Chapter 8

Value Function Approximation

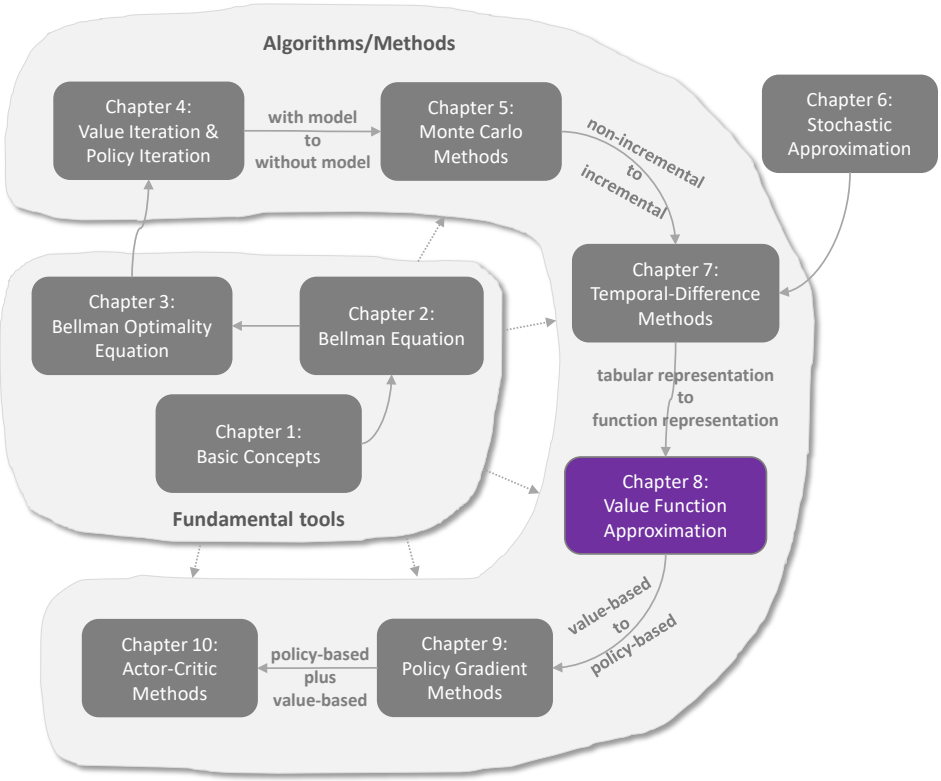


Figure 8.1: Where we are in this book.

In this chapter, we continue to study temporal-difference learning algorithms. However, a different method is used to represent state/action values. So far in this book, state/action values have been represented by *tables*. The tabular method is straightforward to understand, but it is inefficient for handling large state or action spaces. To solve this problem, this chapter introduces the function approximation method, which has become the standard way to represent values. It is also where artificial neural networks are incorporated into reinforcement learning as function approximators. The idea of function approximation can also be extended from representing *values* to representing *policies*, as introduced in Chapter 9.

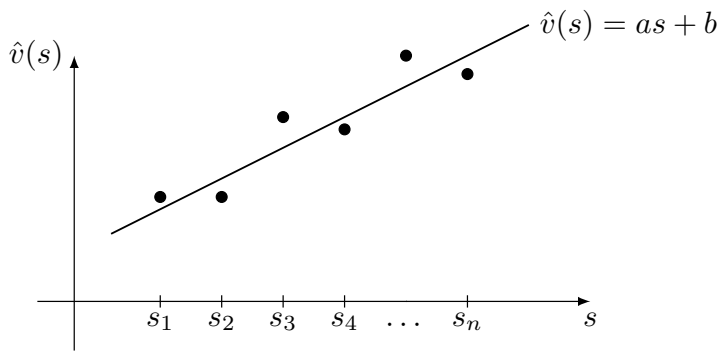


Figure 8.2: An illustration of the function approximation method. The x-axis and y-axis correspond to s and $\hat{v}(s)$, respectively.

8.1 Value representation: From table to function

We next use an example to demonstrate the difference between the tabular and function approximation methods.

Suppose that there are n states $\{s_i\}_{i=1}^n$, whose state values are $\{v_\pi(s_i)\}_{i=1}^n$. Here, π is a given policy. Let $\{\hat{v}(s_i)\}_{i=1}^n$ denote the estimates of the true state values. If we use the tabular method, the estimated values can be maintained in the following table. This table can be stored in memory as an array or a vector. To retrieve or update any value, we can directly read or rewrite the corresponding entry in the table.

State	s_1	s_2	$\cdots$	s_n
Estimated value	$\hat{v}(s_1)$	$\hat{v}(s_2)$	$\cdots$	$\hat{v}(s_n)$

We next show that the values in the above table can be approximated by a function. In particular, $\{(s_i, \hat{v}(s_i))\}_{i=1}^n$ are shown as n points in Figure 8.2. These points can be fitted or approximated by a curve. The simplest curve is a straight line, which can be described as

$$\hat{v}(s, w) = as + b = \underbrace{[s, 1]}_{\phi^T(s)} \underbrace{\begin{bmatrix} a \\ b \end{bmatrix}}_{w} = \phi^T(s)w. \tag{8.1}$$

Here, $\hat{v}(s, w)$ is a function for approximating $v_\pi(s)$. It is determined jointly by the state s and the parameter vector $w \in \mathbb{R}^2$. $\hat{v}(s, w)$ is sometimes written as $\hat{v}_w(s)$. Here, $\phi(s) \in \mathbb{R}^2$ is called the *feature vector* of s.

The first notable difference between the tabular and function approximation methods concerns how they retrieve and update a value.

- ◇ How to *retrieve* a value: When the values are represented by a table, if we want to retrieve a value, we can directly read the corresponding entry in the table. However,

when the values are represented by a function, it becomes slightly more complicated to retrieve a value. In particular, we need to input the state index s into the function and calculate the function value (Figure 8.3). For the example in (8.1), we first need to calculate the feature vector $\phi(s)$ and then calculate $\phi^T(s)w$. If the function is an artificial neural network, a forward propagation from the input to the output is needed.

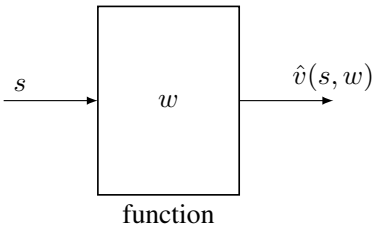


Figure 8.3: An illustration of the process for retrieving the value of s when using the function approximation method.

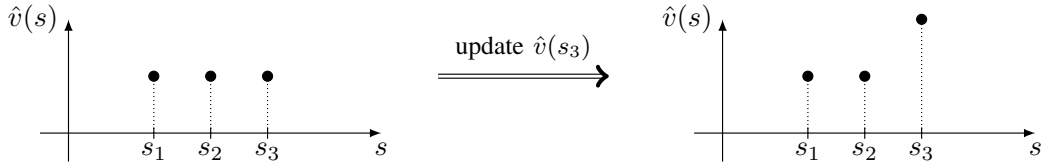
The function approximation method is more efficient in terms of storage due to the way in which the state values are retrieved. Specifically, while the tabular method needs to store n values, we now only need to store a lower dimensional parameter vector w. Thus, the storage efficiency can be significantly improved. Such a benefit is, however, *not* free. It comes with a cost: the state values may not be accurately represented by the function. For example, a straight line is not able to accurately fit the points in Figure 8.2. That is why this method is called approximation. From a fundamental point of view, some information will certainly be lost when we use a low-dimensional vector to represent a high-dimensional dataset. Therefore, the function approximation method enhances storage efficiency by sacrificing accuracy.

◇ How to *update* a value: When the values are represented by a table, if we want to update one value, we can directly rewrite the corresponding entry in the table. However, when the values are represented by a function, the way to update a value is completely different. Specifically, we must update w to change the values indirectly. How to update w to find optimal state values will be addressed in detail later.

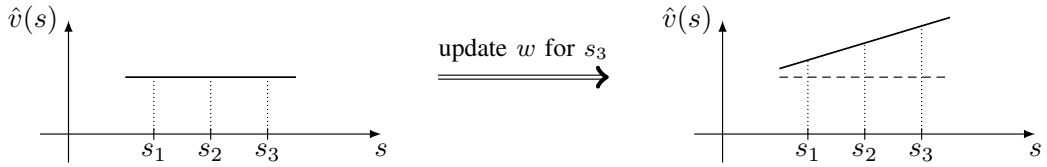
Thanks to the way in which the state values are updated, the function approximation method has another merit: its *generalization ability* is stronger than that of the tabular method. The reason is as follows. When using the tabular method, we can update a value if the corresponding state is visited in an episode. The values of the states that have not been visited cannot be updated. However, when using the function approximation method, we need to update w to update the value of a state. The update of w also affects the values of some other states even though these states have not been visited. Therefore, the experience sample for one state can generalize to help estimate the values of some other states.

The above analysis is illustrated in Figure 8.4, where there are three states $\{s_1, s_2, s_3\}$.

Suppose that we have an experience sample for s_3 and would like to update $\hat{v}(s_3)$. When using the tabular method, we can only update $\hat{v}(s_3)$ without changing $\hat{v}(s_1)$ or $\hat{v}(s_2)$, as shown in Figure 8.4(a). When using the function approximation method, updating w not only can update $\hat{v}(s_3)$ but also would change $\hat{v}(s_1)$ and $\hat{v}(s_2)$, As shown in Figure 8.4(b). Therefore, the experience sample of s_3 can help update the values of its neighboring states.



(a) Tabular method: when $\hat{v}(s_2)$ is updated, the other values remain the same.



(b) Function approximation method: when we update $\hat{v}(s_2)$ by changing w, the values of the neighboring states are also changed.

Figure 8.4: An illustration of how to update the value of a state.

We can use more complex functions that have stronger approximation abilities than straight lines. For example, consider a second-order polynomial:

$$\hat{v}(s,w) = as^2 + bs + c = \underbrace{[s^2, s, 1]}_{\phi^T(s)} \underbrace{\begin{bmatrix} a \\ b \\ c \end{bmatrix}}_{w} = \phi^T(s)w. \tag{8.2}$$

We can use even higher-order polynomial curves to fit the points. As the order of the curve increases, the approximation accuracy can be improved, but the dimension of the parameter vector also increases, requiring more storage and computational resources.

Note that $\hat{v}(s,w)$ in either (8.1) or (8.2) is *linear* in w (though it may be nonlinear in s). This type of method is called *linear function approximation*, which is the simplest function approximation method. To realize linear function approximation, we need to select an appropriate feature vector $\phi(s)$. That is, we must decide, for example, whether we should use a first-order straight line or a second-order curve to fit the points. The selection of appropriate feature vectors is nontrivial. It requires prior knowledge of the given task: the better we understand the task, the better the feature vectors we can select. For instance, if we know that the points in Figure 8.2 are approximately located on a

straight line, we can use a straight line to fit the points. However, such prior knowledge is usually unknown in practice. If we do not have any prior knowledge, a popular solution is to use artificial neural networks as nonlinear function approximations.

Another important problem is how to find the optimal parameter vector. If we know $\{v_\pi(s_i)\}_{i=1}^n$, this is a least-squares problem. The optimal parameter can be obtained by optimizing the following objective function:

$$J_1 = \sum_{i=1}^{n} \left(\hat{v}(s_i, w) - v_\pi(s_i)\right)^2 = \sum_{i=1}^{n} \left(\phi^T(s_i)w - v_\pi(s_i)\right)^2$$
$$= \left\| \begin{bmatrix} \phi^T(s_1) \\ \vdots \\ \phi^T(s_n) \end{bmatrix} w - \begin{bmatrix} v_\pi(s_1) \\ \vdots \\ v_\pi(s_n) \end{bmatrix} \right\|^2 \doteq \|\Phi w - v_\pi\|^2,$$

where

$$\Phi \doteq \begin{bmatrix} \phi^T(s_1) \\ \vdots \\ \phi^T(s_n) \end{bmatrix} \in \mathbb{R}^{n\times 2}, \qquad v_\pi \doteq \begin{bmatrix} v_\pi(s_1) \\ \vdots \\ v_\pi(s_n) \end{bmatrix} \in \mathbb{R}^n.$$

It can be verified that the optimal solution to this least-squares problem is

$$w^* = (\Phi^T\Phi)^{-1}\Phi v_\pi.$$

More information about least-squares problems can be found in [47, Section 3.3] and [48, Section 5.14].

The curve-fitting example presented in this section illustrates the basic idea of value function approximation. This idea will be formally introduced in the next section.

8.2 TD learning of state values based on function approximation

In this section, we show how to integrate the function approximation method into TD learning to estimate the state values of a given policy. This algorithm will be extended to learn action values and optimal policies in Section 8.3.

This section contains quite a few subsections and many coherent contents. It is better for us to review the contents first before diving into the details.

- ◇ The function approximation method is formulated as an optimization problem. The objective function of this problem is introduced in Section 8.2.1. The TD learning algorithm for optimizing this objective function is introduced in Section 8.2.2.

- ◇ To apply the TD learning algorithm, we need to select appropriate feature vectors. Section 8.2.3 discusses this problem.
- ◇ Examples are given in Section 8.2.4 to demonstrate the TD algorithm and the impacts of different feature vectors.
- ◇ A theoretical analysis of the TD algorithm is given in Section 8.2.5. This subsection is mathematically intensive. Readers may read it selectively based on their interests.

8.2.1 Objective function

Let $v_\pi(s)$ and $\hat{v}(s,w)$ be the true state value and approximated state value of $s \in \mathcal{S}$, respectively. The problem to be solved is to find an *optimal* w so that $\hat{v}(s,w)$ can best approximate $v_\pi(s)$ for every s. In particular, the objective function is

$$J(w) = \mathbb{E}[(v_\pi(S) - \hat{v}(S,w))^2], \tag{8.3}$$

where the expectation is calculated with respect to the random variable $S \in \mathcal{S}$. While S is a random variable, what is its probability distribution? This question is important for understanding this objective function. There are several ways to define the probability distribution of S.

- ◇ The first way is to use a *uniform distribution*. That is to treat all the states as *equally important* by setting the probability of each state to $1/n$. In this case, the objective function in (8.3) becomes

$$J(w) = \frac{1}{n}\sum_{s\in\mathcal{S}}(v_\pi(s) - \hat{v}(s,w))^2, \tag{8.4}$$

  which is the average value of the approximation errors of all the states. However, this way does not consider the real dynamics of the Markov process under the given policy. Since some states may be rarely visited by a policy, it may be unreasonable to treat all the states as equally important.

- ◇ The second way, which is the focus of this chapter, is to use the *stationary distribution*. The stationary distribution describes the *long-term* behavior of a Markov decision process. More specifically, after the agent executes a given policy for a sufficiently long period, the probability of the agent being located at any state can be described by this stationary distribution. Interested readers may see the details in Box 8.1.

  Let $\{d_\pi(s)\}_{s\in\mathcal{S}}$ denote the stationary distribution of the Markov process under policy π. That is, the probability for the agent visiting s after a long period of time is $d_\pi(s)$. By definition, $\sum_{s\in\mathcal{S}} d_\pi(s) = 1$. Then, the objective function in (8.3) can be rewritten

as

$$J(w) = \sum_{s \in \mathcal{S}} d_\pi(s)(v_\pi(s) - \hat{v}(s, w))^2, \tag{8.5}$$

which is a weighted average of the approximation errors. The states that have higher probabilities of being visited are given greater weights.

It is notable that the value of $d_\pi(s)$ is nontrivial to obtain because it requires knowing the state transition probability matrix P_π (see Box 8.1). Fortunately, we do not need to calculate the specific value of $d_\pi(s)$ to minimize this objective function as shown in the next subsection. In addition, it was assumed that the number of states was *finite* when we introduced (8.4) and (8.5). When the state space is continuous, we can replace the summations with integrals.

Box 8.1: Stationary distribution of a Markov decision process

The key tool for analyzing stationary distribution is $P_\pi \in \mathbb{R}^{n \times n}$, which is the probability transition matrix under the given policy π. If the states are indexed as $s_1, \ldots, s_n$, then $[P_\pi]_{ij}$ is defined as the probability for the agent moving from s_i to s_j. The definition of P_π can be found in Section 2.6.

◇ Interpretation of P_π^k $(k = 1, 2, 3, \ldots)$.

First of all, it is necessary to examine the interpretation of the entries in P_π^k. The probability of the agent transitioning from s_i to s_j using exactly k steps is denoted as

$$p_{ij}^{(k)} = \Pr(S_{t_k} = j | S_{t_0} = i),$$

where t_0 and t_k are the initial and kth time steps, respectively. First, by the definition of P_π, we have

$$[P_\pi]_{ij} = p_{ij}^{(1)},$$

which means that $[P_\pi]_{ij}$ is the probability of transitioning from s_i to s_j using *a single step*. Second, consider P_π^2. It can be verified that

$$[P_\pi^2]_{ij} = [P_\pi P_\pi]_{ij} = \sum_{q=1}^{n} [P_\pi]_{iq}[P_\pi]_{qj}.$$

Since $[P_\pi]_{iq}[P_\pi]_{qj}$ is the joint probability of transitioning from s_i to s_q and then from s_q to s_j, we know that $[P_\pi^2]_{ij}$ is the probability of transitioning from s_i to s_j

using *exactly two steps*. That is

$$[P_\pi^2]_{ij} = p_{ij}^{(2)}.$$

Similarly, we know that

$$[P_\pi^k]_{ij} = p_{ij}^{(k)},$$

which means that $[P_\pi^k]_{ij}$ is the probability of transitioning from s_i to s_j using *exactly k steps*.

◇ Definition of stationary distributions.

Let $d_0 \in \mathbb{R}^n$ be a vector representing the probability distribution of the states at the initial time step. For example, if s is always selected as the starting state, then $d_0(s) = 1$ and the other entries of d_0 are 0. Let $d_k \in \mathbb{R}^n$ be the vector representing the probability distribution obtained after exactly k steps starting from d_0. Then, we have

$$d_k(s_i) = \sum_{j=1}^{n} d_0(s_j)[P_\pi^k]_{ji}, \quad i = 1, 2, \ldots. \tag{8.6}$$

This equation indicates that the probability of the agent visiting s_i at step k equals the sum of the probabilities of the agent transitioning from $\{s_j\}_{j=1}^n$ to s_i using exactly k steps. The matrix-vector form of (8.6) is

$$d_k^T = d_0^T P_\pi^k. \tag{8.7}$$

When we consider the long-term behavior of the Markov process, it holds under certain conditions that

$$\lim_{k\to\infty} P_\pi^k = \mathbf{1}_n d_\pi^T, \tag{8.8}$$

where $\mathbf{1}_n = [1, \ldots, 1]^T \in \mathbb{R}^n$ and $\mathbf{1}_n d_\pi^T$ is a constant matrix with all its rows equal to d_π^T. The conditions under which (8.8) is valid will be discussed later. Substituting (8.8) into (8.7) yields

$$\lim_{k\to\infty} d_k^T = d_0^T \lim_{k\to\infty} P_\pi^k = d_0^T \mathbf{1}_n d_\pi^T = d_\pi^T, \tag{8.9}$$

where the last equality is valid because $d_0^T \mathbf{1}_n = 1$.

Equation (8.9) means that the state distribution d_k converges to a constant value d_π, which is called the *limiting distribution*. The limiting distribution depends

on the system model and the policy π. Interestingly, it is independent of the initial distribution d_0. That is, regardless of which state the agent starts from, the probability distribution of the agent after a sufficiently long period can always be described by the limiting distribution.

The value of d_π can be calculated in the following way. Taking the limit of both sides of $d_k^T = d_{k-1}^T P_\pi$ gives $\lim_{k\to\infty} d_k^T = \lim_{k\to\infty} d_{k-1}^T P_\pi$ and hence

$$d_\pi^T = d_\pi^T P_\pi. \tag{8.10}$$

As a result, d_π is the left eigenvector of P_π associated with the eigenvalue 1. The solution of (8.10) is called the stationary distribution. It holds that $\sum_{s\in\mathcal{S}} d_\pi(s) = 1$ and $d_\pi(s) > 0$ for all $s \in \mathcal{S}$. The reason why $d_\pi(s) > 0$ (not $d_\pi(s) \geq 0$) will be explained later.

◇ Conditions for the uniqueness of stationary distributions.

The solution d_π of (8.10) is usually called a stationary distribution, whereas the distribution d_π in (8.9) is usually called the limiting distribution. Note that (8.9) implies (8.10), but the converse may not be true. A general class of Markov processes that have unique stationary (or limiting) distributions is *irreducible* (or *regular*) Markov processes. Some necessary definitions are given below. More details can be found in [49, Chapter IV].

- State s_j is said to be *accessible* from state s_i if there exists a finite integer k so that $[P_\pi]_{ij}^k > 0$, which means that the agent starting from s_i can always reach s_j after a finite number of transitions.
- If two states s_i and s_j are mutually accessible, then the two states are said to *communicate*.
- A Markov process is called *irreducible* if all of its states communicate with each other. In other words, the agent starting from an arbitrary state can always reach any other state within a finite number of steps. Mathematically, it indicates that, for any s_i and s_j, there exists $k \geq 1$ such that $[P_\pi^k]_{ij} > 0$ (the value of k may vary for different i, j).
- A Markov process is called *regular* if there exists $k \geq 1$ such that $[P_\pi^k]_{ij} > 0$ for all i, j. Equivalently, there exists $k \geq 1$ such that $P_\pi^k > 0$, where $>$ is elementwise. As a result, every state is reachable from any other state within at most k steps. A regular Markov process is also irreducible, but the converse is not true. However, if a Markov process is irreducible and there exists i such that $[P_\pi]_{ii} > 0$, then it is also regular. Moreover, if $P_\pi^k > 0$, then $P_\pi^{k'} > 0$ for any $k' \geq k$ since $P_\pi \geq 0$. It then follows from (8.9) that $d_\pi(s) > 0$ for every s.

◇ Policies that may lead to unique stationary distributions.

Once the policy is given, a Markov decision process becomes a Markov process, whose long-term behavior is jointly determined by the given policy and the system model. Then, an important question is what kind of policies can lead to regular Markov processes? In general, the answer is *exploratory policies* such as ϵ-greedy policies. That is because an exploratory policy has a positive probability of taking any action at any state. As a result, the states can communicate with each other when the system model allows them to do so.

◇ An example is given in Figure 8.5 to illustrate stationary distributions. The policy in this example is ϵ-greedy with $\epsilon = 0.5$. The states are indexed as s_1, s_2, s_3, s_4, which correspond to the top-left, top-right, bottom-left, and bottom-right cells in the grid, respectively.

We compare two methods to calculate the stationary distributions. The first method is to solve (8.10) to get the theoretical value of d_π. The second method is to estimate d_π numerically: we start from an arbitrary initial state and generate a sufficiently long episode by following the given policy. Then, d_π can be estimated by the ratio between the number of times each state is visited in the episode and the total length of the episode. The estimation result is more accurate when the episode is longer. We next compare the theoretical and estimated results.

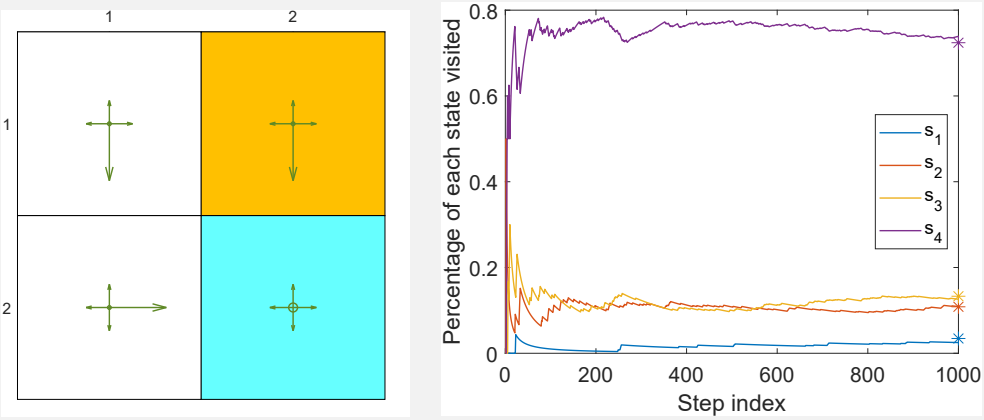


Figure 8.5: Long-term behavior of an ϵ-greedy policy with $\epsilon = 0.5$. The asterisks in the right figure represent the theoretical values of the elements of d_π.

- Theoretical value of d_π: It can be verified that the Markov process induced by the policy is both irreducible and regular. That is due to the following reasons. First, since all the states communicate, the resulting Markov process is irreducible. Second, since every state can transition to itself, the resulting

Markov process is regular. It can be seen from Figure 8.5 that

$$P_\pi^T = \begin{bmatrix} 0.3 & 0.1 & 0.1 & 0 \\ 0.1 & 0.3 & 0 & 0.1 \\ 0.6 & 0 & 0.3 & 0.1 \\ 0 & 0.6 & 0.6 & 0.8 \end{bmatrix}.$$

The eigenvalues of P_π^T can be calculated as $\{-0.0449, 0.3, 0.4449, 1\}$. The unit-length (right) eigenvector of P_π^T corresponding to the eigenvalue 1 is $[0.0463, 0.1455, 0.1785, 0.9720]^T$. After scaling this vector so that the sum of all its elements is equal to 1, we obtain the theoretical value of d_π as follows:

$$d_\pi = \begin{bmatrix} 0.0345 \\ 0.1084 \\ 0.1330 \\ 0.7241 \end{bmatrix}.$$

The ith element of d_π corresponds to the probability of the agent visiting s_i in the long run.

- Estimated value of d_π: We next verify the above theoretical value of d_π by executing the policy for sufficiently many steps in the simulation. Specifically, we select s_1 as the starting state and run 1,000 steps by following the policy. The proportion of the visits of each state during the process is shown in Figure 8.5. It can be seen that the proportions converge to the theoretical value of d_π after hundreds of steps.

8.2.2 Optimization algorithms

To minimize the objective function $J(w)$ in (8.3), we can use the gradient descent algorithm:

$$w_{k+1} = w_k - \alpha_k \nabla_w J(w_k),$$

where

$$\begin{aligned} \nabla_w J(w_k) &= \nabla_w \mathbb{E}[(v_\pi(S) - \hat{v}(S, w_k))^2], \\ &= \mathbb{E}[\nabla_w (v_\pi(S) - \hat{v}(S, w_k))^2] \\ &= 2\mathbb{E}[(v_\pi(S) - \hat{v}(S, w_k))(-\nabla_w \hat{v}(S, w_k))] \\ &= -2\mathbb{E}[(v_\pi(S) - \hat{v}(S, w_k))\nabla_w \hat{v}(S, w_k)]. \end{aligned}$$

Therefore, the gradient descent algorithm is

$$w_{k+1} = w_k + 2\alpha_k \mathbb{E}[(v_\pi(S) - \hat{v}(S, w_k))\nabla_w \hat{v}(S, w_k)], \tag{8.11}$$

where the coefficient 2 before α_k can be merged into α_k without loss of generality. The algorithm in (8.11) requires calculating the expectation. In the spirit of stochastic gradient descent, we can replace the true gradient with a stochastic gradient. Then, (8.11) becomes

$$w_{t+1} = w_t + \alpha_t\big(v_\pi(s_t) - \hat{v}(s_t, w_t)\big)\nabla_w \hat{v}(s_t, w_t), \tag{8.12}$$

where s_t is a sample of S at time t.

Notably, (8.12) is *not* implementable because it requires the true state value v_π, which is unknown and must be estimated. We can replace $v_\pi(s_t)$ with an approximation to make the algorithm implementable. The following two methods can be used to do so.

- ◇ Monte Carlo method: Suppose that we have an episode $(s_0, r_1, s_1, r_2, \dots)$. Let g_t be the discounted return starting from s_t. Then, g_t can be used as an approximation of $v_\pi(s_t)$. The algorithm in (8.12) becomes

  $$w_{t+1} = w_t + \alpha_t\big(g_t - \hat{v}(s_t, w_t)\big)\nabla_w \hat{v}(s_t, w_t).$$

  This is the algorithm of Monte Carlo learning with function approximation.

- ◇ Temporal-difference method: In the spirit of TD learning, $r_{t+1} + \gamma\hat{v}(s_{t+1}, w_t)$ can be used as an approximation of $v_\pi(s_t)$. The algorithm in (8.12) becomes

  $$w_{t+1} = w_t + \alpha_t \left[r_{t+1} + \gamma\hat{v}(s_{t+1}, w_t) - \hat{v}(s_t, w_t)\right]\nabla_w \hat{v}(s_t, w_t). \tag{8.13}$$

  This is the algorithm of TD learning with function approximation. This algorithm is summarized in Algorithm 8.1.

Understanding the TD algorithm in (8.13) is important for studying the other algorithms in this chapter. Notably, (8.13) can only learn the *state values* of a given policy. It will be extended to algorithms that can learn *action values* in Sections 8.3.1 and 8.3.2.

8.2.3 Selection of function approximators

To apply the TD algorithm in (8.13), we need to select appropriate $\hat{v}(s, w)$. There are two ways to do that. The first is to use an artificial neural network as a *nonlinear* function approximator. The input of the neural network is the state, the output is $\hat{v}(s, w)$, and the network parameter is w. The second is to simply use a *linear* function:

$$\hat{v}(s, w) = \phi^T(s)w,$$

Algorithm 8.1: TD learning of state values with function approximation

Initialization: A function $\hat{v}(s, w)$ that is a differentiable in w. Initial parameter w_0.
Goal: Learn the true state values of a given policy π.

For each episode $\{(s_t, r_{t+1}, s_{t+1})\}_t$ generated by π, do
    For each sample (s_t, r_{t+1}, s_{t+1}), do
        In the general case, $w_{t+1} = w_t + \alpha_t \left[r_{t+1} + \gamma \hat{v}(s_{t+1}, w_t) - \hat{v}(s_t, w_t)\right] \nabla_w \hat{v}(s_t, w_t)$
        In the linear case, $w_{t+1} = w_t + \alpha_t \left[r_{t+1} + \gamma \phi^T(s_{t+1}) w_t - \phi^T(s_t) w_t\right] \phi(s_t)$

where $\phi(s) \in \mathbb{R}^m$ is the feature vector of s. The lengths of $\phi(s)$ and w are equal to m, which is usually much smaller than the number of states. In the linear case, the gradient is

$$\nabla_w \hat{v}(s, w) = \phi(s),$$

Substituting which into (8.13) yields

$$w_{t+1} = w_t + \alpha_t \left[r_{t+1} + \gamma \phi^T(s_{t+1}) w_t - \phi^T(s_t) w_t\right] \phi(s_t). \tag{8.14}$$

This is the algorithm of TD learning with linear function approximation. We call it *TD-Linear* for short.

The linear case is much better understood in theory than the nonlinear case. However, its approximation ability is limited. It is also nontrivial to select appropriate feature vectors for complex tasks. By contrast, artificial neural networks can approximate values as black-box universal nonlinear approximators, which are more friendly to use.

Nevertheless, it is still meaningful to study the linear case. A better understanding of the linear case can help readers better grasp the idea of the function approximation method. Moreover, the linear case is sufficient for solving the simple grid world tasks considered in this book. More importantly, the linear case is still powerful in the sense that the tabular method can be viewed as a special linear case. More information can be found in Box 8.2.

Box 8.2: Tabular TD learning is a special case of TD-Linear

We next show that the tabular TD algorithm in (7.1) in Chapter 7 is a special case of the TD-Linear algorithm in (8.14).

Consider the following special feature vector for any $s \in \mathcal{S}$:

$$\phi(s) = e_s \in \mathbb{R}^n,$$

where e_s is the vector with the entry corresponding to s equal to 1 and the other

entries equal to 0. In this case,

$$\hat{v}(s,w) = e_s^T w = w(s),$$

where $w(s)$ is the entry in w that corresponds to s. Substituting the above equation into (8.14) yields

$$w_{t+1} = w_t + \alpha_t\big(r_{t+1} + \gamma w_t(s_{t+1}) - w_t(s_t)\big)e_{s_t}.$$

The above equation merely updates the entry $w_t(s_t)$ due to the definition of e_{s_t}. Motivated by this, multiplying $e_{s_t}^T$ on both sides of the equation yields

$$w_{t+1}(s_t) = w_t(s_t) + \alpha_t\big(r_{t+1} + \gamma w_t(s_{t+1}) - w_t(s_t)\big),$$

which is exactly the tabular TD algorithm in (7.1).

In summary, by selecting the feature vector as $\phi(s) = e_s$, the TD-Linear algorithm becomes the tabular TD algorithm.

8.2.4 Illustrative examples

We next present some examples for demonstrating how to use the TD-Linear algorithm in (8.14) to estimate the state values of a given policy. In the meantime, we demonstrate how to select feature vectors.

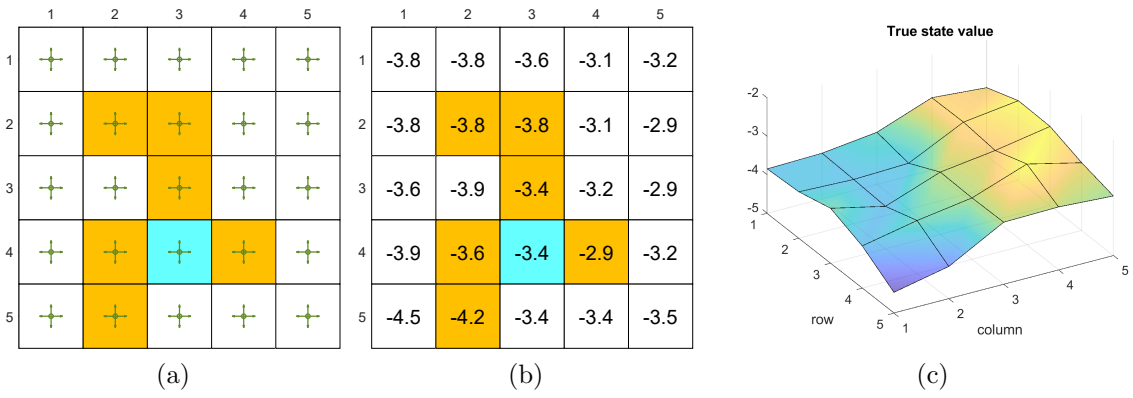


Figure 8.6: (a) The policy to be evaluated. (b) The true state values are represented as a table. (c) The true state values are represented as a 3D surface.

The grid world example is shown in Figure 8.6. The given policy takes any action at a state with a probability of 0.2. Our goal is to estimate the state values under this policy. There are 25 state values in total. The true state values are shown in Figure 8.6(b). The true state values are visualized as a three-dimensional surface in Figure 8.6(c).

We next show that we can use fewer than 25 parameters to approximate these state values. The simulation setup is as follows. Five hundred episodes are generated by the given policy. Each episode has 500 steps and starts from a randomly selected state-action pair following a uniform distribution. In addition, in each simulation trial, the parameter vector w is randomly initialized such that each element is drawn from a standard normal distribution with a zero mean and a standard deviation of 1. We set $r_{\text{forbidden}} = r_{\text{boundary}} = -1$, $r_{\text{target}} = 1$, and $\gamma = 0.9$.

To implement the TD-Linear algorithm, we need to select the feature vector $\phi(s)$ first. There are different ways to do that as shown below.

◇ The first type of feature vector is based on polynomials. In the grid world example, a state s corresponds to a 2D location. Let x and y denote the column and row indexes of s, respectively. To avoid numerical issues, we normalize x and y so that their values are within the interval of $[-1, +1]$. With a slight abuse of notation, the normalized values are also represented by x and y. Then, the simplest feature vector is

$$\phi(s) = \begin{bmatrix} x \\ y \end{bmatrix} \in \mathbb{R}^2.$$

In this case, we have

$$\hat{v}(s, w) = \phi^T(s)w = [x, y] \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = w_1 x + w_2 y.$$

When w is given, $\hat{v}(s, w) = w_1 x + w_2 y$ represents a 2D plane that passes through the origin. Since the surface of the state values may not pass through the origin, we need to introduce a bias to the 2D plane to better approximate the state values. To do that, we consider the following 3D feature vector:

$$\phi(s) = \begin{bmatrix} 1 \\ x \\ y \end{bmatrix} \in \mathbb{R}^3. \tag{8.15}$$

In this case, the approximated state value is

$$\hat{v}(s, w) = \phi^T(s)w = [1, x, y] \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = w_1 + w_2 x + w_3 y.$$

When w is given, $\hat{v}(s, w)$ corresponds to a plane that may not pass through the origin. Notably, $\phi(s)$ can also be defined as $\phi(s) = [x, y, 1]^T$, where the order of the elements does not matter.

The estimation result when we use the feature vector in (8.15) is shown in Fig-

ure 8.7(a). It can be seen that the estimated state values form a 2D plane. Although the estimation error converges as more episodes are used, the error cannot decrease to zero due to the limited approximation ability of a 2D plane.

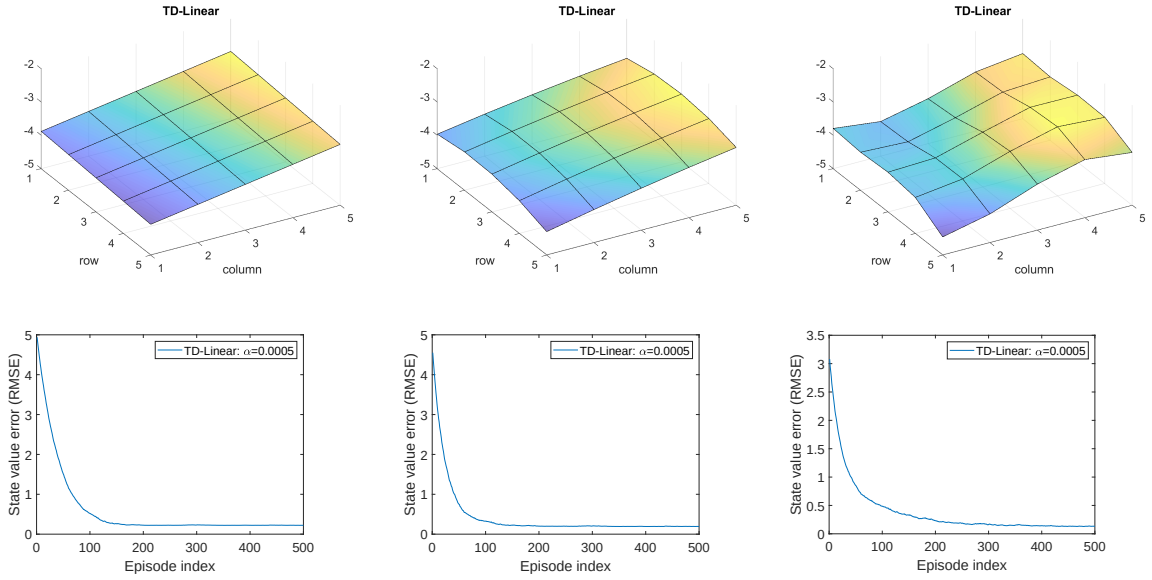
To enhance the approximation ability, we can increase the dimension of the feature vector. To that end, consider

$$\phi(s) = [1, x, y, x^2, y^2, xy]^T \in \mathbb{R}^6. \tag{8.16}$$

In this case, $\hat{v}(s, w) = \phi^T(s)w = w_1 + w_2 x + w_3 y + w_4 x^2 + w_5 y^2 + w_6 xy$, which corresponds to a quadratic 3D surface. We can further increase the dimension of the feature vector:

$$\phi(s) = [1, x, y, x^2, y^2, xy, x^3, y^3, x^2 y, xy^2]^T \in \mathbb{R}^{10}. \tag{8.17}$$

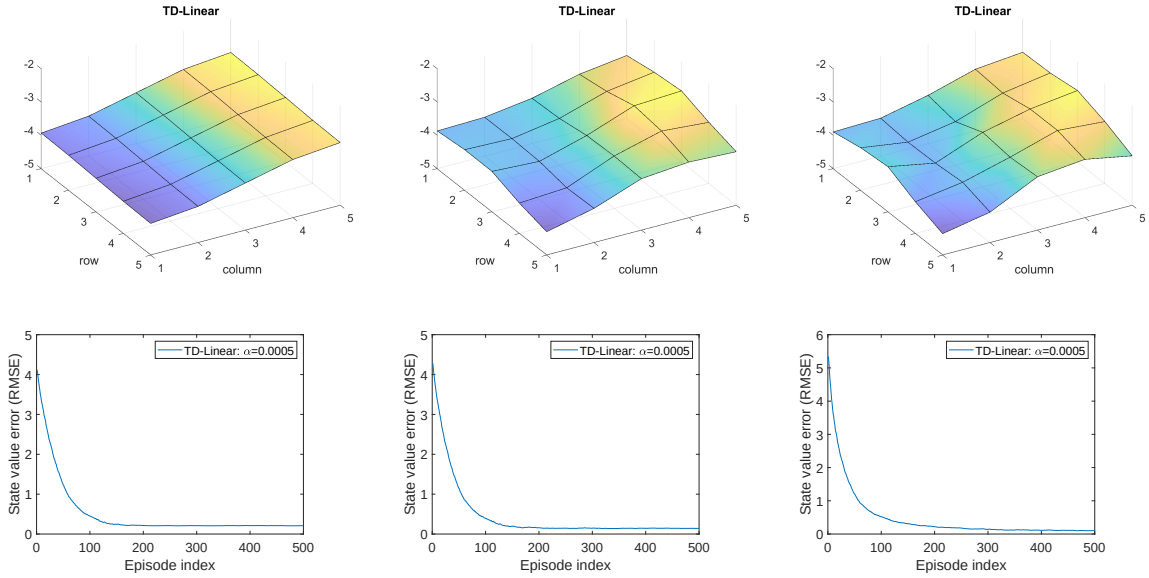
The estimation results when we use the feature vectors in (8.16) and (8.17) are shown in Figures 8.7(b)-(c). As can be seen, the longer the feature vector is, the more accurately the state values can be approximated. However, in all three cases, the estimation error cannot converge to zero because these linear approximators still have limited approximation abilities.



(a) $\phi(s) \in \mathbb{R}^3$ (b) $\phi(s) \in \mathbb{R}^6$ (c) $\phi(s) \in \mathbb{R}^{10}$

Figure 8.7: TD-Linear estimation results obtained with the polynomial features in (8.15), (8.16), and (8.17).

◇ In addition to polynomial feature vectors, many other types of features are available such as Fourier basis and tile coding [3, Chapter 9]. First, the values of x and y of



(a) $q = 1$ and $\phi(s) \in \mathbb{R}^4$ (b) $q = 2$ and $\phi(s) \in \mathbb{R}^9$ (c) $q = 3$ and $\phi(s) \in \mathbb{R}^{16}$

Figure 8.8: TD-Linear estimation results obtained with the Fourier features in (8.18).

each state are normalized to the interval of $[0, 1]$. The resulting feature vector is

$$\phi(s) = \begin{bmatrix} \vdots \\ \cos\left(\pi(c_1 x + c_2 y)\right) \\ \vdots \end{bmatrix} \in \mathbb{R}^{(q+1)^2}, \tag{8.18}$$

where π denotes the circumference ratio, which is $3.1415\ldots$, instead of a policy. Here, c_1 or c_2 can be set as any integers in $\{0, 1, \ldots, q\}$, where q is a user-specified integer. As a result, there are $(q+1)^2$ possible values for the pair (c_1, c_2) to take. Hence, the dimension of $\phi(s)$ is $(q+1)^2$. For example, in the case of $q = 1$, the feature vector is

$$\phi(s) = \begin{bmatrix} \cos\left(\pi(0x + 0y)\right) \\ \cos\left(\pi(0x + 1y)\right) \\ \cos\left(\pi(1x + 0y)\right) \\ \cos\left(\pi(1x + 1y)\right) \end{bmatrix} = \begin{bmatrix} 1 \\ \cos(\pi y) \\ \cos(\pi x) \\ \cos(\pi(x + y)) \end{bmatrix} \in \mathbb{R}^4.$$

The estimation results obtained when we use the Fourier features with $q = 1, 2, 3$ are shown in Figure 8.8. The dimensions of the feature vectors in the three cases are $4, 9, 16$, respectively. As can be seen, the higher the dimension of the feature vector is, the more accurately the state values can be approximated.

8.2.5 Theoretical analysis

Thus far, we have finished describing the story of TD learning with function approximation. This story started from the objective function in (8.3). To optimize this objective

function, we introduced the stochastic algorithm in (8.12). Later, the true value function in the algorithm, which was unknown, was replaced by an approximation, leading to the TD algorithm in (8.13). Although this story is helpful for understanding the basic idea of value function approximation, it is not mathematically rigorous. For example, the algorithm in (8.13) actually does not minimize the objective function in (8.3).

We next present a theoretical analysis of the TD algorithm in (8.13) to reveal why the algorithm works effectively and what mathematical problems it solves. Since general nonlinear approximators are difficult to analyze, this part only considers the linear case. Readers are advised to read selectively based on their interests since this part is mathematically intensive.

Convergence analysis

To study the convergence property of (8.13), we first consider the following deterministic algorithm:

$$w_{t+1} = w_t + \alpha_t \mathbb{E}\Big[\big(r_{t+1} + \gamma \phi^T(s_{t+1}) w_t - \phi^T(s_t) w_t\big)\phi(s_t)\Big], \tag{8.19}$$

where the expectation is calculated with respect to the random variables s_t, s_{t+1}, r_{t+1}. The distribution of s_t is assumed to be the stationary distribution d_π. The algorithm in (8.19) is deterministic because the random variables s_t, s_{t+1}, r_{t+1} all disappear after calculating the expectation.

Why would we consider this deterministic algorithm? First, the convergence of this deterministic algorithm is easier (though nontrivial) to analyze. Second and more importantly, the convergence of this deterministic algorithm implies the convergence of the stochastic TD algorithm in (8.13). That is because (8.13) can be viewed as a stochastic gradient descent (SGD) implementation of (8.19). Therefore, we only need to study the convergence property of the deterministic algorithm.

Although the expression of (8.19) may look complex at first glance, it can be greatly simplified. To do that, define

$$\Phi = \begin{bmatrix} \vdots \\ \phi^T(s) \\ \vdots \end{bmatrix} \in \mathbb{R}^{n \times m}, \quad D = \begin{bmatrix} \ddots & & \\ & d_\pi(s) & \\ & & \ddots \end{bmatrix} \in \mathbb{R}^{n \times n}, \tag{8.20}$$

where Φ is the matrix containing all the feature vectors, and D is a diagonal matrix with the stationary distribution in its diagonal entries. The two matrices will be frequently used.

Lemma 8.1. *The expectation in* (8.19) *can be rewritten as*

$$\mathbb{E}\Big[\big(r_{t+1} + \gamma \phi^T(s_{t+1}) w_t - \phi^T(s_t) w_t\big)\phi(s_t)\Big] = b - A w_t,$$

where

$$A \doteq \Phi^T D(I - \gamma P_\pi)\Phi \in \mathbb{R}^{m\times m},$$
$$b \doteq \Phi^T D r_\pi \in \mathbb{R}^m. \tag{8.21}$$

Here, P_π, r_π are the two terms in the Bellman equation $v_\pi = r_\pi + \gamma P_\pi v_\pi$, and I is the identity matrix with appropriate dimensions.

The proof is given in Box 8.3. With the expression in Lemma 8.1, the deterministic algorithm in (8.19) can be rewritten as

$$w_{t+1} = w_t + \alpha_t(b - Aw_t), \tag{8.22}$$

which is a simple deterministic process. Its convergence is analyzed below.

First, what is the converged value of w_t? Hypothetically, if w_t converges to a constant value w^* as $t \to \infty$, then (8.22) implies $w^* = w^* + \alpha_\infty(b - Aw^*)$, which suggests that $b - Aw^* = 0$ and hence

$$w^* = A^{-1}b.$$

Several remarks about this converged value are given below.

- ◇ Is A invertible? The answer is yes. In fact, A is not only invertible but also positive definite. That is, for any nonzero vector x with appropriate dimensions, $x^T Ax > 0$. The proof is given in Box 8.4.
- ◇ What is the interpretation of $w^* = A^{-1}b$? It is actually the optimal solution for minimizing the *projected Bellman error*. The details will be introduced in Section 8.2.5.
- ◇ The tabular method is a special case. One interesting result is that, when the dimensionality of w equals $n = |\mathcal{S}|$ and $\phi(s) = [0, \dots, 1, \dots, 0]^T$, where the entry corresponding to s is 1, we have

$$w^* = A^{-1}b = v_\pi. \tag{8.23}$$

This equation indicates that the parameter vector to be learned is actually the true state value. This conclusion is consistent with the fact that the tabular TD algorithm is a special case of the TD-Linear algorithm, as introduced in Box 8.2. The proof of (8.23) is given below. It can be verified that $\Phi = I$ in this case and hence $A = \Phi^T D(I - \gamma P_\pi)\Phi = D(I - \gamma P_\pi)$ and $b = \Phi^T D r_\pi = D r_\pi$. Thus, $w^* = A^{-1}b = (I - \gamma P_\pi)^{-1}D^{-1}Dr_\pi = (I - \gamma P_\pi)^{-1}r_\pi = v_\pi$.

Second, we prove that w_t in (8.22) converges to $w^* = A^{-1}b$ as $t \to \infty$. Since (8.22) is a simple deterministic process, it can be proven in many ways. We present two proofs as follows.

◇ Proof 1: Define the convergence error as $\delta_t \doteq w_t - w^*$. We only need to show that δ_t converges to zero. To do that, substituting $w_t = \delta_t + w^*$ into (8.22) gives

$$\delta_{t+1} = \delta_t - \alpha_t A\delta_t = (I - \alpha_t A)\delta_t.$$

It then follows that

$$\delta_{t+1} = (I - \alpha_t A)\cdots(I - \alpha_0 A)\delta_0.$$

Consider the simple case where $\alpha_t = \alpha$ for all t. Then, we have

$$\|\delta_{t+1}\|_2 \leq \|I - \alpha A\|_2^{t+1}\|\delta_0\|_2.$$

When $\alpha > 0$ is sufficiently small, we have that $\|I - \alpha A\|_2 < 1$ and hence $\delta_t \to 0$ as $t \to \infty$. The reason why $\|I - \alpha A\|_2 < 1$ holds is that A is positive definite and hence $x^T(I - \alpha A)x < 1$ for any x.

◇ Proof 2: Consider $g(w) \doteq b - Aw$. Since w^* is the root of $g(w) = 0$, the task is actually a root-finding problem. The algorithm in (8.22) is actually a Robbins-Monro (RM) algorithm. Although the original RM algorithm was designed for stochastic processes, it can also be applied to deterministic cases. The convergence of RM algorithms can shed light on the convergence of $w_{t+1} = w_t + \alpha_t(b - Aw_t)$. That is, w_t converges to w^* when $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$.

Box 8.3: Proof of Lemma 8.1

By using the law of total expectation, we have

$$\begin{aligned}
&\mathbb{E}\Big[r_{t+1}\phi(s_t) + \phi(s_t)\big(\gamma\phi^T(s_{t+1}) - \phi^T(s_t)\big)w_t\Big] \\
&= \sum_{s\in\mathcal{S}} d_\pi(s)\mathbb{E}\Big[r_{t+1}\phi(s_t) + \phi(s_t)\big(\gamma\phi^T(s_{t+1}) - \phi^T(s_t)\big)w_t\Big|s_t = s\Big] \\
&= \sum_{s\in\mathcal{S}} d_\pi(s)\mathbb{E}\Big[r_{t+1}\phi(s_t)\Big|s_t = s\Big] + \sum_{s\in\mathcal{S}} d_\pi(s)\mathbb{E}\Big[\phi(s_t)\big(\gamma\phi^T(s_{t+1}) - \phi^T(s_t)\big)w_t\Big|s_t = s\Big].
\end{aligned} \tag{8.24}$$

Here, s_t is assumed to obey the stationary distribution d_π.

First, consider the first term in (8.24). Note that

$$\mathbb{E}\Big[r_{t+1}\phi(s_t)\Big|s_t = s\Big] = \phi(s)\mathbb{E}\Big[r_{t+1}\Big|s_t = s\Big] = \phi(s)r_\pi(s),$$

where $r_\pi(s) = \sum_a \pi(a|s)\sum_r rp(r|s,a)$. Then, the first term in (8.24) can be rewritten

as

$$\sum_{s\in\mathcal{S}} d_\pi(s)\mathbb{E}\Big[r_{t+1}\phi(s_t)\Big|s_t=s\Big] = \sum_{s\in\mathcal{S}} d_\pi(s)\phi(s)r_\pi(s) = \Phi^T D r_\pi, \tag{8.25}$$

where $r_\pi = [\cdots, r_\pi(s), \cdots]^T \in \mathbb{R}^n$.

Second, consider the second term in (8.24). Since

$$\begin{aligned}
&\mathbb{E}\Big[\phi(s_t)\big(\gamma\phi^T(s_{t+1}) - \phi^T(s_t)\big)w_t\Big|s_t=s\Big] \\
&= -\mathbb{E}\Big[\phi(s_t)\phi^T(s_t)w_t\Big|s_t=s\Big] + \mathbb{E}\Big[\gamma\phi(s_t)\phi^T(s_{t+1})w_t\Big|s_t=s\Big] \\
&= -\phi(s)\phi^T(s)w_t + \gamma\phi(s)\mathbb{E}\Big[\phi^T(s_{t+1})\Big|s_t=s\Big]w_t \\
&= -\phi(s)\phi^T(s)w_t + \gamma\phi(s)\sum_{s'\in\mathcal{S}} p(s'|s)\phi^T(s')w_t,
\end{aligned}$$

the second term in (8.24) becomes

$$\begin{aligned}
&\sum_{s\in\mathcal{S}} d_\pi(s)\mathbb{E}\Big[\phi(s_t)\big(\gamma\phi^T(s_{t+1}) - \phi^T(s_t)\big)w_t\Big|s_t=s\Big] \\
&= \sum_{s\in\mathcal{S}} d_\pi(s)\Big[-\phi(s)\phi^T(s)w_t + \gamma\phi(s)\sum_{s'\in\mathcal{S}} p(s'|s)\phi^T(s')w_t\Big] \\
&= \sum_{s\in\mathcal{S}} d_\pi(s)\phi(s)\Big[-\phi(s) + \gamma\sum_{s'\in\mathcal{S}} p(s'|s)\phi(s')\Big]^T w_t \\
&= \Phi^T D(-\Phi + \gamma P_\pi\Phi)w_t \\
&= -\Phi^T D(I-\gamma P_\pi)\Phi w_t.
\end{aligned} \tag{8.26}$$

Combining (8.25) and (8.26) gives

$$\begin{aligned}
\mathbb{E}\Big[\big(r_{t+1} + \gamma\phi^T(s_{t+1})w_t - \phi^T(s_t)w_t\big)\phi(s_t)\Big] &= \Phi^T D r_\pi - \Phi^T D(I-\gamma P_\pi)\Phi w_t \\
&\doteq b - Aw_t,
\end{aligned} \tag{8.27}$$

where $b \doteq \Phi^T D r_\pi$ and $A \doteq \Phi^T D(I-\gamma P_\pi)\Phi$.

Box 8.4: Proving that $A = \Phi^T D(I-\gamma P_\pi)\Phi$ is invertible and positive definite.

The matrix A is positive definite if $x^T Ax > 0$ for any nonzero vector x with appropriate dimensions. If A is positive (or negative) definite, it is denoted as $A \succ 0$ (or $A \prec 0$). Here, $\succ$ and $\prec$ should be differentiated from $>$ and $<$, which indicate elementwise comparisons. Note that A may not be symmetric. Although positive

definite matrices often refer to symmetric matrices, nonsymmetric ones can also be positive definite.

We next prove that $A \succ 0$ and hence A is invertible. The idea for proving $A \succ 0$ is to show that

$$D(I - \gamma P_\pi) \doteq M \succ 0. \tag{8.28}$$

It is clear that $M \succ 0$ implies $A = \Phi^T M \phi \succ 0$ since Φ is a tall matrix with full column rank (suppose that the feature vectors are selected to be linearly independent). Note that

$$M = \frac{M + M^T}{2} + \frac{M - M^T}{2}.$$

Since $M - M^T$ is skew-symmetric and hence $x^T(M - M^T)x = 0$ for any x, we know that $M \succ 0$ if and only if $M + M^T \succ 0$. To show $M + M^T \succ 0$, we apply the fact that strictly diagonal dominant matrices are positive definite [4].

First, it holds that

$$(M + M^T)\mathbf{1}_n > 0, \tag{8.29}$$

where $\mathbf{1}_n = [1, \ldots, 1]^T \in \mathbb{R}^n$. The proof of (8.29) is given below. Since $P_\pi \mathbf{1}_n = \mathbf{1}_n$, we have $M\mathbf{1}_n = D(I - \gamma P_\pi)\mathbf{1}_n = D(\mathbf{1}_n - \gamma \mathbf{1}_n) = (1 - \gamma)d_\pi$. Moreover, $M^T\mathbf{1}_n = (I - \gamma P_\pi^T)D\mathbf{1}_n = (I - \gamma P_\pi^T)d_\pi = (1 - \gamma)d_\pi$, where the last equality is valid because $P_\pi^T d_\pi = d_\pi$. In summary, we have

$$(M + M^T)\mathbf{1}_n = 2(1 - \gamma)d_\pi.$$

Since all the entries of d_π are positive (see Box 8.1), we have $(M + M^T)\mathbf{1}_n > 0$.

Second, the elementwise form of (8.29) is

$$\sum_{j=1}^{n} [M + M^T]_{ij} > 0, \qquad i = 1, \ldots, n,$$

which can be further written as

$$[M + M^T]_{ii} + \sum_{j \neq i} [M + M^T]_{ij} > 0.$$

It can be verified according to (8.28) that the diagonal entries of M are positive and the off-diagonal entries of M are nonpositive. Therefore, the above inequality can be

rewritten as

$$\left|[M+M^T]_{ii}\right| > \sum_{j\neq i}\left|[M+M^T]_{ij}\right|.$$

The above inequality indicates that the absolute value of the ith diagonal entry in $M+M^T$ is greater than the sum of the absolute values of the off-diagonal entries in the same row. Thus, $M+M^T$ is strictly diagonal dominant and the proof is complete.

TD learning minimizes the projected Bellman error

While we have shown that the TD-Linear algorithm converges to $w^* = A^{-1}b$, we next show that w^* is the optimal solution that minimizes the *projected Bellman error*. To do that, we review three objective functions.

◇ The first objective function is

$$J_E(w) = \mathbb{E}[(v_\pi(S) - \hat{v}(S,w))^2],$$

which has been introduced in (8.3). By the definition of expectation, $J_E(w)$ can be reexpressed in a matrix-vector form as

$$J_E(w) = \|\hat{v}(w) - v_\pi\|_D^2,$$

where v_π is the true state value vector and $\hat{v}(w)$ is the approximated one. Here, $\|\cdot\|_D^2$ is a weighted norm: $\|x\|_D^2 = x^T D x = \|D^{1/2}x\|_2^2$, where D is given in (8.20).

This is the simplest objective function that we can imagine when talking about function approximation. However, it relies on the true state, which is unknown. To obtain an implementable algorithm, we must consider other objective functions such as the Bellman error and projected Bellman error [50–54].

◇ The second objective function is the Bellman error. In particular, since v_π satisfies the Bellman equation $v_\pi = r_\pi + \gamma P_\pi v_\pi$, it is expected that the estimated value $\hat{v}(w)$ should also satisfy this equation to the greatest extent possible. Thus, the *Bellman error* is

$$J_{BE}(w) = \|\hat{v}(w) - (r_\pi + \gamma P_\pi \hat{v}(w))\|_D^2 \doteq \|\hat{v}(w) - T_\pi(\hat{v}(w))\|_D^2. \tag{8.30}$$

Here, $T_\pi(\cdot)$ is the Bellman operator. In particular, for any vector $x \in \mathbb{R}^n$, the Bellman operator is defined as

$$T_\pi(x) \doteq r_\pi + \gamma P_\pi x.$$

Minimizing the Bellman error is a standard least-squares problem. The details of the solution are omitted here.

◇ Third, it is notable that $J_{BE}(w)$ in (8.30) may not be minimized to *zero* due to the limited approximation ability of the approximator. By contrast, an objective function that can be minimized to zero is the *projected Bellman error*:

$$J_{PBE}(w) = \|\hat{v}(w) - MT_\pi(\hat{v}(w))\|_D^2,$$

where $M \in \mathbb{R}^{n\times n}$ is the orthogonal projection matrix that geometrically projects any vector onto the space of all approximations.

In fact, the TD learning algorithm in (8.13) aims to minimize the projected Bellman error J_{PBE} rather than J_E or J_{BE}. The reason is as follows. For the sake of simplicity, consider the linear case where $\hat{v}(w) = \Phi w$. Here, Φ is defined in (8.20). The range space of Φ is the set of all possible linear approximations. Then,

$$M = \Phi(\Phi^T D\Phi)^{-1}\Phi^T D \in \mathbb{R}^{n\times n}$$

is the projection matrix that geometrically projects any vector onto the range space Φ. Since $\hat{v}(w)$ is in the range space of Φ, we can always find a value of w that can minimize $J_{PBE}(w)$ to zero. It can be proven that the solution minimizing $J_{PBE}(w)$ is $w^* = A^{-1}b$. That is

$$w^* = A^{-1}b = \arg\min_w J_{PBE}(w),$$

The proof is given in Box 8.5.

Box 8.5: Showing that $w^* = A^{-1}b$ minimizes $J_{PBE}(w)$

We next show that $w^* = A^{-1}b$ is the optimal solution that minimizes $J_{PBE}(w)$. Since $J_{PBE}(w) = 0 \Leftrightarrow \hat{v}(w) - MT_\pi(\hat{v}(w)) = 0$, we only need to study the root of

$$\hat{v}(w) = MT_\pi(\hat{v}(w)).$$

In the linear case, substituting $\hat{v}(w) = \Phi w$ and the expression of M in (8.28) into the above equation gives

$$\Phi w = \Phi(\Phi^T D\Phi)^{-1}\Phi^T D(r_\pi + \gamma P_\pi \Phi w). \tag{8.31}$$

Since Φ has full column rank, we have $\Phi x = \Phi y \Leftrightarrow x = y$ for any x, y. Therefore, (8.31) implies

$$\begin{aligned}
&w = (\Phi^T D\Phi)^{-1}\Phi^T D(r_\pi + \gamma P_\pi \Phi w) \\
&\iff \Phi^T D(r_\pi + \gamma P_\pi \Phi w) = (\Phi^T D\Phi)w \\
&\iff \Phi^T D r_\pi + \gamma \Phi^T D P_\pi \Phi w = (\Phi^T D\Phi)w \\
&\iff \Phi^T D r_\pi = \Phi^T D(I - \gamma P_\pi)\Phi w \\
&\iff w = (\Phi^T D(I - \gamma P_\pi)\Phi)^{-1}\Phi^T D r_\pi = A^{-1}b,
\end{aligned}$$

where A, b are given in (8.21). Therefore, $w^* = A^{-1}b$ is the optimal solution that minimizes $J_{PBE}(w)$.

Since the TD algorithm aims to minimize J_{PBE} rather than J_E, it is natural to ask how close the estimated value $\hat{v}(w)$ is to the true state value v_π. In the linear case, the estimated value that minimizes the projected Bellman error is $\hat{v}(w) = \Phi w^*$. Its deviation from the true state value v_π satisfies

$$\|\Phi w^* - v_\pi\|_D \leq \frac{1}{1-\gamma} \min_w \|\hat{v}(w) - v_\pi\|_D = \frac{1}{1-\gamma} \min_w \sqrt{J_E(w)}. \tag{8.32}$$

The proof of this inequality is given in Box 8.6. Inequality (8.32) indicates that the discrepancy between Φw^* and v_π is bounded from above by the minimum value of $J_E(w)$. However, this bound is loose, especially when γ is close to one. It is thus mainly of theoretical value.

Box 8.6: Proof of the error bound in (8.32)

Note that

$$\begin{aligned}
\|\Phi w^* - v_\pi\|_D &= \|\Phi w^* - M v_\pi + M v_\pi - v_\pi\|_D \\
&\leq \|\Phi w^* - M v_\pi\|_D + \|M v_\pi - v_\pi\|_D \\
&= \|M T_\pi(\Phi w^*) - M T_\pi(v_\pi)\|_D + \|M v_\pi - v_\pi\|_D, \qquad (8.33)
\end{aligned}$$

where the last equality is due to $\Phi w^* = M T_\pi(\Phi w^*)$ and $v_\pi = T_\pi(v_\pi)$. Substituting

$$M T_\pi(\Phi w^*) - M T_\pi(v_\pi) = M(r_\pi + \gamma P_\pi \Phi w^*) - M(r_\pi + \gamma P_\pi v_\pi) = \gamma M P_\pi(\Phi w^* - v_\pi)$$

into (8.33) yields

$$\begin{aligned}\|\Phi w^* - v_\pi\|_D &\leq \|\gamma M P_\pi(\Phi w^* - v_\pi)\|_D + \|Mv_\pi - v_\pi\|_D \\ &\leq \gamma\|M\|_D\|P_\pi(\Phi w^* - v_\pi)\|_D + \|Mv_\pi - v_\pi\|_D \\ &= \gamma\|P_\pi(\Phi w^* - v_\pi)\|_D + \|Mv_\pi - v_\pi\|_D \qquad (\text{because } \|M\|_D = 1) \\ &\leq \gamma\|\Phi w^* - v_\pi\|_D + \|Mv_\pi - v_\pi\|_D. \qquad (\text{because } \|P_\pi x\|_D \leq \|x\|_D \text{ for all } x)\end{aligned}$$

The proof of $\|M\|_D = 1$ and $\|P_\pi x\|_D \leq \|x\|_D$ are postponed to the end of the box. Recognizing the above inequality gives

$$\begin{aligned}\|\Phi w^* - v_\pi\|_D &\leq \frac{1}{1-\gamma}\|Mv_\pi - v_\pi\|_D \\ &= \frac{1}{1-\gamma}\min_w \|\hat{v}(w) - v_\pi\|_D,\end{aligned}$$

where the last equality is because $\|Mv_\pi - v_\pi\|_D$ is the error between v_π and its orthogonal projection into the space of all possible approximations. Therefore, it is the minimum value of the error between v_π and any $\hat{v}(w)$.

We next prove some useful facts, which have already been used in the above proof.

- ◇ Properties of matrix weighted norms. By definition, $\|x\|_D = \sqrt{x^T D x} = \|D^{1/2}x\|_2$. The induced matrix norm is $\|A\|_D = \max_{x\neq 0}\|Ax\|_D/\|x\|_D = \|D^{1/2}AD^{-1/2}\|_2$. For matrices A, B with appropriate dimensions, we have $\|ABx\|_D \leq \|A\|_D\|B\|_D\|x\|_D$. To see that, $\|ABx\|_D = \|D^{1/2}ABx\|_2 = \|D^{1/2}AD^{-1/2}D^{1/2}BD^{-1/2}D^{1/2}x\|_2 \leq \|D^{1/2}AD^{-1/2}\|_2\|D^{1/2}BD^{-1/2}\|_2\|D^{1/2}x\|_2 = \|A\|_D\|B\|_D\|x\|_D$.
- ◇ Proof of $\|M\|_D = 1$. This is valid because $\|M\|_D = \|\Phi(\Phi^T D\Phi)^{-1}\Phi^T D\|_D = \|D^{1/2}\Phi(\Phi^T D\Phi)^{-1}\Phi^T DD^{-1/2}\|_2 = 1$, where the last equality is valid due to the fact that the matrix in the L_2-norm is an orthogonal projection matrix and the L_2-norm of any orthogonal projection matrix is equal to one.
- ◇ Proof of $\|P_\pi x\|_D \leq \|x\|_D$ for any $x \in \mathbb{R}^n$. First,

$$\|P_\pi x\|_D^2 = x^T P_\pi^T D P_\pi x = \sum_{i,j} x_i [P_\pi^T D P_\pi]_{ij} x_j = \sum_{i,j} x_i \left(\sum_k [P_\pi^T]_{ik}[D]_{kk}[P_\pi]_{kj}\right) x_j.$$

Reorganizing the above equation gives

$$\begin{aligned}
\|P_\pi x\|_D^2 &= \sum_k [D]_{kk} \Big(\sum_i [P_\pi]_{ki} x_i \Big)^2 \\
&\leq \sum_k [D]_{kk} \Big(\sum_i [P_\pi]_{ki} x_i^2 \Big) \qquad \text{(due to Jensen's inequality [55, 56])} \\
&= \sum_i \Big(\sum_k [D]_{kk} [P_\pi]_{ki} \Big) x_i^2 \\
&= \sum_i [D]_{ii} x_i^2 \qquad \text{(due to } d_\pi^T P_\pi = d_\pi^T \text{)} \\
&= \|x\|_D^2.
\end{aligned}$$

Least-squares TD

We next introduce an algorithm called *least-squares TD* (LSTD) [57]. Like the TD-Linear algorithm, LSTD aims to minimize the projected Bellman error. However, it has some advantages over the TD-Linear algorithm.

Recall that the optimal parameter for minimizing the projected Bellman error is $w^* = A^{-1}b$, where $A = \Phi^T D(I - \gamma P_\pi)\Phi$ and $b = \Phi^T D r_\pi$. In fact, it follows from (8.27) that A and b can also be written as

$$\begin{aligned}
A &= \mathbb{E}\Big[\phi(s_t)\big(\phi(s_t) - \gamma\phi(s_{t+1})\big)^T\Big], \\
b &= \mathbb{E}\Big[r_{t+1}\phi(s_t)\Big].
\end{aligned}$$

The above two equations show that A and b are expectations of s_t, s_{t+1}, r_{t+1}. The *idea* of LSTD is simple: if we can use random samples to directly obtain the estimates of A and b, which are denoted as $\hat{A}$ and $\hat{b}$, then the optimal parameter can be directly estimated as $w^* \approx \hat{A}^{-1}\hat{b}$.

In particular, suppose that $(s_0, r_1, s_1, \ldots, s_t, r_{t+1}, s_{t+1}, \ldots)$ is a trajectory obtained by following a given policy π. Let $\hat{A}_t$ and $\hat{b}_t$ be the estimates of A and b at time t, respectively. They are calculated as the averages of the samples:

$$\begin{aligned}
\hat{A}_t &= \sum_{k=0}^{t-1} \phi(s_k)\big(\phi(s_k) - \gamma\phi(s_{k+1})\big)^T, \\
\hat{b}_t &= \sum_{k=0}^{t-1} r_{k+1}\phi(s_k).
\end{aligned} \tag{8.34}$$

Then, the estimated parameter is

$$w_t = \hat{A}_t^{-1}\hat{b}_t.$$

The reader may wonder if a coefficient of $1/t$ is missing on the right-hand side of (8.34). In fact, it is omitted for the sake of simplicity since the value of w_t remains the same when it is omitted. Since $\hat{A}_t$ may not be invertible especially when t is small, $\hat{A}_t$ is usually biased by a small constant matrix σI, where I is the identity matrix and σ is a small positive number.

The *advantage* of LSTD is that it uses experience samples more efficiently and converges faster than the TD method. That is because this algorithm is specifically designed based on the knowledge of the optimal solution's expression. The better we understand a problem, the better algorithms we can design.

The *disadvantages* of LSTD are as follows. First, it can only estimate state values. By contrast, the TD algorithm can be extended to estimate action values as shown in the next section. Moreover, while the TD algorithm allows nonlinear approximators, LSTD does not. That is because this algorithm is specifically designed based on the expression of w^*. Second, the computational cost of LSTD is higher than that of TD since LSTD updates an $m \times m$ matrix in each update step, whereas TD updates an m-dimensional vector. More importantly, in every step, LSTD needs to compute the inverse of $\hat{A}_t$, whose computational complexity is $O(m^3)$. The common method for resolving this problem is to directly update the inverse of $\hat{A}_t$ rather than updating $\hat{A}_t$. In particular, $\hat{A}_{t+1}$ can be calculated recursively as follows:

$$\begin{aligned}
\hat{A}_{t+1} &= \sum_{k=0}^{t} \phi(s_k)\big(\phi(s_k) - \gamma\phi(s_{k+1})\big)^T \\
&= \sum_{k=0}^{t-1} \phi(s_k)\big(\phi(s_k) - \gamma\phi(s_{k+1})\big)^T + \phi(s_t)\big(\phi(s_t) - \gamma\phi(s_{t+1})\big)^T \\
&= \hat{A}_t + \phi(s_t)\big(\phi(s_t) - \gamma\phi(s_{t+1})\big)^T.
\end{aligned}$$

The above expression decomposes $\hat{A}_{t+1}$ into the sum of two matrices. Its inverse can be calculated as [58]

$$\begin{aligned}
\hat{A}_{t+1}^{-1} &= \Big(\hat{A}_t + \phi(s_t)\big(\phi(s_t) - \gamma\phi(s_{t+1})\big)^T\Big)^{-1} \\
&= \hat{A}_t^{-1} + \frac{\hat{A}_t^{-1}\phi(s_t)\big(\phi(s_t) - \gamma\phi(s_{t+1})\big)^T\hat{A}_t^{-1}}{1 + \big(\phi(s_t) - \gamma\phi(s_{t+1})\big)^T\hat{A}_t^{-1}\phi(s_t)}.
\end{aligned}$$

Therefore, we can directly store and update $\hat{A}_t^{-1}$ to avoid the need to calculate the matrix inverse. This recursive algorithm does not require a step size. However, it requires setting the initial value of $\hat{A}_0^{-1}$. The initial value of such a recursive algorithm can be selected as $\hat{A}_0^{-1} = \sigma I$, where σ is a positive number. A good tutorial on the recursive least-squares approach can be found in [59].

8.3 TD learning of action values based on function approximation

While Section 8.2 introduced the problem of *state value* estimation, the present section introduces how to estimate *action values*. The tabular Sarsa and tabular Q-learning algorithms are extended to the case of value function approximation. Readers will see that the extension is straightforward.

8.3.1 Sarsa with function approximation

The Sarsa algorithm with function approximation can be readily obtained from (8.13) by replacing the state values with action values. In particular, suppose that $q_\pi(s,a)$ is approximated by $\hat{q}(s,a,w)$. Replacing $\hat{v}(s,w)$ in (8.13) by $\hat{q}(s,a,w)$ gives

$$w_{t+1} = w_t + \alpha_t \Big[r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, w_t) - \hat{q}(s_t, a_t, w_t) \Big] \nabla_w \hat{q}(s_t, a_t, w_t). \tag{8.35}$$

The analysis of (8.35) is similar to that of (8.13) and is omitted here. When linear functions are used, we have

$$\hat{q}(s,a,w) = \phi^T(s,a)w,$$

where $\phi(s,a)$ is a feature vector. In this case, $\nabla_w \hat{q}(s,a,w) = \phi(s,a)$.

The value estimation step in (8.35) can be combined with a policy improvement step to learn optimal policies. The procedure is summarized in Algorithm 8.2. It should be noted that accurately estimating the action values of a given policy requires (8.35) to be run sufficiently many times. However, (8.35) is executed only once before switching to the policy improvement step. This is similar to the tabular Sarsa algorithm. Moreover, the implementation in Algorithm 8.2 aims to solve the task of finding a good path to the target state from a prespecified starting state. As a result, it cannot find the optimal policy for every state. However, if sufficient experience data are available, the implementation process can be easily adapted to find optimal policies for every state.

An illustrative example is shown in Figure 8.9. In this example, the task is to find a good policy that can lead the agent to the target when starting from the top-left state. Both the total reward and the length of each episode gradually converge to steady values. In this example, the linear feature vector is selected as the Fourier function of order 5. The expression of a Fourier feature vector is given in (8.18).

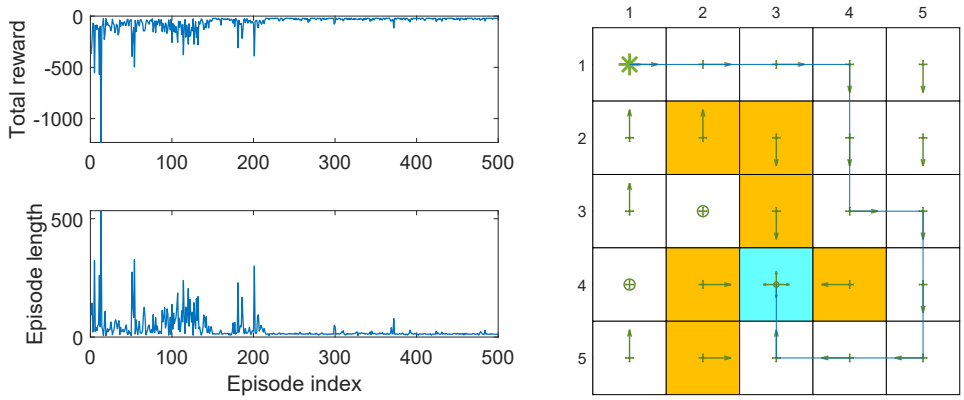


Figure 8.9: Sarsa with linear function approximation. Here, $\gamma = 0.9$, $\epsilon = 0.1$, $r_{\text{boundary}} = r_{\text{forbidden}} = -10$, $r_{\text{target}} = 1$, and $\alpha = 0.001$.

Algorithm 8.2: Sarsa with function approximation

Initialization: Initial parameter w_0. Initial policy π_0. $\alpha_t = \alpha > 0$ for all t. $\epsilon \in (0, 1)$.
Goal: Learn an optimal policy that can lead the agent to the target state from an initial state s_0.
For each episode, do
  Generate a_0 at s_0 following $\pi_0(s_0)$
  If s_t $(t = 0, 1, 2, \dots)$ is not the target state, do
    Collect the experience sample $(r_{t+1}, s_{t+1}, a_{t+1})$ given (s_t, a_t): generate r_{t+1}, s_{t+1} by interacting with the environment; generate a_{t+1} following $\pi_t(s_{t+1})$.
    Update q-value:
    $w_{t+1} = w_t + \alpha_t \Big[r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, w_t) - \hat{q}(s_t, a_t, w_t) \Big] \nabla_w \hat{q}(s_t, a_t, w_t)$
    Update policy:
    $\pi_{t+1}(a|s_t) = 1 - \frac{\varepsilon}{|\mathcal{A}(s_t)|}(|\mathcal{A}(s_t)| - 1)$ if $a = \arg\max_{a \in \mathcal{A}(s_t)} \hat{q}(s_t, a, w_{t+1})$
    $\pi_{t+1}(a|s_t) = \frac{\epsilon}{|\mathcal{A}(s_t)|}$ otherwise
    $s_t \leftarrow s_{t+1}$, $a_t \leftarrow a_{t+1}$

8.3.2 Q-learning with function approximation

Tabular Q-learning can also be extended to the case of function approximation. The update rule is

$$w_{t+1} = w_t + \alpha_t \Big[r_{t+1} + \gamma \max_{a \in \mathcal{A}(s_{t+1})} \hat{q}(s_{t+1}, a, w_t) - \hat{q}(s_t, a_t, w_t) \Big] \nabla_w \hat{q}(s_t, a_t, w_t). \tag{8.36}$$

The above update rule is similar to (8.35) except that $\hat{q}(s_{t+1}, a_{t+1}, w_t)$ in (8.35) is replaced with $\max_{a \in \mathcal{A}(s_{t+1})} \hat{q}(s_{t+1}, a, w_t)$.

Similar to the tabular case, (8.36) can be implemented in either an on-policy or off-policy fashion. An on-policy version is given in Algorithm 8.3. An example for demonstrating the on-policy version is shown in Figure 8.10. In this example, the task is to find a good policy that can lead the agent to the target state from the top-left state.

Algorithm 8.3: Q-learning with function approximation (on-policy version)

Initialization: Initial parameter w_0. Initial policy π_0. $\alpha_t = \alpha > 0$ for all t. $\epsilon \in (0, 1)$.
Goal: Learn an optimal path that can lead the agent to the target state from an initial state s_0.

For each episode, do

If s_t $(t = 0, 1, 2, \dots)$ is not the target state, do

Collect the experience sample (a_t, r_{t+1}, s_{t+1}) given s_t: generate a_t following $\pi_t(s_t)$; generate r_{t+1}, s_{t+1} by interacting with the environment.

Update q-value:

$$w_{t+1} = w_t + \alpha_t \Big[r_{t+1} + \gamma \max_{a \in \mathcal{A}(s_{t+1})} \hat{q}(s_{t+1}, a, w_t) - \hat{q}(s_t, a_t, w_t) \Big] \nabla_w \hat{q}(s_t, a_t, w_t)$$

Update policy:

$\pi_{t+1}(a|s_t) = 1 - \frac{\varepsilon}{|\mathcal{A}(s_t)|}(|\mathcal{A}(s_t)| - 1)$ if $a = \arg\max_{a \in \mathcal{A}(s_t)} \hat{q}(s_t, a, w_{t+1})$

$\pi_{t+1}(a|s_t) = \frac{\varepsilon}{|\mathcal{A}(s_t)|}$ otherwise

As can be seen, Q-learning with linear function approximation can successfully learn an optimal policy. Here, linear Fourier basis functions of order five are used. The off-policy version will be demonstrated when we introduce deep Q-learning in Section 8.4.

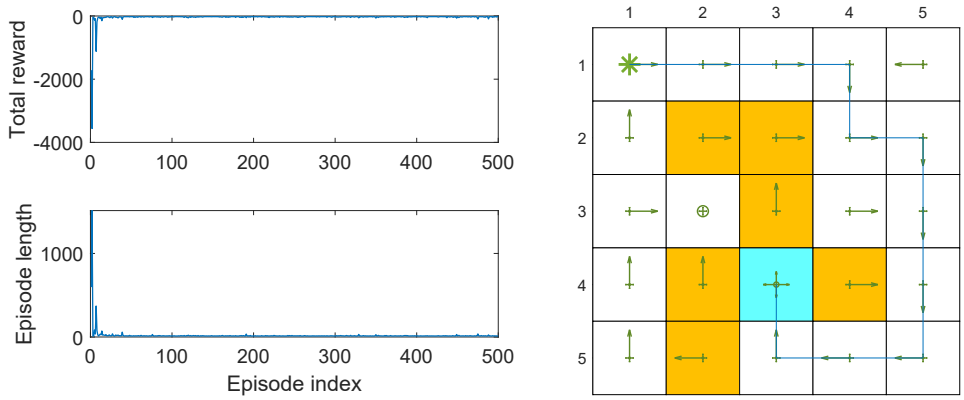


Figure 8.10: Q-learning with linear function approximation. Here, $\gamma = 0.9$, $\epsilon = 0.1$, $r_{\text{boundary}} = r_{\text{forbidden}} = -10$, $r_{\text{target}} = 1$, and $\alpha = 0.001$.

One may notice in Algorithm 8.2 and Algorithm 8.3 that, although the values are represented as functions, the policy $\pi(a|s)$ is still represented as a table. Thus, it still assumes finite numbers of states and actions. In Chapter 9, we will see that the policies can be represented as functions so that continuous state and action spaces can be handled.

8.4 Deep Q-learning

We can integrate deep neural networks into Q-learning to obtain an approach called *deep Q-learning* or *deep Q-network* (DQN) [22, 60, 61]. Deep Q-learning is one of the

earliest and most successful deep reinforcement learning algorithms. Notably, the neural networks do not have to be deep. For simple tasks such as our grid world examples, shallow networks with one or two hidden layers may be sufficient.

Deep Q-learning can be viewed as an extension of the algorithm in (8.36). However, its mathematical formulation and implementation techniques are substantially different and deserve special attention.

8.4.1 Algorithm description

Mathematically, deep Q-learning aims to minimize the following objective function:

$$J = \mathbb{E}\left[\left(R + \gamma \max_{a \in \mathcal{A}(S')} \hat{q}(S', a, w) - \hat{q}(S, A, w)\right)^2\right], \tag{8.37}$$

where (S, A, R, S') are random variables that denote a state, an action, the immediate reward, and the next state, respectively. This objective function can be viewed as the squared Bellman optimality error. That is because

$$q(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a \in \mathcal{A}(S_{t+1})} q(S_{t+1}, a) \middle| S_t = s, A_t = a\right], \quad \text{for all } s, a$$

is the Bellman optimality equation (the proof is given in Box 7.5). Therefore, $R + \gamma \max_{a \in \mathcal{A}(S')} \hat{q}(S', a, w) - \hat{q}(S, A, w)$ should equal zero in the expectation sense when $\hat{q}(S, A, w)$ can accurately approximate the optimal action values.

To minimize the objective function in (8.37), we can use the gradient descent algorithm. To that end, we need to calculate the gradient of J with respect to w. It is noted that the parameter w appears not only in $\hat{q}(S, A, w)$ but also in $y \doteq R + \gamma \max_{a \in \mathcal{A}(S')} \hat{q}(S', a, w)$. As a result, it is nontrivial to calculate the gradient. For the sake of simplicity, it is assumed that the value of w in y is fixed (for a short period of time) so that the calculation of the gradient becomes much easier. In particular, we introduce two networks: one is a *main network* representing $\hat{q}(s, a, w)$ and the other is a *target network* $\hat{q}(s, a, w_T)$. The objective function in this case becomes

$$J = \mathbb{E}\left[\left(R + \gamma \max_{a \in \mathcal{A}(S')} \hat{q}(S', a, w_T) - \hat{q}(S, A, w)\right)^2\right],$$

where w_T is the target network's parameter. When w_T is fixed, the gradient of J is

$$\nabla_w J = \mathbb{E}\left[\left(R + \gamma \max_{a \in \mathcal{A}(S')} \hat{q}(S', a, w_T) - \hat{q}(S, A, w)\right) \nabla_w \hat{q}(S, A, w)\right], \tag{8.38}$$

where some constant coefficients are omitted without loss of generality.

To use the gradient in (8.38) to minimize the objective function in (8.37), we need to

pay attention to the following techniques.

◇ The first technique is to use two networks, a main network and a target network, as mentioned when we calculate the gradient in (8.38). The implementation details are explained below. Let w and w_T denote the parameters of the main and target networks, respectively. They are initially set to the same value.

In every iteration, we draw a mini-batch of samples $\{(s, a, r, s')\}$ from the replay buffer (the replay buffer will be explained soon). The inputs of the main network are s and a. The output $y = \hat{q}(s, a, w)$ is the estimated q-value. The target value of the output is $y_T \doteq r + \gamma \max_{a \in \mathcal{A}(s')} \hat{q}(s', a, w_T)$. The main network is updated to minimize the TD error (also called the loss function) $\sum(y - y_T)^2$ over the samples $\{(s, a, y_T)\}$.

Updating w in the main network does not explicitly use the gradient in (8.38). Instead, it relies on the existing software tools for training neural networks. As a result, we need a mini-batch of samples to train a network instead of using a single sample to update the main network based on (8.38). This is one notable difference between deep and nondeep reinforcement learning algorithms.

The main network is updated in every iteration. By contrast, the target network is set to be the same as the main network every certain number of iterations to satisfy the assumption that w_T is fixed when calculating the gradient in (8.38).

◇ The second technique is *experience replay* [22, 60, 62]. That is, after we have collected some experience samples, we do not use these samples in the order they were collected. Instead, we store them in a dataset called the *replay buffer*. In particular, let (s, a, r, s') be an experience sample and $\mathcal{B} \doteq \{(s, a, r, s')\}$ be the replay buffer. Every time we update the main network, we can draw a mini-batch of experience samples from the replay buffer. The draw of samples, or called *experience replay*, should follow a *uniform distribution*.

Why is experience replay necessary in deep Q-learning, and why must the replay follow a uniform distribution? The answer lies in the objective function in (8.37). In particular, to well define the objective function, we must specify the probability distributions for S, A, R, S'. The distributions of R and S' are determined by the system model once (S, A) is given. The simplest way to describe the distribution of the state-action pair (S, A) is to assume it to be *uniformly* distributed.

However, the state-action samples may *not* be uniformly distributed in practice since they are generated as a sample sequence according to the behavior policy. It is necessary to break the correlation between the samples in the sequence to satisfy the assumption of uniform distribution. To do this, we can use the experience replay technique by uniformly drawing samples from the replay buffer. This is the mathematical reason why experience replay is necessary and why experience replay must follow a uniform distribution. A benefit of random sampling is that each experience sample

Algorithm 8.3: Deep Q-learning (off-policy version)

Initialization: A main network and a target network with the same initial parameter.
Goal: Learn an optimal target network to approximate the *optimal* action values from the experience samples generated by a given behavior policy π_b.

Store the experience samples generated by π_b in a replay buffer $\mathcal{B} = \{(s, a, r, s')\}$

For each iteration, do

Uniformly draw a mini-batch of samples from $\mathcal{B}$

For each sample (s, a, r, s'), calculate the target value as $y_T = r + \gamma \max_{a \in \mathcal{A}(s')} \hat{q}(s', a, w_T)$, where w_T is the parameter of the target network

Update the main network to minimize $(y_T - \hat{q}(s, a, w))^2$ using the mini-batch of samples

Set $w_T = w$ every C iterations

may be used multiple times, which can increase the data efficiency. This is especially important when we have a limited amount of data.

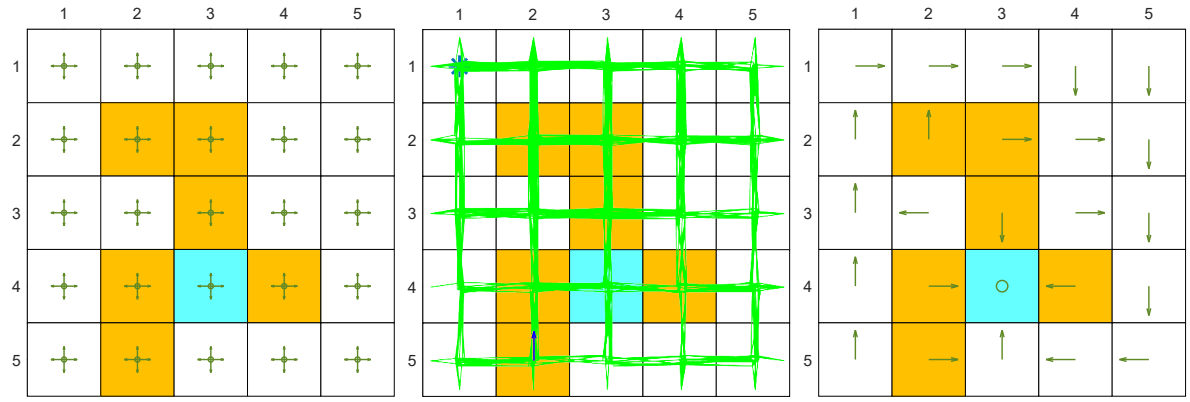
The implementation procedure of deep Q-learning is summarized in Algorithm 8.3. This implementation is off-policy. It can also be adapted to become on-policy if needed.

8.4.2 Illustrative examples

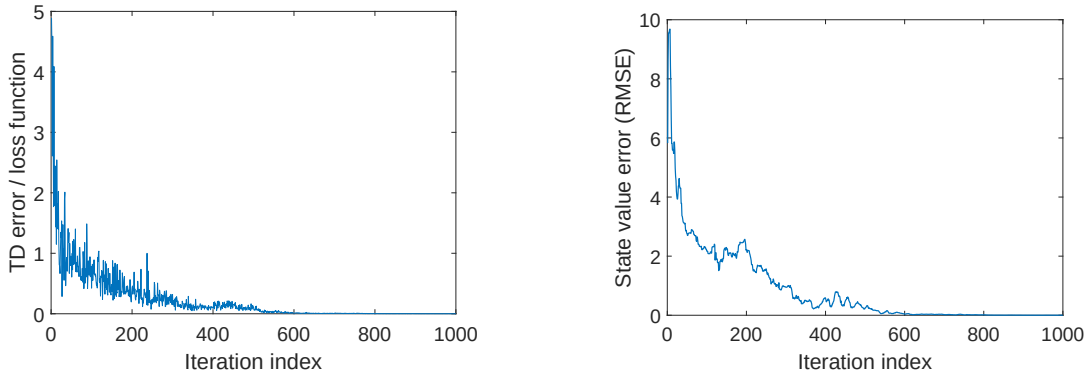
An example is given in Figure 8.11 to demonstrate Algorithm 8.3. This example aims to learn the optimal action values for *every* state-action pair. Once the optimal action values are obtained, the optimal greedy policy can be obtained immediately.

A single episode is generated by the behavior policy shown in Figure 8.11(a). This behavior policy is exploratory in the sense that it has the same probability of taking any action at any state. The episode has only 1,000 steps as shown in Figure 8.11(b). Although there are only 1,000 steps, almost all the state action pairs are visited in this episode due to the strong exploration ability of the behavior policy. The replay buffer is a set of 1,000 experience samples. The mini-batch size is 100, meaning that we uniformly draw 100 samples from the replay buffer every time we acquire samples.

The main and target networks have the same structure: a neural network with one hidden layer of 100 neurons (the numbers of layers and neurons can be tuned). The neural network has three inputs and one output. The first two inputs are the normalized row and column indexes of a state. The third input is the normalized action index. Here, "normalization" means converting a value to the interval of [0,1]. The output of the network is the estimated action value. The reason why we design the inputs as the row and column of a state rather than a state index is that we know that a state corresponds to a two-dimensional location in the grid. The more information about the state we use when designing the network, the better the network can perform. Moreover, the neural



(a) The behavior policy. (b) An episode with 1,000 steps. (c) The final learned policy.



(d) The loss function converges to zero. (e) The value error converges to zero.

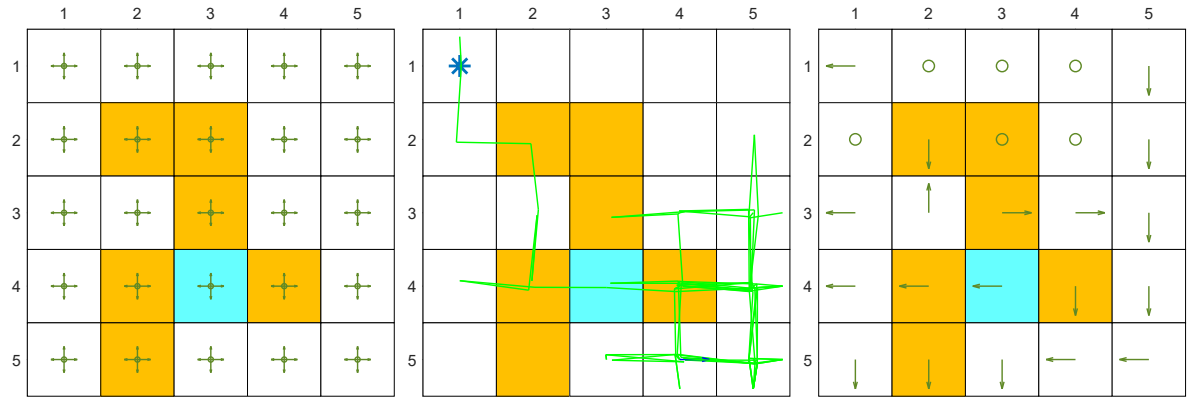
Figure 8.11: Optimal policy learning via deep Q-learning. Here, $\gamma = 0.9$, $r_{\text{boundary}} = r_{\text{forbidden}} = -10$, and $r_{\text{target}} = 1$. The batch size is 100.

network can also be designed in other ways. For example, it can have two inputs and five outputs, where the two inputs are the normalized row and column of a state and the outputs are the five estimated action values for the input state [22].

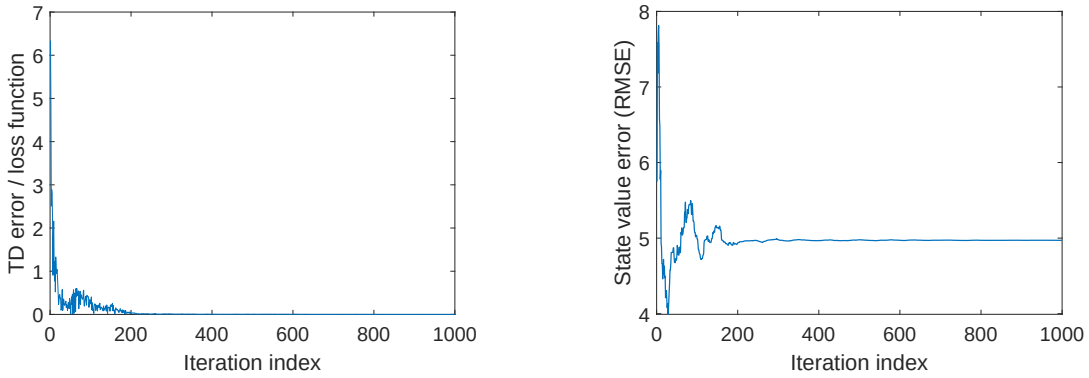
As shown in Figure 8.11(d), the loss function, defined as the average squared TD error of each mini-batch, converges to zero, meaning that the network can fit the training samples well. As shown in Figure 8.11(e), the state value estimation error also converges to zero, indicating that the estimates of the optimal action values become sufficiently accurate. Then, the corresponding greedy policy is optimal.

This example demonstrates the high efficiency of deep Q-learning. In particular, a short episode of 1,000 steps is sufficient for obtaining an optimal policy here. By contrast, an episode with 100,000 steps is required by tabular Q-learning, as shown in Figure 7.4. One reason for the high efficiency is that the function approximation method has a strong generalization ability. Another reason is that the experience samples can be repeatedly used.

We next deliberately challenge the deep Q-learning algorithm by considering a scenario with fewer experience samples. Figure 8.12 shows an example of an episode with merely 100 steps. In this example, although the network can still be well-trained in the sense



(a) The behavior policy. (b) An episode with 100 steps. (c) The final learned policy.



(d) The loss function converges to zero. (e) The value error does not converge to zero.

Figure 8.12: Optimal policy learning via deep Q-learning. Here, $\gamma = 0.9$, $r_{\text{boundary}} = r_{\text{forbidden}} = -10$, and $r_{\text{target}} = 1$. The batch size is 50.

that the loss function converges to zero, the state estimation error cannot converge to zero. That means the network can properly fit the given experience samples, but the experience samples are too few to accurately estimate the optimal action values.

8.5 Summary

This chapter continued introducing TD learning algorithms. However, it switches from the tabular method to the function approximation method. The key to understanding the function approximation method is to know that it is an optimization problem. The simplest objective function is the squared error between the true state values and the estimated values. There are also other objective functions such as the Bellman error and the projected Bellman error. We have shown that the TD-Linear algorithm actually minimizes the projected Bellman error. Several optimization algorithms such as Sarsa and Q-learning with value approximation have been introduced.

One reason why the value function approximation method is important is that it allows artificial neural networks to be integrated with reinforcement learning. For example, deep Q-learning is one of the most successful deep reinforcement learning algorithms.

Although neural networks have been widely used as nonlinear function approximators, this chapter provides a comprehensive introduction to the linear function case. Fully understanding the linear case is important for better understanding the nonlinear case. Interested readers may refer to [63] for a thorough analysis of TD learning algorithms with function approximation. A more theoretical discussion on deep Q-learning can be found in [61].

An important concept named stationary distribution is introduced in this chapter. The stationary distribution plays an important role in defining an appropriate objective function in the value function approximation method. It also plays a key role in Chapter 9 when we use functions to approximate policies. An excellent introduction to this topic can be found in [49, Chapter IV]. The contents of this chapter heavily rely on matrix analysis. Some results are used without explanation. Excellent references regarding matrix analysis and linear algebra can be found in [4, 48].

8.6 Q&A

◇ Q: What is the difference between the tabular and function approximation methods?

A: One important difference is how a value is updated and retrieved.

How to *retrieve* a value: When the values are represented by a table, if we would like to retrieve a value, we can directly read the corresponding entry in the table. However, when the values are represented by a function, we need to input the state index s into the function and calculate the function value. If the function is an artificial neural network, a forward prorogation process from the input to the output is needed.

How to *update* a value: When the values are represented by a table, if we would like to update one value, we can directly rewrite the corresponding entry in the table. However, when the values are represented by a function, we must update the function parameter to change the values indirectly.

◇ Q: What are the advantages of the function approximation method over the tabular method?

A: Due to the way state values are retrieved, the function approximation method is more efficient in storage. In particular, while the tabular method needs to store $|\mathcal{S}|$ values, the function approximation method only needs to store a parameter vector whose dimension is usually much less than $|\mathcal{S}|$.

Due to the way in which state values are updated, the function approximation method has another merit: its generalization ability is stronger than that of the tabular method. The reason is as follows. With the tabular method, updating one state value would not change the other state values. However, with the function approximation method, updating the function parameter affects the values of many states.

Therefore, the experience sample for one state can generalize to help estimate the values of other states.

◇ Q: Can we unify the tabular and the function approximation methods?

A: Yes. The tabular method can be viewed as a special case of the function approximation method. The related details can be found in Box 8.2.

◇ Q: What is the stationary distribution and why is it important?

A: The stationary distribution describes the long-term behavior of a Markov decision process. More specifically, after the agent executes a given policy for a sufficiently long period, the probability of the agent visiting a state can be described by this stationary distribution. More information can be found in Box 8.1.

The reason why this concept emerges in this chapter is that it is necessary for defining a valid objective function. In particular, the objective function involves the probability distribution of the states, which is usually selected as the stationary distribution. The stationary distribution is important not only for the value approximation method but also for the policy gradient method, which will be introduced in Chapter 9.

◇ Q: What are the advantages and disadvantages of the linear function approximation method?

A: Linear function approximation is the simplest case whose theoretical properties can be thoroughly analyzed. However, the approximation ability of this method is limited. It is also nontrivial to select appropriate feature vectors for complex tasks. By contrast, artificial neural networks can be used to approximate values as black-box universal nonlinear approximators, which are more friendly to use. Nevertheless, it is still meaningful to study the linear case to better grasp the idea of the function approximation method. Moreover, the linear case is powerful in the sense that the tabular method can be viewed as a special linear case (Box 8.2).

◇ Q: Why does deep Q-learning require experience replay?

A: The reason lies in the objective function in (8.37). In particular, to well define the objective function, we must specify the probability distributions of S, A, R, S'. The distributions of R and S' are determined by the system model once (S, A) is given. The simplest way to describe the distribution of the state-action pair (S, A) is to assume it to be *uniformly* distributed. However, the state-action samples may *not* be uniformly distributed in practice since they are generated as a sequence by the behavior policy. It is necessary to *break the correlation* between the samples in the sequence to satisfy the assumption of uniform distribution. To do this, we can use the experience replay technique by uniformly drawing samples from the replay buffer. A benefit of experience replay is that each experience sample may be used multiple times, which can increase the data efficiency.

◇ Q: Can tabular Q-learning use experience replay?

A: Although tabular Q-learning does not require experience replay, it can also use experience relay without encountering problems. That is because Q-learning has no requirements about how the samples are obtained due to its off-policy attribute. One benefit of using experience replay is that the samples can be used repeatedly and hence more efficiently.

◇ Q: Why does deep Q-learning require two networks?

A: The fundamental reason is to simplify the calculation of the gradient of (8.37). Specifically, the parameter w appears not only in $\hat{q}(S, A, w)$ but also in $R+\gamma \max_{a\in\mathcal{A}(S')} \hat{q}(S', a, w)$. As a result, it is nontrivial to calculate the gradient with respect to w. On the one hand, if we fix w in $R + \gamma \max_{a\in\mathcal{A}(S')} \hat{q}(S', a, w)$, the gradient can be easily calculated as shown in (8.38). This gradient suggests that two networks should be maintained. The main network's parameter is updated in every iteration. The target network's parameter is fixed within a certain period. On the other hand, the target network's parameter cannot be fixed forever. It should be updated every certain number of iterations.

◇ Q: When an artificial neural network is used as a nonlinear function approximator, how should we update its parameter?

A: It must be noted that we should not directly update the parameter vector by using, for example, (8.36). Instead, we should follow the network training procedure to update the parameter. This procedure can be realized based on neural network training toolkits, which are currently mature and widely available.