent to the number of cans the robot finds at state 5, and

2. the number of cans at state 5 follows the distribution as shown in Table 2.1.

Determine the expected reward $r(5)$.

Table 2.1: Distribution of number of cans at State 5.

Number of cans	Reward, r	Probability, $\rho(r \mid 5)$
0	0	0.4
1	1	0.2
3	3	0.3
5	5	0.1

Solution: The expected reward that the robot receives when it enters state 5 is (as given in Equation (2.19)):

$$\begin{aligned} r(5) &= \mathbb{E}\big[\, R_t \big|\, S_t = 5 \,\big] \\ &= \sum_r r\, \rho(r \mid 5) \\ &= 0 \cdot 0.4 + 1 \cdot 0.2 + 3 \cdot 0.3 + 5 \cdot 0.1 \\ &= 1.6 \end{aligned} \tag{2.29}$$

□

Example 2.2.4. Consider the cleaning robot problem in Example 2.2.3, with the reward probabilities for states 3, 4, and 5 as shown in Table 2.2. Suppose that the transition function for the robot taking action $+1$ at state 4 is now non-deterministic, with the transition probabilities as shown in Figure 2.6. Determine the expected reward $r(4, +1)$.

Table 2.2: Distribution of number of cans at state 3, 4, and 5.

State, s	Number of cans	Reward, r	Probability, $\rho(r \mid s)$
3	0	0	1
4	0	0	1
5	0	0	0.4
	1	1	0.2
	3	3	0.3
	5	5	0.1

Solution: From Figure 2.6, we have the transition probabilities of the agent at state 4 as shown in Table 2.3.

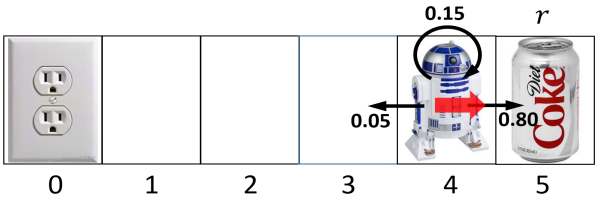


Figure 2.6: Cleaning robot with non-deterministic transitions.

Table 2.3: Transition probabilities.

s	s'	$f(s,+1,s')$
4	5	0.80
4	4	0.15
4	3	0.05

The expected reward for the state-action pair $(4,+1)$ is

$$
\begin{aligned}
r(s,a) &= r(4,+1) \\
&= \mathbb{E}\big[\, R_t \,|\, S_{t-1}=4,\ A_{t-1}=+1 \,\big] \\
&= \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} r \cdot p(s', r \,|\, s, a) \qquad \big(\text{Note: } r\in\{0,1,3,5\};\ s'\in\{3,4,5\}\big) \\
&= \sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} r \cdot f(s,a,s') \cdot \rho(r \,|\, s') \\
&= \sum_{r\in\mathcal{R}} r\, f(4,+1,5)\, \rho(r \,|\, 5) + r\, f(4,+1,4)\, \rho(r \,|\, 4) + r\, f(4,+1,3)\, \rho(r \,|\, 3) \\
&= 0\cdot f(4,+1,5)\, \rho(0 \,|\, 5) + 0\cdot f(4,+1,4)\, \rho(0 \,|\, 4) + 0\cdot f(4,+1,3)\, \rho(0 \,|\, 3) + \\
&\quad 1\cdot f(4,+1,5)\, \rho(1 \,|\, 5) + 1\cdot f(4,+1,4)\, \rho(1 \,|\, 4) + 1\cdot f(4,+1,3)\, \rho(1 \,|\, 3) + \\
&\quad 3\cdot f(4,+1,5)\, \rho(3 \,|\, 5) + 3\cdot f(4,+1,4)\, \rho(3 \,|\, 4) + 3\cdot f(4,+1,3)\, \rho(3 \,|\, 3) + \\
&\quad 5\cdot f(4,+1,5)\, \rho(5 \,|\, 5) + 5\cdot f(4,+1,4)\, \rho(5 \,|\, 4) + 5\cdot f(4,+1,3)\, \rho(5 \,|\, 3) \\
&= 1\times 0.8\times 0.2 + 3\times 0.8\times 0.3 + 5\times 0.8\times 0.1 \qquad \textit{(All other terms are } 0\textit{)} \\
&= 1.28 \qquad\qquad (2.30)
\end{aligned}
$$

□

In this module, we will focus on the class of Markov Decision Processes that are finite in the size of state, action and reward, i.e., $|\mathcal{S}|, |\mathcal{A}|, |\mathcal{R}| < \infty$.

finite Markov Decision Process

2.2.3 The Markov property

A state S_t is said to be *Markov* if and only if

$$\mathbb{P}(S_{t+1}|S_t) = \mathbb{P}(S_{t+1}|S_t, S_{t-1}, \ldots, S_1) \tag{2.31}$$

That is, the next state is completely determined by only the current state.

2.2.4 Return

We have discussed the reward that the agent will receive when it interacts with the environment by taking actions. Each reward r is received after one transition. We next consider the scenario where the agent starts from an initial state and reaches some state s after moving t steps. We now ask the question: What is the total reward the agent can obtain *from this point onward* (i.e., from state s) if the agent continues to make transitions and collect a reward after each transition? We call this total reward the *return* (denoted by G_t) with respect to the time step t, and is defined as:

return

$$G_t \triangleq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.32}$$

where k is the index of the time steps taken after time step t, and γ (with $0 \leq \gamma \leq 1$) is called the *discount rate*. Figure 2.7 illustrates this definition of G_t.

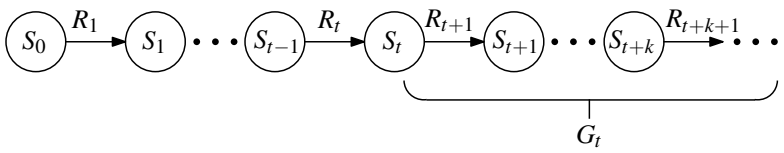


Figure 2.7: Return.

The term G_t thus determines the (discounted) present value of future rewards. A reward received k steps in the future is discounted by a factor of γ^{k-1}. A small value for the discount rate γ emphasizes the preference that the agent is to focus more on the immediate rewards received from the next few steps and to heavily discount the rewards received from subsequent future steps. Conversely, a large γ forces the agent to take into account future rewards more strongly, i.e., the agent becomes more far-sighted.

The upper limit of infinity for the summation operator in Equation (2.32) corresponds to the case where the agent interacts with the environment continuously, such as a cleaning robot operating in an office building on a 24/7 basis. These are referred to as *continuing tasks*[2]. In many application settings, there is a natural way that the agent will terminate the interaction at some (goal) state and re-start the interaction at some other state, such as in playing the board game of Go. For such cases, the summation operation for G_t in Equation

[2]For problems involving continuing tasks, the condition of $\gamma < 1$ is required for the existence of a solution [22].

(2.32) will end at some R_T, where T is the time step of termination. A reinforcement learning task that terminates this way is referred to as an *episodic task*. episodic task

Example 2.2.5. Consider the cleaning robot problem (illustrated in Figure 2.8) that we had discussed earlier, with the state set $\mathcal{S} = \{0, 1, 2, 3, 4, 5\}$ and the action set $\mathcal{A} = \{-1, 1\}$. Suppose that the transition function is:

$$\bar{f}(s, a) = \begin{cases} s + a & \text{if } 1 \leq s \leq 4 \\ s & \text{if } s = 0 \text{ or } s = 5 \end{cases} \tag{2.33}$$

Calculate the return (with a discount rate of $\gamma = 0.5$) if the robot starts at state 2 and reaches state 5 without changing its direction of motion.

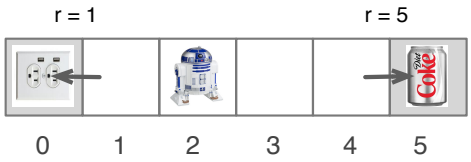


Figure 2.8: Cleaning robot problem.

Solution: We first express the reward function as

$$\bar{\rho}(s, a, s') = \begin{cases} 5 & \text{if } s \neq 5 \text{ and } s' = 5 \\ 1 & \text{if } s \neq 0 \text{ and } s' = 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.34}$$

To reach state 5 from state 2 without any change of direction, the robot will make three transitions, i.e., $\bar{f}(2, 1) = 3$, $\bar{f}(3, 1) = 4$ and $\bar{f}(4, 1) = 5$. The specified return therefore is

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \\ &= \bar{\rho}(2, 1, 3) + 0.5 \cdot \bar{\rho}(3, 1, 4) + 0.5^2 \cdot \bar{\rho}(4, 1, 5) \\ &= 0 + 0 + 0.25 \cdot 5 \\ &= 1.25 \end{aligned} \tag{2.35}$$

□

Remark 2.2.3. *We note that the return at time step t can be expressed in terms of the return in*

the next time step $t+1$, i.e.,

$$\begin{aligned} G_t &\triangleq \sum_{k=0}^{\infty} \gamma^{\,k} R_{t+k+1} \\ &= R_{t+1} + \gamma\, R_{t+2} + \gamma^{\,2} R_{t+3} + \gamma^{\,3} R_{t+4} \cdots \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma\, R_{t+3} + \gamma^{\,2} R_{t+4} \cdots \right) \\ &= R_{t+1} + \gamma \Bigg(\underbrace{\sum_{k=0}^{\infty} \gamma^{\,k+1} R_{t+k+2}}_{G_{t+1}} \Bigg) \\ &= R_{t+1} + \gamma\, G_{t+1} \end{aligned} \tag{2.36}$$

□

2.2.5 Policy

When an agent interacts with the environment, it may do so aimlessly by randomly taking an action at any given state. To control a Markov Decision Process is to impose a decision-making mechanism on the process. This decision-making mechanism is generally referred to as a *policy*, i.e., a policy specifies the action to be taken by the agent at each state. policy

Policies can be deterministic or non-deterministic. A deterministic policy, denoted by $\pi(s)$, specifies a particular action to be taken at a particular state, i.e., $\pi(s)$

$$\pi : \mathcal{S} \to \mathcal{A} \tag{2.37}$$

So under a policy π, the agent at state s is to execute the action a give by $\pi(s)$, i.e.,

$$\pi(s) = a \tag{2.38}$$

A non-deterministic policy specifies the probability of an action a being taken at state s at time t, i.e., $\pi(a \,|\, s)$

$$\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]; \tag{2.39}$$

So the probability of the agent taking action a at state s is given by the policy $\pi(a \,|\, s)$, i.e.,

$$\pi(a \,|\, s) = \mathbb{P}\{A_t = a \,|\, S_t = s\} \tag{2.40}$$

By convention, we use π to refer to a policy in the general context.

Example 2.2.6. In Example 2.2.5, suppose that the behavior of the robot is restricted to the following specifications:

1. In state 1, the robot must take the action $a = -1$.
2. In state 0 or 5, the robot can take either action.
3. In any of the remaining states, the robot must take the action $a = +1$.

Then the policy that governs the behavior of the robot can be expressed as:

$$\begin{cases} \pi(1) = -1 \\ \pi(s) = \pm 1 \quad \text{if} \quad s = 0, 5 \\ \pi(s) = 1 \quad\quad \text{if} \quad s = 2, 3, 4 \end{cases}$$

□

2.3 Value Functions and the Bellman Equation

problem of reinforcement learning

As we discussed earlier in Section 1.3 of Chapter 1, the objective of the agent is to perform its task while maximizing the return, and solving a reinforcement learning problem means finding a policy that enables the agent to achieve its objective. Such a policy is called an *optimal policy*.

An intuitive way to describe an optimal policy is that it is 'better' than any other polices that we can come up with. This requires some method for comparing any two polices and decide which one is 'better'. The concept of value functions enables us to do that.

2.3.1 State and action value functions

There two types of value functions. One is called the *state value function*; the other is called the *action value function*[3].

Consider the treasure-hunting problem discussed in Chapter 1, whose states are as shown in Figure 1.4. Suppose that the robot is following a policy π that results in the state transition sequence $0 \to \cdots \to 7 \to 11 \to 12$. If we just want to compare states 7 and 11, we can see intuitively that state 11 has a higher 'worth' than state 7, because the former is both nearer to the treasure box and further away from the deathtrap than the latter. It is in this context that the value function is defined.

state value functrion $v_\pi(s)$

The state value function, denoted by $v_\pi(s)$, represents the 'worth' of a state s under some policy π. For the case just discussed above, we have $v_\pi(11) > v_\pi(7)$. Specifically, v_π is the expected return the agent will obtain when it starts from state s and follow the

[3]Also called *state-action value function*. We use the shorter form throughout these notes.

policy π thereafter, i.e.,

$$\begin{aligned} v_\pi(s) &\triangleq \mathbb{E}_\pi\big[G_t \,|\, S_t = s\big] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^{\,k} R_{t+k+1} \,\middle|\, S_t = s\right], \quad \text{for all } s \in \mathcal{S} \end{aligned} \tag{2.41}$$

Example 2.3.1. Determine the value of $v_\pi(2)$ for the policy as given in Example 2.2.5, with the assumption that the robot stops after reaching state 5.

Solution: The given policy is $\pi(2) = \pi(3) = \pi(4) = +1$, which is deterministic. Since all transitions in this example are deterministic, the state value function v_π degenerates to just G_t itself. Hence, there is no need to consider the expectation operator $\mathbb{E}$, and so the value for $v_\pi(2)$ is the same as the value for G_t calculated earlier in Example 2.2.5, i.e.,

$$\begin{aligned} v_\pi(2) &= \mathbb{E}[G_t|\, s_t = 2] = G_t|_{s_t=2} \\ &= R_{t+1} + \gamma R_{t+2} + \gamma^{\,2} R_{t+3} \\ &= \bar{\rho}(2,1,3) + 0.5 \cdot \bar{\rho}(3,1,4) + 0.5^2 \cdot \bar{\rho}(4,1,5) \\ &= 0 + 0 + 0.25 \cdot 5 \\ &= 1.25 \end{aligned} \tag{2.42}$$

□

Remark 2.3.1. *Note that the value of a state is policy-dependent. Consider the cleaning robot example as described in Example 2.2.5 and two policies* $\pi_1(s) = 1$ *and* $\pi_2(s) = -1$*, with* $s = 1, 2, 3, 4$*. The values of state* 4 *under these two policies are:* $v_{\pi_1}(4) = R_{t+1} = \bar{\rho}(4, +1, 5) = 5$*, and* $v_{\pi_2}(4) = R_{t+1} + \gamma\, R_{t+2} + \gamma^2\, R_{t+3} + \gamma^3\, R_{t+4} = \bar{\rho}(4, -1, 3) + \gamma\, \bar{\rho}(3, -1, 2) + \gamma^2\, \bar{\rho}(2, -1, 1) + \gamma^3\, \bar{\rho}(1, -1, 0) = (0.5)^3 \times 1 = 0.125$. □

When the transitions are not deterministic, calculating $v_\pi(s)$ is more complicated. One way to determining $v_\pi(s)$ is by solving the so-called Bellman Equation, which we will discuss in the next section.

The state value function v_π thus defined enables us to compare two policies in terms of which one produces a larger return. We say that policy π_i is better than policy π_j, written[4] as $\pi_i > \pi_j$, if and only if $v_{\pi_i}(s) > v_{\pi_j}(s)$ for all $s \in \mathcal{S}$.

The second type of value function is the *action value function*. It represents the 'worth' of a state-action pair (s, a) under some policy π. This 'worth' is represented by a value given by the function $q_\pi(s, a)$, with $q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, which is defined as the expected return the agent will obtain from taking action a at state s at time step t, and thereafter

action value function $q_\pi(s, a)$

[4]It would be incorrect to write $\pi_i(s) > \pi_j(s)$ because we are then comparing two actions.

following policy π, i.e.,

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi\big[G_t \big| S_t = s, A_t = a\big] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^{\,k} R_{t+k+1} \;\Big|\; S_t = s, A_t = a\right] \end{aligned} \tag{2.43}$$

Remark 2.3.2. For two deterministic policies π_i and π_j, we can use the action value function $q_\pi(s,a)$ to compare them, i.e., $\pi_i > \pi_j$ if $q_{\pi_i}(s,a) > q_{\pi_j}(s,a)$ for all (s,a) pairs, since $v_\pi(s) = q_\pi(s,\pi(s))$ if π is deterministic. □

Usually for a given reinforcement learning problem formulated as a finite Markov Decision Process, the number of possible policies n is finite. So conceptually we could just compare the v_π values of *all* possible policies, i.e., v_{π_1}, v_{π_2}, $\cdots$, v_{π_n}, to find an optimal policy. This requires us to first determine v_π for any given policy π. The Bellman Equation provides one way of doing this.

2.3.2 The Bellman Equation

In general, the Bellman Equation describes a relationship between the value of a state $v_\pi(s)$ (or of a state-action pair $q_\pi(s,a)$) and the values of its successor states $v_\pi(s')$ (or state-action pairs $q_\pi(s',a')$). We first consider the case of state value, under the general assumption that the transition, reward and policy are all non-deterministic,

We start with the definition of the state value function under policy π given in Equation (2.41). Now

$$\begin{aligned} v_\pi(s) &\triangleq \mathbb{E}_\pi\Big[G_t \,\Big|\, S_t = s\Big] \\ &= \mathbb{E}_\pi\Big[R_{t+1} + \gamma R_{t+2} + \gamma^{\,2} R_{t+3} + \gamma^{\,3} R_{t+4} + \cdots \,\Big|\, S_t = s\Big] \quad \text{(from definition of } G_t\text{)} \\ &= \mathbb{E}_\pi\Big[R_{t+1} \,\Big|\, S_t = s\Big] + \mathbb{E}_\pi\Big[\gamma\, R_{t+2} + \gamma^{\,2} R_{t+3} + \gamma^{\,3} R_{t+4} + \cdots \,\Big|\, S_t = s\Big] \\ &= \mathbb{E}_\pi\Big[R_{t+1} \,\Big|\, S_t = s\Big] + \gamma\, \mathbb{E}_\pi\Big[\underbrace{\big(R_{t+2} + \gamma\, R_{t+3} + \gamma^{\,2} R_{t+4} + \cdots\big)}_{G_{t+1}\text{ (with } S_{t+1} = s'\text{)}} \,|\, S_t = s\Big] \quad \text{(factoring out } \gamma\text{)} \\ &= \mathbb{E}_\pi\Big[R_{t+1} \,\Big|\, S_t = s\Big] + \gamma\, \mathbb{E}_\pi\Big[G_{t+1} \,\Big|\, S_t = s\Big] \end{aligned} \tag{2.44}$$

The portion between states s and s' in Figure 2.9 illustrates the terms involved in Equation (2.44). The first term in Equation (2.44) is the expected reward received by the agent after taking an action at state s. Since the transition, reward and policy are all non-deterministic, we have

$$\mathbb{E}_\pi\Big[R_{t+1} \,\Big|\, S_t = s\Big] = \sum_a \pi(a \,|\, s) \sum_{s'} \sum_r r \cdot p(s', r \,|\, s, a) \tag{2.45}$$

This case is similar to that discussed in Example 2.2.4, except that in that example the action is already specified (i.e., $a = +1$) but here the action is given by a non-deterministic policy $\pi(a \,|\, s)$, which explains the summation over all a in Equation (2.45).

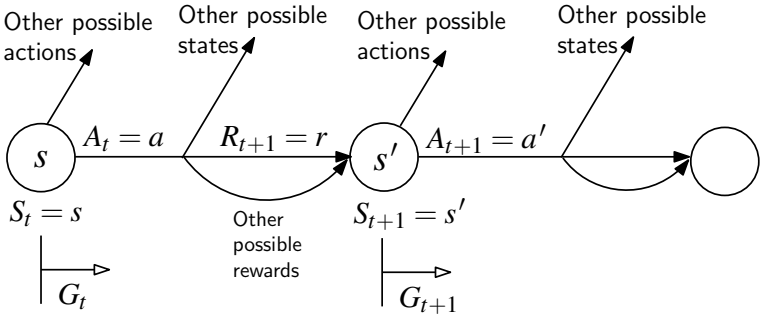


Figure 2.9: The general setting of a Markov Decision Process.

We next examine the term $\mathbb{E}_\pi[G_{t+1} \,|\, S_t = s]$ in Equation (2.44). We note that G_{t+1} is the return obtained by the agent from step $t + 1$ onward, i.e., when the agent is at state $S_{t+1} = s'$, as is illustrated in Figure 2.9. However, in the term $\mathbb{E}_\pi[G_{t+1} \,|\, S_t = s]$, G_{t+1} is with reference to step t, i.e., when the agent is still at state $S_t = s$. So we proceed to express $\mathbb{E}_\pi\Big[G_{t+1} \,|\, S_t = s\Big]$ in terms of $\mathbb{E}_\pi\Big[G_{t+1} \,|\, S_{t+1} = s'\Big]$.

Since there are multiple possible actions a to be taken at s and multiple possible next states s' (as is illustrated in Figure 2.9), we can write out the probabilities with respect to each such a and s', then combine the resulting terms to obtain a simplified expression.

Now,

$$
\begin{aligned}
& \mathbb{E}_\pi\Big[G_{t+1}\Big|\, S_t = s\Big] \\
= \; & \underbrace{\pi(a_1|s)\left(\sum_r p(s_1', r|s, a_1)\right)}_{②} \times \underbrace{\mathbb{E}_\pi\Big[G_{t+1}|S_{t+1} = s_1'\Big]}_{①} + \\
& \pi(a_1|s)\left(\sum_r p(s_2', r|s, a_1)\right) \times \mathbb{E}_\pi\Big[G_{t+1}|S_{t+1} = s_2'\Big] + \\
& \pi(a_1|s)\left(\sum_r p(s_3', r|s, a_1)\right) \times \mathbb{E}_\pi\Big[G_{t+1}|S_{t+1} = s_3'\Big] + \\
& \cdots + \\
& \pi(a_2|s)\left(\sum_r p(s_1', r|s, a_2)\right) \times \mathbb{E}_\pi\Big[G_{t+1}|S_{t+1} = s_1'\Big] + \\
& \cdots \; \cdots \\
= \; & \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \,|\, s, a) \underbrace{\mathbb{E}_\pi\left[G_{t+1}|S_{t+1} = s'\right]}_{v_\pi(s')} \\
= \; & \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \,|\, s, a)\, v_\pi(s') \qquad (2.46)
\end{aligned}
$$

where the term indicated by ① is the expectation of the return obtained by the agent as it follows the policy π from the state s_1' onward (as is illustrated in Figure 2.9), while the term indicated by ② is the probability[5] of the agent taking a_1 to reach s_1' from s, as is illustrated in the shaded portion in the backup diagram[6] shown in Figure 2.10.

With Equations (2.45) and (2.46), Equation (2.44) can be written as

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\Big[R_{t+1} \,\Big|\, S_t = s\Big] + \gamma\, \mathbb{E}_\pi\Big[G_{t+1} \,\Big|\, S_t = s\Big] \\
&= \sum_a \pi(a \,|\, s) \sum_{s'} \sum_r r \cdot p(s', r \,|\, s, a) + \gamma \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \,|\, s, a)\, v_\pi(s') \\
&= \sum_a \pi(a \,|\, s) \sum_{s', r} p(s', r \,|\, s, a)\big[r + \gamma\, v_\pi(s')\big], \qquad \text{for all } s \in \mathcal{S} \qquad (2.47)
\end{aligned}
$$

Equation (2.47) is the *Bellman Equation for* v_π. It specifies that, under a given policy π, the value of a state $S_t = s$ must equal to the expected reward of transitioning into a new state $S_{t+1} = s'$ plus the discounted value of the state s'.

Bellman Equation for v_π

[5]Note that the summation term inside the brackets is in fact the probability of the agent reaching s_1' from s, under the condition that a_1 is *already the chosen action*, as shown earlier in Equation (2.24).

[6]A backup diagram is an intuitive visualization of a reinforcement learning process. A state value is represented by a hollow circle while a action value is represented by a solid circle. Action is represented by a arrow starting from a state. Reward is conventionally shown after the action value. Action that result in maximum action value is shown as a arc starting from a state.

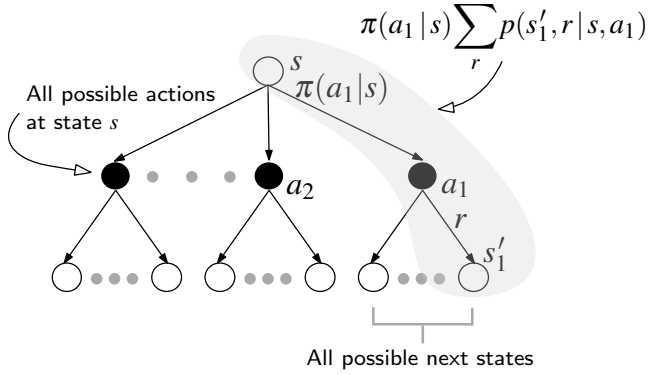


Figure 2.10: Illustration of Bellman Equation using backup diagram.

The Bellman Equation for the action value function $q_\pi(s,a)$ can be derived in a similar way. From the definition of $q_\pi(s,a)$, we have

Bellman Equation for q_π

$$\begin{aligned} q_\pi(s,a) &\triangleq \mathbb{E}_\pi\big[G_t \,|\, S_t, A_t = a)\big] \\ &= \mathbb{E}_\pi\Big[R_{t+1} \,\Big|\, S_t = s, A_t = a\Big] + \gamma\, \mathbb{E}_\pi\Big[G_{t+1} \,\Big|\, S_t = s, A_t = a\Big] \end{aligned} \tag{2.48}$$

Now

$$\mathbb{E}_\pi\Big[R_{t+1} \,\Big|\, S_t = s, A_t = a\Big] = \sum_{s'}\sum_r r \cdot p(s', r \,|\, s, a) \tag{2.49}$$

and

$$\begin{aligned} \mathbb{E}_\pi\Big[G_{t+1} \,\Big|\, S_t = s, A_t = a\Big] &= \sum_r p(s'_1, r|s, a) \times \mathbb{E}_\pi\Big[G_{t+1}|S_{t+1} = s'_1, A_{t+1} = a'_1\Big] + \\ &\quad \sum_r p(s'_1, r|s, a) \times \mathbb{E}_\pi\Big[G_{t+1}|S_{t+1} = s'_1, A_{t+1} = a'_2\Big] + \cdots + \\ &\quad \sum_r p(s'_2, r|s, a) \times \mathbb{E}_\pi\Big[G_{t+1}|S_{t+1} = s'_2, A_{t+1} = a'_1\Big] + \cdots + \\ &\quad \cdots \\ &= \sum_{s',a'}\sum_r p(s', r \,|\, s, a)\, \pi(a' \,|\, s')\, \underbrace{\mathbb{E}_\pi\left[G_{t+1}|S_{t+1} = s', A_{t+1} = a'\right]}_{q_\pi(s', a')} \\ &= \sum_{s',a'}\sum_r p(s', r \,|\, s, a)\pi(a' \,|\, s')\, q_\pi(s', a') \end{aligned} \tag{2.50}$$

With Equations (2.49) and (2.50), Equation (2.48) can be written as

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi\Big[R_{t+1} \,\Big|\, S_t = s, A_t = a\Big] + \gamma\, \mathbb{E}_\pi\Big[G_{t+1} \,\Big|\, S_t = s, A_t = a\Big] \\ &= \sum_{s'}\sum_{r} r \cdot p(s', r \mid s, a) + \gamma \sum_{s',a'}\sum_{r} p(s', r \mid s, a)\pi(a' \mid s')\, q_\pi(s', a') \\ &= \sum_{s',r} p(s',\, r \mid s, a) \left[r + \gamma \sum_{a'} \pi(a' \mid s')\, q_\pi(s', a') \right] \end{aligned} \tag{2.51}$$

Equation (2.51) is the *Bellman Equation for* q_π.

For a deterministic policy, we have,

$$\begin{aligned} q_\pi(s,a) &\triangleq \mathbb{E}_\pi\big[G_t \,|\, S_t, A_t = a)\big] \\ &= \sum_{s',r} p(s',\, r \mid s, a)\big[r + \gamma\, q_\pi(s', a')\big] \end{aligned} \tag{2.52}$$

We next establish two relationships between $v_\pi(s)$ and $q_\pi(s,a)$. Recall that

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\Big[R_{t+1} \,\Big|\, S_t = s\Big] + \gamma\, \mathbb{E}_\pi\Big[G_{t+1} \,\Big|\, S_t = s\Big] \\ q_\pi(s,a) &= \mathbb{E}_\pi\Big[R_{t+1} \,\Big|\, S_t = s, A_t = a\Big] + \gamma\, \mathbb{E}_\pi\Big[G_{t+1} \,\Big|\, S_t = s, A_t = a\Big] \end{aligned}$$

Now

$$\begin{aligned} & \sum_a \pi(a \mid s)\, q_\pi(s,a) \\ = & \sum_a \pi(a \mid s) \left(\mathbb{E}_\pi \left[R_{t+1} \,\middle|\, S_t = s, A_t = a \right] + \gamma\, \mathbb{E}_\pi \left[G_{t+1} \,\middle|\, S_t = s, A_t = a \right] \right) \\ = & \sum_a \pi(a \mid s) \left(\mathbb{E}_\pi \left[R_{t+1} \,\middle|\, S_t = s, A_t = a \right] \right) + \sum_a \pi(a \mid s) \left(\gamma\, \mathbb{E}_\pi \left[G_{t+1} \,\middle|\, S_t = s, A_t = a \right] \right) \\ = & \mathbb{E}_\pi \left[R_{t+1} \,\middle|\, S_t = s \right] + \gamma\, \mathbb{E}_\pi \left[G_{t+1} \,\middle|\, S_t = s \right] \\ = & v_\pi(s) \end{aligned} \tag{2.53}$$

that is,

$$v_\pi(s) = \sum_a \pi(a \mid s)\, q_\pi(s,a) \tag{2.54}$$

When the policy π is deterministic[7], i.e., $\pi(a \mid s) = 1$, we have

$$v_\pi(s) = q_\pi(s, \pi(s)) \tag{2.55}$$

[7]Since $v_\pi(s)$ is a weighted sum of $q_\pi(s,a)$ with $\sum_a \pi(a \mid s) = 1$, we have $v_\pi(s) \le \max q_\pi(s,a)$.

From Equations (2.51) and (2.54), we have

$$\begin{aligned} q_\pi(s,a) &= \sum_{s',r} p(s',\, r \,|\, s,a)\Big[r + \gamma \underbrace{\sum_{a'} \pi(a' \,|\, s')\, q_\pi(s',a')}_{v_\pi(s')}\Big] \\ &= \sum_{s',r} p(s',\, r \,|\, s,a)\Big[r + \gamma\, v_\pi(s')\Big] \end{aligned} \tag{2.56}$$

If we know the full dynamics of a Markov Decision Process, i.e., the p-function, then we can obtain the values of $v_\pi(s)$ or $q_\pi(s,a)$ for a given policy π by computing the closed-form solution to the corresponding Bellman Equation, as is demonstrated in Example 2.3.2.

Example 2.3.2. Consider the cleaning task described earlier in Example 2.2.1, with the following deterministic reward function and policy, and the state-transition probabilities, as shown in Figure 2.11. Assuming that the robot stops after reaching state 0 or 5, determine the value of $q_\pi(2,1)$.

$$\bar{\rho}(s,a,s') = \begin{cases} 5 & \text{if } s \neq 5 \text{ and } s' = 5 \\ 1 & \text{if } s \neq 0 \text{ and } s' = 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.57}$$

$$\begin{cases} \pi(1) = -1 & \\ \pi(s) = 1 & \text{if} \quad s = 2,3,4 \\ \pi(s) = \pm 1 & \text{if} \quad s = 0,5 \end{cases} \tag{2.58}$$

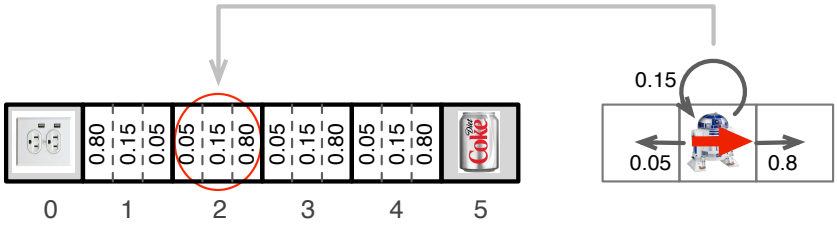


Figure 2.11: Transition probabilities. The three numbers in each square indicate the probabilities of the robot, when taking the action prescribed by the policy, reaching the state to its left, remaining at the same state, and reaching the state to its right, respectively.

Solution: Since both the reward function and the policy are deterministic, we will use the notations $f(s,a,s')$ and $P^a_{ss'}$, instead of the general p-function, to represent the transition probabilities in the calculations.

From Figure 2.11, we have the following transition probabilities:

$$\begin{aligned} f(1,-1,0) &= 0.80 \equiv P_{10}^{-1} \qquad f(2,1,1) = 0.05 \equiv P_{21}^{1} \\ f(1,-1,1) &= 0.15 \equiv P_{11}^{-1} \qquad f(2,1,2) = 0.15 \equiv P_{22}^{1} \\ f(1,-1,2) &= 0.05 \equiv P_{12}^{-1} \qquad f(2,1,3) = 0.80 \equiv P_{23}^{1} \end{aligned}$$

$$\begin{aligned} f(3,1,2) &= 0.05 \equiv P_{32}^{1} \qquad f(4,1,3) = 0.05 \equiv P_{43}^{1} \\ f(3,1,3) &= 0.15 \equiv P_{33}^{1} \qquad f(4,1,4) = 0.15 \equiv P_{44}^{1} \\ f(3,1,4) &= 0.80 \equiv P_{34}^{1} \qquad f(4,1,5) = 0.80 \equiv P_{45}^{1} \end{aligned}$$

We can interpret the assumption that the robot will stop once it reaches state 0 or 5 to mean that the probability of the robot remaining at either of these two states is 1, i.e.,

$$f(0,\pm 1,0) = f(5,\pm 1,5) = 1$$

Also from the reward function as given in Equation (2.57), we have

$$\begin{array}{lll} \bar{\rho}(0,1,0) = 0 & \bar{\rho}(0,-1,0) = 0 & \bar{\rho}(1,1,2) = 0 \\ \bar{\rho}(1,-1,0) = 1 & \bar{\rho}(2,-1,1) = 0 & \bar{\rho}(2,1,2) = 0 \\ \bar{\rho}(3,-1,2) = 0 & \bar{\rho}(3,1,4) = 0 & \bar{\rho}(4,1,5) = 5 \\ \bar{\rho}(4,-1,3) = 0 & \bar{\rho}(5,1,5) = 0 & \bar{\rho}(5,-1,5) = 0 \end{array} \tag{2.59}$$

Applying the Bellman Equation on $q_\pi(2,1)$ yields

$$\begin{aligned} q_\pi(2,1) &= \sum_{s'} P_{ss'}^{a} \Big(\bar{\rho}(s,a,s') + \gamma q_\pi(s',a') \Big) \\ &= \sum_{j=1}^{3} P_{2j}^{1} \Big(\bar{\rho}(2,1,j) + \gamma q_\pi(j,\pi(j)) \Big) \\ &= P_{21}^{1} \Big(\bar{\rho}(2,1,1) + \gamma q_\pi(1,-1) \Big) \ \text{(Note: } j=1 \text{ and } \pi(1)=-1) \\ &+ P_{22}^{1} \Big(\bar{\rho}(2,1,2) + \gamma q_\pi(2,1) \Big) \ \text{(Note: } j=2 \text{ and } \pi(2)=1) \\ &+ P_{23}^{1} \Big(\bar{\rho}(2,1,3) + \gamma q_\pi(3,1) \Big) \ \text{(Note: } j=3 \text{ and } \pi(3)=1) \\ &= 0.05\Big(0 + 0.5 q_\pi(1,-1)\Big) + 0.15\Big(0 + 0.5 q_\pi(2,1)\Big) + 0.80\Big(0 + 0.5 q_\pi(3,1)\Big) \\ &= 0.025 q_\pi(1,-1) + 0.075 q_\pi(2,1) + 0.4 q_\pi(3,1) \end{aligned} \tag{2.60}$$

Similarly, applying the Bellman Equation on $q_\pi(1,-1)$, $q_\pi(3,1)$, and $q_\pi(4,1)$ results in

four equations with four unknowns, i.e.,

$$\begin{aligned} q_\pi(2,1) &= 0.025q_\pi(1,-1) + 0.075q_\pi(2,1) + 0.4q_\pi(3,1) && (2.61)\\ q_\pi(1,-1) &= 0.8 + 0.075q_\pi(1,-1) + 0.025q_\pi(2,1) && (2.62)\\ q_\pi(3,1) &= 0.025q_\pi(2,1) + 0.075q_\pi(3,1) + 0.4q_\pi(4,1) && (2.63)\\ q_\pi(4,1) &= 0.025q_\pi(3,1) + 0.075q_\pi(4,1) + 4 && (2.64) \end{aligned}$$

Solving Equations (2.61)-(2.64) yields,

$$\begin{aligned} q_\pi(1,-1) &= 0.888\\ q_\pi(2,1) &= 0.852\\ q_\pi(3,1) &= 1.915\\ q_\pi(4,1) &= 4.376 && (2.65) \end{aligned}$$

Hence, the answer is: $q_\pi(2,1) = 0.852$. □

2.4 Optimal Polices and Optimal Value Functions

Since the Bellman Equation provides a method to determine the state values $v_\pi(s)$ or action values $q_\pi(s,a)$ of a given policy π, we can use either of these two value functions to establish a partial order[8] over a set of policies; that is we can compare two policies and decide which one is better by comparing their respective $v_\pi(s)$ or $q_\pi(s,a)$ values. For two policies π_i and π_j, we say that $\pi_i \geq \pi_j$ (i.e., π_i is as good as or better than π_j) if and only if $v_{\pi_i}(s) \geq v_{\pi_j}(s)$ for all $s \in \mathcal{S}$. We note that such a comparison is made here under the assumption that all the value functions we are dealing with can be evaluated in some numerical system.

Recall (form our discussion earlier) that the objective of the agent is to perform its task while maximizing the return, and solving a reinforcement learning problem means finding an optimal policy that enables the agent to achieve its objective. For a given finite Markov Decision Process, the number of possible policies is finite. Therefore, conceptually we can calculate the $v_\pi(s)$ or $q_\pi(s,a)$ values of *all* these individual policies to identify the policy (or policies) whose $v_\pi(s)$ or $q_\pi(s,a)$ values are the largest. Such a policy is called an *optimal policy*, denoted by π_*, and can be defined either in terms of the state value optimal policy

[8] A relation $\leq$ is called a *partial order* on a set X if it has: (1) reflexivity: $a \leq a$ for all $a \in X$; (2) antisymmetry: $a \leq b$ and $b \leq a$ implies $a = b$; and (3) transitivity: $a \leq b$ and $b \leq c$ implies $a \leq c$. An intuitive way to interpret a partial order is that it is a ranking of a set of entities in a certain way.

function or the action value function, i.e.,

$$v_*(s) \triangleq \max_{\pi} v_\pi(s) \tag{2.66}$$

$$q_*(s,a) \triangleq \max_{\pi} q_\pi(s,a) \tag{2.67}$$

Since an optimal policy is still a policy, it satisfies Equation (2.56); that is,

$$q_*(s,a) = \sum_{s',r} p(s',r \mid s,a)\big[r + \gamma\, v_*(s')\big] \tag{2.68}$$

Now suppose that we have already calculated the values for $q_*(s,a)$ for all (s,a) pairs. Then we can obtain an optimal policy by simply picking the action at each state s corresponding the largest value of $q_*(s,a)$, i.e.,

$$\pi_*(a \mid s) = 1\,, \qquad \text{where } a \in \arg\max_{\sigma \in \mathcal{A}(s)} q_*(s,\sigma) \tag{2.69}$$

This is what we had done in the treasure-hunting example in Chapter 1, as is illustrated in Figure 2.12 (which is the same as Figure 1.7), where the policy was considered to be deterministic in the original discussion.

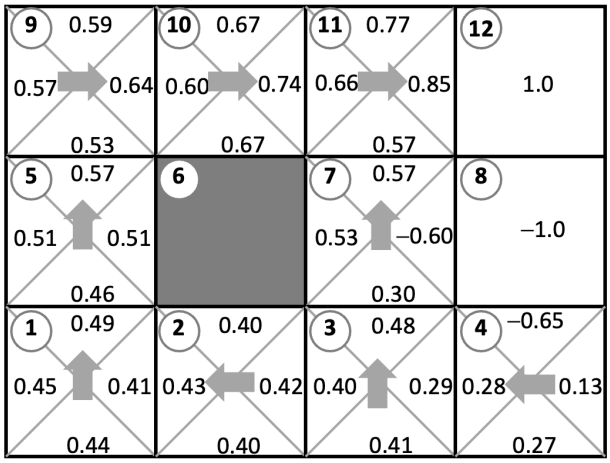


Figure 2.12: Estimated $q_*(s,a)$ and optimal policy $\pi_*(s)$ for the treasure-hunting problem.

Remark 2.4.1. : *If we substitute Equation (2.68) into the expression for a in Equation (2.69), then we have*

$$a \in \arg\max_{\sigma \in \mathcal{A}(s)} q_*(s,\sigma) = \arg\max_{\sigma \in \mathcal{A}(s)} \left(\sum_{s',r} p(s',r \mid s,\sigma)\Big[r + \gamma\, v_*(s')\Big] \right)$$

which indicates that we can extract an optimal policy from the optimal state values $v_*(s)$ only if we know the p-function $p(s',r \mid s,a)$. □

Remark 2.4.2. *It can be shown that, for a given reinforcement learning problem with finite state and action sets, (i) the optimal state values v_* and the optimal action values q_* are unique, and (ii) there may exist multiple optimal policies [22].* □

Note that the term $\arg\max_\sigma q_*(s,\sigma)$ in Equation (2.69) yields a set of all actions at state s corresponding to the maximum value of $q_*(s,a)$. In case that this set has more than one element, it is understood that only one of them is arbitrarily chosen by the agent to form an optimal policy.

We next express Equation (2.54) in terms of π_*, i.e.,

$$v_*(s) = \sum_a \pi_*(a \mid s)\, q_*(s,a) \tag{2.70}$$

If we construct an optimal policy π_* according to Equations (2.69), then Equation (2.70) can be written as

$$v_*(s) = \max_{a\in\mathcal{A}(s)} q_*(s,a) \tag{2.71}$$

where $\mathcal{A}(s)$ denotes the set of actions available for selection by the agent at state s. Equation (2.71) is referred to as a *consistency condition* on the optimal values $v_*(s)$ and $q_*(s,a)$. Substituting Equation (2.68) into Equation (2.71) yields

$$\begin{aligned} v_*(s) &= \max_{a\in\mathcal{A}(s)} q_*(s,a) \\ &= \max_a \sum_{s',r} p(s',r \mid s,a)\big[r + \gamma\, v_*(s')\big] \end{aligned} \tag{2.72}$$

From Equations (2.68) and (2.71), we have

$$\begin{aligned} q_*(s,a) &= \sum_{s',r} p(s',r \mid s,a)\big[r + \gamma\, v_*(s')\big] \\ &= \sum_{s',r} p(s',r \mid s,a)\big[r + \gamma \max_{a'} q_*(s',a')\big] \end{aligned} \tag{2.73}$$

Bellman Optimality Equation

Equations (2.72) and (2.73) are the *Bellman Optimaility Equation* for v_* and q_*, respectively. They play an important role in solving reinforcement learning problems.

Solution approaches for reinforcement learning problems can be classified into two types. The first is for determining an optimal policy under the condition that a model of the system is available; that is, the function $p(s',r \mid s,a)$ is known *a priori*. This type of solution approach, referred to as *model-based*, is discussed next in Chapter 3.

The second type of solution approach concerns the situation where the states of a Markov model are defined but all other information must be "learned" while the agent tries to find an optimal policy. Solution approached of this type (called model-free reinforcement

learning) are discussed in Chapters 4-6.

Chapter 3

Dynamic Programming

If we know the dynamics of a Markov Decision Process as described by the p-function $p(s', r \mid s, a)$, we can apply the Bellman Optimality Equation for v_* on each state s, or for q_* on each state-action pair (s, a), to yield a set of equations. This set of equations can be solved explicitly to produce the optimal state values $v_*(s)$ or the optimal action values $q_*(s, a)$, from which an optimal policy can be extracted.

Example 3.0.1. Consider the cleaning task described earlier in Example 2.3.2. Note that $q_*(0, \pm 1) = q_*(5, \pm 1) = 0$. To set up the Bellman Optimality Equations for v_*, we need to write out the expressions $v_*(s)$ for all $s = 1, 2, 3, 4$. Take $s = 2$ for example, we have

$$\begin{aligned} v_*(2) &= \max_a \sum_{s', r} p(s', r \mid s, a)\big[r + \gamma\, v_*(s')\big] \\ &= \max_a \sum_{s'} p(s', r \mid 2, a)\big[r + \gamma\, v_*(s')\big] \\ &= \max_a \sum_{s'} P^a_{2s'}\big[r + \gamma\, v_*(s')\big] \\ &= \max \Big\{ P^1_{21}\big[r + \gamma\, v_*(1)\big],\, P^1_{22}\big[r + \gamma\, v_*(2)\big],\, P^1_{23}\big[r + \gamma\, v_*(3)\big], \\ &\qquad\qquad P^{-1}_{21}\big[r + \gamma\, v_*(1)\big],\, P^{-1}_{22}\big[r + \gamma\, v_*(2)\big],\, P^{-1}_{23}\big[r + \gamma\, v_*(3)\big] \Big\} \end{aligned} \tag{3.1}$$

Similarly, to set up the Bellman Optimality Equations for q_*, we need to write out the expressions for $q_*(s, a)$ for all (s, a) pairs with $s = 1, 2, 3, 4$ and $a = \pm 1$. Take $s = 2$ and

$a = 1$ for example, we have

$$\begin{aligned} q_*(2,1) &= \sum_{s',r} p(s',r \mid s,a)\big[r + \gamma \max_{a'} q_*(s',a')\big] \\ &= \sum_{s'} p(s',r \mid 2,1)\big[r + \gamma \max_{a'} q_*(s',a')\big] \\ &= \sum_{s'} P^1_{2s'}\big[r + \gamma \max_{a'} q_*(s',a')\big] \\ &= P^1_{21}\big[r + \gamma \max\{q_*(1,1), q_*(1,-1)\}\big] \\ &+ P^1_{22}\big[r + \gamma \max\{q_*(2,1),\ q_*(2,-1)\}\big] \\ &+ P^1_{23}\big[r + \gamma \max\{q_*(3,1),\ q_*(3,-1)\}\big] \end{aligned}$$

For this simple task, we will have four equations for v_* and eight equations for q_*. □

In practice, explicitly solving the Bellman Optimality Equations for a given reinforcement learning problem to obtain an optimal policy is usually not possible for two reasons. First, these equations are nonlinear due to the max operation. Second, the large numbers of states and actions in practically meaningful applications often make computing the explicit solution for the Bellman Optimality Equations intractable, e.g., as in the case of the game of Go for which the estimated number of state-action pairs is in the order of 10^{170}. This necessitates the use of approximate methods for solving the Bellman Optimality Equation.

Approximate methods are especially desirable for solving reinforcement learning problems (that are formulated as Markov Decision Processes) for two main reasons. First, such methods usually require less computation and so the solution process becomes tractable. Second, in many reinforcement learning problems, many states are not important to the solution process. Approximate methods can leverage on this fact, whereby the agent may ignore some states but still able to come up with a "good enough" solution quickly. This is particularly desirable for applications that involve online learning.

In this chapter, we discuss two approximate methods for solving the Bellman Optimality Equation, namely, *Policy Iteration* and *Value Iteration*. These methods are also known as Dynamic Programming methods for solving reinforcement learning problems.

3.1 Policy Iteration

Policy iteration consists of two sub-processes that interact iteratively in order to yield an optimal policy. The first sub-process is called *policy evaluation*; the second *policy improvement*. The objective of policy evaluation is to determine the state values $v_\pi(s)$

policy iteration

for a given policy π, while the objective of policy improvement is to find a new policy $\pi'(s)$ such that $\pi' \geq \pi$.

Policy iteration involves the following steps:

1. Arbitrarily choose an initial policy π_0. Let $\pi_k = \pi_0$.
2. Perform policy evaluation with respect to π_k to obtain the state values v_{π_k}.
3. Based on v_{π_k}, perform policy improvement on π_k to produce a new policy π_{k+1} that is at least as good as π_k, i.e., $\pi_{k+1} \geq \pi_k$.
4. If π_{k+1} is as good as π_k, i.e., $\pi_{k+1} = \pi_k$, then stop — we now have an optimal policy $\pi_* = \pi_k$; otherwise, assign π_{k+1} to be π_k and go to Step 2.

Figure 3.1 illustrates this iterative process.

$$\pi_0 \xrightarrow{\mathsf{PE}} v_{\pi_0} \xrightarrow{\mathsf{PI}} \pi_1 \xrightarrow{\mathsf{PE}} v_{\pi_1} \xrightarrow{\mathsf{PI}} \pi_2 \xrightarrow{\mathsf{PE}} \cdots \xrightarrow{\mathsf{PI}} \pi_* \xrightarrow{\mathsf{PE}} v_*$$

Figure 3.1: The policy iteration process: PE = policy evaluation; PI = policy improvement.

3.1.1 Policy evaluation

Policy evaluation is the process of determining the state values $v_\pi(s)$ for a given policy π. It is also often referred to as the *prediction problem*. This process itself is also iterative.

The key idea in policy evaluation is to turn the Bellman Equation for v_π, i.e., Equation (2.47), into an update rule in an iterative process that eventually produces the values of $v_\pi(s)$ for a given π, i.e.,

$$v_{k+1}(s) = \sum_a \pi(a \,|\, s) \sum_{s', r} p(s', r \,|\, s, a) \big[r + \gamma \, v_k(s') \big], \qquad \text{for all } s \in \mathcal{S} \tag{3.2}$$

Table 3.1 shows the procedure for implementing an algorithm for policy evaluation. We use $V(s)$ and $Q(s)$ to represent an estimate of $v(s)$ and $q(s)$, respectively, as discussed earlier in Remark 1.4.1.

Example 3.1.1. Consider an environment consisting of the 4-by-4 grid world[1] as shown in Figure 3.2, where the shaded squares indicate terminal states (i.e., 0 and 15). The expected reward is $r(s, a, s') = -1$ for all states s, s' and action a; that is, every time the agent takes an action it receives a reward of -1. At each state, the agent can take any of the four actions: {up, right, down, left}, and transition into the next state in the direction of the

[1]This is Example 4.1 in [22].

Table 3.1: Algorithm for policy evaluation.

Input:	∘ π (the policy to be evaluated) ∘ a small threshold θ ∘ $V_0(s)$, initial values for $V(s)$ for all $s \in \mathcal{S}$; all initial values are arbitrary except $V(s) = 0$ for all $s \in \hat{\mathcal{S}}$, with $\hat{\mathcal{S}}$ being set of terminal states
Repeat	
	$\Delta \leftarrow 0$
	Loop through all $s \in \mathcal{S}$:
	$v \leftarrow V_(s)$
	$V(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma V(s')\Big]$ $\Leftarrow$ This is Equation (3.2)
	$\Delta \leftarrow \max\big(\Delta, \lvert v - V(s)\rvert\big)$
until	$\Delta < \theta$
Output:	$V(s)$, which is the estimated value of $v_\pi(s)$

action, except for the cases where the action takes the agent off the gird; for those cases the agent remains in the same state before the action was taken, e.g, $p(6, -1 \mid 5, \mathsf{right}) = 1$ and $p(4, -1 \mid 4, \mathsf{left}) = 1$. Let $\gamma = 1$. Assume that the agent follows a uniform random policy, i.e., $\pi(a|s) = 0.25$ for all s.

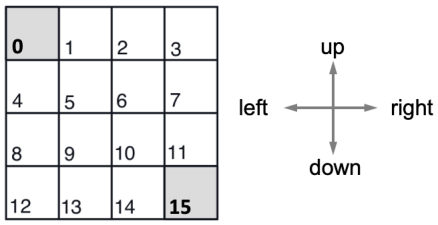


Figure 3.2: A simple grid world example.

Figure 3.3 shows some sample state values during the iterative process, where k is the index for the number of passes through the state set. As illustrative examples, we show the calculations for the state value of $s = 5$ for the first three iterations, i.e., $v_k(5)$ with $k = 1, 2, 3$.

$$
\begin{aligned}
V_1(5) &= \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V_k(s')\big] \\
&= \pi(\mathsf{up} \mid 5) \cdot p(1, -1 \mid 5, \mathsf{up}) \cdot [-1 + \gamma V_0(1)] \\
&+ \pi(\mathsf{right} \mid 5) \cdot p(6, -1 \mid 5, \mathsf{right}) \cdot [-1 + \gamma V_0(6)] \\
&+ \pi(\mathsf{left} \mid 5) \cdot p(4, -1 \mid 5, \mathsf{left}) \cdot [-1 + \gamma V_0(4)] \\
&+ \pi(\mathsf{down} \mid 5) \cdot p(9, -1 \mid 5, \mathsf{down}) \cdot [-1 + \gamma V_0(9)] \\
&= 0.25 \cdot 1 \cdot [-1 + 1 \cdot 0] + 0.25 \cdot 1 \cdot [-1 + 1 \cdot 0] + 0.25 \cdot 1 \cdot [-1 + 1 \cdot 0] + 0.25 \cdot 1 \cdot [-1 + 1 \cdot 0] \\
&= -1.0
\end{aligned} \tag{3.3}
$$

$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Figure 3.3: Sample state values during policy evaluation. The values shown are those calculated at the end of the indicated iteration.

$$
\begin{aligned}
V_2(5) &= 0.25 \cdot 1 \cdot [-1 + 1 \cdot V_1(1)] \\
&+ 0.25 \cdot 1 \cdot [-1 + 1 \cdot V_1(6)] \\
&+ 0.25 \cdot 1 \cdot [-1 + 1 \cdot V_1(4)] \\
&+ 0.25 \cdot 1 \cdot [-1 + 1 \cdot V_1(9)] \\
&= 0.25 \cdot 1 \cdot [-1 + 1 \cdot (-1)] \times 4 \\
&= -2.0 \qquad (3.4) \\
V_3(5) &= 0.25 \cdot 1 \cdot [-1 + 1 \cdot V_2(1)] \\
&+ 0.25 \cdot 1 \cdot [-1 + 1 \cdot V_2(6)] \\
&+ 0.25 \cdot 1 \cdot [-1 + 1 \cdot V_2(4)] \\
&+ 0.25 \cdot 1 \cdot [-1 + 1 \cdot V_2(9)] \\
&= 0.25 \cdot 1 \cdot [-1 + 1 \cdot (-1.7)] \times 2 + 0.25 \cdot 1 \cdot [-1 + 1 \cdot (-2.0)] \times 2 \\
&= -2.85 \approx -2.9 \qquad (3.5)
\end{aligned}
$$

□

In the algorithm described in Table 3.1, the stopping condition is that the difference in the value of v_π between two successive iterations is less than the threshold θ, i.e., $\Delta < \theta$. How do we know that this condition will be reached as k increases? To answer this question is to prove that the policy evaluation process will converge as k increases.

Rigorous mathematical proof for such convergence can be found in many well known textbooks about Dynamic Programming. The key steps are:

1. Express the Bellman Equation for v_π in a form involving an operator called the *Bellman policy operator* (denoted by $\mathbb{T}_\pi$), i.e.,

$$\mathbb{T}_\pi V_k = \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \big[r + \gamma V_k \big] \tag{3.6}$$

   where V_k is a vector of the state values.

2. Show that $\mathbb{T}_\pi$ is a contraction mapping, i.e.,

$$\big\| \mathbb{T}_\pi V_k - \mathbb{T}_\pi V_{k+1} \big\|_\infty \leq \gamma \big\| V_k - V_{k+1} \big\|_\infty \tag{3.7}$$

   where $\|(\cdot)\|_\infty$ denote the L^∞-norm, i.e., Equation (2.5).

3. By invoking the *Contraction Mapping Theorem* (also known as the *Fixed Point Theorem* [9]), it can be concluded that $\mathbb{T}_\pi$ has a unique fixed point which is v_π; that is, the policy evaluation algorithm as shown in Table 3.1 converges to v_π.

We will not reproduce a formal proof here. For the purpose of this module, it suffice to gain an intuitive appreciation of the logical arguments behind the mathematical proof. The line of reasoning goes as follows. For the sake of simplicity, we assume that (i) the reward values are all positive, and (ii) $V_0(s) = 0$ for all states. We know that for a given policy π, the state values v_π are finite — although we do not yet know what they are before running the policy evaluation algorithm. So at the start of the policy evaluation process (i.e., $k = 0$), the difference between $v_\pi(s)$ and $V_k(s) = V_0(s) = 0$ is finite. Now if we can show that at the completion of each iteration k the difference between $v_\pi(s)$ and $V_k(s)$ becomes smaller, then as k increases $V_k(s)$ will approach $v_\pi(s)$ and the policy evaluation algorithm (i.e., Table 3.1) will converge. Figure 3.4 illustrates this situation.

We next show that the update rule, i.e., Equation (3.2), makes the difference between $v_\pi(s)$ and $V_k(s)$ smaller as k increases. Let δ_k denote the maximum of the the difference between $V_k(s)$ and $v_\pi(s)$ at the end of iteration k, i.e.,

$$\delta_k \triangleq \max_s |v_\pi(s) - V_k(s)| \tag{3.8}$$

Removing the maximum operator yields

$$|v_\pi(s) - V_k(s)| \leq \delta_k \tag{3.9}$$

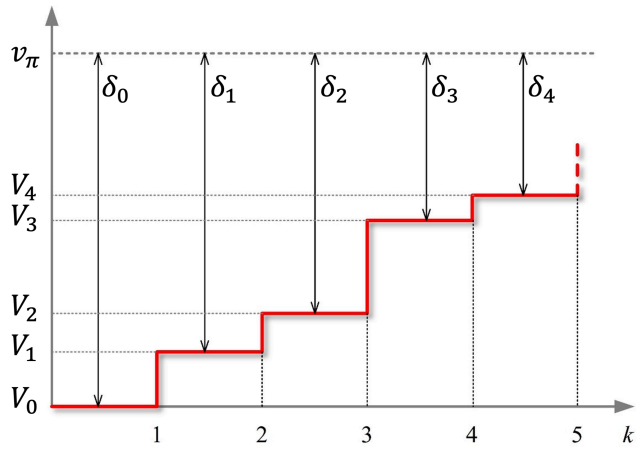


Figure 3.4: Convergence of policy evaluation algorithm.

From this inequality, we have

$$\begin{aligned} & -\delta_k \le v_\pi(s) - V_k(s) \le \delta_k \\ \Rightarrow \quad & -v_\pi(s) - \delta_k \le -V_k(s) \le -v_\pi(s) + \delta_k \\ \Rightarrow \quad & v_\pi(s) + \delta_k \ge V_k(s) \ge v_\pi(s) - \delta_k \\ \Rightarrow \quad & v_\pi(s) - \delta_k \le V_k(s) \le v_\pi(s) + \delta_k \end{aligned} \tag{3.10}$$

Since Equation (3.10) holds for any state at iteration k, for a state s' we have[2]

$$V_k(s') \ge v_\pi(s') - \delta_k \tag{3.11}$$

Now consider iteration $(k+1)$. The update rule can be expressed as

$$\begin{aligned} V_{k+1}(s) &= \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V_k(s') \big] \\ &\ge \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma \, (v_\pi(s') - \delta_k) \big] \\ &= \underbrace{\sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma v_\pi(s') \big]}_{\text{This is } v_\pi(s)} - \gamma \delta_k \\ &= v_\pi(s) - \gamma \delta_k \end{aligned} \tag{3.12}$$

that is,

$$v_\pi(s) - \gamma \delta_k \le V_{k+1}(s) \tag{3.13}$$

[2]Here we are referring to the value of state s' at iteration k. If we think of iteration k as taking place at time step t, then we are referring to the value of $V_k(s')$ at time step t here, not at time step $t+1$ (with which the notation s' is often associated).

or

$$v_\pi(s) - V_{k+1}(s) \leq \gamma\, \delta_k \tag{3.14}$$

Therefore,

$$\delta_{k+1} \triangleq \max_s |v_\pi(s) - V_{k+1}(s)| \leq \max_s |\gamma\, \delta_k| = \gamma\, \delta_k \tag{3.15}$$

i.e.,

$$\delta_{k+1} \leq \gamma\, \delta_k \tag{3.16}$$

If we set $0 \leq \gamma < 1$, then $\delta_k \to 0$, and consequently $V_k(s) \to v_\pi(s)$, as $k \to \infty$. In practice, the iteration can be stopped when δ_k drops below a prescribed tolerance.

After we have obtained $v_\pi(s)$ for a given π, the next step in the policy iteration process is to modify π to produce an improved policy π', i.e., $\pi' \geq \pi$. This is done in the sub-process of policy improvement.

3.1.2 Policy improvement

Suppose that we currently have a deterministic policy π and we have evaluated its state values $v_\pi(s)$ for all $s \in \mathcal{A}(s)$. We now ask the question: How do we know whether this current policy is optimal, and if it is not optimal, how do we improve on it to generate a better (or optimal) one?

The policy improvement sub-process (of policy iteration) deals with this two-part question. The answer to the first part can be found in the answer to the second part. So we now address the second part.

Policy Improvement Theorem

We will do this in two steps. In the first step, we will show that *if we choose the new policy π' such that it satisfies the condition*

$$q_\pi\left(s, \pi'(s)\right) \geq v_\pi(s) \tag{3.17}$$

then we have $v_{\pi'}(s) \geq v_\pi(s)$; that is, the new policy π' is as good as or better than the current policy π, i.e., $\pi' \geq \pi$. This is known as the Policy Improvement Theorem. In the second step we will present a way to choose the new policy π' such that it satisfies Equation (3.17). To expedite the discussion, we will consider only deterministic polices.

We start with the first step to show that Equation (3.17) implies $v_{\pi'}(s) \geq v_\pi(s)$. We will be using Equation (2.56), so we show it again below for ease of reference:

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma\, v_\pi(S_{t+1}) \,\middle|\, S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma\, v_\pi(s')\right] \end{aligned} \tag{3.18}$$

Now from Equation (3.17), we have

$$\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\
&= \mathbb{E}\Big[R_{t+1} + \gamma\, v_\pi(S_{t+1}) \,\Big|\, S_t = s, A_t = \pi'(s)\Big] \quad \text{(Taking } a = \pi'(s) \text{ then follow } \pi) \\
&= \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma\, v_\pi(S_{t+1}) \,\Big|\, S_t = s\Big] \qquad \leftarrow (\dagger) \\
&\leq \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma \underbrace{q_\pi\left(S_{t+1}, \pi'(S_{t+1})\right)}_{\text{Eqn. (3.17) w.r.t. } S_{t+1}} \,\Big|\, S_t = s\Big] \\
&= \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma \underbrace{\mathbb{E}_{\pi'}\left[R_{t+2} + \gamma\, v_\pi(S_{t+2}) \,|\, S_{t+1}, A_{t+1} = \pi'(S_{t+1})\right]}_{\text{Eqn. (3.18) w.r.t. } S_{t+1}} \,\Big|\, S_t = s\Big] \\
&= \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma\, R_{t+2} + \gamma^2\, v_\pi(S_{t+2}) \,\Big|\, S_t = s\Big] \qquad \leftarrow (\ddagger) \\
&\leq \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma\, R_{t+2} + \gamma^2\, R_{t+3} + \gamma^3\, v_\pi(S_{t+3}) \,\Big|\, S_t = s\Big] \qquad \leftarrow \text{repeat } (\dagger) \text{ to } (\ddagger) \\
&\;\;\vdots \\
&\leq \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma\, R_{t+2} + \gamma^2\, R_{t+3} + \gamma^3\, R_{t+4} + \cdots \,\Big|\, S_t = s\Big] \\
&= v_{\pi'}(s) \qquad (3.19)
\end{aligned}$$

Hence, we can conclude that Equation (3.17) implies $v_{\pi'}(s) \geq v_\pi(s)$, which means $\pi' \geq \pi$. We have completed the first step.

Now, for the second step, the question is how to select the new policy π' such that it satisfies Equation (3.17). One way is to choose $\pi'(s)$ according to

$$\begin{aligned}
\pi'(s) &\triangleq \arg\max_a\, q_\pi(s, a) \\
&= \arg\max_a\, \mathbb{E}\Big[R_{t+1} + \gamma\, v_\pi(S_{t+1}) \,\Big|\, S_t = s, A_t = a\Big] \\
&= \arg\max_a\, \sum_{s', r} p(s', r \,|\, s, a)\big[r + \gamma\, v_\pi(s')\big] \qquad (3.20)
\end{aligned}$$

This means that we can just modify π to produce a better policy π', by simply selecting at s the action that looks best (in terms of one step look-ahead) according to the current values of $v_\pi(s)$, as is illustrated in Example 3.1.2. Such a new policy π' is called a *greedy policy*. This process of creating a new policy π' that selects a 'greedy action' at s based on the value function v_π of the current policy π is called *policy improvement*.

greedy policy

policy improvement

Note that by the way π' is chosen, we have $q_\pi\left(s, \pi'(s)\right) \geq q_\pi\left(s, \pi(s)\right)$. It can be shown that the new greedy policy $\pi'(s)$ satisfies Equation (3.17). For the simpler case of deterministic policies, we have, from Equation (2.55),

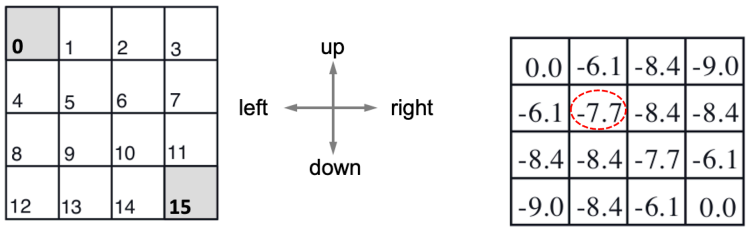
$$q_\pi\left(s, \pi'(s)\right) \geq q_\pi(s, \pi(s)) = v_\pi(s). \qquad (3.21)$$

Proof for the general case involving non-deterministic policies is more complicated and can be found in various sources, such as [10].

If applying the greedy policy at s results in $\pi'(s) \neq \pi(s)$, then taking action $\pi'(s)$ at s and following π thereafter guarantees that $v_{\pi'}(s) > v_\pi(s)$ and so π' is better than π. If on the other hand it turns out that $\pi'(s) = \pi(s)$, i.e., $\pi(s)$ is already a 'greedy action', then we have $v_{\pi'} = v_\pi$ and so $\pi' = \pi$, which means π cannot be improved.

Example 3.1.2. Consider an environment consisting of the 4-by-4 grid world as shown in Figure 3.5(a), where the shaded squares indicate terminal states. The reward function is defined such that every time the agent takes an action it receives a reward of -1. At each state, the agent can take any of the four actions: $\{\mathsf{up}, \mathsf{right}, \mathsf{down}, \mathsf{left}\}$, and transition into the next state in the direction of the action, except for the cases where the action takes the agent off the gird; for those cases the agent remains in the same state before the action was taken, e.g, $p(6, -1 \,|\, 5, \mathsf{right}) = 1$ and $p(4, -1 \,|\, 4, \mathsf{left}) = 1$. Let $\gamma = 1$.

Assume that the agent is currently following a deterministic policy π, and that the current state values $v_\pi(s)$ are as shown in Figure 3.5(b). Suppose that the current deterministic policy π is such that $\pi(5) = \mathsf{right}$ and we would like to improve π for state 5. We note that $v_\pi(5) = -7.7$, and $q_\pi(5, \pi(5)) = v_\pi(5) = -7.7$. The four states adjacent to state 5 are $s' = 1, 4, 6, 9$, with $v_\pi(s') = -6.1, -6.1, -8.4, -8.4$, respectively.



(a) (b)

Figure 3.5: (a) A 4×4 grid environment. (b) Values of $v_\pi(s)$.

To improve π at state 5, we follow the greedy policy as given in Equation (3.20) to determine the action that the new policy π' should take at state 5. Now

$$\begin{aligned}
\pi'(s) &\triangleq \arg\max_a q_\pi(s,a) \\
&= \arg\max_a \sum_{s',r} p(s',r \mid s,a)\big[r + \gamma\, v_\pi(s')\big] \\
&= \arg\max_a \sum_{s'} p(s',-1 \mid 5,a)\big[-1 + 1 \cdot v_\pi(s')\big] \\
&= \arg\max_a \Big[\overline{f}(5,a,s')\,[-1 + 1 \cdot v_\pi(s')]\Big]_{s'=1,4,6,9} \\
&= \{\mathsf{up}, \mathsf{left}\} \qquad (3.22)
\end{aligned}$$

Hence, the best action for the new greedy policy π' to take at state 5 is either $\pi'(5) = \mathsf{up}$ or $\pi'(5) = \mathsf{left}$.

We next show that the condition in the Policy Improvement Theorem, i.e., Equation (3.17), is satisfied. We will look at the case $\pi'(5) = \mathsf{up}$ and show that $q_{\pi'}(5, \pi'(5)) \geq v_\pi(5)$. For the case of $\pi'(5) = \mathsf{left}$, the same procedure applies.

Now

$$\begin{aligned}
q_{\pi'}(s,a) &= q_{\pi'}(5, \pi'(5)) \\
&= q_{\pi'}(5, \mathsf{up}) \\
&= \sum_{s',r} p(s',r \mid s,a)\big[r + \gamma\, v_\pi(s')\big] \quad \text{(from Equation (2.56))} \\
&= p(1,-1 \mid 5, \mathsf{up}) \times \big[-1 + 1 \cdot v_\pi(1)\big] \\
&= 100\% \times \big[-1 - 6.1\big] \\
&= -7.1 \\
&> -7.7 = v_\pi(5) \qquad (3.23)
\end{aligned}$$

Since π' is deterministic, we have $q_{\pi'}(5, \mathsf{up}) = v_{\pi'}(5)$. Consequently, $v_{\pi'}(5) = -7.1 > -7.7 = v_\pi(5)$; that is, π' is better than π. In practice, the decision to execute up or left is made arbitrarily. □

3.1.3 Convergence of policy iteration

We have now shown that, given a policy π, we can determine v_π and subsequently obtain a new policy π' which is as good as or better then π, by replacing the action $a = \pi(s)$ with an action $a = \pi'(s)$ that is greedy with respect to v_π. Now the question is: Does policy iteration guarantee that an optimal policy will be found?

To address this question, we refer to Figure 3.1. We see that each time a policy π is

improved, we will have a new and better policy π'. Now what if we reach a point where the current policy cannot be improved, i.e., $\pi'(s) = \pi(s)$ for all s? In this case, we have $v_\pi(s) = v_{\pi'}(s)$. Since at each state s we always pick the 'greedy action' according to the values of v_π, we now have (following the same argument presented when discussing Equations (2.69)-(2.72)),

$$\begin{aligned} v_{\pi'} &= \max_a \mathbb{E}\Big[R_{t+1} + \gamma\, v_{\pi'}(S_{t+1}) \,\Big|\, S_t = s, A_t = a\Big] \\ &= \max_a \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma\, v_{\pi'}(s')\big] \end{aligned} \tag{3.24}$$

which is the same as the Bellman Optimality Equation with $v_* = v_{\pi'}$, and so $\pi' = \pi_*$. Therefore, policy iteration is guaranteed to yield an optimal policy.

3.2 Value Iteration

Policy iteration comprises cycles of a complete run of the policy evaluation sub-process, followed by the policy improvement process. The policy evaluation sub-process may take a large number of iterations to complete. In practice, it is often the case that the policy evaluation sub-process can be truncated and policy improvement can be applied on some intermediate state values (i.e., before convergence) to yield an optimal policy. Example 3.2.1 illustrates this point.

Example 3.2.1. The left column in Figure 3.6 shows sample state values during the policy evaluation sub-process under a random policy $\pi = \pi_0$ for the grid world problem described in Example 3.1.1. The right column shows the greedy policy generated based on the state values obtained at the end of the iteration as indicated. The last row shows the converged state values v_{π_0} under the random policy π_0, and the new policy π_1 which is greedy with respect to the converged state values v_{π_0}. It can be seen that if we were to truncate policy evaluation after the third iteration (i.e., $k = 3$) and then carry out policy improvement, we could have obtained the same π_1. □

3.2.1 Update rules

If we conduct policy iteration by truncating policy evaluation after just one iteration and then carrying out policy improvement as usual, then we will have the following update rule

value iteration on $v(s)$

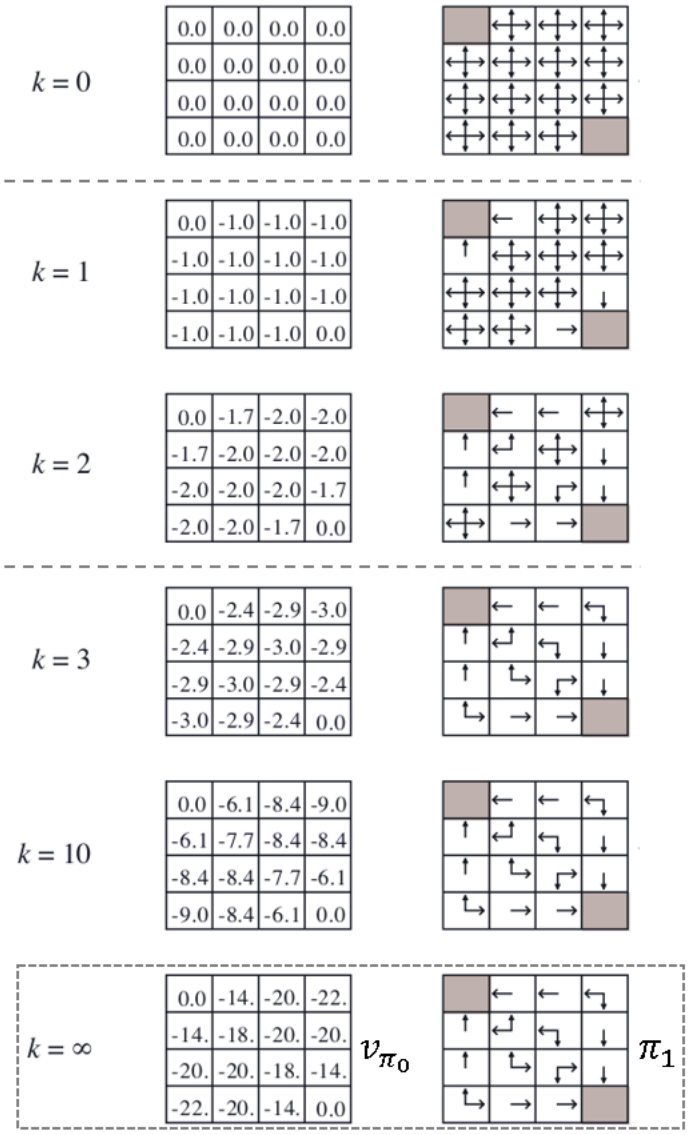


Figure 3.6: Sample state values and their corresponding greedy policies during the policy evaluation sub-process for the grid world problem described in Example 3.1.1.

that combines (truncated) policy evaluation and policy improvement:

$$\begin{aligned} V_{k+1}(s) &= \max_a \mathbb{E}\Big[R_{t+1} + \gamma\, v_k(S_{t+1}) \,\Big|\, S_t = s, A_t = a)\Big] \\ &= \max_a \sum_{s',r} p\,(s', r \mid s, a)\,\Big[r + \gamma\, V_k(s')\Big], \quad \text{for all } s \in \mathcal{S} \end{aligned} \tag{3.25}$$

This is called *value iteration on state value* $v(s)$. It can be shown (see [15]) that starting with an arbitrary V_0, the sequence $\{V_k\}$ generated by Equation (3.25) converges to the optimal value v_* as $k \to \infty$.

Note that Equation (3.25) is the same as the update rule for policy evaluation, i.e., Equation (3.2), except that the maximum is to be taken over all actions. Another way to

view Equation (3.25) is that it is a result of turning the Bellman Optimality Equation for v_*, i.e., Equation (2.72), into an update rule.

The same idea can be applied to the action values $Q(s,a)$. That is, we can turn the Bellman Optimality Equation for q_*, i.e., Equation (2.73), in to an update rule for $Q(s,a)$, i.e.,

value iteration on $Q(s,a)$

$$Q_{k+1}(s,a) = \sum_{s',r} p(s',r \mid s,a)\Big[r + \gamma \max_{a'} Q_k(s',a')\Big] \tag{3.26}$$

This is called *value iteration on action value* $q(s,a)$. It can be shown [15] that starting with an arbitrary Q_0, the sequence $\{Q_k\}$ generated by Equation (3.26) converges to the optimal value q_*. We will present a simplified proof later.

Once the optimal values $v_*(s)$ or $q_*(s,a)$ are found, an optimal policy can be extracted based on these values, i.e.,

$$\pi_*(s) = \begin{cases} \arg\max_a \sum_{s',r} p\left(s',r \mid s,a\right)\Big[r + \gamma\, v_*(s')\Big] \\ \arg\max_a \; q_*(s,a) \end{cases} \tag{3.27}$$

An algorithm for implementing value iteration on the action value $q(s,a)$ is shown in Table 3.2. The algorithm for implementing value iteration on $v(s)$ is very similar and so will be omitted here.

Table 3.2: Algorithm for value evaluation on $q(s,a)$.

Input:	◦ A small threshold θ
	◦ Initial values $Q_0(s,a)$ for all $(s,a) \in \mathcal{S} \times \mathcal{A}(s)$; all $Q_0(s,a)$ are arbitrary except $Q(s,a) = 0$ for all $(s,a) \in \hat{\mathcal{S}} \times \mathcal{A}(s)$, with $\hat{\mathcal{S}}$ being the set of terminal states
Repeat	$\Delta \leftarrow 0$
	Loop through all pairs $(s,a) \in \mathcal{S} \times \mathcal{A}(s)$:
	$q \leftarrow Q(s,a)$
	$Q(s,a) \leftarrow \sum_{s',r} p(s',r \mid s,a)\Big[r + \gamma \max_{a'} Q(s',a')\Big]$ $\Leftarrow$ This is Equation (3.26)
	$\Delta \leftarrow \max\big(\Delta, \lvert q - Q(s,a)\rvert\big)$
until	$\Delta < \theta$
Output	$Q(s,a)$, which the estimated value of $q_*(s,a)$

The detailed calculations involved in value iteration on $q(s,a)$ are illustrated in Example 3.2.2 (for the case of deterministic state-transitions) and Example 3.2.3 (for the case of non-deterministic state-transitions). In these two examples, the policy and the reward function are deterministic, so the update rule can be expressed in terms of the transition probabilities

$P^a_{ss'}$ and the reward function $\rho(s,a,s')$ instead of the general p-function, i.e.,

$$Q(s,a) \leftarrow \sum_{s'} P^a_{ss'} \left(\rho(s,a,s') + \gamma \max_{a'} Q\left(s',a'\right) \right) \tag{3.28}$$

3.2.2 Illustrative examples

Example 3.2.2. Consider the cleaning robot problem described earlier in Example 2.2.1, with the following deterministic state transition function $\bar{f}$ and reward function ρ:

$$\bar{f}(s,a) = \begin{cases} s+a & \text{if } 1 \le s \le 4 \\ s & \text{if } s = 0 \text{ or } s = 5 \end{cases} \tag{3.29}$$

$$\rho(s,a,s') = \begin{cases} 5 & \text{if } s \neq 5 \text{ and } s' = 5 \\ 1 & \text{if } s \neq 0 \text{ and } s' = 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.30}$$

Let $\gamma = 0.5$. Show the calculations for the first two iterations of the value iteration algorithm on $q(s,a)$.

Solution: For deterministic transitions, the update rule degenerates to

$$Q(s,a) = \rho(s,a,s') + \gamma \max_{a'} Q\left(\bar{f}(s,a), a'\right) \tag{3.31}$$

The iterative algorithm proceeds as follows. First, the values for the $Q(s,a)$ are initialized to be identically 0 for all states and actions, i.e.,

$$Q_0(s,a) = 0 \quad \text{for} \quad s = 0,1,2,3,4,5 \quad \text{and} \quad a = \pm 1 \tag{3.32}$$

Iteration $k = 1$:

$$\begin{aligned} Q(0,1) &= \rho(0,1,0) + 0.5 \max_{a'} Q\left(\bar{f}(0,1), a'\right) \quad \Big(\text{Note: } \bar{f}(0,1) = 0\Big) \\ &= \rho(0,1,0) + 0.5 \max \underbrace{\left[Q\left(0,-1\right), Q\left(0,1\right)\right]}_{\text{for } a' = \pm 1 \text{ at state } 0} \\ &= 0 + 0.5 \cdot \max\left[0,0\right] \quad \Big(\text{Note: } \max[(\cdot)] \text{ means the maximum in } (\cdot)\Big) \\ &= 0 \end{aligned} \tag{3.33}$$

$$\begin{aligned} Q(0,-1) &= \rho(0,-1,0) + 0.5 \max_{a'} Q\left(\bar{f}(0,-1), a'\right) \quad \Big(\text{Note: } \bar{f}(0,-1) = 0\Big) \\ &= \rho(0,-1,0) + 0.5 \text{ max}\left[Q\left(0,-1\right), Q\left(0,1\right)\right] \\ &= 0 + 0.5 \text{max}\left[0,0\right] \\ &= 0 \end{aligned} \tag{3.34}$$

$$\begin{aligned} Q(1,1) &= \rho(1,1,2) + 0.5 \max_{a'} Q\left(\bar{f}(1,1), a'\right) \quad \Big(\text{Note: } \bar{f}(1,1) = 2\Big) \\ &= \rho(1,1,2) + 0.5 \text{ max} \underbrace{\left[Q\left(2,-1\right), Q\left(2,1\right)\right]}_{\text{for } a' = \pm 1 \text{ at state 2}} \\ &= 0 + 0.5 \cdot \text{max}\left[0,0\right] \\ &= 0 \end{aligned} \tag{3.35}$$

$$\begin{aligned} Q(1,-1) &= \rho(1,-1,0) + 0.5 \max_{a'} Q\left(\bar{f}(1,-1), a'\right) \quad \Big(\text{Note: } \bar{f}(1,-1) = 0\Big) \\ &= \rho(1,-1,0) + 0.5 \text{ max}\left[Q\left(0,-1\right), Q\left(0,1\right)\right] \\ &= 1 + 0.5 \cdot \text{max}\left[0,0\right] \\ &= 1 \end{aligned} \tag{3.36}$$

$$\begin{aligned} Q(2,1) &= \rho(2,1,3) + 0.5 \max_{a'} Q\left(\bar{f}(2,1), a'\right) \quad \Big(\text{Note: } \bar{f}(2,1) = 3\Big) \\ &= \rho(2,1,3) + 0.5 \text{ max}\left[Q\left(3,-1\right), Q\left(3,1\right)\right] \\ &= 0 + 0.5 \cdot \text{max}\left[0,0\right] \\ &= 0 \end{aligned} \tag{3.37}$$

$$\begin{aligned} Q(2,-1) &= \rho(2,-1,1) + 0.5 \max_{a'} Q\left(\bar{f}(2,-1), a'\right) \quad \Big(\text{Note: } \bar{f}(2,-1) = 1\Big) \\ &= \rho(2,-1,1) + 0.5 \text{ max}\left[Q\left(1,-1\right), Q\left(1,1\right)\right] \\ &= 0 + 0.5 \cdot \text{max}\left[1,0\right] \\ &= 0.5 \end{aligned} \tag{3.38}$$

Note that in Equation (3.38) above we use the updated value of $Q(1,-1) = 1$ as calculated in Equation (3.36), and not $Q(1,-1) = 0$ as initially assigned in Equation (3.32) before the

start of the iteration. This is an asynchronous implementation of Dynamic Programming.

$$\begin{aligned} Q(3,1) &= \rho(3,1,4) + 0.5 \max_{a'} Q\left(\bar{f}(3,1), a'\right) \quad \Big(\text{Note: } \bar{f}(3,1) = 4\Big) \\ &= \rho(3,1,4) + 0.5 \text{ max}\left[Q\left(4,-1\right), Q\left(4,1\right)\right] \\ &= 0 + 0.5 \cdot \text{max}\left[0,0\right] \\ &= 0 \end{aligned} \tag{3.39}$$

$$\begin{aligned} Q(3,-1) &= \rho(3,-1,2) + 0.5 \max_{a'} Q\left(\bar{f}(3,-1), a'\right) \quad \Big(\text{Note: } \bar{f}(3,-1) = 2\Big) \\ &= \rho(3,-1,2) + 0.5 \text{ max}\left[Q\left(2,-1\right), Q\left(2,1\right)\right] \\ &= 0 + 0.5 \cdot \text{max}\left[0.5,0\right] \\ &= 0.25 \end{aligned} \tag{3.40}$$

$$\begin{aligned} Q(4,1) &= \rho(4,1,5) + 0.5 \max_{a'} Q\left(\bar{f}(4,1), a'\right) \quad \Big(\text{Note: } \bar{f}(4,1) = 5\Big) \\ &= \rho(4,1,5) + 0.5 \text{ max}\left[Q\left(5,-1\right), Q\left(5,1\right)\right] \\ &= 5 + 0.5 \cdot \text{max}\left[0,0\right] \\ &= 5 \end{aligned} \tag{3.41}$$

$$\begin{aligned} Q(4,-1) &= \rho(4,-1,3) + 0.5 \max_{a'} Q\left(\bar{f}(4,-1), a'\right) \quad \Big(\text{Note: } \bar{f}(4,-1) = 3\Big) \\ &= \rho(4,-1,3) + 0.5 \text{ max}\left[Q\left(3,-1\right), Q\left(3,1\right)\right] \\ &= 0 + 0.5 \cdot \text{max}\left[0.25,0\right] \\ &= 0.125 \end{aligned} \tag{3.42}$$

$$\begin{aligned} Q(5,1) &= \rho(5,1,5) + 0.5 \max_{a'} Q\left(\bar{f}(5,1), a'\right) \quad \Big(\text{Note: } \bar{f}(5,1) = 5\Big) \\ &= \rho(5,1,5) + 0.5 \text{ max}\left[Q\left(5,-1\right), Q\left(5,1\right)\right] \\ &= 0 + 0.5 \cdot \text{max}\left[0,0\right] \\ &= 0 \end{aligned} \tag{3.43}$$

$$\begin{aligned} Q(5,-1) &= \rho(5,-1,5) + 0.5 \max_{a'} Q\left(\bar{f}(5,-1), a'\right) \quad \Big(\text{Note: } \bar{f}(5,-1) = 5\Big) \\ &= \rho(5,-1,5) + 0.5 \text{ max}\left[Q\left(5,-1\right), Q\left(5,1\right)\right] \\ &= 0 + 0.5 \text{max}\left[0,0\right] \\ &= 0 \end{aligned} \tag{3.44}$$

Iteration $k = 2$:

$$\begin{aligned} Q(0,1) &= \rho(0,1,0) + 0.5 \max_{a'} Q\left(\bar{f}(0,1), a'\right) \\ &= \rho(0,1,0) + 0.5 \text{ max}\left[Q\left(0,-1\right), Q\left(0,1\right)\right] \\ &= 0 + 0.5 \cdot \text{max}\left[0,0\right] \\ &= 0 \end{aligned} \tag{3.45}$$

$$\begin{aligned} Q(0,-1) &= \rho(0,-1,0) + 0.5 \max_{a'} Q\left(\bar{f}(0,-1), a'\right) \\ &= \rho(0,-1,0) + 0.5 \text{ max}\left[Q\left(0,-1\right), Q\left(0,1\right)\right] \\ &= 0 + 0.5 \text{max}\left[0,0\right] \\ &= 0 \end{aligned} \tag{3.46}$$

$$\begin{aligned} Q(1,1) &= \rho(1,1,2) + 0.5 \max_{a'} Q\left(\bar{f}(1,1), a'\right) \\ &= \rho(1,1,2) + 0.5 \text{ max}\left[Q\left(2,-1\right), Q\left(2,1\right)\right] \\ &= 0 + 0.5 \cdot \text{max}\left[0.5,0\right] \\ &= 0.25 \end{aligned} \tag{3.47}$$

$$\begin{aligned} Q(1,-1) &= \rho(1,-1,0) + 0.5 \max_{a'} Q\left(\bar{f}(1,-1), a'\right) \\ &= \rho(1,-1,0) + 0.5 \text{ max}\left[Q\left(0,-1\right), Q\left(0,1\right)\right] \\ &= 1 + 0.5 \cdot \text{max}\left[0,0\right] \\ &= 1 \end{aligned} \tag{3.48}$$

$$\begin{aligned} Q(2,1) &= \rho(2,1,3) + 0.5 \max_{a'} Q\left(\bar{f}(2,1), a'\right) \\ &= \rho(2,1,3) + 0.5 \text{ max}\left[Q\left(3,-1\right), Q\left(3,1\right)\right] \\ &= 0 + 0.5 \cdot \text{max}\left[0.25,0\right] \\ &= 0.125 \end{aligned} \tag{3.49}$$

$$\begin{aligned} Q(2,-1) &= \rho(2,-1,1) + 0.5 \max_{a'} Q\left(\bar{f}(2,-1), a'\right) \\ &= \rho(2,-1,1) + 0.5 \text{ max}\left[Q\left(1,-1\right), Q\left(1,1\right)\right] \\ &= 0 + 0.5 \cdot \text{max}\left[1,0.25\right] \\ &= 0.5 \end{aligned} \tag{3.50}$$

$$\begin{aligned} Q(3,1) &= \rho(3,1,4) + 0.5 \max_{a'} Q\left(\bar{f}(3,1), a'\right) \\ &= \rho(3,1,4) + 0.5 \text{ max}\left[Q\left(4,-1\right), Q\left(4,1\right)\right] \\ &= 0 + 0.5 \cdot \text{max}\left[0.125, 5\right] \\ &= 2.5 \end{aligned} \tag{3.51}$$

$$\begin{aligned} Q(3,-1) &= \rho(3,-1,2) + 0.5 \max_{a'} Q\left(\bar{f}(3,-1), a'\right) \\ &= \rho(3,-1) + 0.5 \text{ max}\left[Q\left(2,-1\right), Q\left(2,1\right)\right] \\ &= 0 + 0.5 \cdot \text{max}\left[0.5, 0.125\right] \\ &= 0.25 \end{aligned} \tag{3.52}$$

$$\begin{aligned} Q(4,1) &= \rho(4,1,5) + 0.5 \max_{a'} Q\left(\bar{f}(4,1), a'\right) \\ &= \rho(4,1,5) + 0.5 \text{ max}\left[Q\left(5,-1\right), Q\left(5,1\right)\right] \\ &= 5 + 0.5 \cdot \text{max}\left[0, 0\right] \\ &= 5 \end{aligned} \tag{3.53}$$

$$\begin{aligned} Q(4,-1) &= \rho(4,-1,3) + 0.5 \max_{a'} Q\left(\bar{f}(4,-1), a'\right) \\ &= \rho(4,-1,3) + 0.5 \text{ max}\left[Q\left(3,-1\right), Q\left(3,1\right)\right] \\ &= 0 + 0.5 \cdot \text{max}\left[0.25, 2.5\right] \\ &= 1.25 \end{aligned} \tag{3.54}$$

$$\begin{aligned} Q(5,1) &= \rho(5,1,5) + 0.5 \max_{a'} Q\left(\bar{f}(5,1), a'\right) \\ &= \rho(5,1) + 0.5 \text{ max}\left[Q\left(5,-1\right), Q\left(5,1\right)\right] \\ &= 0 + 0.5 \cdot \text{max}\left[0, 0\right] \\ &= 0 \end{aligned} \tag{3.55}$$

$$\begin{aligned} Q(5,-1) &= \rho(5,-1,5) + 0.5 \max_{a'} Q\left(\bar{f}(5,-1), a'\right) \\ &= \rho(5,-1) + 0.5 \text{ max}\left[Q\left(5,-1\right), Q\left(5,1\right)\right] \\ &= 0 + 0.5 \text{max}\left[0, 0\right] \\ &= 0 \end{aligned} \tag{3.56}$$

The iterations are carried out until the values of $Q(s,a)$ converge, which are shown in

Table 3.3. The optimal values for $Q(s,a)$ are obtained after five iterations, i.e., $Q^* = Q_5$. The first row of Table 3.3 lists all six states. The five rows below the row of initial values (i.e., Q_0) show the values of $Q(s,a)$ for five iterations, with the two numbers (separated by a vertical bar "|") in each state column indicate the values of $Q(s,a)$ when taking action $a = -1$ and $a = 1$ at that state. For instance, the two numbers for Q_2 under state 1 are 1.000 and 0.250. This means that $Q_2(1,-1) = 1.000$ and $Q_2(1,1) = 0.250$.

Table 3.3: Values of $Q(s,a)$ for robot with deterministic state transitions.

Action Value	State 0	1	2	3	4	5
Q_0	0.000|0.000	0.000|0.000	0.000|0.000	0.000|0.000	0.000|0.000	0.000|0.000
Q_1	0.000|0.000	1.000|0.000	0.500|0.000	0.250|0.000	0.125|5.000	0.000|0.000
Q_2	0.000|0.000	1.000|0.250	0.500|0.125	0.250|0.250	1.250|5.000	0.000|0.000
Q_3	0.000|0.000	1.000|0.250	0.500|1.250	0.625|2.500	1.250|5.000	0.000|0.000
Q_4	0.000|0.000	1.000|0.625	0.500|1.250	0.625|2.500	1.250|5.000	0.000|0.000
Q_5	0.000|0.000	1.000|0.625	0.500|1.250	0.625|2.500	1.250|5.000	0.000|0.000

From the table, we see that at state 1, the optimal values for the $Q(s,a)$ for the two actions are 1.000 (for $a = -1$) and 0.625 (for $a = 1$). So we have $\pi_*(1) = -1$. Applying the same reasoning on other states yields the following optimal policy (which is illustrated in Figure 3.7): $\pi_*(1) = -1$, $\pi_*(2) = 1$, $\pi_*(3) = 1$, and $\pi_*(4) = 1$. □

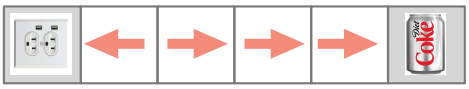


Figure 3.7: Optimal policy.

Example 3.2.3. Consider the problem described in Example 3.2.2. Suppose that the behavior of the robot is now stochastic with the state transition probabilities as given in Table 3.4 below. Determine the optimal values of $Q(s,a)$.

Table 3.4: State-transition probabilities.

(s,a)	$f(s,a,0)$	$f(s,a,1)$	$f(s,a,2)$	$f(s,a,3)$	$f(s,a,4)$	$f(s,a,5)$
$(0,-1)$	1	0	0	0	0	0
$(1,-1)$	0.8	0.15	0.05	0	0	0
$(2,-1)$	0	0.8	0.15	0.05	0	0
$(3,-1)$	0	0	0.8	0.15	0.05	0
$(4,-1)$	0	0	0	0.8	0.15	0.05
$(5,-1)$	0	0	0	0	0	1
$(0,1)$	1	0	0	0	0	0
$(1,1)$	0.05	0.15	0.8	0	0	0
$(2,1)$	0	0.05	0.15	0.8	0	0
$(3,1)$	0	0	0.05	0.15	0.8	0
$(4,1)$	0	0	0	0.05	0.15	0.8
$(5,1)$	0	0	0	0	0	1

Solution: The solution process is similar to that shown in Example 3.2.2. The difference is that now when updating the values for $Q(s,a)$ we need to take into account the stochastic state transitions by using the general form of the Bellman Equation:

$$Q(s,a) = \sum_{s'} P^a_{ss'} \left(\rho(s,a,s') + \gamma \max_{a'} Q\left(s',a'\right) \right) \tag{3.57}$$

where the values for $P^a_{ss'} \equiv f(s,a,s')$ are as given in Table 3.4. The initial values for $Q(s,a)$ are set to identically 0, i.e., $Q_0(s,a) = 0$ for $s = 0,1,2,3,4,5$ and $a = \pm 1$.

Iteration 1:

$$\begin{aligned}
Q(0,-1) &= \sum_{s'} P^{-1}_{0s'} \left(\rho(0,-1,s') + \gamma \max_{a'} Q\left(s',a'\right) \right) \\
&= \sum_{j=0}^{0} P^{-1}_{0j} \left(\rho(0,-1,j) + \gamma \max_{a'} Q\left(j,a'\right) \right) \\
&= P^{-1}_{00} \left(\rho(0,-1,0) + \gamma \max_{a'} Q\left(0,a'\right) \right) \quad \text{(Since 0 is the only } s'\text{)} \\
&= 1 \cdot \left(0 + 0.5 \max\left[Q(0,-1), Q(0,1) \right] \right) \\
&= 0.5 \max\left[0,0\right] \\
&= 0
\end{aligned} \tag{3.58}$$

$$\begin{aligned}
Q(0,1) &= \sum_{s'} P^{1}_{0s'} \left(\rho(0,1,s') + \gamma \max_{a'} Q\left(s',a'\right) \right) \\
&= \sum_{j=0}^{0} P^{1}_{0j} \left(\rho(0,1,j) + \gamma \max_{a'} Q\left(j,a'\right) \right) \\
&= P^{1}_{00} \left(\rho(0,1,0) + \gamma \max_{a'} Q\left(0,a'\right) \right) \quad \text{(Since 0 is the only } s'\text{)} \\
&= 1 \cdot \left(0 + 0.5 \max\left[Q(0,-1), Q(0,1) \right] \right) \\
&= 0.5 \max\left[0,0\right] \\
&= 0
\end{aligned} \tag{3.59}$$

$$
\begin{aligned}
Q(1,-1) &= \sum_{s'} P_{1s'}^{-1}\Big(\rho(1,-1,s')+\gamma \max_{a'} Q\left(s',a'\right)\Big) \\
&= \sum_{j=0}^{2} P_{1j}^{-1}\Big(\rho(1,-1,j)+\gamma \max_{a'} Q\left(j,a'\right)\Big) \\
&= P_{10}^{-1}\Big(\rho(1,-1,0)+\gamma \max_{a'} Q\left(0,a'\right)\Big) \quad \Big(\text{Note: } j=0\Big) \\
&+ P_{11}^{-1}\Big(\rho(1,-1,1)+\gamma \max_{a'} Q\left(1,a'\right)\Big) \quad \Big(\text{Note: } j=1\Big) \\
&+ P_{12}^{-1}\Big(\rho(1,-1,2)+\gamma \max_{a'} Q\left(2,a'\right)\Big) \quad \Big(\text{Note: } j=2\Big) \\
&= 0.8\cdot\Big(1+0.5\cdot \max\big[Q(0,-1),Q(0,1)\big]\Big) \quad \Big(\text{Note: } \rho(1,-1,0)=1\ \Big) \\
&+ 0.15\cdot\Big(0+0.5\cdot \max\big[Q(1,-1),Q(1,1)\big]\Big) \\
&+ 0.05\cdot\Big(0+0.5\cdot \max\big[Q(2,-1),Q(2,1)\big]\Big) \\
&= 0.8\cdot\Big(1+0.5\cdot \max\big[0,0\big]\Big)+0.15\cdot 0.5\cdot \max\big[0,0\big]+0.05\cdot 0.5\cdot \max\big[0,0\big] \\
&= 0.8 \qquad (3.60)
\end{aligned}
$$

$$
\begin{aligned}
Q(1,1) &= \sum_{s'} P_{1s'}^{1}\Big(\rho(1,1,s')+\gamma \max_{a'} Q\left(s',a'\right)\Big) \\
&= \sum_{j=0}^{2} P_{1j}^{1}\Big(\rho(1,1,j)+\gamma \max_{a'} Q\left(j,a'\right)\Big) \\
&= P_{10}^{1}\Big(\rho(1,1,0)+\gamma \max_{a'} Q\left(0,a'\right)\Big) \quad \Big(\text{Note: } j=0\Big) \\
&+ P_{11}^{1}\Big(\rho(1,1,1)+\gamma \max_{a'} Q\left(1,a'\right)\Big) \quad \Big(\text{Note: } j=1\Big) \\
&+ P_{12}^{1}\Big(\rho(1,1,2)+\gamma \max_{a'} Q\left(2,a'\right)\Big) \quad \Big(\text{Note: } j=2\Big) \\
&= 0.05\cdot\Big(1+0.5\cdot \max\big[Q(0,-1),Q(0,1)\big]\Big) \quad \Big(\text{Note: } \rho(1,1,0)=1\Big) \\
&+ 0.15\cdot\Big(0+0.5\cdot \max\big[Q(1,-1),Q(1,1)\big]\Big) \quad \Big(\text{Note: } Q(1,-1)=0.8\Big) \\
&+ 0.8\cdot\Big(0+0.5\cdot \max\big[Q(2,-1),Q(2,1)\big]\Big) \\
&= 0.05\Big(1+0.5\cdot \max\big[0,0\big]\Big)+0.15\cdot 0.5\cdot \max\big[0.8,0\big]+0.8\cdot 0.5\cdot \max\big[0,0\big] \\
&= 0.11 \qquad (3.61)
\end{aligned}
$$

Note that in Equation (3.61) above, we use the $Q(1,-1)$ value just updated from Equation

(3.60), i.e., $Q(1,-1)=0.8$, instead of $Q(1,-1)=0$ as initially assigned.

$$\begin{aligned}
Q(2,-1) &= \sum_{s'} P_{2s'}^{-1}\left(\rho(2,-1,s')+\gamma\max_{a'} Q\left(s',a'\right)\right)\\
&= \sum_{j=1}^{3} P_{2j}^{-1}\left(\rho(2,-1,j)+\gamma\max_{a'} Q\left(j,a'\right)\right)\\
&= P_{21}^{-1}\left(\rho(2,-1,1)+\gamma\max_{a'} Q\left(1,a'\right)\right) \quad \left(\text{Note: } j=1\right)\\
&+ P_{22}^{-1}\left(\rho(2,-1,2)+\gamma\max_{a'} Q\left(2,a'\right)\right) \quad \left(\text{Note: } j=2\right)\\
&+ P_{23}^{-1}\left(\rho(2,-1,3)+\gamma\max_{a'} Q\left(3,a'\right)\right) \quad \left(\text{Note: } j=3\right)\\
&= 0.8\cdot\left(0+0.5\cdot\max\left[Q(1,-1),Q(1,1)\right]\right)\\
&+ 0.15\cdot\left(0+0.5\cdot\max\left[Q(2,-1),Q(2,1)\right]\right)\\
&+ 0.05\cdot\left(0+0.5\cdot\max\left[Q(3,-1),Q(3,1)\right]\right)\\
&= 0.8\cdot 0.5\cdot\max\left[0.8,0.11\right]+0+0\\
&= 0.32
\end{aligned} \tag{3.62}$$

$$\begin{aligned}
Q(2,1) &= \sum_{s'} P_{2s'}^{1}\left(\rho(2,1,s')+\gamma\max_{a'} Q\left(s',a'\right)\right)\\
&= \sum_{j=1}^{3} P_{2j}^{1}\left(\rho(2,1,j)+\gamma\max_{a'} Q\left(j,a'\right)\right)\\
&= P_{21}^{1}\left(\rho(2,1,1)+\gamma\max_{a'} Q\left(1,a'\right)\right)\\
&+ P_{22}^{1}\left(\rho(2,1,2)+\gamma\max_{a'} Q\left(2,a'\right)\right)\\
&+ P_{23}^{1}\left(\rho(2,1,3)+\gamma\max_{a'} Q\left(3,a'\right)\right)\\
&= 0.05\cdot\left(0+0.5\cdot\max\left[Q(1,-1),Q(1,1)\right]\right)\\
&+ 0.15\cdot\left(0+0.5\cdot\max\left[Q(2,-1),Q(2,1)\right]\right)\\
&+ 0.8\cdot\left(0+0.5\cdot\max\left[Q(3,-1),Q(3,1)\right]\right)\\
&= 0.05\cdot 0.5\cdot\max\left[0.8,0.11\right]+0.15\cdot 0.5\cdot\max\left[0.32,0\right]+0\\
&= 0.044
\end{aligned} \tag{3.63}$$

$$
\begin{aligned}
Q(3,-1) &= \sum_{s'} P_{3s'}^{-1}\left(\rho(3,-1,s')+\gamma \max_{a'} Q\left(s',a'\right)\right) \\
&= \sum_{j=2}^{4} P_{3j}^{-1}\left(\rho(3,-1,j)+\gamma \max_{a'} Q\left(j,a'\right)\right) \\
&= P_{32}^{-1}\left(\rho(3,-1,2)+\gamma \max_{a'} Q\left(2,a'\right)\right) \\
&+ P_{33}^{-1}\left(\rho(3,-1,3)+\gamma \max_{a'} Q\left(3,a'\right)\right) \\
&+ P_{34}^{-1}\left(\rho(3,-1,4)+\gamma \max_{a'} Q\left(4,a'\right)\right) \\
&= 0.8\cdot\left(0+0.5\cdot \max\left[Q(2,-1),Q(2,1)\right]\right) \\
&+ 0.15\cdot\left(0+0.5\cdot \max\left[Q(3,-1),Q(3,1)\right]\right) \\
&+ 0.05\cdot\left(0+0.5\cdot \max\left[Q(4,-1),Q(4,1)\right]\right) \\
&= 0.8\cdot 0.5\cdot \max\left[0.32,0.044\right]+0+0 \\
&= 0.128
\end{aligned} \tag{3.64}
$$

$$
\begin{aligned}
Q(3,1) &= \sum_{s'} P_{3s'}^{1}\left(\rho(3,1,s')+\gamma \max_{a'} Q\left(s',a'\right)\right) \\
&= \sum_{j=2}^{4} P_{3j}^{1}\left(\rho(3,1,j)+\gamma \max_{a'} Q\left(j,a'\right)\right) \\
&= P_{32}^{1}\left(\rho(3,1,2)+\gamma \max_{a'} Q\left(2,a'\right)\right) \\
&+ P_{33}^{1}\left(\rho(3,1,3)+\gamma \max_{a'} Q\left(3,a'\right)\right) \\
&+ P_{34}^{1}\left(\rho(3,1,4)+\gamma \max_{a'} Q\left(4,a'\right)\right) \\
&= 0.05\cdot\left(0+0.5\cdot \max\left[Q(2,-1),Q(2,1)\right]\right) \\
&+ 0.15\cdot\left(0+0.5\cdot \max\left[Q(3,-1),Q(3,1)\right]\right) \\
&+ 0.8\cdot\left(0+0.5\cdot \max\left[Q(4,-1),Q(4,1)\right]\right) \\
&= 0.05\cdot 0.5\cdot \max\left[0.32,0.044\right]+0.15\cdot 0.5\cdot \max\left[0.128,0\right]+0 \\
&= 0.018
\end{aligned} \tag{3.65}
$$

$$
\begin{aligned}
Q(4,-1) &= \sum_{s'} P_{4s'}^{-1}\left(\rho(4,-1,s')+\gamma \max_{a'} Q\left(s',a'\right)\right) \\
&= \sum_{j=3}^{5} P_{4j}^{-1}\left(\rho(4,-1,j)+\gamma \max_{a'} Q\left(j,a'\right)\right) \\
&= P_{43}^{-1}\left(\rho(4,-1,3)+\gamma \max_{a'} Q\left(3,a'\right)\right) \\
&+ P_{44}^{-1}\left(\rho(4,-1,4)+\gamma \max_{a'} Q\left(4,a'\right)\right) \\
&+ P_{45}^{-1}\left(\rho(4,-1,5)+\gamma \max_{a'} Q\left(5,a'\right)\right) \\
&= 0.8\cdot\left(0+0.5\cdot \max\left[Q(3,-1),Q(3,1)\right]\right) \\
&+ 0.15\cdot\left(0+0.5\cdot \max\left[Q(4,-1),Q(4,1)\right]\right) \\
&+ 0.05\cdot\left(5+0.5\cdot \max\left[Q(5,-1),Q(5,1)\right]\right) \\
&= 0.8\cdot 0.5\cdot \max\left[0.128,0.018\right]+0+0.05\cdot 5 \\
&= 0.301
\end{aligned} \tag{3.66}
$$

$$
\begin{aligned}
Q(4,1) &= \sum_{s'} P_{4s'}^{1}\left(\rho(4,1,s')+\gamma \max_{a'} Q\left(s',a'\right)\right) \\
&= \sum_{j=3}^{5} P_{4j}^{1}\left(\rho(4,1,j)+\gamma \max_{a'} Q\left(j,a'\right)\right) \\
&= P_{43}^{1}\left(\rho(3,1,3)+\gamma \max_{a'} Q\left(3,a'\right)\right) \\
&+ P_{44}^{1}\left(\rho(4,1,4)+\gamma \max_{a'} Q\left(4,a'\right)\right) \\
&+ P_{45}^{1}\left(\rho(4,1,5)+\gamma \max_{a'} Q\left(5,a'\right)\right) \\
&= 0.05\cdot\left(0+0.5\cdot \max\left[Q(3,-1),Q(3,1)\right]\right) \\
&+ 0.15\cdot\left(0+0.5\cdot \max\left[Q(4,-1),Q(4,1)\right]\right) \\
&+ 0.8\cdot\left(5+0.5\cdot \max\left[Q(5,-1),Q(5,1)\right]\right) \\
&= 0.05\cdot 0.5\cdot \max\left[0.128,0.018\right]+0.15\cdot 0.5\cdot \max\left[0.301,0\right]+0.8\cdot 5 \\
&= 4.026
\end{aligned} \tag{3.67}
$$

$$\begin{aligned} Q(5,-1) &= \sum_{s'} P_{5s'}^{-1}\left(\rho(5,-1,s')+\gamma \max_{a'} Q\left(s',a'\right)\right) \\ &= \sum_{j=5}^{5} P_{5j}^{-1}\left(\rho(5,-1,j)+\gamma \max_{a'} Q\left(j,a'\right)\right) \\ &= P_{55}^{-1}\left(\rho(5,-1,5)+\gamma \max_{a'} Q\left(5,a'\right)\right) \\ &= P_{55}^{-1}\left(\rho(5,-1,j)+0.5\max\left[Q(5,-1),Q(5,1)\right]\right) \\ &= 1\cdot\left(0+0.5\max\left[0,0\right]\right) \\ &= 0 \end{aligned} \tag{3.68}$$

$$\begin{aligned} Q(5,1) &= \sum_{s'} P_{5s'}^{1}\left(\rho(5,1,s')+\gamma \max_{a'} Q\left(s',a'\right)\right) \\ &= \sum_{j=5}^{5} P_{5j}^{1}\left(\rho(5,1,j)+\gamma \max_{a'} Q\left(j,a'\right)\right) \\ &= P_{55}^{1}\left(\rho(5,1,5)+\gamma \max_{a'} Q\left(5,a'\right)\right) \\ &= P_{55}^{1}\left(\rho(5,1,j)+0.5\max\left[Q(5,-1),Q(5,1)\right]\right) \\ &= 1\cdot\left(0+0.5\max\left[0,0\right]\right) \\ &= 0 \end{aligned} \tag{3.69}$$

Such iteration repeats until the values of $Q(s,a)$ converge to the optimal values after 22 iterations (i.e., $Q^* = Q_{22}$), which are shown in Table 3.5. An optimal policy is: $\pi_*(1) = -1$, $\pi_*(2) = 1$, $\pi_*(3) = 1$, and $\pi_*(4) = 1$, which happens to be identical to that for Example 3.2.2 as shown in Figure 3.7.

Table 3.5: Values of $Q(s,a)$ for non-deterministic state transitions.

State	0	1	2	3	4	5
Q_0	0.000|0.000	0.000|0.000	0.000|0.000	0.000|0.000	0.000|0.000	0.000|0.000
Q_1	0.000|0.000	0.800|0.110	0.320|0.044	0.128|0.018	0.301|4.026	0.000|0.000
Q_2	0.000|0.000	8.868|0.243	0.374|0.101	0.260|1.639	1.208|4.343	0.000|0.000
Q_3	0.000|0.000	0.874|0.265	0.419|0.709	0.515|1.878	1.327|4.373	0.000|0.000
Q_4	0.000|0.000	0.883|0.400	0.453|0.826	0.581|1.911	1.342|4.376	0.000|0.000
...	...	...	...	...	...	...
Q_{12}	0.000|0.000	0.888|0.458	0.467|0.852	0.594|1.915	1.344|4.376	0.000|0.000
...	...	...	...	...	...	...
Q_{22}	0.000|0.000	0.888|0.458	0.467|0.852	0.594|1.915	1.344|4.376	0.000|0.000

□

Note that in both examples no action is selected for either state 0 or 5, since the

reinforcement learning task is considered terminated at those states.

3.2.3 Convergence of value iteration

We now show that the value iteration algorithm as given in Table 3.2 converges to $q_*(s,a)$ as the number of iteration k increases. The main argument to support the claim that this algorithm converges is the same as that given for the case of the policy evaluation algorithm (as presented in Section 3.1.1). To strengthen the argument for this case, we add another step to first show that the values of $Q(s,a)$—which are the intermediate estimates of $q_*(s,a)$—are bounded from one iteration to the next[3].

So we will investigate the convergence of the value iteration algorithm given in Table 3.2 in two steps. We will first show that values of $Q(s,a)$ are bounded from one iteration to the next. We will then show that, under the update rule as given in Equation (3.26), $Q_k(s,a)$ approaches $q_*(s,a)$ monotonically as k increases, as is illustrated in Figure 3.8.

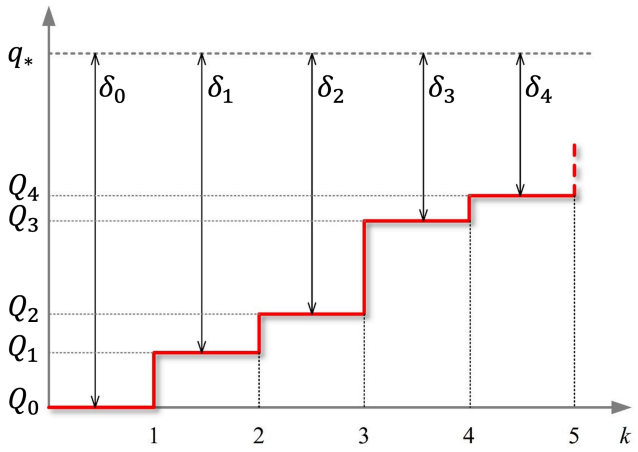


Figure 3.8: Convergence of value iteration algorithm.

For the sake of simplicity, we assume that (i) the reward values are all positive, and (ii) $Q_0(s,a) = 0$ for all state-action pairs. We know that for a finite Markov Decision Process $q_*(s,a)$ are finite.

Step 1*:* Suppose that at iteration k, the action value $Q_k(s,a)$ for some state s and action a is not optimal, i.e., $Q_k(s,a) \neq q_*(s,a)$. We can express the maximum of the difference between $Q_k(s,a)$ and $q_*(s,a)$ as

$$\delta_k \triangleq \max_{s,a} |q_*(s,a) - Q_k(s,a)| \tag{3.70}$$

[3]This step was not included Section 3.1.1 in order to keep the discussion focused on the main issue of convergence at that point. The reader may add this step to the discussion on the convergence of the policy evaluation algorithm as an exercise.

Removing the maximum operator yields

$$|q_*(s,a) - Q_k(s,a)| \leq \delta_k$$

From this inequality, we have

$$\begin{aligned} & -\delta_k \leq q_*(s,a) - Q_k(s,a) \leq \delta_k \\ \Rightarrow \quad & -q_*(s,a) - \delta_k \leq -Q_k(s,a) \leq -q_*(s,a) + \delta_k \\ \Rightarrow \quad & q_*(s,a) + \delta_k \geq Q_k(s,a) \geq q_*(s,a) - \delta_k \\ \Rightarrow \quad & q_*(s,a) - \delta_k \leq Q_k(s,a) \leq q_*(s,a) + \delta_k \end{aligned} \tag{3.71}$$

Now consider the iteration $(k+1)$. The update rule is:

$$Q_{k+1}(s,a) = \sum_{s',r} p(s',r \mid s,a)\Big[r + \gamma \max_{a'} Q_k(s',a')\Big] \tag{3.72}$$

Since Equation (3.71) holds for any given state-action pair (s',a'), we have

$$Q_k(s',a') \leq q_*(s',a') + \delta_k \tag{3.73}$$

With Equations (3.72) and (3.73), we have

$$\begin{aligned} Q_{k+1}(s,a) &= \sum_{s',r} p(s',r \mid s,a)\Big[r + \gamma \max_{a'} Q_k(s',a')\Big] \\ &\leq \sum_{s',r} p(s',r \mid s,a)\Big[r + \gamma \max_{a'} \left(q_*(s',a') + \delta_k\right)\Big] \\ &= \sum_{s',r} p(s',r \mid s,a)\Big(r + \gamma \max_{a'} q_*\left(s',a'\right) + \gamma\delta_k\Big) \\ &= \underbrace{\sum_{s',r} p(s',r \mid s,a)\Big(r + \gamma \max_{a'} q_*\left(s',a'\right)\Big)}_{\text{This is } q_*(s,a)} + \gamma\delta_k \\ &= q_*(s,a) + \gamma\delta_k \end{aligned} \tag{3.74}$$

that is,

$$Q_{k+1}(s,a) \leq q_*(s,a) + \gamma\delta_k \tag{3.75}$$

Since $q_*(s,a)$ is finite, Equation (3.75) indicates that $Q_{k+1}(s,a)$ is bounded if δ_k is bounded. Since initially $Q_0(s,a)$ is set to zero, and δ_k is the maximum absolute difference between $Q_k(s,a)$ and $q_*(s,a)$ in each iteration, we can conclude that Q_{k+1} is also bounded.

Step 2: We will next show that $Q_k(s,a)$ approaches $q_*(s,a)$ monotonically as k increases,

as is illustrated in Figure 3.8. From Equation (3.71), we have

$$Q_k(s', a') \geq q_*(s', a') - \delta_k$$

So

$$\begin{aligned} Q_{k+1}(s,a) &= \sum_{s',r} p(s', r \mid s, a)\Big(r + \gamma \max_{a'} Q_k(s', a')\Big) \\ &\geq \sum_{s',r} p(s', r \mid s, a)\Big(r + \gamma \max_{a'} \Big(q_*(s', a') - \delta_k\Big)\Big) \\ &= \underbrace{\sum_{s',r} p(s', r \mid s, a)\Big(r + \gamma \max_{a'} q_*(s', a')\Big)}_{\text{This is } q_*(s,a)} - \gamma\delta_k \\ &= q_*(s,a) - \gamma\delta_k \end{aligned} \tag{3.76}$$

that is,

$$q_*(s,a) - \gamma\delta_k \leq Q_{k+1}(s,a)$$

or

$$q_*(s,a) - Q_{k+1}(s,a) \leq \gamma\delta_k$$

Therefore,

$$\delta_{k+1} \triangleq \max_{s,a} |q_*(s,a) - Q_{k+1}(s,a)| \leq \max_{s,a} |\gamma\,\delta_k| = \gamma\,\delta_k$$

i.e.,

$$\delta_{k+1} \leq \gamma\,\delta_k$$

If we set $0 \leq \gamma < 1$, then $\delta_k \to 0$, and consequently $Q_k(s,a) \to q_*(s,a)$, as $k \to \infty$. In practice, the iteration can be stopped when δ_k drops below a prescribed tolerance. □

3.3 Generalized Policy Iteration

Recall that when presenting the update rule for value iteration, i.e., Equation(3.25), it was pointed out that value iteration can be considered as equivalent to policy iteration but with policy evaluation truncated after one iteration. This view leads to the unified scheme called *generalized policy iteration*. Under this scheme, a reinforcement learning algorithm is considered to be an implementation of two interacting sub-processes, namely, policy evaluation and policy improvement; the specific form of the implementation (i.e., whether policy evaluation is truncated or not) determines how the algorithm is further classified (i.e., as policy iteration or value iteration). At the "unified level", the algorithm just implements the scheme of generalized policy iteration.

Dynamic Programming methods require a complete model of the environment represented by the function $p = (s', r \mid s, a)$. When this p-function is not available, the methods of policy iteration and value iteration are not applicable in solving a reinforcement learning problem. In such situations, we turn to a class of so-called model-free approaches that (still operating under the scheme of generalized policy iteration) enables us to find an optimal policy without a model.

Part B

Model-free Approaches

Chapter 4

Monte-Carlo Methods

Monte Carlo methods in reinforcement learning are based on the general *Monte Carlo technique* of finding approximate solutions to mathematical problems using 'experience' gained by random sampling processes. In this module, we will use the term "Monte Carlo technique" to refer to the general mathematical technique, and use the term "Monte Carlo methods" to refer to the methods (based on the Monte Carlo technique) that had been developed in the context of reinforcement learning.

Monte Carlo technique

Monte Carlo methods

In the absence of prior knowledge about the transition probabilities and the reward function, Monte Carol methods rely on sampled data obtained from direct *experience* to estimate the state or action values, from which an optimal policy can be obtained. By experience we mean that in these methods the agent executes randomly chosen episodes (i.e., sequences of transitions that end at a terminal state) and estimates the state or action values based on the mean return calculated from the sampled data.

Monet Carlo methods and episodic tasks

The operational characteristics of the set of Monte Carlo methods discussed in this chapter can be interpreted in term of scheme of generalized policy iteration. However, the terminologies used here are slightly different from that used in the context of Dynamic Programming. The process of policy evaluation is now called *Monet Carlo prediction*, and the process of policy iteration is now called *Monte Carlo control*. To put it simply, in Monte Carlo prediction we aim to estimate the state or action values of a given policy, while in Monte Carlo control we aim to find an optimal policy.

terminologies

Before we discuss these specific methods, we briefly introduce the Monte Carlo technique through a simple example.

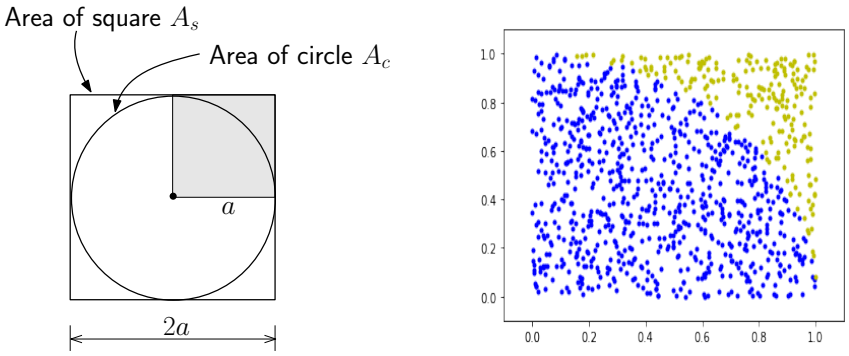
4.1 The Monte Carlo Technique

The Monte Carlo technique uses random sampling to estimate numerical results. For example, to estimate the probability of getting a 6 from throwing a dice, we can throw a dice many times (say, n), count the number of times (m) when the dice shows 6, and estimate this probability as: m/n. In this example, throwing the dice is the 'experience', while the m and n are the sampled data obtained from these experience. Such experience does not have to be physical; they can be events simulated in a computer, for instance. The following example involves a setting more elaborate than that for the case of throwing a dice, but it illustrates the same basic principle of estimating some entity via random sampling.

Suppose that we would like to find the value of π (the mathematical constant of $3.1416\ldots$, not a policy) based on data generated by some experience. One way to start is by recognizing the ratio between the area A_c of a circle with radius a and the area A_s of the smallest square that encases the circle (as is illustrated in Figure 4.1(a)), i.e.,

$$\frac{A_s}{A_c} = \frac{(2a)^2}{\pi a^2} = \frac{4}{\pi} \tag{4.1}$$

This ratio also applies to the shaded portions of the circle and the square. Suppose that we make a dartboard in the shape of the shaded area, and start throwing darts at it in a random manner, i.e., without purposely aiming at any particular spot in that shaded area. After a large number of throws (say, n), we count the number of hits (m) inside the quadrant to estimate the value $\pi \approx 4\,m/n$.



(a) (b)

Figure 4.1: (a) A circle encased by the smallest square. (b) Simulation of $1,000$ random points inside the shaded square with 790 points inside the quadrant, yielding $\pi = 790/1000 = 3.16$.

We can run computer simulations to generate random points representing the hits. As the number of such random points increases, the estimate for the value of π becomes more

accurate. Figure 4.1(b) shows one such simulation with 1000 points generated, for which the estimated value for π is 3.16.

4.2 Monte Carlo Prediction

We first look at the the problem of estimating the state value of a given policy. Here is the situation:

1. We have a set of state $\mathcal{S}$ and a set of action $\mathcal{A}$.
2. We do not have prior knowledge about the transition function or the reward function.
3. The agent at state s takes an action in $\mathcal{A}(s)$ to possibly move around in the state space.
4. Each time the agent takes an action, it receives a reward only after it has taken the action.

Now suppose that we are given a policy π, how do we estimate the state value v_π of this policy? Monte Carlo prediction is about answering this question.

In Monte Carlo prediction, the general idea is to let the agent move around in episodes by following the given policy π while collecting the rewards along the way. We consider an episode to consist of a sequence of steps, with each step involving a state, an action, and a reward. When the agent at the state S_t take the action A_t and receives a reward R_{t+1}, we refer to this as the t^{th} step, denoted by the triple $\{S_t, A_t, R_{t+1}\}$. So an episode with T steps that starts at the initial state S_0 and terminates at state S_{T-1} can be expressed as

$$\Big\{S_t, A_t, R_{t+1}\Big\}_{t=0}^{T-1} = \{S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T\} \tag{4.2}$$

Figure 4.2 shows four episodes in the treasure-hunting problem which was initially described in Section 1.3.2.

Remark 4.2.1. *If the transitions are probabilistic, the agent may enter a different new state from the same 'departing' state. State* 9 *in Figure 4.2 is an example of such a departing state.* □

Suppose that the robot now receives a reward of -0.04 for each step (except those resulting in the robot entering a terminal state) that it takes. Then Episode 3 (which involves $T = 7$ steps, as depicted in the figure), for instance, can be expressed as

$$\begin{aligned}\Big\{S_t, A_t, R_{t+1}\Big\}_{t=0}^{6} \;=\; &\big\{1, \mathsf{up}, -0.04,\;\; 5, \mathsf{up}, -0.04,\;\; 9, \mathsf{down}, -0.04,\;\; 5, \mathsf{up}, -0.04,\\ &9, \mathsf{right}, -0.04,\;\; 10, \mathsf{right}, -0.04,\;\; 11, \mathsf{right}, +1\big\}\end{aligned} \tag{4.3}$$

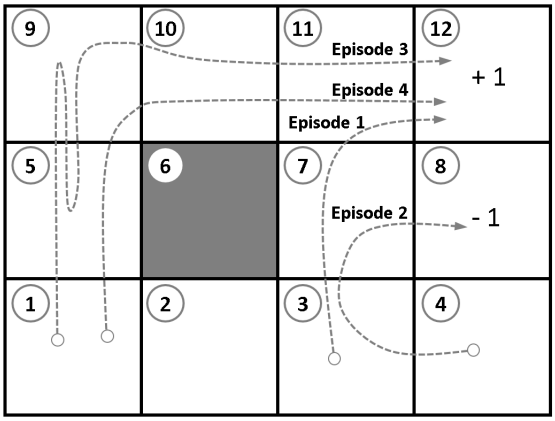


Figure 4.2: Four episodes generated by following some fixed policy.

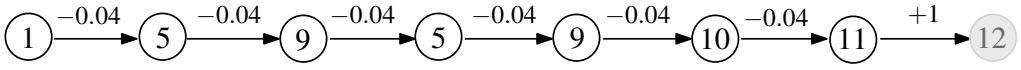
At the end of each episode, we can "look back" and calculate the return G_t for each state $S_t = s$ visited by the agent in the episode, i.e.,

$$G_t = \sum_{k=0}^{T-t-1} \left(\gamma^{\,k} R_{t+k+1} \,\middle|\, S_t = s \right) = R_{t+1} + \gamma\, G_{t+1} \tag{4.4}$$

For example, for state 1 in Episode 3, we have $t = 0$, and the return for state 1 is

$$G_t|_{S_t=1} = \left(\sum_{k=0}^{7-0-1} \gamma^{\,k} R_{0+k+1} \,\middle|\, S_t = 1 \right) = \sum_{k=0}^{6} \gamma^{\,k} R_{k+1} \tag{4.5}$$

as is illustrated in Figure 4.3.



$$G_t\big|_{S_t=1} = \; 0.9^0 \cdot (-0.04) + 0.9^1 \cdot (-0.04) + 0.9^2 \cdot (-0.04) + 0.9^3 \cdot (-0.04) + 0.9^4 \cdot (-0.04) + 0.9^5 \cdot (-0.04) + 0.9^6 \cdot (+1)$$

Figure 4.3: Return starting from state 1.

A more efficient way is to use the relationship between G_t and G_{t+1} as described in Equation (2.36) to calculate G_t for each state S_t, i.e.,

calculating G_t for one episode

$$G_t = \sum_{k=0}^{T-t-1} \left(\gamma^{\,k} R_{t+k+1} \,\middle|\, S_t = s \right) = R_{t+1} + \gamma\, G_{t+1} \tag{4.6}$$

We can start from the last step (i.e., $t = T - 1$) and work backward to $t = 0$. Take Episode 3 again for example, we have $T = 7$, and the value of the terminal state is zero by definition, i.e., $G_7 = 0$. So starting from the last step, i.e., $t = T - 1 = 7 - 1 = 6$, we

have

$$\begin{aligned} G_6\,|_{S_6=11} &= R_7 + \gamma\, G_7 = +1 + 0.9 \times 0 = +1 \\ G_5\,|_{S_5=10} &= R_6 + \gamma\, G_6 = -0.04 + 0.9 \times 1 = 0.86 \\ G_4\,|_{S_4=9} &= R_5 + \gamma\, G_5 = -0.04 + 0.9 \times 0.86 = 0.82 \\ G_3\,|_{S_3=5} &= R_4 + \gamma\, G_4 = -0.04 + 0.9 \times 0.82 = 0.70 \\ G_2\,|_{S_2=9} &= R_3 + \gamma\, G_3 = -0.04 + 0.9 \times 0.70 = 0.59 \\ G_1\,|_{S_1=5} &= R_2 + \gamma\, G_2 = -0.04 + 0.9 \times 0.59 = 0.49 \\ G_0\,|_{S_0=1} &= R_1 + \gamma\, G_1 = -0.04 + 0.9 \times 0.49 = 0.40 \end{aligned} \tag{4.7}$$

Note that a state s may be visited by the agent multiple times during the same episode, as in the case of states 5 and 9 in Episode 3 illustrated in Figure 4.2. Each visit to s is recorded by the occurrence of $S_t = s$ in an episode $\{S_t, A_t, R_{t+1}\}_{t=0}^{T-1}$. For each visit, there will be one state value calculated. For example, the values of state 5 for the two visits in Episode 3 (as illustrated in Figure 4.2) are: $G_1 = 0.49$ and $G_3 = 0.70$. If we use a list g_s^j to hold the G_t values for state $s = 5$ during episode $j = 3$, then we can write $g_5^3 = [0.49, 0.70]$. We also use $g_s^j(i)$ to denote the i^{th} element of g_s^j (as illustrated in Figure ?); for instance, $g_5^3(2) = 0.70$. For such lists, we define the append operation $[\![(\cdot),(\cdot)]\!]$ as

list notation and *append* operation

$$[\![g_1, g_2]\!] = [a, b] \quad \text{if} \quad g_1 = [a] \quad \text{and} \quad g_2 = [b] \tag{4.8}$$

and denote the number of element in a list g_s^j by $|g_s^j|$. Let [ ] denote the empty list.

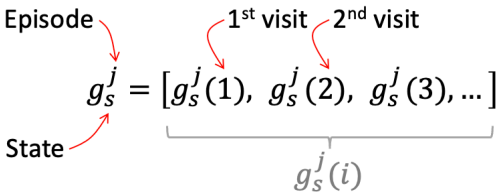


Figure 4.4: List for holding G_t.

Suppose that we generate m episodes (by following the given policy π) and do the same state value calculations for all m episodes to obtain m lists[1] of G_t values for s, i.e., $g_s^1, g_s^2, \ldots, g_s^m$. We then take the average of all the G_t values for s (generated from the m episodes) to obtain the estimated state value $V_\pi(s)$, also called the *empirical mean return*, i.e.,

$$V_\pi(s) = \frac{\text{total return starting from } s \text{ and following } \pi \text{ over } m \text{ episodes}}{\text{number of visits to } s \text{ over } m \text{ episodes}} = \frac{\sum_{i=1}^{|g_s|} g_s(i)}{|g_s|} \tag{4.9}$$

[1]Some of them may be empty.

where $g_s = [\![g_s^1, g_s^2, \ldots, g_s^m]\!]$. By the *law of large numbers*, we have $V_\pi(s) \to v_\pi(s)$ as $m \to \infty$. This in essence is how Monte Carlo prediction works.

every-visit

Equation (4.9) involves the total number of visits to state s over m episodes. This way of calculating $V_\pi(s)$ is called *every-visit* Monte Carlo prediction. There is another version called *first-visit* Monte Carlo prediction, where only the first visit to state s in an episode is considered in the calculation of $V_\pi(s)$. So for this version the value of n in Equation (4.9) is the total number of first visits to s in all m episodes; that is, each of the individual lists $g_s^1, g_s^2, \ldots, g_s^m$ (if not empty) will contain only one element, which is the value of G_t for state s, calculated when the agent visits the state s for the very first time in the corresponding episode. It can be shown [20] that both versions result in the convergence of V_π to v_π. Their difference is subtle and not of our main concern here. In this module, it suffices to consider just the first visit version.

first-visit

Table 4.1 shows the procedure for implementing the first-visit Monte Carlo prediction. It uses the list Returns to hold the G_t values of all the states. This list is updated in each step.

Table 4.1: First-visit Monte Carlo prediction.

Input: ∘ π (the policy to be evaluated)
Initialize: ∘ $V(s) \in \mathbb{R}$ to any arbitrary values, for all $s \in \mathcal{S}$
∘ Returns(s) as an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
  Generate an episode with T steps following π:
    $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless S_t appears in state set of episode $\{S_0, S_1, \ldots, S_{t-1}\}$
      Append G to Return(S_t)
      $V(S_t) \leftarrow$ average(Return(S_t))

Remark 4.2.2. *An important point to note here is that the estimates of the state values V_π can be updated only* after the completion of an episode. *Such a method is often characterized as operating in an* off-line *mode. In the case of the treasure-hunting problem, this means that when the robot is carrying out Episode 4, we must wait for the robot to fall into the deathtrap and only then can we update the state values. This makes the Monte Carlo methods in general less efficient than other methods such as Temporal Difference and Q-learning.* □

Monte Carlo methods are off-line methods

Example 4.2.1. Determine the values of V_π after completing Episodes 1 and 2 (as shown in Figure 4.2) of an every-visit Monte Carlo prediction process.

Solution: We see that no state is visited more than once in either episode. So in this case the calculation of V under consideration of every-visit and first-visit are equivalent.

Initially we set $V(s) = 0$ for all $s \in \mathcal{S}$, i.e.,

$$V = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.10}$$

Episode 1 is described by the following state-action-reward sequence:

$$\begin{array}{lll} S_0 = 3 & A_0 = \mathsf{up} & R_1 = -0.04 \\ S_1 = 7 & A_1 = \mathsf{up} & R_2 = -0.04 \\ S_2 = 11 & A_2 = \mathsf{right} & R_3 = +1 \end{array} \tag{4.11}$$

So we have $T = 3$. We first apply Equation (4.6) to calculate the return G_t for each step t of the episode, starting from the last step where $t = T - 1 = 3 - 1 = 2$, i.e.,

$$\begin{aligned} G_2 &= \sum_{k=0}^{T-t-1} \gamma^{\,k} R_{t+k+1} = \sum_{k=0}^{3-2-1} \gamma^{\,k} R_{2+k+1} = R_3 = +1 && \text{(at state } S_2 = 11) \\ G_1 &= \sum_{k=0}^{3-1-1} \gamma^{\,k} R_{1+k+1} = R_2 + \gamma \underbrace{R_3}_{G_2} = R_2 + \gamma\, G_2 \\ &= -0.04 + 0.9 \times (+1) = 0.86 && \text{(at state } S_1 = 7) \\ G_0 &= \sum_{k=0}^{3-0-1} \gamma^{\,k} R_{0+k+1} = R_1 + \gamma\, R_2 + \gamma^{\,2} R_3 = R_1 + \gamma\,(R_2 + \gamma \underbrace{R_3}_{G_2}) \\ &= R_1 + \gamma\,(\underbrace{R_2 + \gamma G_2}_{G_1}) = R_1 + \gamma G_1 \\ &= -0.04 + 0.9 \times 0.86 = 0.73 && \text{(at state } S_0 = 3) \end{aligned} \tag{4.12}$$

At the end of Episode 1, we have $g_3^1 = [0.73] = g_3$, $g_7^1 = [0.86] = g_7$, $g_{11}^1 = [1] = g_{11}$, and $g_s^1 = [\,]$ for all other states; and the V values are:

$$V = \begin{bmatrix} 0 & 0 & 0.73 & 0 & 0 & 0 & -0.86 & 0 & 0 & 0 & 1.00 & 0 \end{bmatrix} \tag{4.13}$$

Episode 2:

$$\begin{array}{lll} S_0 = 4 & A_0 = \mathsf{left} & R_1 = -0.04 \\ S_1 = 3 & A_1 = \mathsf{up} & R_2 = -0.04 \\ S_2 = 7 & A_2 = \mathsf{right} & R_3 = -1 \end{array} \tag{4.14}$$

So $T = 3$, and

$$\begin{aligned} G_2 &= R_3 + \gamma G_3 = -1 + 0.9 \times 0 = -1 && \text{(at state } S_2 = 7) \\ G_1 &= R_2 + \gamma\, G_2 = -0.04 + 0.9 \times (-1) = -0.90 && \text{(at state } S_1 = 3) \\ G_0 &= R_1 + \gamma\, G_1 = -0.04 + 0.9 \times (-0.9) = -0.85 && \text{(at state } S_0 = 4) \end{aligned} \tag{4.15}$$

At the end of Episode 2, we have $g_3^2 = [-0.90]$, $g_4^2 = [-0.85]$, $g_7^2 = [-1]$, and $g_s^2 = 0$ for all other states.

Since states 3 and 7 both appear in Episodes 1 and 2, we first append their G_t value lists, i.e.,

$$g_3 = [\![g_3^1,\ g_3^2]\!] = [0.73,\ -0.90] \tag{4.16}$$
$$g_7 = [\![g_7^1,\ g_7^2]\!] = [0.86,\ -1] \tag{4.17}$$

Consequently,

$$V(3) = \frac{\sum_{i=1}^{2} g_3(i)}{|g_3|} = \frac{0.73 - 0.90}{2} = -0.09 \tag{4.18}$$
$$V(7) = \frac{\sum_{i=1}^{2} g_7(i)}{|g_7|} = \frac{0.86 - 1}{2} = -0.07 \tag{4.19}$$

Each of states 4 and 11 was visited only once over the two episodes, so no averaging needs to be done for those states. Thus, after completing Episodes 1 and 2 the V values are:

$$V = \begin{bmatrix} 0 & 0 & -0.09 & -0.85 & 0 & 0 & -0.07 & 0 & 0 & 0 & 1.00 & 0 \end{bmatrix} \tag{4.20}$$

□

Remark 4.2.3. *If we just want to find the state value of a specific state with respect to a given policy, we can simply generate randomly many episodes starting from that state and following that policy. A good estimate of the value of that state with respect the given policy can thus be obtained after running a large number of such episodes.* □

4.3 Monte Carlo Control

Monte Carlo control is a model-free method for finding an optimal policy by random sampling. Similar to the method of policy iteration in Dynamic Programming, Monte Carl control consists of two interacting sub-processes, namely,

Monte Carlo control

1. Monte Carlo prediction, which corresponds to the sub-process of policy evaluation in policy iteration, and
2. Policy improvement.

Recall that in model-based policy iteration, we start with an initial (random) policy π_0 and conduct policy evaluation on the state values to obtain $v_{\pi_0}(s)$. We then conduct policy improvement, by modifying π_0 to make it "greedy" with respect to $v_{\pi_0}(s)$, in order to generate a new policy $\pi_1 \geq \pi_0$. We repeat this evaluation-improvement cycle until a policy

cannot be improved, at which point we have obtained an optimal policy π_*, as is illustrated in Figure 3.1 — which is shown again below (i.e., Figure 4.5). It is to be emphasized here that during policy improvement, to select the "greedy action" we use Equation (3.20) which involves the p-function $p(s', r \mid s, a)$ that models the dynamics of the agent-environment interaction. In the model-based setting, this function is complete known. In the model-free setting, however, we do not have such a model. Therefore, even though we can conduct Monte Carlo prediction to obtain v_{π_i} (as discussed earlier in Section 4.2), we will not be able to improve a policy π_i by modifying it to make it greedy with respect to v_{π_i}, simply because we do not know $p(s', r \mid s, a)$.

why knowing v_π is not enough

$$\pi_0 \xrightarrow{\text{PE}} v_{\pi_0} \xrightarrow{\text{PI}} \pi_1 \xrightarrow{\text{PE}} v_{\pi_1} \xrightarrow{\text{PI}} \pi_2 \xrightarrow{\text{PE}} \cdots \xrightarrow{\text{PI}} \pi_* \xrightarrow{\text{PE}} v_*$$

Figure 4.5: The policy iteration process: PE = policy evaluation; PI = policy improvement.

An obvious approach to deal with the lack of a model (in the form of the p-function) is to estimate the transition probabilities using the Monte Carlo technique. This is a valid approach. However, doing so would involve estimating *all* the transition probabilities for a given environment. For reinforcement learning problems with a large number of states and actions, this approach can quickly become computationally intractable.

why not just estimate p?

A more efficient way (in policy iteration) to compensate for the lack of a model is to conduct Monte Carlo prediction on the action values $q_{\pi_k}(s, a)$ for a given policy π_k, i.e., the expected return when the agent takes action a at state s and thereafter following policy π_k. Since $q_{\pi_k}(s, a)$ contains the values of all the actions $\mathcal{A}(s)$ at state s, we can generate a new policy π_{k+1} by modifying π_k to make it greedy with respect to the action values for π_k; that is, we can just choose an action with the maximum action value $q_{\pi_k}(s, a)$ at s, i.e.,

$$\pi_{k+1}(s) = \arg\max_a q_{\pi_k}(s, a) \tag{4.21}$$

We follow similar arguments presented in Section 3.1.2 to show that π_{k+1} thus obtained is as good as or better than π_k, i.e., $\pi_{k+1} \geq \pi_k$. To do that we invoke the Policy Improvement Theorem (with deterministic policies) which states that[2]

$$\begin{array}{ll} \text{IF} & \pi_{k+1} \text{ satisfies } q_{\pi_k}\big(s, \pi_{k+1}(s)\big) \geq v_{\pi_k}(s) \\ \text{THEN} & v_{\pi_{k+1}}(s) \geq v_{\pi_k}(s) \end{array} \tag{4.22}$$

It suffices here for us to show that the IF clause in Equation (4.22) is true in Monte Carlo

[2]The proof is shown in Section 3.1.2, specifically see Equation (3.19).

policy improvement. Now

$$\begin{aligned} q_{\pi_k}\big(s,\, \pi_{k+1}(s)\big) &= q_{\pi_k}\Big(s,\, \arg\max_a q_{\pi_k}(s,a)\Big) \\ &= \max_a q_{\pi_k}(s,a) \\ &\geq q_{\pi_k}(s, \pi_k(s)) \\ &= v_{\pi_k}(s) \qquad \text{(from Equation (2.55))} \end{aligned} \tag{4.23}$$

Hence, we can conclude that by choosing, according to Equation (4.21), a new policy π_{k+1} that is greedy with respect to the value $q_{\pi_k}(s,a)$ of the current policy π, it can be ensured that $v_{k+1}(s) \geq v_{\pi_k}(s)$, which implies that $\pi_{k+1} \geq \pi_k$.

4.3.1 Monte Carlo control with exploring starts

We now turn to the question of how to estimate $q_\pi(s,a)$ for a given policy π. The procedure for Monte Carlo prediction to estimate $v_\pi(s)$, as discussed in Section 4.2, can be readily applied here to estimate $q_\pi(s,a)$, except that the consideration on the "visits to state s" is now replaced by the consideration on the "visits to the state-action pair (s,a)". Specifically, a state-action pair (s,a) is said to have been visited if the agent visits state s and takes action a while in s, with the notions of first-visit and every-visit similarly applied.

In the case of using Monte Carlo prediction to estimate v_π discussed earlier in Section 4.2, the argument was that, if the number of episodes goes to infinity, each state will be visited an infinite number of times; consequently, the empirical mean return $V_\pi(s)$ for each state will converge to $v_\pi(s)$. In the case of estimating $q_\pi(s,a)$, the argument is similar; that is, for the empirical mean return of $Q_\pi(s,a)$ to converge to $q_\pi(s,a)$, it is required that all the state-action pairs be visited.

One way to satisfy the requirement of the agent visiting all state-action pairs is to assume that every state-action pair has a non-zero probability of being selected at the start of an episode, i.e., as (S_0, A_0). We refer to this as the *exploring-starts assumption*. Under this assumption, it is guaranteed that all state-action pairs will be visited an infinitely number of times when the number of episodes goes to infinity. This is called *Monte Carlo control with exploring starts*, or simply, *Monte Carlo ES*.

exploring-starts assumption

Table 4.2 shows the procedure for implementing an algorithm for Monte Carlo ES.

Table 4.2: First-visit Monte Carlo ES for estimating $\pi \approx \pi_*$.

Initialize:
- $\pi(s) \in \mathcal{A}(s)$, arbitrarily, for all $s \in \mathcal{S}$
- $Q(s,a) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
- $\mathsf{Returns}(s,a)$ as an empty list, for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Loop forever (for each episode):
  Select $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ randomly so that every (s,a) pair has non-zero probability of being selected
  Generate an episode with T steps starting from (S_0, A_0) following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless pair (S_t, A_t) appears in $S_0, A_0, S_1, A_1, \ldots, S_{t-1}, A_{t-1}$:
      Append G to $\mathsf{Return}(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow \mathsf{average}(\mathsf{Return}(S_t, A_t))$
      $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

4.3.2 Monte Carlo control without exploring starts

Monte Carlo ES assumes that the agent can start an episode from any state. This is an unrealistic assumption. For example, in a robotics competition that requires a robot to navigate through a maze from a *given starting location*, this assumption cannot be met.

To get around this assumption, we can try to make the agent visit various state-action pairs during an episode. Specifically, we can assign, for every state s and every action a available at s, a non-zero probability that the state-action pair (s,a) will be visited by the agent during an episode, i.e.,

$$\pi(a \mid s) > 0\,, \qquad \text{for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}(s) \tag{4.24}$$

Definition 4.3.1. (*ϵ-soft policy):* Consider a set of states $\mathcal{S}$ and the set of actions $\mathcal{A}(s)$ available at each state $s \in \mathcal{S}$. Let $\epsilon \in \mathbb{R}$ and $0 < \epsilon \leq 1$. A policy π that assigns a probability of at least $\epsilon/|\mathcal{A}(s)|$ to each action in $\mathcal{A}(s)$ at each state s, i.e., ϵ-soft policy

$$\pi(a|s) \geq \frac{\epsilon}{|\mathcal{A}(s)|}\,, \quad 0 < \epsilon \leq 1 \tag{4.25}$$

is called an ϵ-soft policy. □

We now summarize what we have discussed up to this point in order to provide a clear context for the discussion to follow hereafter. Recall that Monte Carlo control involves two sub-processes. The first is *Monte Carlo prediction*, where we estimate $q_\pi(s,a)$ for a given policy π. The second is *policy improvement*, where we try to find a new policy $\pi' \geq \pi$

by making π greedy with respect to $q_\pi(s,a)$. Monte Carlo ES is one method under the category of Monte Carlo control, so it also has these two sub-processes. To use Monte Carlo ES, we need the *exploring-starts assumption* when estimating $q_\pi(s,a)$ in the first sub-process, i.e., Monte Carlo prediction.

Monte Carlo control without exploring starts

Adopting an ϵ-soft policy enables us to get around the difficulty of meeting the exploring-starts assumption when estimating $q_\pi(s,a)$. This is called *Monet Carlo control without exploring starts*. The question now is: How do we conduct policy improvement, now that the "old" policy π is a non-deterministic ϵ-soft policy?

We have shown earlier in this section via Equation (4.23) that a new (deterministic) policy π' that is as good as or better than π can be generated, by making the current (also deterministic) policy π greedy with respect to $q_\pi(s,a)$. If we follow the same approach, i.e., by making the current ϵ-soft policy π greedy with respect to $q_\pi(s,a)$, we face the following two immediate questions:

1. What kind of new policy π' will we get?
2. How do we know that π' will still be as good as or better than the current ϵ-soft policy π?

To answer the first question, we start with the fact that, under a deterministic greedy policy π, the action associated with the largest value of $q_\pi(s,a)$ at a state s is always selected. We call this action the *greedy action*, denoted by A^*, i.e.,

$$A^* \triangleq \operatorname*{argmax}_a q_\pi(s,a) \in \mathcal{A}(s) \tag{4.26}$$

and write $\pi(s) = A^*$, or in the form of a non-deterministic policy, $\pi(A^*|\, s) = 1 = 100\%$. If there are more than one actions associated with the largest value of $q_\pi(s,a)$, only one of them is selected by the agent arbitrarily.

ϵ-greedy policy

Under an ϵ-soft policy, each action at a state s has at least a $\epsilon/|\mathcal{A}(s)|$ probability of being selected. If we want to make an ϵ-soft policy greedy, we would give preference to A^* by giving it a high probability. One way to do this is to assign a probability of $\epsilon/|\mathcal{A}(s)|$ to each of the non-greedy actions at s and assign the remaining probability to A^*.

Example 4.3.1. Show that under an ϵ-greedy policy the probability of the agent selecting the greedy action A^* at s is

$$1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} \tag{4.27}$$

Solution: The total number of actions at s is $|\mathcal{A}(s)|$, of which $(|\mathcal{A}(s)| - 1)$ are non-greedy. To each of these non-greedy actions, we assign the probability of $\epsilon/|\mathcal{A}(s)|$. So

the remaining probability is

$$1 - \frac{\epsilon}{|\mathcal{A}(s)|}\Big(|\mathcal{A}(s)| - 1\Big) = 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} \tag{4.28}$$

which is assigned to the greedy action A^*. □

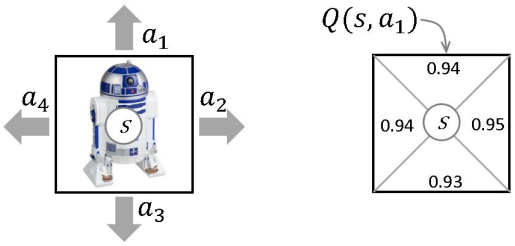
We can now express an ϵ-greedy policy as

$$\pi(a \,|\, s) = \begin{cases} 1 - \epsilon + \dfrac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = A^* \\ \dfrac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a \neq A^* \end{cases} \tag{4.29}$$

By definition, an ϵ-greedy policy is an ϵ-soft policy, i.e., an ϵ-greedy policy belongs to the class of ϵ-soft polices. This answers the first question about the form of the greedy policy we will get when conducting policy improvement.

Example 4.3.2. Consider the situation as illustrated in Figure 4.6. The greedy action is $A^* = a_2$. If $\epsilon \ = 0.1$, then the ϵ-greedy policy for this state is

$$\pi(a \,|\, s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} = 1 - 0.1 + \frac{0.1}{4} = 0.925 \\ \\ \frac{\epsilon}{|\mathcal{A}(s)|} = \frac{0.1}{4} = 0.025 \end{cases} \tag{4.30}$$



(a) (b)

Figure 4.6: (a) Actions and (b) action values at state s.

□

We next look at the second question raised earlier. It is about whether the new policy π', generated by making the current policy π ϵ-greedy with respect to $q_\pi(s, a)$, guarantees that $\pi' \geq \pi$. Here we invoke the *Policy Improvement Theorem*, and show that the ϵ-greedy policy satisfies Equation (3.17), i.e., $q_{\pi'}(s, a)) \geq v_\pi(s)$.

Let π' be an ϵ-greedy policy with respect to $q_\pi(s, a)$. For better readability, we just

write a instead of $a \in \mathcal{A}(s)$ in the summations in the following derivation. Now

$$\begin{aligned} q_{\pi'}(s,a) &= \sum_a \pi'(a \mid s)\, q_\pi(s,a) \\ &= \underbrace{\frac{\epsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a)}_{\text{for all actions, incl. } A^*} + \underbrace{(1-\epsilon)\, \max_a q_\pi(s,a)}_{\text{extra for } A^* \text{ only}} \end{aligned} \tag{4.31}$$

In the last line of Equation (4.31), the probability $\left(1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|}\right)$ of selecting the greedy action A^* is separated into two parts. One part, i.e., $\frac{\epsilon}{|\mathcal{A}(s)|}$, is lumped with the first term for all the non-greedy actions, while the other part, i.e., $(1-\epsilon)$, is specifically assigned to A^* as an extra probability. Before we manipulate Equation (4.31) further, we first establish a proposition about the term $\max_a q_\pi(s,a)$ in that equation.

Proposition 4.3.1. Following the discussion on the derivation of Equation (4.31), show that

$$\max_a q_\pi(s,a) \geq \sum_a \left[\left(\frac{\pi(a|s) - \frac{\epsilon}{|\mathcal{A}(s)|}}{1-\epsilon} \right) q_\pi(s,a) \right] \tag{4.32}$$

Proof: Let w_a be the first term of the summation in Equation (4.32) , i.e.,

$$w_a = \frac{\pi(a|s) - \frac{\epsilon}{|\mathcal{A}(s)|}}{1-\epsilon} \tag{4.33}$$

Then the summation can be expressed as

$$\sum_a \left[\left(\frac{\pi(a|s) - \frac{\epsilon}{|\mathcal{A}(s)|}}{1-\epsilon} \right) q_\pi(s,a) \right] = \sum_a w_a\, q_\pi(s,a) \tag{4.34}$$

which is a weighted sum of $q_\pi(s,a)$, with w_a being the weights.

Now for an ϵ-soft policy π we have (by definition): $\pi(a|s) \geq \epsilon/|\mathcal{A}(s)|$. Hence, $\pi(a|s) - \frac{\epsilon}{|\mathcal{A}(s)|} \geq 0$. Since $(1-\epsilon) \geq 0$, we can conclude that w_a is non-negative (i.e.,

$w_a \geq 0$). Moreover,

$$\begin{aligned}
\sum_a w_a &= \sum_a \left(\frac{\pi(a|s) - \frac{\epsilon}{|\mathcal{A}(s)|}}{1-\epsilon} \right) \\
&= \frac{1}{1-\epsilon} \sum_a \left(\pi(a|s) - \frac{\epsilon}{|\mathcal{A}(s)|} \right) \\
&= \frac{1}{1-\epsilon} \left(\underbrace{\pi(a_1\,|s) + \cdots + \pi(a_{|\mathcal{A}|}\,|s)}_{|\mathcal{A}| \text{ terms sum to } 1} - \underbrace{\left(\frac{\epsilon}{|\mathcal{A}(s)|} + \cdots + \frac{\epsilon}{|\mathcal{A}(s)|} \right)}_{|\mathcal{A}(s)| \text{ terms sum to } \epsilon} \right) \\
&= \frac{1}{1-\epsilon} \Big(1 - \epsilon \Big) \\
&= 1
\end{aligned} \tag{4.35}$$

Now $w_a \geq 0$ and $\sum_a w_a = 1$ imply that $w_a \leq 1$. Since the term $\sum_a w_a \, q_\pi(s,a)$ in Equation (4.34) is a weighted sum of $q_\pi(s,a)$ with the weights w_a summing to 1, this weighted sum cannot be greater[3] than the largest $q_\pi(s,a)$, i.e., $\max_a q_\pi(s,a)$. Therefore,

$$\max_a q_\pi(s,a) \geq \sum_a w_a \, q_\pi(s,a) = \sum_a \left[\left(\frac{\pi(a|s) - \frac{\epsilon}{|\mathcal{A}(s)|}}{1-\epsilon} \right) q_\pi(s,a) \right] \tag{4.36}$$

□

We now continue our derivation from Equation (4.31).

$$\begin{aligned}
q_{\pi'}(s,a) &= \sum_a \pi'(a \,|\, s) \, q_\pi(s,a) \\
&= \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\epsilon) \cdot \underbrace{\max_a q_\pi(s,a)}_{\text{Proposition 4.3.1}} \\
&\geq \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\epsilon) \sum_a \left[\left(\frac{\pi(a|s) - \frac{\epsilon}{|\mathcal{A}(s)|}}{1-\epsilon} \right) q_\pi(s,a) \right] \\
&= \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + \sum_a \pi(a \,|\, s) \, q_\pi(s,a) - \frac{\epsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) \\
&= \sum_a \pi(a \,|\, s) \, q_\pi(s,a) \\
&= v_\pi(s) \qquad \text{(from Equation (2.54))}
\end{aligned} \tag{4.37}$$

Therefore, we can conclude that the new ϵ-greedy policy π' satisfies the condition for

[3]This is akin to saying that the expectation of a set of values for a discrete random variable cannot be greater than the largest value in that set.

the policy improvement theorem, and consequently, $\pi' \geq \pi$. It can be shown that this implementation of *Monte Carlo control without exploring starts* will reach a point where the current police cannot be improved [22], i.e., $\pi' = \pi$, in which case we have an optimal policy $\pi_* = \pi$.

Table 4.3 shows the procedure for implementing an algorithm for Monte Carlo control without exploring starts.

Table 4.3: First-visit Monte Carlo control without ES for estimating $\pi \approx \pi_*$.

Paremeter: A small $\epsilon > 0$

Initialize: An arbitrarily ϵ-soft policy π
- $Q(s,a) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
- $\mathsf{Returns}(s,a)$ as an empty list, for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
- Generate an episode with T steps following π:
  $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
- $G \leftarrow 0$
- Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
  - $G \leftarrow \gamma G + R_{t+1}$
  - Unless pair (S_t, A_t) appears in $S_0, A_0, S_1, A_1, \ldots, S_{t-1}, A_{t-1}$:
    - Append G to $\mathsf{Return}(S_t, A_t)$
    - $Q(S_t, A_t) \leftarrow \text{average}(\mathsf{Return}(S_t, A_t))$
    - $A^* \leftarrow \arg\max_a Q(S_t, a)$
    - For all $a \in \mathcal{A}(S_t)$:

$$\pi(a \mid S_t) \leftarrow \begin{cases} 1 - \epsilon + \dfrac{\epsilon}{|\mathcal{A}(S_t)|} & \text{if } a = A^* \\ \dfrac{\epsilon}{|\mathcal{A}(S_t)|} & \text{if } a \neq A^* \end{cases}$$

In this chapter, we have discussed how to solve a reinforcement learning problem using the Monte Carlo technique. We have presented the main solution approach of Monte Carlo control (with or without the assumption of exploring starts), which requires only information about the states, the actions and the rewards (received after taking an action).

A key advantage of this approach lies in its simplicity (both conceptual and implementational). The main disadvantage of this approach is the fact that the empirical mean return for either the state values $v_\pi(s)$ or the action values $q_\pi(s,a)$ can only be updated after an episode has been completed. In other words, learning under Monte Carlo methods is conducted off-line. This limits both its efficiency and scope of practical application — especially for situations where the agent is expected to carry out a task while still learning how best to do it, i.e., learning is conducted *online*. A class of methods, called *Temporal Difference learning*, aims to overcome this disadvantage so as to enable online learning.

We will discuss Temporal Difference learning in the next chapter.

Chapter 5

Temporal Difference Learning

Unlike Monte Carlo methods that update the state values $V(s)$ or action values $Q(s, a)$ iteratively on a per episode basis, Temporal Difference methods update them after one or a few time steps within an episode, before these values converge to a close approximate of their true values. This is referred to as *bootstrapping*. bootstrapping

The core idea of Temporal Difference methods is to update these values iteratively using an update rule (involving an error term) that can be expressed in the following general form:

$$\text{updated value} \leftarrow \text{current value} + \text{learning rate} \times \underbrace{(\text{estimated target value} - \text{current value})}_{\text{temporal difference error}} \quad (5.1)$$

Temporal Difference methods that apply this update rule after one time step are referred to as TD(0) methods. Convergence of TD(0) algorithms, under a set of conditions known as the Robbins-Monro conditions, can be proved using results from stochastic approximation [16]. TD(0)

This chapter presents three approaches of implementing TD(0) for reinforcement learning. The first approach, called *TD(0) prediction*, is for policy evaluation. The second and third, called *SARSA* and Q-learning respectively, are for finding an optimal policy, and so are categorized under *TD(0) control*.

5.1 TD(0) Prediction

In TD(0) prediction, the update rule in Equation (5.1) is applied to obtain an estimate $V(s)$ of the state values $v_\pi(s)$ for a given policy π, i.e.,

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{estimated target value}} - V(s_t)\Big] \tag{5.2}$$

It works as follows. The agent at state S_t takes an action $\pi(S_t)$, then finds itself in state S_{t+1} and receives a reward R_{t+1}. At this point, a "estimated target" for V is formed by adding the reward R_{t+1} to $\gamma V(S_{t+1})$, i.e., the discounted state value of S_{t+1}. This estimated target is usually different at each time step—we can think of it as a moving target. What we want to see is that, as the number of steps increases, the moving target itself will approach the true value of v_π and the estimate V will then approach this true value to reduce the temporal difference error to zero; that is, $V(s)$ converges to $v_\pi(s)$ for the given policy π.

Table 5.1 shows the procedure of implementing a TD(0) prediction algorithm for v_π.

Table 5.1: TD(0) prediction algorithm for estimating v_π.

Input:	π (the policy to be evaluated)
Parameter:	step size $\alpha \in (0, 1]$
Initialize:	$V(s) \in \mathbb{R}$ for all $s \in \mathcal{S} \cup \hat{\mathcal{S}}$, with $\hat{\mathcal{S}}$ being the set of terminal states; all initial values are arbitrary except $V(s) = 0$ for all $s \in \hat{\mathcal{S}}$

	Loop for each episode:
	  Initialize S
	  Loop for each step of episode:
	  $A \leftarrow \pi(S)$
	  Take action A; observe R, S'
	  $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
	  $S \leftarrow S'$
	until $S \in \hat{\mathcal{S}}$

5.2 TD(0) Control

The two TD(0) control methods (namely, SARSA and Q-learning) discussed in this section aim to find a close estimate $Q(s, a)$ of the optimal action values $q_*(s, a)$, from which a corresponding optimal policy π_* can be extracted. Both implement the update rule in Equation (5.1) in an iterative process that involves exploration of the state-action space—usually via an ϵ-soft policy, in order to ensure that all state-action pairs (s, a) have a non-

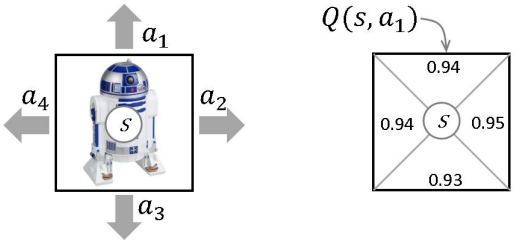
zero probability of being visited by the agent.

Although upon convergence both methods lead to an optimal (or near-optimal) policy, there are operational differences that could result in their respective solutions being different for the same reinforcement learning problem. We will first look at their implementation and then discuss their differences.

5.2.1 Exploration vs exploitation

We have encountered the notion of exploration in Chapter 4 when we discussed the method of *Monte Carlo control without exploring starts*. We now re-visit this notion here because it plays a central role in gaining an understanding of the behavior of the SARSA and Q-learning methods that we will discuss subsequently.

Consider the situation where the robot is at a state s whose action values are as shown in Figure 5.1. Suppose that those action values are not optimal; that is, the robot is to continue (from s) the learning process to find the optimal values. Now the robot needs to decide which action to take next.



(a) (b)

Figure 5.1: (a) Actions and (b) action values at state s.

The current $Q(s, a)$ values indicate that, to maximize the expected return starting from s, the robot should take action a_2 because $\max_a Q(s, a) = Q(s, a_2) = 0.95 > 0.94 > 0.93$. If the robot takes action a_2, then we describe this decision behavior as *exploitation* because the decision is made by exploiting the knowledge, in the form of the action values $Q(s, a)$, that the robot has learned up to this point. Since the current action values $Q(s, a)$ are not optimal and will be updated in subsequent visits by the agent, taking a_2 at s now is the "best" policy choice only with respect to these currently sub-optimal action values. It is possible that, by taking an action different from a_2 at s now, the robot may subsequently find higher rewards in some states that it will not encounter if it takes a_2 now. This decision behavior of the agent purposely not taking the "best" action with respect to the action values at a state is what we have known as *exploration*.

exploitation

exploration

Exploration is an essential characteristic of reinforcement learning. In general, if we do

not try new things (i.e., explore), then we will never develop new capabilities and skills. However, if we always explore and do not utilize what we have learned (i.e., exploitation), then we will most likely not be able to attain any significant achievement. The necessity for balancing this trade-off between exploitation and exploration is a key issue in reinforcement learning. One simple way[1] to balance this trade-off is to use the ϵ-greedy policy as shown in Equation (4.29) and reproduced here for easy reference:

$$\pi(a \,|\, s) = \begin{cases} 1 - \epsilon + \dfrac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = A^* \triangleq \arg\max_a Q(s,a) \\ \dfrac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a \neq A^* \end{cases} \tag{5.3}$$

where $0 < \epsilon < 1$.

In implementation, the probability ϵ is kept small so that the agent will focus on executing (in the best way it could) the task at hand most of the time, and occasionally will wander off to explore by taking an action that it is not necessarily the "best" based on what it knows at moment, as is illustrated in Figure 5.2.

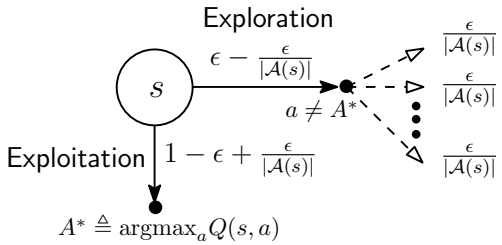


Figure 5.2: Exploration and exploitation.

The ϵ-greedy policy for action selection is often used in the implementations of SARSA and Q-learning. We next discuss each method in turn.

5.2.2 SARSA

One simple way to view SARSA is to think of it as a model-free implementation of the generalized policy iteration scheme, with the (normally iterative) policy evaluation sub-process truncated after one time step. Only that this time the policy evaluation/prediction sub-process is carried out for estimating the optimal action values $q_*(s,a)$ — or some action values close to $q_*(s,a)$, depending on the way exploration is conducted, and this sub-process is done via Temporal Difference learning, i.e, Equation (5.1). So we can simply

[1]Other methods are available. For example, a different strategy for exploration is to use the Boltzmann distribution to set the probability of the agent taking action a at state s_k as: $\mathbb{P}(a|s_k) = \exp\left(Q_k(s_k,a)/\tau_k\right) / \sum_{\hat{a}} \exp\left(Q_k(s_k,\hat{a})/\tau_k\right)$, where the "temperature" $\tau_k \geq 0$ controls the randomness of the exploration. The two extreme cases of $\tau_k \to 0$ and $\tau_k \to \infty$ correspond to greedy exploration and uniformly random action selection, respectively.

replace $V(s)$ in Equation (5.2) with $Q(s,a)$ to produce the update rule in SARSA:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\Big[\underbrace{R_{t+1} + \gamma\, Q(S_{t+1},A_{t+1})}_{\text{estimated target value}} - Q(S_t,A_t)\Big] \tag{5.4}$$

To apply this rule, the action values $Q(s,a)$ are initialized to some arbitrary values for non-terminal states and to zero otherwise. Starting at a non-terminal state S_t, the agent takes an action A_t according to some policy π (such as an ϵ-greedy policy) that enables the agent to explore, i.e., to visit every state-action pairs with a non-zero probability. After taking the action A_t the agent finds itself in state S_{t+1} and receives a reward R_{t+1}. The value of $Q(S_t,A_t)$ is then updated according to Equation (5.4). In state S_{t+1} the agent take another action A_{t+1}, again according to π, and so on until a terminal state is reached. For a terminal state s, its action value $Q(s,a)$ is defined as zero for all $a \in \mathcal{A}(s)$.

Note that this update rule uses all elements of the quintuple $\{S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}\}$ that are involved in a transition from S_t to S_{t+1}; for this reason it acquires the name *SARSA*. Table 5.2 shows the procedure of implementing the SARSA algorithm for estimating $Q \approx q_*$.

Convergence of SARSA to the optimal action values of a ϵ-soft policy (e.g., ϵ-greedy) is guaranteed under the following *Robbins-Monro conditions* [16]:

1. $\sum_{t=0}^{\infty}\alpha_t^2 < \infty$ and $\sum_{t=0}^{\infty}\alpha_t \to \infty$, where α_t is the learning rate with k being the time step.
2. All the state-action pairs (s,a) are visited infinitely often.

The first condition can be met by setting (among other possible choices) $\alpha_t = 1/t$, with the requirement that the initial value of α (i.e., α_0) is set to some finite value. The second condition can be satisfied by using an ϵ-soft policy for exploration.

5.2.3 Q-learning

In Q-learning the general update rule for Temporal Difference learning has the following specific form:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\Big[\underbrace{R_{t+1} + \gamma \max_{a'} Q\left(S_{t+1},a'\right)}_{\text{estimated target value}} - Q(S_t,A_t)\Big] \tag{5.5}$$

This form looks similar to that in SARSA; the only difference is that the term $Q(S_{t+1},A_{t+1})$ in SARSA is replaced by the term $\max_{a'} Q(S_{t+1},a')$ in Q-learning. This difference may seem minor in appearance, but its implication is significant in the context

Table 5.2: SARSA for estimating $Q \approx q_*$.

Parameters:	Step size $\alpha \in (0, 1]$, and small $\epsilon > 0$
Initialize:	$Q(s, a)$ for all $s \in \mathcal{S} \cup \hat{\mathcal{S}}$ and $a \in \mathcal{A}(s)$, with $\hat{\mathcal{S}}$ being the set of terminal states; all initial values are arbitrary except $Q(s, a) = 0$ for all $s \in \hat{\mathcal{S}}$
	Loop for each episode: Initialize S Choose A from S using policy derived from Q (e.g., ϵ-greedy) Loop for each step of episode: Take action A; observe R, S' Choose A' from S' using policy derived from Q (e.g., ϵ-greedy) $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma\, Q(S',\, A') - Q(S, A)\big]$ $S \leftarrow S'$; $A \leftarrow A'$ until $S \in \hat{\mathcal{S}}$

of the optimal policies obtained by these two methods — we will discuss this in the next section.

Table 5.3 shows the procedure for implementing a Q-learning algorithm for estimating $\pi \approx \pi_*$. The same Robbins-Monro conditions presented in the discussion above about SARSA apply to Q-learning.

Table 5.3: Q-learning for estimating $\pi \approx \pi_*$.

Parameters:	Step size $\alpha \in (0, 1]$, and small $\epsilon > 0$
Initialize:	$Q(s, a)$ for all $s \in \mathcal{S} \cup \hat{\mathcal{S}}$ and $a \in \mathcal{A}(s)$, with $\hat{\mathcal{S}}$ being the set of terminal states; all initial values are arbitrary except $Q(s, a) = 0$ for all $s \in \hat{\mathcal{S}}$
	Loop for each episode: Initialize S Loop for each step of episode: Choose A from S using policy derived from Q (e.g., ϵ-greedy) Take action A; receive R and observe S' $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma\ \max_{a'} Q(S',\, a') - Q(S, A)\big]$ $S \leftarrow S'$ until $S \in \hat{\mathcal{S}}$

Example 5.2.1. Suppose that a robot is to learn an optimal policy on a 3×3 grid for reaching the terminal state (the gray square) as shown in Figure 5.3. At a state, the robot can take one of four actions to move up ($a = 1$), right ($a = 2$), down ($a = 3$), or left ($a = 4$) and into the corresponding adjacent state. If no such adjacent state exists, then the robot's state remains unchanged after the execution of the action. The reward function ρ

is given as follows:

$$\rho(s,a,s') = \begin{cases} 1 & \text{if } s \neq 8 \text{ and } s' = 8 \\ 0 & \text{otherwise} \end{cases} \tag{5.6}$$

Let $\gamma = 0.9$ and $\alpha \triangleq \alpha_t = 1/t$, where t is the time step, with an initial value of $\alpha_0 = 1$.

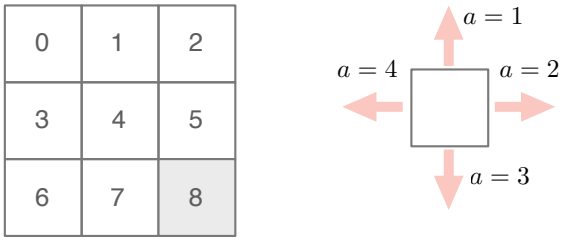
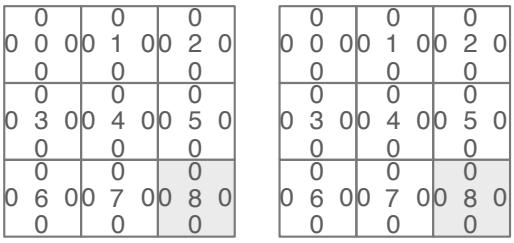


Figure 5.3: Reinforcement learning on a 3×3 grid.

The learning process will consist of a series of episodes. In an episode the robot will start at an initial state and make transitions according to the algorithm given in Table 5.3 until it reaches the terminal state ($s = 8$), upon which the episode ends.

In this example, we will focus on the calculation of the action values $Q(s,a)$; that is, we will not actually simulate the probabilistic selection of an action at each state during an episode. Instead we will simply assume a sequence of actions for an episode (as if they were observed to have occurred) in order to carry out the calculation of the $Q(s,a)$ values.

For record keeping, we define a function $N(s,a)$ which counts the accumulative number of times an action a has been taken at state s over a set of episodes, i.e., it is not reset to 0 after an episode. For graphical illustration of the learning process, we will use two types of diagrams: one showing the values of $N(s,a)$, and the other showing the values of $Q(s,a)$, at the end of an episode. Figure 5.4 shows the diagrams for $N(s,a)$ and $Q(s,a)$ before learning starts.



(a) $N(s,a)$ (b) $Q(s,a)$

Figure 5.4: Initial values of $N(s,a)$ and $Q(s,a)$. Each value at an edge of a state corresponds to the action in the outward direction perpendicular to that edge.

Episode 1: The (assumed) action sequence for Episode 1 is shown in Figure 5.5. Note that $\alpha_0 = 1$. We will use $Q_k(s,a)$ to denote the state-action value during time step k of an episode.

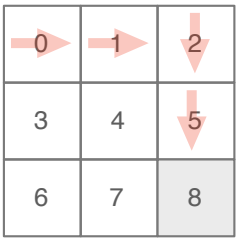


Figure 5.5: Action sequence in Episode 1.

From the update rule given in Equation (5.5), we have

$$\begin{aligned} Q_1(0,2) &= Q_0(0,2) + \alpha_0\Big(\rho(0,2,1) + \gamma \max_{a'} Q_0(1,a') - Q_0(0,2)\Big) \\ &= 0 + 1 \cdot (0 + 0.9 \cdot 0 - 0) \quad \big(\text{Note: } \alpha_0 = 1\big) \\ &= 0 \end{aligned} \tag{5.7}$$

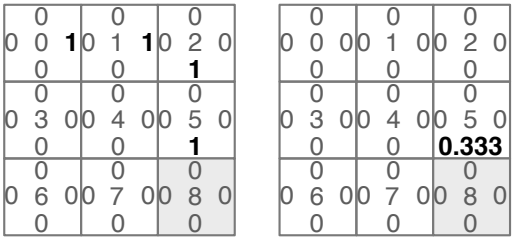
$$\begin{aligned} Q_2(1,2) &= Q_1(1,2) + \alpha_1\Big(\rho(1,2,2) + \gamma \max_{a'} Q_1(2,a') - Q_1(1,2)\Big) \\ &= 0 + \frac{1}{1}\Big(0 + 0.9 \cdot 0 - 0\Big) \quad \big(\text{Note: } k = 1;\ \alpha_k = 1/k\big) \\ &= 0 \end{aligned} \tag{5.8}$$

$$\begin{aligned} Q_3(2,3) &= Q_2(2,3) + \alpha_2\Big(\rho(2,3,5) + \gamma \max_{a'} Q_2(5,a') - Q_2(2,3)\Big) \\ &= 0 + \frac{1}{2}\Big(0 + 0.9 \cdot 0 - 0\Big) \\ &= 0 \end{aligned} \tag{5.9}$$

$$\begin{aligned} Q_4(5,3) &= Q_3(5,3) + \alpha_3\Big(\rho(5,3,8) + \gamma \max_{a'} Q_3(8,a') - Q_3(5,3)\Big) \\ &= 0 + \frac{1}{3}\Big(1 + 0.9 \cdot 0 - 0\Big) \\ &= 0.333 \end{aligned} \tag{5.10}$$

The episode ends when the robot enters the terminal state $s = 8$. The values of $N(s,a)$ and $Q(s,a)$ at the end of the episode are as shown in Figure 5.6.

Episode 2: The (assumed) action sequence for Episode 2 is shown in Figure 5.7. The initial learning rate is reset to $\alpha_0 = 1$ and the initial values of $Q(s,a)$ are those obtained from Episode 1 as shown in Figure 5.6(b).



(a) $N(s,a)$ (b) $Q(s,a)$

Figure 5.6: Values of $N(s,a)$ and $Q(s,a)$ at the end of Episode 1.

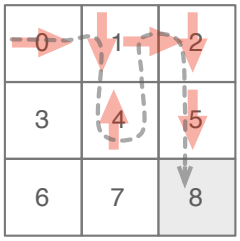


Figure 5.7: Action sequence in Episode 2.

$$\begin{aligned} Q_1(0,2) &= Q_0(0,2) + \alpha_0\Big(\rho(0,2,1) + \gamma \max_{a'} Q_0(1,a') - Q_0(0,2)\Big) \\ &= 0 + 1 \cdot (0 + 0.9 \cdot 0 - 0) \\ &= 0 \end{aligned} \tag{5.11}$$

$$\begin{aligned} Q_2(1,3) &= Q_1(1,3) + \alpha_1\Big(\rho(1,3,4) + \gamma \max_{a'} Q_1(4,a') - Q_1(1,3)\Big) \\ &= 0 + \frac{1}{1}\Big(0 + 0.9 \cdot 0 - 0\Big) \\ &= 0 \end{aligned} \tag{5.12}$$

$$\begin{aligned} Q_3(4,1) &= Q_2(4,1) + \alpha_2\Big(\rho(4,1,1) + \gamma \max_{a'} Q_2(1,a') - Q_2(4,1)\Big) \\ &= 0 + \frac{1}{2}\Big(0 + 0.9 \cdot 0 - 0\Big) \\ &= 0 \end{aligned} \tag{5.13}$$

$$\begin{aligned} Q_4(1,2) &= Q_3(1,2) + \alpha_3\Big(\rho(1,2,2) + \gamma \max_{a'} Q_3\left(2,a'\right) - Q_3(1,2)\Big) \\ &= 0 + \frac{1}{3}\Big(0 + 0.9 \cdot 0 - 0\Big) \\ &= 0 \end{aligned} \tag{5.14}$$

$$\begin{aligned} Q_5(2,3) &= Q_4(2,3) + \alpha_4\Big(\rho(2,3,5) + \gamma \max_{a'} Q_4\left(5,a'\right) - Q_4(2,3)\Big) \\ &= 0 + \frac{1}{4}\Big(0 + 0.9 \max\left[0,0,0.333,0\right] - 0\Big) \\ &= \frac{1}{4} \cdot 0.9 \cdot 0.333 \\ &= 0.075 \end{aligned} \tag{5.15}$$

$$\begin{aligned} Q_6(5,3) &= Q_5(5,3) + \alpha_5\Big(\rho(5,3,8) + \gamma \max_{a'} Q_5\left(8,a'\right) - Q_5(5,3)\Big) \\ &= 0.333 + \frac{1}{5}\Big(1 + 0.9 \cdot 0 - 0.333\Big) \\ &= 0.466 \end{aligned} \tag{5.16}$$

The values of $N(s,a)$ and the $Q(s,a)$ at the end of the episode are shown in Figure 5.8.

0 0 0 **2** 0	0 0 1 **2** **1**	0 0 2 0 **2**
0 0 3 0 0	**1** 0 4 0 0	0 0 5 0 **2**
0 0 6 0 0	0 0 7 0 0	0 0 8 0 0

(a) $N(s,a)$

0 0 0 0 0	0 0 1 0 0	0 0 2 0 **0.075**
0 0 3 0 0	0 0 4 0 0	0 0 5 0 **0.466**
0 0 6 0 0	0 0 7 0 0	0 0 8 0 0

(b) $Q(s,a)$

Figure 5.8: Values of $N(s,a)$ and $Q(s,a)$ at the end of Episode 2.

The above process repeats until, for every (s,a) pair, the difference in $Q(s,a)$ between two successive updates is smaller than some prescribed threshold. The action values $Q(s,a)$ are then considered to have converged to its optimal values $q_*(s,a)$, from which an optimal policy π_* can be then extracted. □

Remark 5.2.1. *If a pair (s,a) is not visited by the agent during a step, then the corresponding value $Q_k(s,a)$ does not change for that step. In Example 5.2.1, the state-action pair $(5,3)$ is visited during Episode 1 at time step 4 and was updated in Equation (5.10). After that, it remains unchanged until it is visited again in Episode 2 at time step 6. So from time step 4 of Episode 1 until time step 5 of Episode 2, the vale of $Q(5,3)$ remains at the same value as that*

calculated in Equation (5.10). □

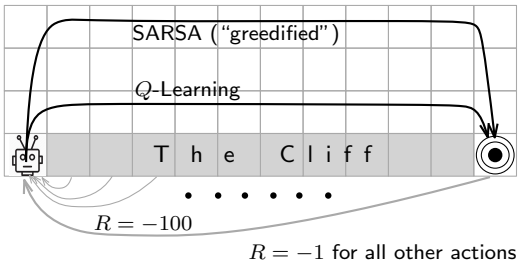
5.3 Target Policy and Behavior Policy

One way to characterize the behavior of a reinforcement learning algorithm is to look at its so-called *target policy* and *behavior policy*. In the context of this chapter, these two types of policies are fundamental to the understanding of the difference in learning behavior of SARSA and Q-learning.

We have pointed out earlier in Section 5.2.3 that the only difference between the update rules of SARSA and Q-learning is that the term $Q(S_{t+1}, A_{t+1})$ in SARSA is replaced by the term $\max_a Q(S_{t+1}, a)$ in Q-learning. This seemingly minor difference in appearance in fact explains a significant difference in the behavior of the agent when under the control of the optimal polices obtained by these two methods. We will illustrate this behavioral difference by the well-known *Cliff Walking* example.

Example 5.3.1. Consider the situation where a robot is to learn to move from one point on a cliff to another point, as is illustrated in Figure 5.9(a). This can be formulated as a typical undiscounted episodic reinforcement-learning task, with the usual actions in four directions at a state, and a reward of $R = -1$ for all transitions except those that cause the robot to enter the region labeled as "The Cliff". If the robot enters this region, it receives a reward of $R = -100$ and is instantly sent back to the starting state (as is indicated in Figure 5.9(b)).

(a) A Cliff near Latrabjarg, Iceland [4].



(b) Optimal policies.

Figure 5.9: Cliff walking example.

The solutions obtained by SARSA and Q-learning, both using the ϵ-greedy policy with $\epsilon = 0.1$ for action selection [22], are illustrated in Figure 5.9(b). We can see that Q-learning learns the optimal action values q_* of the optimal policy π_*, which is for the robot to move right next to the cliff along the edge, while SARSA learns the action values of a near-optimal policy that directs the robot to first move away from the cliff, then turn to move parallel to the edge, and finally turn towards the cliff to reach the goal state. □

Looking at the solutions as illustrated in Figure 5.9(b), the obvious question is: What causes SARSA to converge to the action values of the near-optimal policy as shown, instead of the action values $q_*(s,a)$ of the optimal policy π_* as Q-learning does? The answer to this question lies in the fact that SARSA is a so-called on-policy method while Q-learning is a so-called off-policy method. To explain what constitutes an on- or off-policy method, we need to first clarify the notions of *target policy* and *behavior policy* in the context of a reinforcement learning problem.

target policy

We note that both SARSA and Q-learning involve the iterative process for estimating the action values $Q(s,a)$. Once this process converges, a policy can be extracted based on the value of $Q(s,a)$. This policy is called the *target policy*. In the case of Q-learning, the algorithm for this iterative process (as described in Table 5.3) converges to the optimal action values $q_*(s,a)$, from which an optimal policy π_* can be extracted as $\pi_*(s) = \arg\max_a q_*(s,a)$, and is illustrated in Figure 5.9(b). This optimal policy π_* is the target policy of Q-learning for the cliff-walking problem. In other words, the target policy of Q-learning is the deterministic greedy policy with respect to the optimal action values $q_*(s,a)$. It is important to note that the Q-learning algorithm (as shown in Table 5.3) assumes that the agent is following π_* when updating $Q(s,a)$ using the term $\max_{a'} Q(s',a')$, but in fact the policy that the agent is following (before convergence) is not yet optimal.

target policy of Q-learning

behavior policy

We also note that, during the iterative process for estimating the action values $Q(s,a)$, both SARSA and Q-learning conduct exploration. Recall that the purpose of conducting exploration is to enable the agent to visit every state-action pairs (a,s) with a non-zero probability[2]. We can think of exploration as a process in which the agent follows a specifically chosen policy in order to generate sufficiently "rich" data samples for use by the update rule to find its target policy. This "specifically chosen" policy used for exploration is called the *behavior policy*. Here we take for example that such exploration is done under an ϵ-greedy policy as described in Equation (5.3). Therefore, in this case the behavior policy of Q-learning is the ϵ-greedy policy.

The situation for the case of Q-learning can be summarized as follows: The aim of Q-learning is to find its target policy, which is also the optimal policy π_* for a given reinforcement learning problem. The Q-learning algorithm achieves this aim by using an ϵ-soft policy (such as ϵ-greedy) as its behavior policy to generate the data samples needed in the learning process.

The same description given above applies to SARSA, with the exception that in SARSA the agent does not assume that it is following π_* as its target policy during the iteractive learning process; instead, it follows an ϵ-soft policy (e.g., ϵ-greedy) when updating the

[2]For example, in the method of Monet Carlo without exploring starts, the agent follows the ϵ-greedy policy during Monet Carlo prediction when the agent collects the expected return G_t by exploring various states.

values of $Q(s,a)$.

Figure 5.10 illustrates an intuitive representation of the agent's behavior when under the control of the target policy and the behavior policy in SARSA and Q-learning, for the case where SARSA uses the ϵ-greedy policy for both policies while Q-learning uses it for only for its behavior policy.

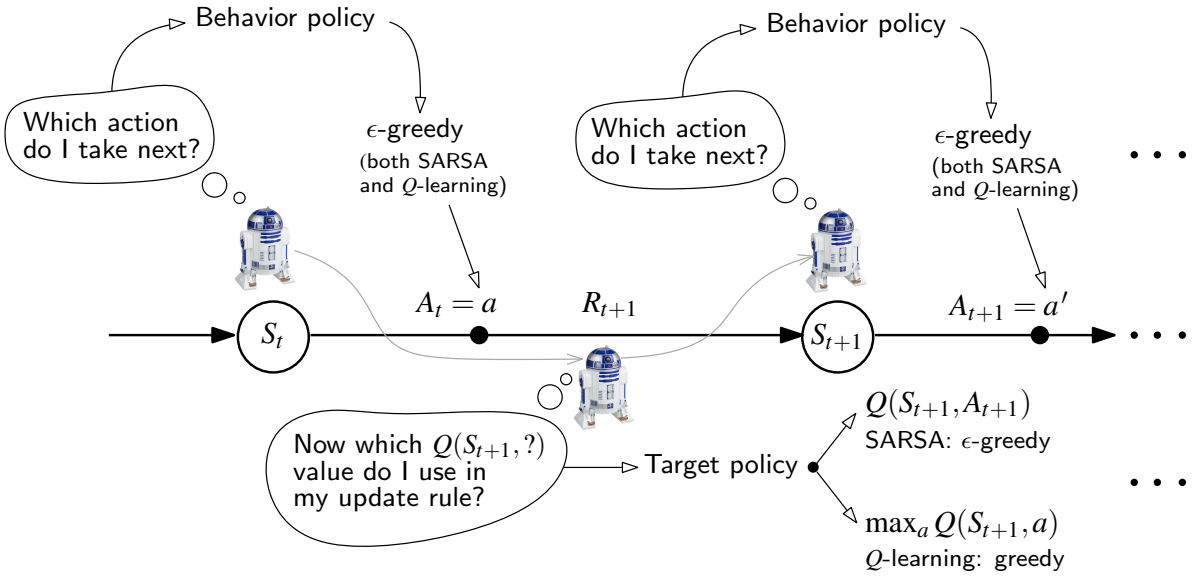


Figure 5.10: The roles of behavior policy and target policy in SARSA and Q-learning, for the situation where SARSA uses the ϵ-greedy policy for both while Q-learning uses it for its behavior policy.

Definition 5.3.1. *(On-policy and off-policy methods[3]):* A reinforcement learning method is considered *on-policy* if its behavior policy and target policy are the same, and *off-policy* otherwise. on-policy and off-policy

For the situation as illustrated in Figure 5.10, SARSA is on-policy, while Q-learning is off-policy.

In general, SARSA and Q-learning will converge to the optimal action values $Q_\pi(s,a)$ of their respective target policy π under the so-call *Robboins-Monro conditions* [16]. For Q-learning, since its target policy is the greedy policy, convergence to the optimal action values of the target policy means convergence to $q_*(s,a)$, from which a greedy policy can be extracted as: $\pi_*(s) = \arg\max_a q_*(s,a)$; that is, the policy we obtain upon convergence of Q-learning is an overall optimal policy π_* for the given reinforcement learning problem. In the cliff-walking situation described in Example 5.3.1, this optimal policy π_* directs the robot to move along (and right next) to the edge of the cliff, as is illustrated in Figure

[3]The method of Monte Carlo without exploring starts is an on-policy method that learns the action values of not the optimal policy but a near-optimal policy that still explores.

5.9(b). Obviously, even though this policy incurs the least cost (i.e., negative reward), it is highly risky. However, Q-learning does not take this risk into consideration when estimating the action values.

For SARSA, the issue of convergence is not as straightforward to describe as that for Q-learning, because of possible variation in the implementation of SARSA that can lead to subtle implications. SARSA does not necessarily converge to the overall optimal policy π_*. Using an algorithm as described in Table 5.2 with an ϵ-greedy policy as the target policy as well as the behavior policy, SARSA will converge to the optimal $Q(s,a)$ values for this ϵ-greedy policy. We can then obtain — based on these optimal $Q(s,a)$ values — an optimal ϵ-greedy policy, which in general is not the same as the optimal policy π_* we would obtain from Q-learning[4].

If we "greedify" an optimal ϵ-greedy policy by taking the greedy action at each state based on the optimal $Q(s,a)$ values determined by SARSA, we will in general obtain only a near-optimal policy for the given reinforcement learning problem. The policy (labeled SARSA) illustrated in Figure 5.9(b) is such a near-optimal policy. Since an optimal ϵ-greedy policy still explores with a probability of ϵ, SARSA recognizes the risk that, if the robot were to move right next to the edge of the cliff (by following π_*), such exploration would result in the robot falling off the cliff. By keeping the robot further away from the edge of the cliff, the optimal ϵ-greedy policy generated by SARSA reduces this risk. This answers the question raised earlier about what causes SARSA to converge to the action values of a near-optimal policy instead of $q_*(s,a)$ of the optimal policy π_*, as Q-learning does.

5.4 Practical Implications

Since Q-learning will yield the optimal policy π_* while SARSA produces (after greedification) only a near-optimal policy, what would be the advantage for using SARSA in the first place? The answer to this question lies in the way SARSA mitigates the risk of the robot falling off the cliff during learning.

Suppose that we replace the cliff in the cliff-walking example with a shallow trench, and we want the robot (at the starting location) to learn to move around the trench in order

[4]It is possible to make SARSA converge to the action values $q_*(s,a)$ of the optimal policy π_* as Q-learning does, if the assumption of *Greedy in the Limit with Infinite Exploration (GLIE)* — in addition to the Robbins-Monro conditions — is satisfied. Satisfying this assumption means that the agent carries out exploration infinitely, but its behavior policy will tend towards a greedy policy as time goes to infinity. One way to satisfy GLIE is to set $\epsilon = \epsilon_k = 1/k$, where k is the time step. In practice, it may not be always possible to satisfy this assumption. For example, if the robot falls off the cliff and is destroyed, then the learning will end. In this scenario, we will need to have multiple spare robots ready for use so that a replacement is available whenever a robot is put out of action; such a learning process, however, would be expensive to complete.

to reach the goal location. Because the robot must explore during learning, it is inevitable that it will fall into the trench from time to time. Although it is possible that whatever slight damage sustained by the robot could be fixed after each fall, we do not want it to fall too often so as to avoid major malfunction due to repeated damages.

If we use Q-learning to train the robot, the robot will favor the parallel path right next to the trench because the states along this path tend to have higher action values. However, being so close the edge incurs a high risk of falling into the trench because the robot may suddenly decides to explore (i.e., to act randomly under an ϵ-greedy behavior policy) and falls into the trench.

On the other hand, if we use SARSA to train the robot, the robot will learn to recognize the fact that moving too close to the edge increases the risk (due to the need for exploration) of falling into the trench, and so it will settle on a near-optimal path that leaves a buffer zone between the robot and the trench. This way if the robot needs to explore, the exploration will most likely to be safe due to the existence of the buffer zone.

For reinforcement learning problems with a small state space and few choices of actions at each state, Temporal Difference control methods (such as Q-learning and SARSA) are effective in generating acceptable solutions. For practical problems that have a very large state space and action set, such methods may become ineffective. Approaches of "deep" reinforcement learning are then needed to deal with such problems of a larger scale.

Chapter 6

Deep Reinforcement Learning

Deep reinforcement learning in general refers to machine learning methods that combine techniques of reinforcement learning and deep learning. Currently there exists a variety of definitions about "deep learning". For the purpose of this module, we adopt the following definition.

Definition 6.0.1. *(Deep learning):* Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction [11].

An intuitive example of deep learning is the use of a network (consisting of multiple layers of processing units) to extract high level features from pixel-level data in a raw image in order to classify the objects in that image. Figure 6.1 illustrates the use of a deep learning scheme for object recognition; it involves a type of layered neural network called a convolution neural network (CNN) .

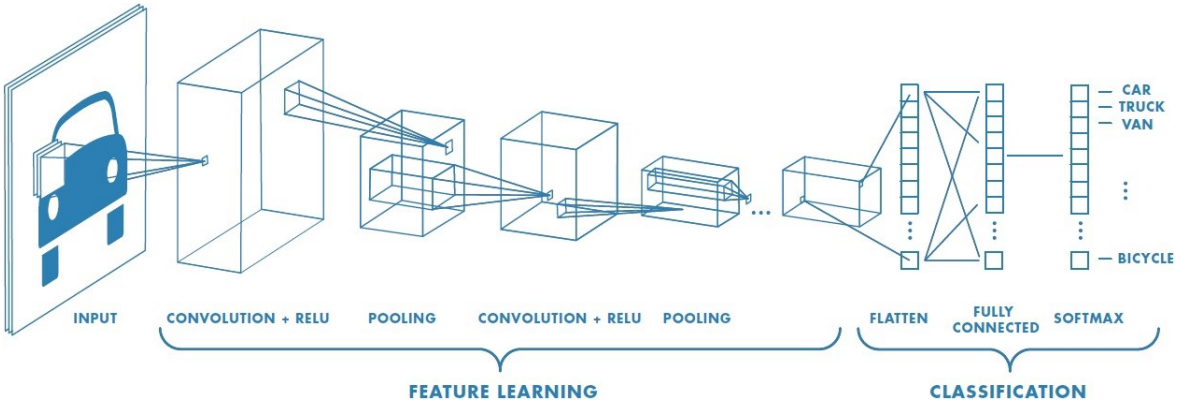


Figure 6.1: A deep learning scheme for object recognition using a CNN [12].

In this chapter, we will first introduce the scheme of multilayer perceptrons[1](MLP), then

[1]Convolutional neural networks (CNN) are similar to MLP in structure but their processing units have the type

discuss how a MLP can be incorporated into Q-learning to form the deep reinforcement learning technique called *Deep-Q-Networks* (DQN).

6.1 What Makes Reinforcement Learning "Deep"

In Q-learning, the main task is to find the optimal state-action values $Q_*(s, a)$. Implementation of Q-learning — using the update rule as given in Equation (1.4) — requires storage and retrieval of intermediate values of $Q(s_i, a_j)$ for state-action pairs (s_i, a_j). The usual practice is to use some data structure (e.g., a matrix) for this purpose. We can simply think of the data structure as a "look-up table" for the values of $Q(s_i, a_j)$, or simply a Q-table. When provided with the input s_i and a_j, we just search through this Q-table to find the value of $Q(s_i, a_j)$, as is illustrated in Figure 6.2.

Q-table

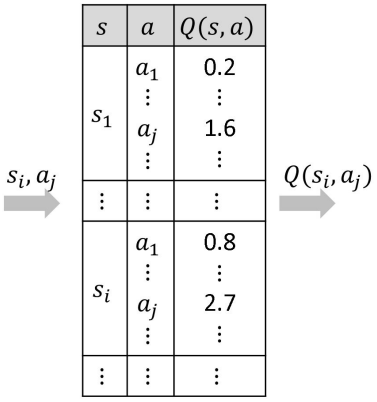


Figure 6.2: An example of a Q-table for obtaining $Q(s_i, a_j)$ for a given input state-action pair (a_i, s_j).

A major obstacle in using the Q-learning method to solve a practically meaningful problem is that the size of the Q-table can become so huge that it is not possible to "look up" $Q(s_i, a_j)$ given s_i and a_j.

To get a sense of proportion about this issue, consider the ancient game of Go. In this game, two players take turn to place black and white stones at the vacant intersections of a grid marked on a board. The act of placing a stone is called a *move*, i.e., an action. At any given time during the game, the collective positions of all the stones on the grid constitute a *board position*, which can be thought of as a *state* in a reinforcement learning problem. The number of legal board positions (for the standard size of a 19×19 grid) has been estimated[2] to be approximately 2×10^{170}. The average number of possible moves from a board position is considered to be around 250. So the size of a Q-table for the

of operation characteristics that are uniquely suitable for image processing.

[2]For a comparison, the number of atoms in the known observable universe is estimated to be about 10^{80} [8].

game of Go would be about $250 \times 2 \times 10^{170}$ [25, 5], making searching the Q-table to find $Q(s_i, a_j)$ computationally intractable.

The Deep Q-Network (DQN) method provides a computational solution for dealing with this issue. It has produced substantial impact on practical implementation of reinforcement-learning solutions for real-life problems [19]. This computation solution is based on a biologically inspired computing system called *(artificial) neural networks*.

6.2 Multilayer Perceptrons

Multilayer percetrons (MLP) are also known as multilayer feedforward neural networks. The name MLP finds its root in the original device called Percetrons, introduced by Frank Rosenblatt in 1957 [17]. Percetrons and MLP share a rich history which accounts for the majority of key results in machine learning from the 60's all the way to the present day [13, 18]

6.2.1 Structure

A neural network consists of a collection of *processing units* (also referred to as *neurons*). The units are arranged in layers, with each unit in a layer being connected to all the units in the next layer. Each such connection has a strength, represented by a *weight* value denoted by w. Signals flow from layer to layer (typically visualized as from left to right) in a single direction. Each unit usually implements a nonlinear function (denoted by φ), such as $\varphi(\cdot) = \tanh(\cdot)$. This nonlinear function is called an *activation function*. The input to the activation function φ of a unit is the weighted sum of the input signals to that unit. Figure 6.3 illustrates a processing unit, where $(\cdot)_j^{(l)}$ represents a signal $(\cdot)$ associated with unit j in layer l, and $w_{ji}^{(l)}$ denotes the weight from unit i in layer $(l-1)$ to unit j in layer l.

processing unit

connection weight

activation function

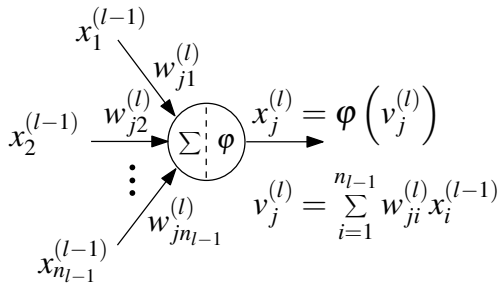


Figure 6.3: A processing unit j in a layer l.

As is illustrated in Figure 6.3, the input to the unit j are $x_1^{(l-1)}$, $x_2^{(l-1)}$, ..., $x_{n_{l-1}}^{(l-1)}$, which are outputs of units $1, 2, \ldots, n_{l-1}$ in the preceding layer $(l-1)$. The output of unit j is

$x_j^{(l)}$, i.e.,

$$x_j^{(l)} = \varphi^{(l)}\left(v_j^{(l)}\right), \quad \text{where} \quad v_j^{(l)} = \sum_{i=1}^{n_{l-1}} w_{ji}^{(l)} x_i^{(l-1)} \tag{6.1}$$

In a layered network structure, $x_j^{(l)}$ becomes an input to all units in the next layer, i.e., layer $(l+1)$. With the components as shown in Figure 6.4, we can construct a MLP as depicted in Figure 6.5. The layer that receives signal from some external source is called the *input layer* — normally indicated by $l = 0$. The layer that sends out the signal to some external source is called the *output layer*. Any layer located between the input and output layers are called a *hidden layer*. Although the MLP depicted in Figure 6.5 has four layers, it is actually called a three-layer MLP, i.e., two hidden layers plus one output layer, because by convention only layers with weights are counted. The superscript (l) in $\varphi^{(l)}(\cdot)$ is often omitted if the same activation function φ is used for all units (except those in the input layer) in the network

number of layers

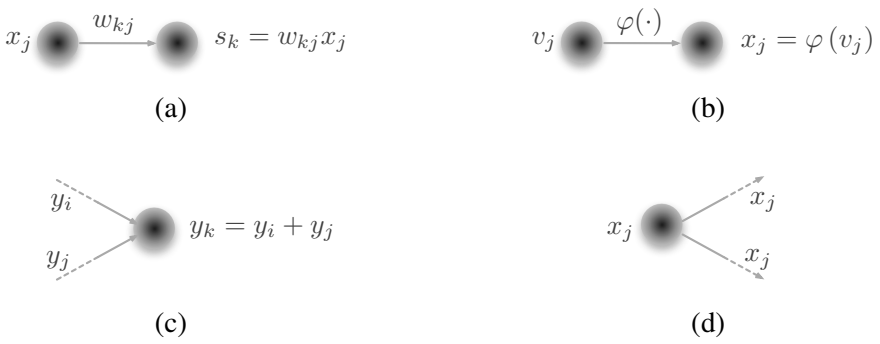


Figure 6.4: Components of signal flow graph for neural networks: (a) synaptic link, (b) activation link, (c) synaptic convergence, and (d) synaptic divergence.

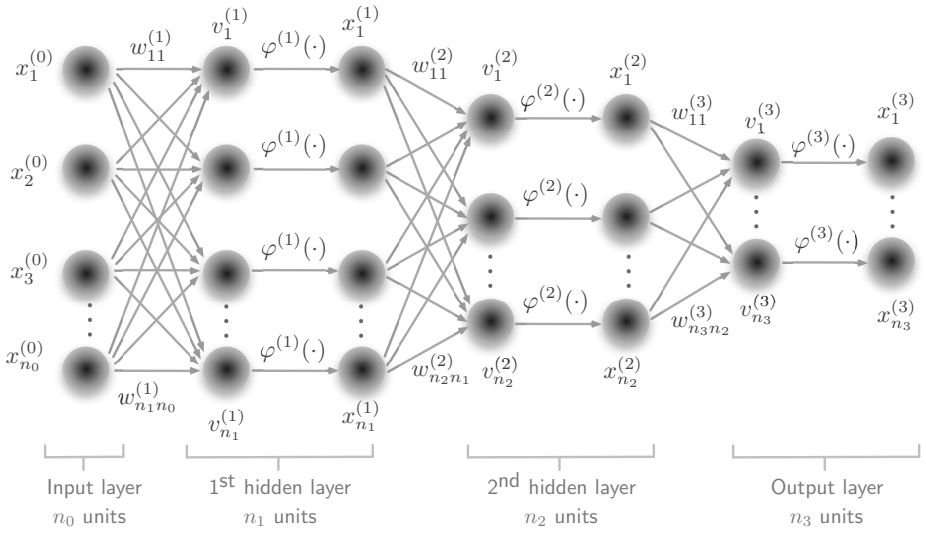


Figure 6.5: A three-layer MLP.

If, in addition to the inputs from the units in the preceding layer, a unit j in layer l also has a bias b (as an extra input), then the input signal $x_0^{(l-1)} = b$ and the weight $w_{j0}^{(l)} = 1$ can be added to unit j to account for this bias, as is illustrated in Example 6.2.1 below.

Example 6.2.1. Consider the MLP shown in Figure 6.6, with the weight values as indicated. Suppose that the activation function is $\varphi(v) = \tanh(v)$.

(a) Derive the input-output mapping defined by the network.

(b) Suppose that biases equal to -1 and $+1$ are applied to the top and bottom neurons of the first hidden layer, respectively, and biases equal to $+1$ and -2 are applied to the top and bottom neurons of the second hidden layer, respectively. Determine the input-output mapping defined by the network.

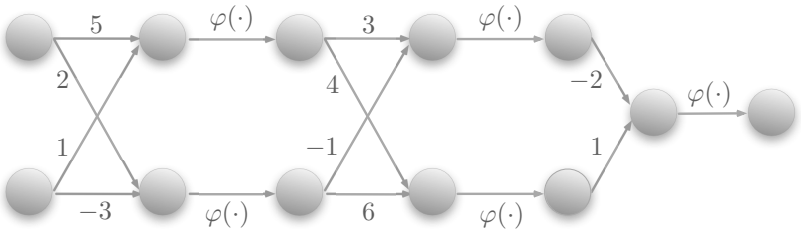


Figure 6.6: A MLP.

Solution: (a) Let x_1 and x_2 be the inputs to the MLP, i.e., $x_1^{(0)} = x_1$ and $x_2^{(0)} = x_2$. Let $v_1^{(l)}$ and $v_2^{(l)}$ be the inputs to φ, and $x_1^{(l)}$ and $x_2^{(l)}$ be the output, of the units in layer l, with $l = 1, 2, 3$. The output of the MLP is $x_1^{(3)}$. The input-output mapping defined by this MLP is the relationship between x_1, x_2, and $x_1^{(3)}$. Now,

$$
\begin{aligned}
v_1^{(1)} &= 5x_1^{(0)} + x_2^{(0)} = 5x_1 + x_2 & x_1^{(1)} &= \varphi\left(v_1^{(1)}\right) \\
v_2^{(1)} &= 2x_1^{(0)} - 3x_2^{(0)} = 2x_1 - 3x_2 & x_2^{(1)} &= \varphi\left(v_2^{(1)}\right) \\
v_1^{(2)} &= 3x_1^{(1)} - x_2^{(1)} & x_1^{(2)} &= \varphi\left(v_1^{(2)}\right) \\
v_2^{(2)} &= 4x_1^{(1)} + 6x_2^{(1)} & x_2^{(2)} &= \varphi\left(v_2^{(2)}\right) \\
v_1^{(3)} &= -2x_1^{(2)} + x_2^{(2)} & x_1^{(3)} &= \varphi\left(v_1^{(3)}\right)
\end{aligned}
\tag{6.2}
$$

(b) The solution for this part is similar to that for (a), except the introduction of the bias terms that are to be added, as is illustrated in Figure 6.7. For example, for the expression of $v_1^{(1)}$, we have $v_1^{(1)} = 5x_1 + x_2 - 1$.

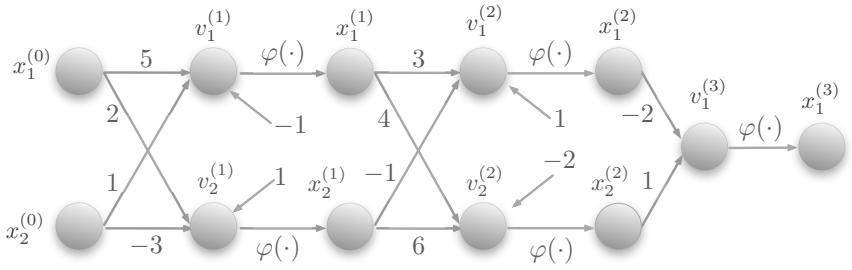


Figure 6.7: A MLP with bias terms.

The complete solution is as follows.

$$
\begin{aligned}
v_1^{(1)} &= 5x_1 + x_2 - 1 & x_1^{(1)} &= \varphi\left(v_1^{(1)}\right) \\
v_2^{(1)} &= 2x_1 - 3x_2 + 1 & x_2^{(1)} &= \varphi\left(v_2^{(1)}\right) \\
v_1^{(2)} &= 3x_1^{(1)} - x_2^{(1)} + 1 & x_1^{(2)} &= \varphi\left(v_1^{(2)}\right) \\
v_2^{(2)} &= 4x_1^{(1)} + 6x_2^{(1)} - 2 & x_2^{(2)} &= \varphi\left(v_2^{(2)}\right) \\
v_1^{(3)} &= -2x_1^{(2)} + x_2^{(2)} & x_1^{(3)} &= \varphi\left(v_1^{(3)}\right)
\end{aligned}
\tag{6.3}
$$

□

From Example 6.2.1 and Figure 6.5, we can see that the output of a MLP depends on the input $x_{(\cdot)}^{(0)}$ to the network and and the weights $w_{(\cdot)}^{(l)}$ of all layers. Using a vector $\mathbf{x}^{(0)}$ to represent the inputs to the network, and use a suitably defined matrix $\mathbf{W}$ to represent all these weights collectively, we can express the relationship between the input $\mathbf{x}^{(0)}$ and the output $\mathbf{x}^{(L)}$ (where L is the number of layers[3]) of a MLP in the form of a vector function $g(\mathbf{x}^{(0)}, \mathbf{W})$, i.e.,

$$\mathbf{x}^{(L)} = g\left(\mathbf{x}^{(0)}, \mathbf{W}\right) \tag{6.4}$$

MLP as universal approximator

It has been formally established that a MLP is an universal approximator. That is, it is inherently capable of approximating any function. Specifically, a theorem has been proved by Cybenko [7], which in effect states that *a MLP with only one layer of hidden units is capable of approximating any function with finitely many discontinuities to arbitrary precision, provided there are enough number of hidden units whose activation functions are nonlinear*.

Figure 6.8 illustrates the idea of using a MLP to approximate a singe-input (i.e., x),

[3]Recall that the input layer is not counted since it does not have weights.

single-output (y) function $y = f(x)$. The solid curve represents the output y of the function $f(x)$, while the dash curve represents the output y_{nn} of the MLP. The accuracy of the approximation is reflected by the deviation of y_{nn} from y.

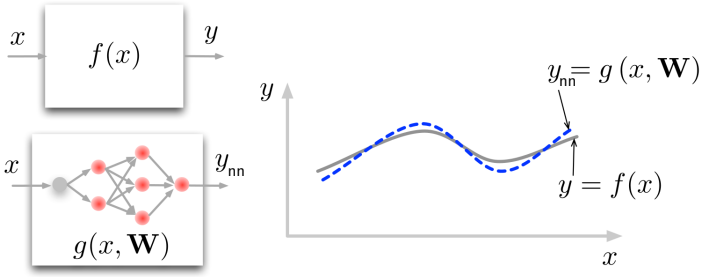


Figure 6.8: Using a MLP to approximate a single-inout single-output function.

Two factors determine the accuracy of the approximation: (1) the number of hidden units, and (2) the values of the weights $\mathbf{W}$. These raise the questions of how to determine the number of hidden units for a particular function to be approximated, and how to determine suitable weight values. The first question concerns the structure of the MLP — the answer to which is usually obtained by empirical means (e.g., trial and error). The second question concerns the issue of neural network learning.

Assuming that there are a sufficient number of hidden units in a MLP, the key to achieve a close approximation of a given function is to determine a set of values $\mathbf{W}^*$ for the weights $\mathbf{W}$ such that, for a given a function $\mathbf{y} = f(\mathbf{x})$,

optimal weights

$$\mathbf{x}^{(L)} = g\left(\mathbf{x}^{(0)}, \mathbf{W}^*\right) \approx f\left(\mathbf{x}^{(0)}\right) = \mathbf{y} \tag{6.5}$$

The values of $\mathbf{W}^*$ are called the *optimal weights*. The objective of neural network learning is to find these values for a given function.

6.2.2 Learning

We will introduce the idea of neural network learning using the following simple example. Suppose that we want to build a MLP that can approximate the parabolic function

$$y = x^2 \tag{6.6}$$

Since this function has one input and one output, we will use a MLP with a single unit in its input and output layer. Let the output of the MLP be y_{nn}, i.e.,

$$y_{nn} = g(x, \mathbf{W}) \tag{6.7}$$

and let e denote the difference between y and y_{nn}, i.e.,

$$e = y - y_{\text{nn}} \tag{6.8}$$

We call e the (network) *output error*. The objective then is to find a set of optimal values $\mathbf{W}^*$ for the weights $\mathbf{W}$, i.e., $\mathbf{W} = \mathbf{W}^*$, such that for any given x, the output error e is minimized, i.e.,

$$e = y - y_{\text{nn}} = x^2 - g(x, \mathbf{W}^*) \to 0 \tag{6.9}$$

Figure 6.9 illustrates this concept of neural network learning. It is in this context that we call y the desired output of the MLP during learning, because we want the the output y_{nn} of the MLP to approach y.

desired output

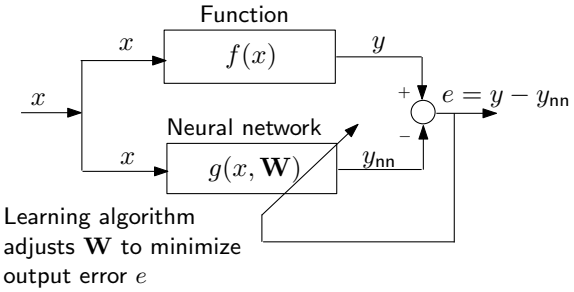


Figure 6.9: Neural network for function approximation.

The problem we face now is this: How do we find the optimal weight values? Intuitively, we can randomly select some value for $\mathbf{W}$, calculate (for a given input x) the MLP output y_{nn} based on these weight values, and check the error $e = y - y_{\text{nn}}$ to see if it is sufficiently small. If it is not sufficiently small, then we can select some new values for $\mathbf{W}$, and repeat the process. Obviously, this is not an efficient process, because we are just searching around aimlessly and hoping to hit the optimal weight values by luck. To conduct this search for optimal weight values more efficiently, we need some kind of indicator to guide us in selecting the values for the weights. In the well-known method of neural network learning called *error-backpropagation*, the indicator is the gradient of a measure of output error with respect to the weights.

The measure of output error used to define the gradient is commonly referred to as the *energy function*, denoted by E, and is defined as

energy function

$$E = \frac{1}{2} \sum_{j=1}^{n_L} e_j^2 \tag{6.10}$$

where e_j is the output error of unit j, and n_L is the total number of units, in the output layer L. In the case of using a MLP to approximate a parabolic function discussed above,

we have

$$E = \frac{1}{2}\sum_{j=1}^{n_L} e_j^2 = \frac{1}{2}e^2 = \frac{1}{2}\big(y - y_{\text{nn}}\big)^2 = \frac{1}{2}\left(x^2 - g(x, \mathbf{W})\right)^2 \tag{6.11}$$

We see that E is a function of the input x and the weight $\mathbf{W}$. Our objective now is to adjust $\mathbf{W}$ in order to minimize E.

To achieve this objective, we can adjust the values of the individual weights $w_{ji}^{(l)}$ along the negative gradient of E with respect to $w_{ji}^{(l)}$, i.e., $-\partial E/\partial w_{ji}^{(l)}$, according to the so-called Delta Rule:

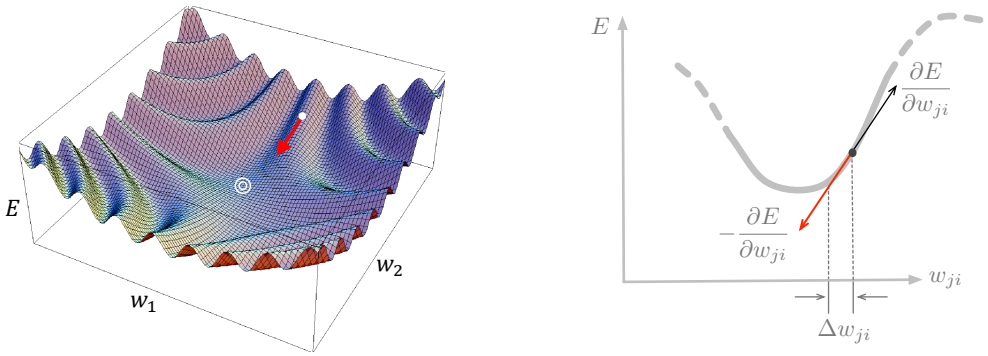
Delta Rule

$$\left[w_{ji}^{(l)}\right]_{\text{new}} = \left[w_{ji}^{(l)}\right]_{\text{old}} + \Delta w_{ji}^{(l)} = \left[w_{ji}^{(l)}\right]_{\text{old}} - \eta\left(\frac{\partial E}{\partial w_{ji}^{(l)}}\right) \tag{6.12}$$

where $l = 1, 2, \ldots, L$; the term $\eta \in (0, 1]$ is called the *learning rate* — it determines how big an adjustment (in the value of $w_{ji}^{(l)}$) is made each time the Delta Rule is applied.

learning rate

Figure 6.10 illustrates how the Delta Rule works. For easy visualization, we just assume some relationship between the energy function E and two weights w_1 and w_2 (among all the weights in $\mathbf{W}$) as represented by the ridged surface depicted in Figure 6.10(a). Imagine that we are currently located at the point represented by the solid white circle, and we want to move down to the location (indicated by the double white circle) where $E = 0$. It is obvious that, to reduce E, we can move in the direction of the red arrow, which is the "composite direction" formed by the negative gradients of E with respective to w_1 and w_2. If we just consider one of these gradients on its own, as is illustrated in Figure 6.10(b), we can see that by adjusting the current weight value $[w_{ji}]_{\text{old}}$ by an amount Δw_{ji} as depicted in the figure, we will obtain a new weight $[w_{ji}]_{\text{new}}$ corresponding to a smaller E. This in essence is how the error-backpropagation algorithm works.



(a) (b)

Figure 6.10: Gradient of output error with respect to weight parameter.

We now look at the implementation of the error-backpropagation algorithm. Consider a MLP with L layers, indexed as $l = 0, 1, 2, \ldots, L$, with layer 0 being the input layer, and layer L the output layer. Each layer has $(n_l + 1)$ units, numbered from 0 to n_{l+1}, with the 0^{th} unit representing the bias. For a MLP to learn, it requires a set of data, called the

training dataset

training dataset. A training dataset consists of a collection of *examples*. Each example consists of two elements: (1) a vector $\mathbf{x} = [x_1, x_2, \ldots, x_{n_0}]^T \in \mathbb{R}^{n_0 \times 1}$, and (2) a desired output (also called a *label*) $\mathbf{d} \in \mathbb{R}^{n_L \times 1}$ corresponding to $\mathbf{x}$. For instance, in the case of using a MLP to approximate $y = f(x) = x^2$ discussed earlier, we can discretize $f(x)$ to generate a training dataset with seven examples as

example

$$\{(-3, 9), (-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4), (3, 9)\} \tag{6.13}$$

Consider a training dataset with K examples, i.e.,

$$\{(\mathbf{x}_k, \mathbf{d}_k)\}, \quad k = 1, 2, \ldots, K \tag{6.14}$$

During learning, each example in the training dataset is presented to network in turn. Such a presentation is called an *iteration*. A complete round of presenting all K examples of the training dataset to the network is called an *epoch*. In other words, one epoch consists of K iterations. We use the notation $[\,\cdot\,](k)$ to represent the value of an entity $[\,\cdot\,]$ during the k^{th} iteration. For instance, $\mathbf{x}^{(0)}(k)$ represents the input to the units of the input layer of the network during the k^{th} iteration. Specifically, presenting an example to the network means setting $\mathbf{x}^{(0)}(k) = \mathbf{x}_k$; that is, the input to the i^{th} unit in the input layer of the network is the i^{th} element of the vector $\mathbf{x}_k$ (i.e., $\mathbf{x}_k[\,i\,]$) of the k^{th} example in the training dataset.

iteration

epoch

Table 6.1 shows the procedure for training a MLP using the Delta Rule. (We will discuss the details of how to derive the gradient terms associated with the Delta Rule shortly.) The steps indicated by the circled numbers in Table 6.1 correspond to the numerical labels in Figure 6.11. Once a MLP has been trained to approximate a function, it can be used to compute the output for an input that is not in the training dataset. For example, once we have trained a MLP to approximate the parabolic function $y = f(x) = x^2$ using the training dataset as given in Equation (6.13), we can use the MLP to compute the output corresponding to an input that is not in that dataset, e.g., $x = 1.2$. If the MLP has a sufficiently large number of hidden units and if a set of optimal weights have been found upon training, then for the input $x = 1.2$ the MLP would be able to produce an output very close to the exact value of $1.44 = 1.2^2$, even though the MLP has not been trained using $x = 1.2$ as an input. This ability of a neural network to produce a "good enough" output for an input that it has not "seen" during learning is referred to as *generalization*. This is the situation illustrated in Figure 6.11(b).

generalization

To complete the discussion on the error-backpropagation algorithm, we next derive the gradient terms in the Delta Rule for a three-layer MLP. The steps are similar for a MLP with more layers.

Consider a unit j in the output layer, i.e., $l = L$. The output error of unit j at iteration

Table 6.1: Neural network learning and generalization. The circled numbers correspond to the numerical labels in Figure 6.11.

Input:	A training dataset with K examples: $\{(\mathbf{x}_k, \mathbf{d}_k)\}$, $k = 1, 2, \ldots, K$
Parameters:	Learning rate $\eta > 0$, and error threshold θ
Initialize:	Weights $\mathbf{W}$ to arbitrary values, typically 0
	Loop for each epoch, i.e., $k = 1, 2, \ldots, K$:
	① Feed $\mathbf{x}_k$ as input to network, i.e., $\mathbf{x}^{(0)}(k) \leftarrow \mathbf{x}_k$
	② Compute network output $\mathbf{x}^{(L)}(k) = g(\mathbf{x}^{(0)}(k), \mathbf{W})$
	③ ④ Compare $\mathbf{x}^{(L)}(k)$ with $\mathbf{d}_k$ to determine $e_j(k)$ for all output units j
	⑤ Apply Delta Rule to adjust weights $\mathbf{W}$
	Until $E(k) < \theta$
	$\mathbf{W}^* \leftarrow \mathbf{W}$
	After training, fix weight values at $\mathbf{W}^*$
	⑥ Feed input not in training dataset into network
	⑦ Compute network output

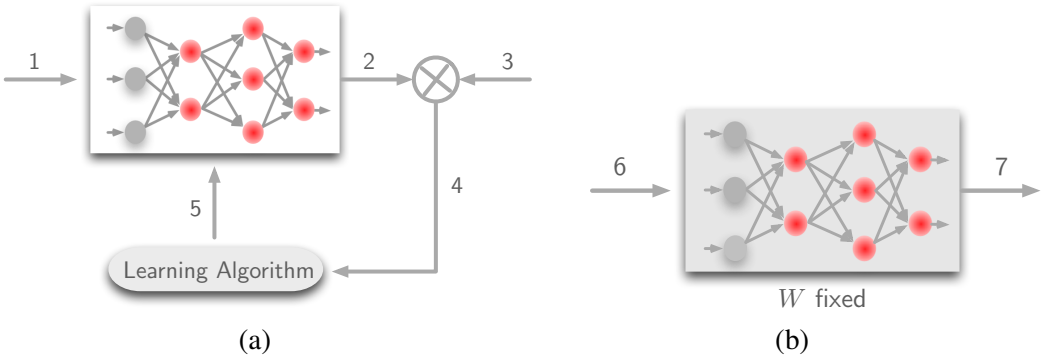


Figure 6.11: Neural network learning and generalization.

k can be expressed as

$$e_j(k) = d_j(k) - x_j^{(L)}(k) \tag{6.15}$$

where $d_j(k)$ denotes the j^{th} element of the vector of labels $\mathbf{d}_k$ for example k, i.e., $d_j(k) = \mathbf{d}_k[\,j\,]$. The energy function at iteration k is

$$E(k) = \frac{1}{2}\sum_{j=1}^{n_L} e_j^2(k) = \frac{1}{2}\sum_{j=1}^{n_L}\Big(d_j(k) - x_j^{(L)}(k)\Big)^2 \tag{6.16}$$

In general, E is a function (denoted by h) of $\mathbf{W}$ and the examples $(\mathbf{x}_k, \mathbf{d}_k)$ in the training dataset, i.e.,

$$E(k) = h\Big(\mathbf{x}_k, \mathbf{d}_k, \mathbf{W}(k)\Big) \tag{6.17}$$

Ideally, when a set of optimal weights $\mathbf{W}^*$ has been found, we have

$$E(k) = h\Big(\mathbf{x}_k, \mathbf{d}_k, \mathbf{W}^*\Big) \to 0, \qquad \text{for all } k \in \{1, 2, \ldots, K\} \tag{6.18}$$

To find a set of optimal weights, we start with some arbitrary initial values for each $w_{ji}^{(l)}(0)$, for instance, $w_{ji}^{(l)}(0) = 0$, and update each weight $w_{ji}^{(l)}$ using the Delta Rule to obtain the new weight $w_{ji}^{(l)}(k+1)$ after each iteration k, i.e.,

$$w_{ji}^{(l)}(k+1) = w_{ji}^{(l)}(k) + \Delta w_{ji}^{(l)}(k) \tag{6.19}$$

where the change in the value of $w_{ji}^{(l)}$ after iteration k is

$$\Delta w_{ji}^{(l)}(k) = -\eta \left(\frac{\partial E(k)}{\partial w_{ji}^{(l)}(k)} \right) \tag{6.20}$$

Note that $E(k)$ is evaluated at the output layer, i.e., $l = L$. So for the weights $w_{ji}^{(L)}(k)$ of the output layer, this partial derivative can be evaluated readily. For the weights of the hidden layers, the evaluation of this gradient is more complicated. The basic idea is to propagate the network output error backward from the output layer through the hidden layers, so that the gradients of E with respect to the weights of the hidden layers can be evaluated — thus the name error-backpropagation for this neural network learning method. The way to effectively propagate the output error backward through the layers is to apply the Chain Rule to evaluate $\partial E(k)/\partial w_{ji}^{(l)}(k)$ for $0 < l < L$.

We now derive all these gradient terms, staring from the output layer.

For a unit j in the *output layer* (i.e., $l = 3$), the change in the value of the weight from unit i in the 2nd hidden layer (i.e., $l = 2$) to unit j is:

$$\Delta w_{ji}^{(3)}(k) = -\eta \left(\frac{\partial E(k)}{\partial w_{ji}^{(3)}(k)} \right) \tag{6.21}$$

Figure 6.12 illustrates the structural relationship between unit i in the 2nd hidden layer and unit j in the output layer. By the definition as given in Equation (6.16), E is a function of $x_j^{(3)}$, and $x_j^{(3)}$ is a function of $v_j^{(3)}$ through the activation function φ, i.e., $x_j^{(3)} = \varphi\left(v_j^{(3)}\right)$, while $v_j^{(3)}$ is a function of $w_{ji}^{(3)}$ since $v_j^{(3)} = \sum_{i=1}^{n_2} w_{ji}^{(3)} x_i^{(2)}$, where n_2 is the total number of units in layer $l = 2$, i.e., the 2nd hidden layer. So we can apply the Chain Rule on the gradient term in Equation (6.21) by introducing an intermediate variable $v_j^{(3)}$ as

Chain Rule

$$\Delta w_{ji}^{(3)}(k) = -\eta \left(\frac{\partial E(k)}{\partial w_{ji}^{(3)}(k)} \right) = -\eta \left(\frac{\partial E(k)}{\partial v_j^{(3)}(k)} \right) \left(\frac{\partial v_j^{(3)}(k)}{\partial w_{ji}^{(3)}(k)} \right) \tag{6.22}$$

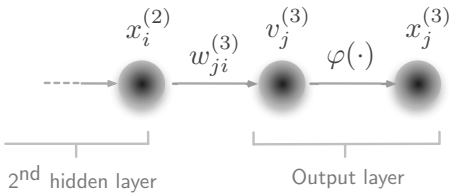


Figure 6.12: Weights from the second hidden layer to the output layer.

In the derivation below, we omit the iteration index k for readability. Now

$$\begin{aligned}
\frac{\partial v_j^{(3)}}{\partial w_{ji}^{(3)}} &= \frac{\partial}{\partial w_{ji}^{(3)}}\left(\sum_{m=0}^{n_2} w_{jm}^{(3)} x_m^{(2)}\right) \\
&= \frac{\partial\left(w_{j0}^{(3)} x_0^{(2)}\right)}{\partial w_{ji}^{(3)}} + \frac{\partial\left(w_{j1}^{(3)} x_1^{(2)}\right)}{\partial w_{ji}^{(3)}} + \cdots + \frac{\partial\left(w_{ji}^{(3)} x_i^{(2)}\right)}{\partial w_{ji}^{(3)}} + \cdots + \frac{\partial\left(w_{jn_2}^{(3)} x_{n_2}^{(2)}\right)}{\partial w_{ji}^{(3)}} \\
&= \frac{\partial\left(w_{ji}^{(3)} x_i^{(2)}\right)}{\partial w_{ji}^{(3)}} = x_i^{(2)} \quad \text{(i.e., for } m = i\text{)} \qquad (6.23) \\
\frac{\partial E}{\partial v_j^{(3)}} &= \frac{1}{2}\frac{\partial}{\partial v_j^{(3)}}\left(\sum_{m=1}^{n_3} e_m^2\right) \quad (m \text{ starts from 1 since no bias for output layer}) \\
&= \frac{1}{2}\frac{\partial}{\partial v_j^{(3)}}\left(\sum_{m=1}^{n_3}\left(d_m - x_m^{(3)}\right)^2\right) \quad \text{(from definition of } e\text{)} \\
&= \frac{1}{2}\sum_{m=1}^{n_3}\left(\frac{\partial\left(d_m - x_m^{(3)}\right)^2}{\partial v_j^{(3)}}\right) \\
&= \frac{1}{2}\sum_{m=1}^{n_3}\left(\frac{\partial\left(d_m - x_m^{(3)}\right)^2}{\partial\left(d_m - x_m^{(3)}\right)}\,\frac{\partial\left(d_m - x_m^{(3)}\right)}{\partial x_m^{(3)}}\,\frac{\partial x_m^{(3)}}{\partial v_j^{(3)}}\right) \\
&= -\left(d_j - x_j^{(3)}\right)\frac{\partial x_j^{(3)}}{\partial v_j^{(3)}} \quad \text{(for } m = j \text{ ; note that } \partial x_m^{(3)}/\partial v_j^{(3)} = 0 \text{ for all } m \neq j\text{)} \\
&= -\left(d_j - x_j^{(3)}\right)\varphi'\left(v_j^{(3)}\right) \equiv -\delta_j^{(3)} \quad \left(\text{where } \varphi'(\cdot) \equiv \tfrac{\partial\varphi(\cdot)}{\partial(\cdot)}\right) \qquad (6.24)
\end{aligned}$$

Therefore, Equation (6.22) can be expressed as

$$\Delta w_{ji}^{(3)} = -\eta\left(\frac{\partial E}{\partial w_{ji}^{(3)}}\right) = -\eta\left(\frac{\partial E}{\partial v_j^{(3)}}\right)\left(\frac{\partial v_j^{(3)}}{\partial w_{ji}^{(3)}}\right) = \eta\,\delta_j^{(3)}\,x_i^{(2)} \qquad (6.25)$$

Figure 6.13 illustrates the structural relationship between a unit i in the 1st hidden layer and a unit j in the 2nd hidden layer. For a unit j in the *2nd hidden layer* (i.e., $l = 2$), the change in the value of the weight from unit i in the 1st hidden layer (i.e., $l = 1$) to unit j

is:

$$\Delta w_{ji}^{(2)} = -\eta \left(\frac{\partial E}{\partial w_{ji}^{(2)}} \right) = -\eta \left(\frac{\partial E}{\partial v_j^{(2)}} \right) \left(\frac{\partial v_j^{(2)}}{\partial w_{ji}^{(2)}} \right) \tag{6.26}$$

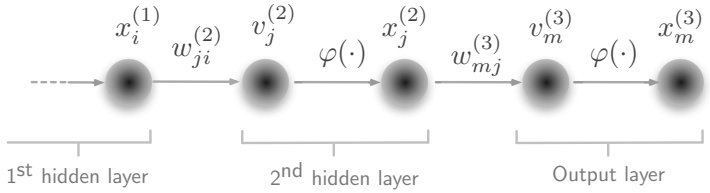


Figure 6.13: Weights from the first hidden layer to the second hidden layer.

Now

$$\frac{\partial v_j^{(2)}}{\partial w_{ji}^{(2)}} = \frac{\partial}{\partial w_{ji}^{(2)}} \left(\sum_{m=0}^{n_1} w_{jm}^{(2)} x_m^{(1)} \right) = \frac{\partial \left(w_{ji}^{(2)} x_i^{(1)} \right)}{\partial w_{ji}^{(2)}} = x_i^{(1)} \quad (\text{for } m = i) \tag{6.27}$$

$$\begin{aligned}
\frac{\partial E}{\partial v_j^{(2)}} &= \frac{1}{2} \frac{\partial}{\partial v_j^{(2)}} \left(\sum_{m=1}^{n_3} e_m^2 \right) \quad (\text{from definition of } E \text{ using } m \text{ as unit index}) \\
&= \frac{1}{2} \frac{\partial}{\partial v_j^{(2)}} \left(\sum_{m=1}^{n_3} \left(d_m - x_m^{(3)} \right)^2 \right) \\
&= \frac{1}{2} \sum_{m=1}^{n_3} \left(\frac{\partial \left(d_m - x_m^{(3)} \right)^2}{\partial v_j^{(2)}} \right) \\
&= \frac{1}{2} \sum_{m=1}^{n_3} \left(\frac{\partial \left(d_m - x_m^{(3)} \right)^2}{\partial \left(d_m - x_m^{(3)} \right)} \frac{\left(d_m - x_m^{(3)} \right)}{\partial x_m^{(3)}} \frac{\partial x_m^{(3)}}{\partial v_m^{(3)}} \frac{\partial v_m^{(3)}}{\partial x_j^{(2)}} \frac{\partial x_j^{(2)}}{\partial v_j^{(2)}} \right) \\
&= \frac{1}{2} \sum_{m=1}^{n_3} \left(\frac{\partial \left(d_m - x_m^{(3)} \right)^2}{\partial \left(d_m - x_m^{(3)} \right)} \frac{\left(d_m - x_m^{(3)} \right)}{\partial x_m^{(3)}} \frac{\partial x_m^{(3)}}{\partial v_m^{(3)}} \frac{\partial v_m^{(3)}}{\partial x_j^{(2)}} \right) \frac{\partial x_j^{(2)}}{\partial v_j^{(2)}} \\
&= \sum_{m=1}^{n_3} \left(\left(d_m - x_m^{(3)} \right) (-1) \varphi' \left(v_m^{(3)} \right) \underbrace{\frac{\partial \left(\sum_{q=0}^{n_2} w_{mq}^{(3)} x_q^{(2)} \right)}{\partial x_j^{(2)}}}_{\text{for } q = j} \right) \frac{\partial x_j^{(2)}}{\partial v_j^{(2)}} \\
&= - \left(\sum_{m=1}^{n_3} \underbrace{\left(d_m - x_m^{(3)} \right) \varphi' \left(v_m^{(3)} \right)}_{\delta_m^{(3)}} \frac{\partial \left(w_{mj}^{(3)} x_j^{(2)} \right)}{\partial x_j^{(2)}} \right) \frac{\partial \varphi \left(v_j^{(2)} \right)}{\partial v_j^{(2)}} \\
&= - \left(\sum_{m=1}^{n_3} \delta_m^{(3)} w_{mj}^{(3)} \right) \varphi' \left(v_j^{(2)} \right) \equiv -\delta_j^{(2)}
\end{aligned} \tag{6.28}$$

Therefore, Equation (6.26) can be expressed as

$$\Delta w_{ji}^{(2)} = -\eta \left(\frac{\partial E}{\partial w_{ji}^{(2)}} \right) = -\eta \left(\frac{\partial E}{\partial v_j^{(2)}} \right) \left(\frac{\partial v_j^{(2)}}{\partial w_{ji}^{(2)}} \right) = \eta \, \delta_j^{(2)} \, x_i^{(1)} \tag{6.29}$$

Figure 6.14 illustrates the structural relationship between a unit i in the input layer and a unit j in the 1st hidden layer. For a unit j in the *1st hidden layer* (i.e., $l = 1$), the change in the value of the weight from unit i in the input layer (i.e., $l = 0$) to unit j is:

$$\Delta w_{ji}^{(1)} = -\eta \left(\frac{\partial E}{\partial w_{ji}^{(1)}} \right) = -\eta \left(\frac{\partial E}{\partial v_j^{(1)}} \right) \left(\frac{\partial v_j^{(1)}}{\partial w_{ji}^{(1)}} \right) \tag{6.30}$$

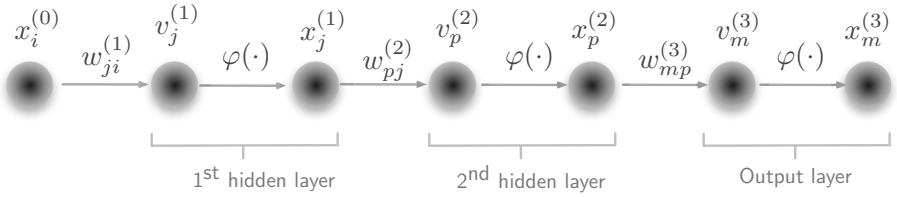


Figure 6.14: Weights from the input layer to the first hidden layer.

$$\frac{\partial v_j^{(1)}}{\partial w_{ji}^{(1)}} = \frac{\partial}{\partial w_{ji}^{(1)}}\left(\sum_{m=0}^{n_0} w_{jm}^{(1)} x_m^{(0)}\right) = \frac{\partial\left(w_{ji}^{(1)} x_i^{(0)}\right)}{\partial w_{ji}^{(1)}} = x_i^{(0)} \tag{6.31}$$

$$\begin{aligned}
\frac{\partial E}{\partial v_j^{(1)}} &= \frac{1}{2}\frac{\partial}{\partial v_j^{(1)}}\left(\sum_{m=1}^{n_3} e_m^2\right) \\
&= \frac{1}{2}\frac{\partial}{\partial v_j^{(1)}}\left(\sum_{m=1}^{n_3}\left(d_m - x_m^{(3)}\right)^2\right) \\
&= \frac{1}{2}\sum_{m=1}^{n_3}\left(\frac{\partial\left(d_m - x_m^{(3)}\right)^2}{\partial v_j^{(1)}}\right) \\
&= \frac{1}{2}\sum_{m=1}^{n_3}\left(\frac{\partial\left(d_m - x_m^{(3)}\right)^2}{\partial\left(d_m - x_m^{(3)}\right)}\ \frac{\left(d_m - x_m^{(3)}\right)}{\partial x_m^{(3)}}\ \frac{\partial x_m^{(3)}}{\partial v_m^{(3)}}\ \frac{\partial v_m^{(3)}}{\partial v_j^{(1)}}\right) \\
&= \frac{1}{2}\sum_{m=1}^{n_3}\left(\frac{\partial\left(d_m - x_m^{(3)}\right)^2}{\partial\left(d_m - x_m^{(3)}\right)}\ \underbrace{\frac{\left(d_m - x_m^{(3)}\right)}{\partial x_m^{(3)}}}_{-1}\ \frac{\partial x_m^{(3)}}{\partial v_m^{(3)}}\ \frac{\partial \sum_{p=0}^{n_2} w_{mp}^{(3)} x_p^{(2)}}{\partial v_j^{(1)}}\right) \\
&= -\sum_{m=1}^{n_3}\left(\underbrace{\left(d_m - x_m^{(3)}\right)\varphi'\left(v_m^{(3)}\right)}_{\delta_m^{(3)}}\sum_{p=0}^{n_2}\frac{\partial w_{mp}^{(3)} x_p^{(2)}}{\partial v_j^{(1)}}\right) \\
&= -\sum_{m=1}^{n_3}\sum_{p=0}^{n_2}\left(\delta_m^{(3)}\ \frac{\partial w_{mp}^{(3)} x_p^{(2)}}{\partial x_p^{(2)}}\ \frac{\partial x_p^{(2)}}{\partial v_p^{(2)}}\ \frac{\partial v_p^{(2)}}{\partial x_j^{(1)}}\frac{\partial x_j^{(1)}}{\partial v_j^{(1)}}\right) \\
&= -\sum_{p=0}^{n_2}\left(\underbrace{\left(\sum_{m=1}^{n_3}\delta_m^{(3)} w_{mp}^{(3)}\right)\varphi'\left(v_p^{(2)}\right)}_{\delta_p^{(2)}}\frac{\partial v_p^{(2)}}{\partial x_j^{(1)}}\right)\frac{\partial x_j^{(1)}}{\partial v_j^{(1)}} \\
&= -\sum_{p=0}^{n_2}\left(\delta_p^{(2)}\ \underbrace{\frac{\partial \sum_{q=0}^{n_1} w_{pq}^{(2)} x_q^{(1)}}{\partial x_j^{(1)}}}_{\text{for } q=j}\right)\frac{\partial x_j^{(1)}}{\partial v_j^{(1)}} \\
&= -\left(\sum_{p=0}^{n_2}\delta_p^{(2)}\ w_{pj}^{(2)}\right)\varphi'\left(v_j^{(1)}\right) \equiv -\delta_j^{(1)}
\end{aligned} \tag{6.32}$$

Therefore, Equation (6.30) can be expressed as

$$\Delta w_{ji}^{(1)} = -\eta \left(\frac{\partial E}{\partial w_{ji}^{(1)}} \right) = -\eta \left(\frac{\partial E}{\partial v_j^{(1)}} \right) \left(\frac{\partial v_j^{(1)}}{\partial w_{ji}^{(1)}} \right) = \eta \, \delta_j^{(1)} \, x_i^{(0)} \tag{6.33}$$

We have derived all the $\Delta w_{ji}^{(l)}$ terms, i.e., Equations (6.25), (6.29), and (6.33), for a three-layer MLP. We now make two observations. First, all $\Delta w_{ji}^{(l)}$ terms for a layer contain the output of the preceding layer; for instance, the expression for $\Delta w_{ji}^{(1)}$ contains $x_i^{(0)}$. Second, in the expression of $\Delta w_{ji}^{(l)}$ for a layer, there is a corresponding δ term, i.e., $\delta_{(\cdot)}^{(l)}$, for the same layer. In particular, the δ-term for a layer is a function of the δ-term for the preceding layer; for instance, the expression for $\delta_{(\cdot)}^{(3)}$ contains $\delta_{(\cdot)}^{(2)}$. These two observations provide the rationale to consider the error-backpropagation algorithm as having two procedural passes (as illustrated in Figure 6.15): One going forward from the input layer through the hidden layer(s) to the output layer in order to compute the output $\mathbf{x}^{(l)}$ of each layer, i.e., forward pass

$$v_j^{(l)} = \sum_{i=1}^{n_{l-1}} w_{ji}^{(l)} x_i^{(l-1)} \,, \quad x_j^{(l)} = \varphi \left(v_j^{(l)} \right) \tag{6.34}$$

while the other pass going backwards in order to compute the δ terms for updating the weights using the Delta Rule, i.e., backward pass

$$w_{ji}^{(l)}(k+1) = w_{ji}^{(l)}(k) + \eta \, \delta_j^{(l)}(k) \, x_i^{(l-1)}(k) \tag{6.35}$$

where (with k omitted)

$$\delta_j^{(l)} = \begin{cases} \left(d_j - x_j^{(l)} \right) \varphi' \left(v_j^{(l)} \right) = e_j^{(l)} \varphi' \left(v_j^{(l)} \right) & \text{for } l = L \\ \left(\sum_{p=0}^{n_{l+1}} \delta_p^{(l+1)} w_{pj}^{(l+1)} \right) \varphi' \left(v_j^{(l)} \right) & \text{for } 0 < l < L \end{cases} \tag{6.36}$$

Remark 6.2.1. *From Equations (6.36), we see that if we know $e_j^{(L)}$ then we do not need to have the labels d_j (which is part of the training dataset $\{(\mathbf{x}, \mathbf{d})\}$) when implementing the Delta Rule. In other words, if we have some way of determining the network output error $e_j^{(L)}$, then we will be able to run the error-backprogagation algorithm with just the input vector $\mathbf{x}$; that is, there is no need to have the complete training dataset $\{(\mathbf{x}, \mathbf{d})\}$.* □.

From Equation (6.36) we see that $\delta_j^{(l)}$ is a function of the partial derivative of φ, i.e., $\varphi'(\cdot) \triangleq \partial\varphi(\cdot)/\partial(\cdot)$. This means that in order to implement the Delta Rule, we must use a activation function that is differentiable. There are a variety of choices for φ; which one

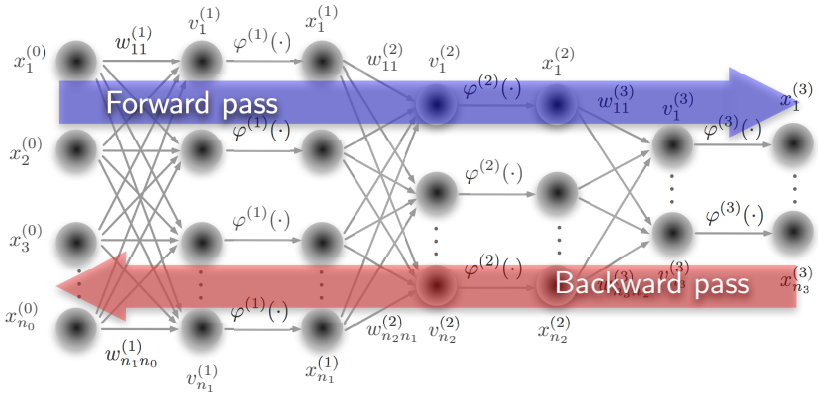


Figure 6.15: Error-backpropagation: Forward pass and backward pass.

to use will often depend on the application for which the MLP is intended. The common types of activation function include the sigmoidal function $\varphi(x) = 1/(1 + \exp(-ax))$, the hyperbolic tangent function $\varphi(x) = \tanh(ax)$, and the Gaussian radial basis function $\varphi(x) = \exp(-(x-c)^2/r^2)$, with $a, c, r > 0$.

Example 6.2.2. Figure 6.16 illustrates a sigmoidal function

$$\varphi(v) = \frac{1}{1+e^{-av}}, \qquad a > 0 \tag{6.37}$$

Determine $\partial\varphi(v)/\partial v$.

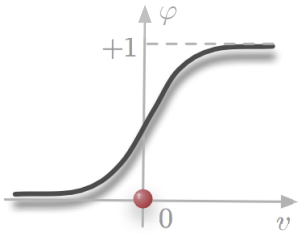


Figure 6.16: A sigmoidal function.

Solution:

$$\begin{aligned}
\varphi'(v) &= \frac{\partial\varphi(v)}{\partial v} = \frac{\partial}{\partial v}\left(\frac{1}{1+e^{-av}}\right) \\
&= -\frac{(-a)e^{-av}}{(1+e^{-av})^2} \qquad \left(\text{Note: } \frac{\partial}{\partial x}\left(\frac{1}{f(x)}\right) = -\frac{1}{f^2(x)}\left(\frac{\partial f(x)}{\partial x}\right)\right) \\
&= a\left(\frac{e^{-av}}{1+e^{-av}}\right)\left(\frac{1}{1+e^{-av}}\right) \\
&= a\left(1-\varphi(v)\right)\varphi(v)
\end{aligned} \tag{6.38}$$

□

Remark 6.2.2. *Note that, for the sigmoidal function, $\varphi'(v)$ can be easily evaluated if the value of $\varphi(v)$ is already available. Now $x_j^{(l)} = \varphi^{(l)}\left(v_j^{(l)}\right)$, which is the output of unit j in layer l. Since the outputs of the units in each layer are calculated during the forward pass, the value of $\varphi(v)$ is already available when the backward pass is executed.* □

Example 6.2.3. Consider the three-layer MLP as shown in Figure 6.17. Suppose that the activation function of the hidden units and the output unit is $\varphi(\cdot) = \tanh(\cdot)$. Determine:

$$\textit{(i)}\ \frac{\partial v_1^{(3)}}{\partial w_{12}^{(3)}} \qquad \textit{(ii)}\ \frac{\partial v_1^{(3)}}{\partial x_2^{(2)}} \qquad \textit{(iii)}\ \frac{\partial v_1^{(3)}}{\partial x_2^{(1)}}$$

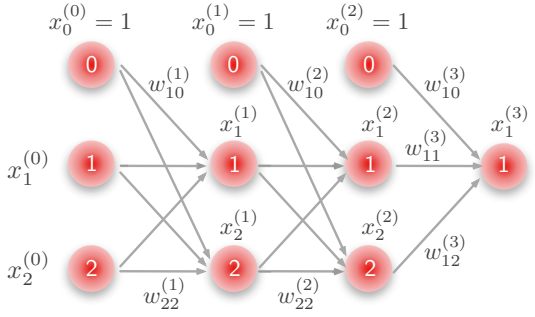


Figure 6.17: A three-layer MLP. (Note that the signal v is not explicitly shown in this figure.)

Solution: For a MLP, we have,

$$v_j^{(l)} = \sum_{i=0}^{n_{l-1}} w_{ji}^{(l)} x_i^{(l-1)}, \qquad x_j^{(l)} = \varphi^{(l)}\left(v_j^{(l)}\right) \tag{6.39}$$

(i) Now

$$\begin{aligned} v_1^{(3)} &= \sum_{i=0}^{n_2} w_{1i}^{(3)} x_i^{(2)} \\ &= \sum_{i=0}^{2} w_{1i}^{(3)} x_i^{(2)} \quad \text{(Since in the given network, } n_2 = 2\text{)} \\ &= w_{10}^{(3)} x_0^{(2)} + w_{11}^{(3)} x_1^{(2)} + w_{12}^{(3)} x_2^{(2)} \end{aligned} \tag{6.40}$$

Note that the term $w_{12}^{(3)}$ appears explicitly in the expression for $v_1^{(3)}$ while all other entities

(i.e., the other w's and x's) in that expression are not a function of $w_{12}^{(3)}$. Hence, we have

$$\frac{\partial v_1^{(3)}}{\partial w_{12}^{(3)}} = \frac{\partial}{\partial w_{12}^{(3)}}\left(w_{10}^{(3)}x_0^{(2)} + w_{11}^{(3)}x_1^{(2)} + w_{12}^{(3)}x_2^{(2)}\right) = x_2^{(2)} \tag{6.41}$$

□

(ii) From the expression for $v_1^{(3)}$ obtained from (i) above, we have

$$\frac{\partial v_1^{(3)}}{\partial x_2^{(2)}} = \frac{\partial}{\partial x_2^{(2)}}\left(w_{10}^{(3)}x_0^{(2)} + w_{11}^{(3)}x_1^{(2)} + w_{12}^{(3)}x_2^{(2)}\right) = w_{12}^{(3)} \tag{6.42}$$

(iii) Now

$$\frac{\partial v_1^{(3)}}{\partial x_2^{(1)}} = \frac{\partial}{\partial x_2^{(1)}}\left(w_{10}^{(3)}x_0^{(2)} + w_{11}^{(3)}x_1^{(2)} + w_{12}^{(3)}x_2^{(2)}\right) \tag{6.43}$$

We note that although $x_2^{(1)}$ does not explicitly appear in the above expression, it is implicitly related to $v_1^{(3)}$ through $x_1^{(2)}$ and $x_2^{(2)}$ as shown in the relationships below.

$$\begin{aligned} x_1^{(2)} &= \varphi\left(v_1^{(2)}\right) = \tanh\left(v_1^{(2)}\right) \\ &= \tanh\left(\sum_{i=0}^{n_1} w_{1i}^{(2)}x_i^{(1)}\right) = \tanh\left(\sum_{i=0}^{2} w_{1i}^{(2)}x_i^{(1)}\right) \\ &= \tanh\left(w_{10}^{(2)}x_0^{(1)} + w_{11}^{(2)}x_1^{(1)} + w_{12}^{(2)}x_2^{(1)}\right) \end{aligned} \tag{6.44}$$

$$\begin{aligned} x_2^{(2)} &= \varphi\left(v_2^{(2)}\right) = \tanh\left(v_2^{(2)}\right) \\ &= \tanh\left(\sum_{i=0}^{n_1} w_{2i}^{(2)}x_i^{(1)}\right) = \tanh\left(\sum_{i=0}^{2} w_{2i}^{(2)}x_i^{(1)}\right) \\ &= \tanh\left(w_{20}^{(2)}x_0^{(1)} + w_{21}^{(2)}x_1^{(1)} + w_{22}^{(2)}x_2^{(1)}\right) \end{aligned} \tag{6.45}$$

Thus, we can write

$$\begin{aligned} \frac{\partial v_1^{(3)}}{\partial x_2^{(1)}} &= \frac{\partial}{\partial x_2^{(1)}}\left(w_{10}^{(3)}x_0^{(2)} + w_{11}^{(3)}x_1^{(2)} + w_{12}^{(3)}x_2^{(2)}\right) \\ &= w_{11}^{(3)}\left(\frac{\partial x_1^{(2)}}{\partial x_2^{(1)}}\right) + w_{12}^{(3)}\left(\frac{\partial x_2^{(2)}}{\partial x_2^{(1)}}\right) \\ &= w_{11}^{(3)}\left(\frac{\partial x_1^{(2)}}{\partial v_1^{(2)}}\frac{\partial v_1^{(2)}}{\partial x_2^{(1)}}\right) + w_{12}^{(3)}\left(\frac{\partial x_2^{(2)}}{\partial v_2^{(2)}}\frac{\partial v_2^{(2)}}{\partial x_2^{(1)}}\right) \end{aligned} \tag{6.46}$$

From Equations (6.44) and (6.45), we have

$$\begin{aligned}
\frac{\partial x_1^{(2)}}{\partial v_1^{(2)}} &= \varphi'\left(v_1^{(2)}\right) \\
\frac{\partial x_2^{(2)}}{\partial v_2^{(2)}} &= \varphi'\left(v_2^{(2)}\right) \\
\frac{\partial v_1^{(2)}}{\partial x_2^{(1)}} &= w_{12}^{(2)} \\
\frac{\partial v_2^{(2)}}{\partial x_2^{(1)}} &= w_{22}^{(2)}
\end{aligned} \tag{6.47}$$

$$\begin{aligned}
\frac{\partial v_1^{(3)}}{\partial x_2^{(1)}} &= w_{11}^{(3)}\left(\frac{\partial x_1^{(2)}}{\partial v_1^{(2)}}\frac{\partial v_1^{(2)}}{\partial x_2^{(1)}}\right) + w_{12}^{(3)}\left(\frac{\partial x_2^{(2)}}{\partial v_2^{(2)}}\frac{\partial v_2^{(2)}}{\partial x_2^{(1)}}\right) \\
&= w_{11}^{(3)}\varphi'\left(v_1^{(2)}\right)w_{12}^{(1)} + w_{12}^{(3)}\varphi'\left(v_2^{(2)}\right)w_{22}^{(1)}
\end{aligned} \tag{6.48}$$

□

Example 6.2.4. Consider the three-layer MLP as shown in Figure 6.17, with the activation function $\varphi(v) = \tanh(v)$. Derive the expressions for all the $\Delta w_{ji}^{(l)}$ terms.

Solution: We first carry out the forward pass as follows.

$$\begin{aligned}
v_j^{(1)} &= \sum_{i=0}^{2} w_{ji}^{(1)} x_i^{(0)}, \quad x_j^{(1)} = \varphi\left(v_j^{(1)}\right), \quad j = 1, 2 \\
v_j^{(2)} &= \sum_{i=0}^{2} w_{ji}^{(2)} x_i^{(1)}, \quad x_j^{(2)} = \varphi\left(v_j^{(2)}\right), \quad j = 1, 2 \\
v_j^{(3)} &= \sum_{i=0}^{2} w_{ji}^{(3)} x_i^{(2)}, \quad x_j^{(3)} = \varphi\left(v_j^{(3)}\right), \quad j = 1
\end{aligned} \tag{6.49}$$

Next the backward pass is executed to yield the expressions for all the $\Delta w_{ji}^{(l)}$ terms,

i.e.,

$$\delta_1^{(3)} = \left(d_1 - x_1^{(3)}\right)\varphi'\left(v_1^{(3)}\right) \tag{6.50}$$

$$\delta_1^{(2)} = \left(\sum_{m=1}^{1} w_{m1}^{(3)}\delta_m^{(3)}\right)\varphi'\left(v_1^{(2)}\right) = w_{11}^{(3)}\delta_1^{(3)}x_1^{(2)}\left(1 - x_1^{(2)}\right) \tag{6.51}$$

$$\delta_2^{(2)} = \left(\sum_{m=1}^{1} w_{m2}^{(3)}\delta_m^{(3)}\right)\varphi'\left(v_2^{(2)}\right) = w_{12}^{(3)}\delta_1^{(3)}x_2^{(2)}\left(1 - x_2^{(2)}\right) \tag{6.52}$$

$$\delta_1^{(1)} = \left(\sum_{m=0}^{2} w_{m1}^{(2)}\delta_m^{(2)}\right)\varphi'\left(v_1^{(1)}\right) \tag{6.53}$$

$$= \left(w_{11}^{(2)}\delta_1^{(2)} + w_{21}^{(2)}\delta_2^{(2)}\right)x_1^{(1)}\left(1 - x_1^{(1)}\right) \tag{6.54}$$

$$\delta_2^{(1)} = \left(\sum_{m=0}^{2} w_{m2}^{(2)}\delta_m^{(2)}\right)\varphi'\left(v_2^{(1)}\right) \tag{6.55}$$

$$= \left(w_{12}^{(2)}\delta_1^{(2)} + w_{22}^{(2)}\delta_2^{(2)}\right)x_2^{(1)}\left(1 - x_2^{(1)}\right) \tag{6.56}$$

Hence, the expressions for all the $\Delta w_{ji}^{(l)}$ terms are:

$$\Delta w_{10}^{(3)} = \eta\,\delta_1^{(3)} \quad \Delta w_{11}^{(3)} = \eta\,\delta_1^{(3)}x_1^{(2)} \quad \Delta w_{12}^{(3)} = \eta\,\delta_1^{(3)}x_2^{(2)}$$

$$\Delta w_{10}^{(2)} = \eta\,\delta_1^{(2)} \quad \Delta w_{11}^{(2)} = \eta\,\delta_1^{(2)}x_1^{(1)} \quad \Delta w_{12}^{(2)} = \eta\,\delta_1^{(2)}x_2^{(1)}$$

$$\Delta w_{20}^{(2)} = \eta\,\delta_2^{(2)} \quad \Delta w_{21}^{(2)} = \eta\,\delta_2^{(2)}x_1^{(1)} \quad \Delta w_{22}^{(2)} = \eta\,\delta_2^{(2)}x_2^{(1)}$$

$$\Delta w_{10}^{(1)} = \eta\,\delta_1^{(1)} \quad \Delta w_{11}^{(1)} = \eta\,\delta_1^{(1)}x_1^{(0)} \quad \Delta w_{12}^{(1)} = \eta\delta_1^{(1)}x_2^{(0)}$$

$$\Delta w_{20}^{(1)} = \eta\,\delta_2^{(1)} \quad \Delta w_{21}^{(1)} = \eta\,\delta_2^{(1)}x_1^{(0)} \quad \Delta w_{22}^{(1)} = \eta\,\delta_2^{(1)}x_2^{(0)}$$

□

6.2.3 Implementation

Multilayer perceptrons have been used for finding good solutions to practical problems in many domains. Figures 6.18-6.20 illustrate three simple examples. In the example illustrated in Figure 6.18, a three-layer MLP is used to represent the nonlinear XOR relationship. The network consists of two input units, two hidden layers each with two units, and a single output unit. The MLP has 15 weights, including those due to the bias terms.

In the example illustrated in Figure 6.19, a MLP is used to recognize the scanned images of digits 0 to 9. The scanned image is discretized into a grid, with each cell in the

grid assigned a binary value of 0 (light) or 1 (dark) to form a feature vector for the image. A MLP is trained to map this feature vector into a 10-element binary output vector, whose only non-zero element indicates the digit being recognized.

In the example illustrated in Figure 6.20, a MLP is used to model the relationship between a set of biophysical variables and the body fat percentage of a person. Measurements of these biophysical variables and the body fat percentage of a group of people are first collected and used to train the MLP. Once trained, the MLP is used to generate prediction about the body fat percentage of a non-sampled person based on his/her biophysical measurements.

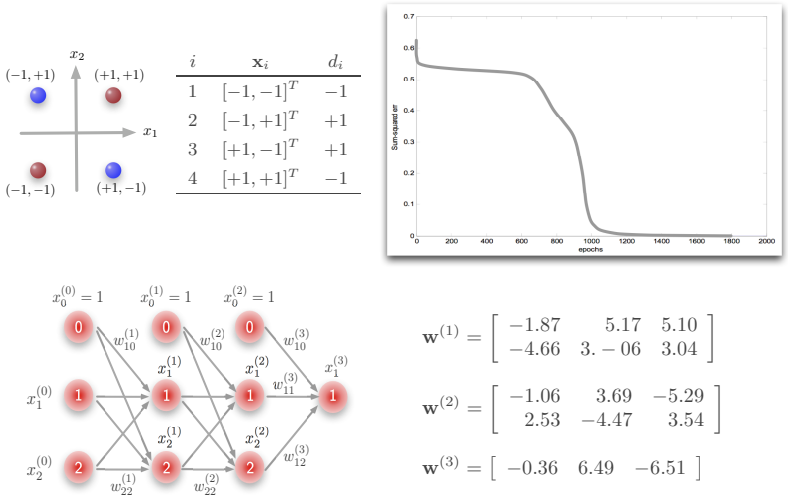


Figure 6.18: The XOR problem.

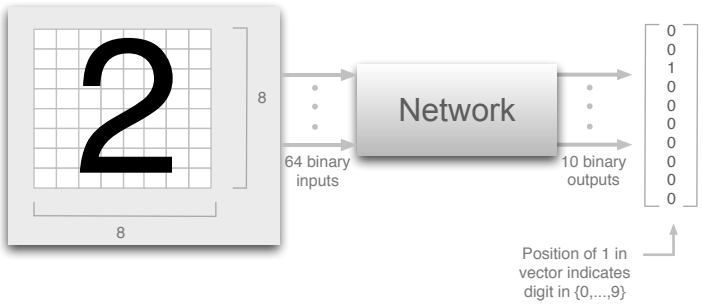


Figure 6.19: Optical character recognition.

Various factors influence the operational characteristics of a MLP-based solution. We highlight three here. The first is the learning rate η. In the Delta Rule, the learning rate scales down the gradient of E with respect to a weight $w_{ji}^{(l)}$ to control the amount of change made to the weight, i.e.,

$$\Delta w_{ji}^{(l)}(k) = -\eta \left(\frac{\partial E(k)}{\partial w_{ji}^{(l)}(k)} \right) \tag{6.57}$$

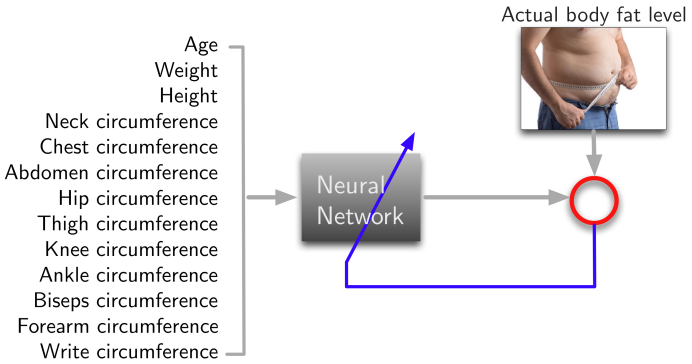
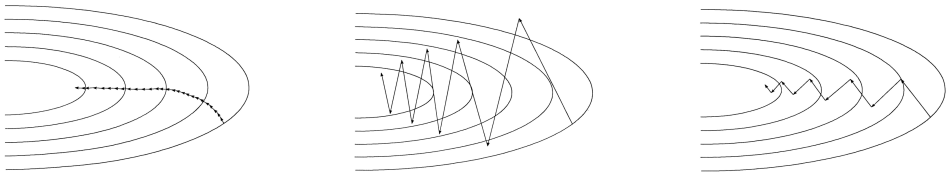


Figure 6.20: Human body fat prediction.

The role of the learning rate is to prevent changing the weight too drastically such that the gradient decent may, instead of converge to, oscillate significantly around a "valley" (see Figure 6.10) where the energy function E has a minimum. For this reason η is usually set to a small value, i.e., $0 < \eta < 1$, but too small a value tends to curtail the speed of the gradient descent towards a minimum of E. Figure 6.21 illustrates three typical scenarios qualitatively, where the inner circle represents the region in which a minimum of E is located and the zig-zag lines reflect the degree of weight change during learning. While a small learning rate results in a longer time to convergence, a large learning rate results in oscillatory behavior and may lead to instability of the learning process. The preferred scenario is one where the oscillation abates as the gradient decent approaches a minimum of E.



(a) η too small (b) η too large (c) η preferred

Figure 6.21: Qualitative description of convergence behavior. The inner circle represents the region where a minimum of E is located. The zig-zag lines reflect the degree of weight change during learning.

The second factor is the presence of local minima in the energy function E. Recall from Equations (6.17) that in general, E is a function of $\mathbf{W}$ and the examples $(\mathbf{x}_k, \mathbf{d}_k)$ in the training dataset, i.e.,

$$E(k) = h\Big(\mathbf{x}_k, \mathbf{d}_k, \mathbf{W}(k)\Big) \tag{6.58}$$

If a set of weight values $\mathbf{W}^*$ found by gradient decent is globally optimal, then we have

$$E(k) = h\Big(\mathbf{x}_k, \mathbf{d}_k, \mathbf{W}^*\Big) \to 0, \qquad \text{for all } k \in \{1, 2, \ldots, K\} \tag{6.59}$$

However, the error-backpropagation algorithm does not guarantee convergence to the global

minimum of $E = 0$. In fact, it is not uncommon that the learning process gets trapped in some local minima, which can be visualized as the "valleys" shown in Figure 6.22. (This figure is the same as Figure 6.10(a); we shown it again below for easy reference.)

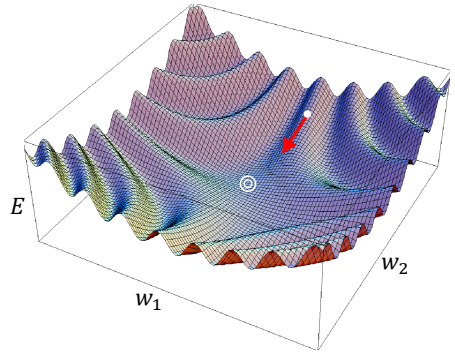


Figure 6.22: Energy function E containing multiple local minima.

The third factor that influences the operational characteristics of a MLP-based solution is the stopping criterion. Conceptually, if the gradient decent converges to the global minimum of E, then we can set $E = 0$ as the stopping criterion. However, convergence may occur at some local minimum, i.e., $E > 0$. In such cases, an empirical stopping criterion is required. Possible choices (or some combination of these) include:

1. Absolute rate of change in mean squared error per epoch is sufficiently small.
2. Change in the values of weights per epoch is sufficiently small.
3. The value of E of an epoch is less than some threshold. This is the criterion indicated in Table 6.1, where the parameter θ is the threshold.

When implementing a MLP for solving practical problems, the following set of issues need to be taken into consideration. There are no formal methods for resolving these issues analytically; their resolution often relies on experience and are highly dependent on the specific characteristics of the problem.

- *Number of hidden layers*: More hidden layers tend to give more accurate approximation but may also exacerbate problem of local minima.
- *Number of units in hidden layer*: Too few will lead to high training and generalization error due to under-fitting while too many may lead to low training error but still lead to high generalization error due to over-fitting.
- *Convergence is not guaranteed*: Although convergence can usually be achieved in practice by selecting a suitable learning rate and the initial values for the weights, there is no general proof that the learning process will converge.

- *Convergence to global minimum is not guaranteed*: Gradient descent (if convergent) only guarantees reduction of network output error down to a minimum, but does not guarantee that such a minimum is global.
- *Learning often takes a long time to converge*: Complex problems often need a large number epochs, from hundreds of thousands to even millions.
- *Network is essentially a "black box"*: It does not provide an intuitive (e.g., causal) explanation for the its computed result because what the MLP learns are operational parameters, not general abstract knowledge of a domain.

deep neural network

Neural networks with many hidden layers, where the layers are designed to successively process the input to extract (from the input) high-level features, is often described as *deep neural networks*. One example is a network that processes the raw pixel-level data of an image (i.e., the input) to produce a list of labels (i.e, the output, such as "car", "cat", "dog", etc.) for certain regions of that image. What makes a reinforcement learning method "deep" can be attributed to the use of such a deep neural network in the process of finding an optimal policy. Deep Q-Network is one such method.

6.3 The Concept of Deep Q-Network (DQN)

In Q-learning, we "look up" the value $Q(s_i, a_j)$ from the Q-table for a given state-action pair (s_i, a_j). Suppose that there are M actions that the agent can choose from at state s_i. We can think of the Q-table as a vector function, denoted by $f(\cdot)$, that when provided with a state s_i as input produces the vector $\mathbf{q} = [Q(s_i, a_1), Q(s_i, a_2), \ldots, Q(s_i, a_M)]^T$ as output, i.e.,

$$\mathbf{q} = \begin{bmatrix} Q(s_i, a_1) \\ Q(s_i, a_2) \\ \vdots \\ Q(s_i, a_M) \end{bmatrix} = f(s_i) \tag{6.60}$$

6.3.1 Approximating the Q-table

The key idea in the DQN method is to use a neural network to approximate $f(s_i)$, as is illustrated in Figure 6.23.

Because neural networks are universal approximators, we can be assured that a suitably designed neural network $g(s_i, \mathbf{W})$ is capable of approximating the function $f(s_i)$ that represents the Q-table as discussed earlier. Hence, we can set up a MLP and try to find a

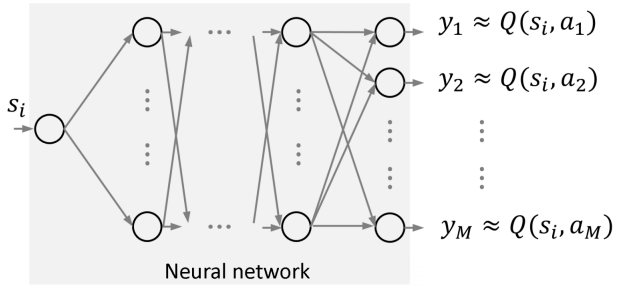


Figure 6.23: The approach used in a DQN for generating the output $Q(s_i, a)$ for all actions at a state s_i that is given as the input to the neural network.

set of optimal weights $\mathbf{W}^*$ such that

$$\mathbf{q} - \mathbf{y}_{\text{nn}} = f(s_i) - g(s_i, \mathbf{W}^*) = \begin{bmatrix} Q(s_i, a_1) \\ Q(s_i, a_2) \\ \vdots \\ Q(s_i, a_M) \end{bmatrix} - \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} \rightarrow \mathbf{0} \tag{6.61}$$

We are now in a position to address the question: *how does using a neural network enable us to avoid the intractable computation issue associated with searching the Q-table for the values of $Q(s_i, a_j)$*? The answer is that, for a given input s_i, computing the vector $\mathbf{y}_{\text{nn}}$ using a neural network $g(s_i, \mathbf{W}^*)$, i.e., $\mathbf{y}_{\text{nn}} = g(s_i, \mathbf{W}^*)$, is much less computationally intensive than using the original Q-table, because the size of the neural network (as reflected by $\mathbf{W}^*$) is much smaller than the size of the Q-table. For example, the size of the network used by DeepMind's *AlphaGO* (as listed in Table 6.2) may seem large but is still manageable using modern computers [6].

Table 6.2: Configuration of value network in AlphaGO.

Input layer	$19 \times 19 \times 49$
Hidden layer	$19 \times 19 \times 192$ (12 layers) 19×19 (1 layer) 256 (1 layer)
Output layer	1

When the state s_i is represented by low-level information (e.g., raw pixel data of an image), a deep neural network, as briefly described in the previous section, can be used to approximate the "Q-table function" $\mathbf{q} = f(s_i)$. This is the key idea in the Deep Q-Network method.

Besides having the inherent property of universal function approximation, a particular advantage of using a deep neural network to approximate the Q-table is that the network can implicitly determine the states s_i from inputs of a visual form, such as a set of images.

A popular type of such networks is the convolutional neural network (CNN). Figure 6.24 illustrates the use of a CNN to estimate the action values $Q(s, a)$ for a reinforcement-learning controller that plays the Atari games [14].

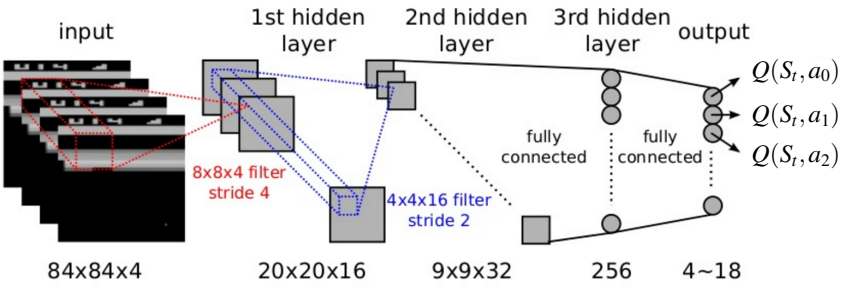


Figure 6.24: Using a CNN to learn the action values from a stack of four raw images [1].

6.3.2 Learning the action values

Recall that to train a MLP we need a training dataset $\{(\mathbf{x}_k, \mathbf{d}_k)\}$, where $\mathbf{x}_k$ is the input to the network and $\mathbf{d}_k$ is the desired output (or label) corresponding to $\mathbf{x}_k$. Hence, to train a MLP to approximate the Q-table, we need to specify the input to the network and the corresponding desired output.

The input to the network is the current state s_i (as we had discussed in Section 6.3.1). Let $g(s_i, \mathbf{W})$ denote the input-output mapping realized by the MLP for approximating the Q-table, then the output of the MLP can be expressed as

$$\mathbf{y}_{\text{nn}} = g(s_i, \mathbf{W}) \tag{6.62}$$

As was discussed in Section 6.3.1, $\mathbf{y}_{\text{nn}}$ by design is a vector containing the estimated action values at state s_i as generated by the network, i.e.,

$$\mathbf{y}_{\text{nn}} = \begin{bmatrix} Q(s_i, a_1) \\ Q(s_i, a_2) \\ \vdots \\ Q(s_i, a_j) \\ \vdots \\ Q(s_i, a_M) \end{bmatrix} = g(s_i, \mathbf{W}) \tag{6.63}$$

where M is the total number of actions at s. Next we need to specify the desired output

corresponding to the input s_i. Ideally the desired output of the network corresponding to the state s_i would be the true optimal action values $q_*(s_i, a_k)$ with $k = 1, 2, \ldots, M$, but we have no idea what those values are — otherwise the problem is already solved. However, we can exploit the update rule in Q-learning, i.e., Equation (5.5),

$$Q(s_i, a_j) \leftarrow Q(s_i, a_j) + \alpha \Big[\underbrace{R_{t+1} + \gamma \max_{a'} Q(s', a')}_{\text{estimated target value}} - Q(s_i, a_j) \Big] \tag{6.64}$$

(where s' is the next state reached by the agent upon taking action a_j at state s_i) and use the "estimated target value" as the estimated desired output for training the MLP. Specifically, let

desired output

$$\hat{Q}(s_i, a_j) = r + \gamma \max_{a'} Q(s', a') \tag{6.65}$$

Then we have the MLP training scheme as illustrated in Figure 6.25.

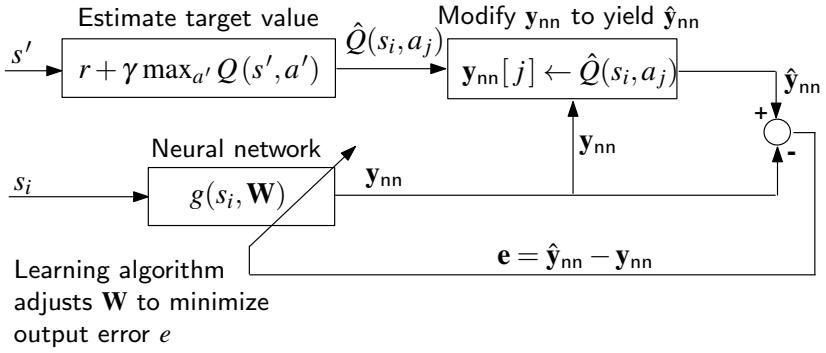


Figure 6.25: Neural network learning in DQN.

Training of the MLP proceeds as follows. We first initialize the weights $\mathbf{W}$ of the MLP to some arbitrary small values (usually zero), and starts Q-learning. At a non-terminal state s_i the agent takes action a_j, then finds itself in the new state $S_{t+1} = s'$ and receives a reward $R_{t+1} = r$. The state s_i is inputted into the MLP to generate the network output $\mathbf{y}_{nn} = g(s_i, \mathbf{W})$. A target value $\hat{Q}(s_i, a_j)$ for the action value $Q(s_i, a_j)$ is estimated as $\hat{Q}(s_i, a_j) = r + \gamma \max_{a'} Q(s', a')$ if s' is not a terminal state, and as $\hat{Q}(s_i, a_j) = r$ otherwise. The network output $\mathbf{y}_{nn}$ is then modified by replacing the j^{th} element of the vector $\mathbf{y}_{nn}$ with $\hat{Q}(s_i, a_j)$. This modified $\mathbf{y}_{nn}$ (now denoted by $\hat{\mathbf{y}}_{nn}$) is used as the desired output of the MLP in the error-backpropagation algorithm to update the weights $\mathbf{W}$. This process repeats until the weights $\mathbf{W}$ converge to some optimal values $\mathbf{W}^*$, in which case

training a DQN

we consider the MLP has been trained to approximate the Q-table, i.e.,

$$\mathbf{y}_{\mathsf{nn}} \triangleq \begin{bmatrix} Q(s,a_1) \\ Q(s,a_2) \\ \vdots \\ Q(s,a_M) \end{bmatrix} = g(s,\mathbf{W}^*) \approx \begin{bmatrix} q_*(s,a_1) \\ q_*(s,a_2) \\ \vdots \\ q_*(s,a_M) \end{bmatrix}, \quad \text{for all } s \in \mathcal{S} \tag{6.66}$$

Note that, in the learning scheme as illustrated in Figure 6.25, the estimated target value $\hat{Q}(s,a_i)$ could change from one time step to the next. Conducting learning with such a (possibly) constantly changing target may lead to unstable behavior. This is an important issue that can cause a Deep-Q-Network to perform poorly in its application. The technique of *experience replay* was proposed by the team at DeepMind to deal with this issue [14]. It represents a key contribution to the major advance in the field of reinforcement learning in recent years. This technique will be presented in Part II of this module, where the applications covered therein provide a rich context for its presentation.

experience replay

Bibliography

[1] https://wiki.math.uwaterloo.ca/statwiki/index.php?title=File:DQN_arch.png

[2] https://github.com/jassiay/CliffWalking/blob/master/cliff2.py

[3] https://www.youtube.com/watch?time_continue=230&v=Vto8n9C7DSQ&feature=emb_logo

[4] https://en.wikipedia.org/wiki/L%C3%A1trabjarg

[5] https://tromp.github.io/go/gostate.pdf

[6] http://ktiml.mff.cuni.cz/~bartak/ui_seminar/talks/2017ZS/YuuSakaguchi_AlphaGo.pdf

[7] Cybenko, G. "Approximations by superpositions of sigmoidal functions", Mathematics of Control, Signals, and Systems, 2(4), 303–314, 1989.

[8] Eddington, A.S. The Constants of Nature. In J. R. Newman (ed.) *The World of Mathematics*. Simon & Schuster, 1074–1093, 1956.

[9] Istratescu, V. I. *Fixed Point Theory: An Introduction*. D.Reidel, the Netherlands, 1981.

[10] Kakade, S. and Langford, J. Approximately optimal approximate reinforcement learning. *Proc. of the 19th International Conference On Machine Learning*, vol. 2, 267–274, 2002.

[11] LeCun, Y., Bengio, Y. and Hinton, G. Deep learning. *Nature*, 521, 436–444, 2015.

[12] MathWorks, Introducing Deep Learning with MATLAB. https://www.mathworks.com/content/dam/mathworks/tag-team/Objects/d/80879v00_Deep_Learning_ebook.pdf

[13] Minsky, M. and Papert, S. *Perceptrons: An Introduction to Computational Geometry*. 2ed, MIT Press, 1987.

[14] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M.A., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S.

and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[15] Puterman, M.L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. 1st ed., John Wiley & Sons, Inc., 1994.

[16] Robbins, H. and Monro, S. A Stochastic Approximation Method. *Annals of Mathematical Statistics*, 22 (3): 400-407, 1951.

[17] Rosenblatt, F. *The Perceptron – a perceiving and recognizing automaton*. Report 85-460-1. Cornell Aeronautical Laboratory, 1957.

[18] Rumelhart, D.E., McClelland, J.L. and the PDP Research Group. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Volume 1: Foundations, Cambridge, Massachusetts: MIT Press, 1986.

[19] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Driessche, G.V., Graepel, T. and Hassabis, D. Mastering the game of Go without human knowledge. *Nature*, 550 (7676): 354–359, 2017.

[20] Singh, S.P. and Sutton, R.S. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158, 1996.

[21] Singh, S., Jaakkola, T., Littman, M.L. and Szepesvari, C. Convergence results for single-step on-policy reinforcement-learning algorithms, *Machine Learning*, 38, 287–308, 2000.

[22] Sutton, Richard S. and Barto, Andrew G. *Reinforcement Learning: An Introduction*, MIT Press, 2ed., 2018.

[23] Thompson, W. R. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25:285–294, 1933.

[24] Tokic, M., Ertel, W., and Fessler, J. The crawler, a class room demonstrator for reinforcement learning. *Proc. of the* 22^{nd} *International Florida Artificial Intelligence Research Society Conference*, 160-165, 2009. (Video link on YouTube: https://www.youtube.com/watch?v=2VjNTQ5cGzM.)

[25] Tromp J. and Farneback G. Combinatorics of Go. In: van den Herik H.J., Ciancarini P., Donkers H.H.L.M.. (eds) *Computers and Games*, Lecture Notes in Computer Science, vol. 4630. Springer, Berlin, Heidelberg, 2007.

[26] Van Kampen, N. G., Remarks on non-Markov processes. *Brazilian Journal of Physics*, vol. 28, no. 2, 90-96, 1998.