Chapter 8

Value Function Approximation

So far in this book, state and action values are represented by *tables*. Although such a tabular representation is intuitive, it would encounter some problems when the state or action space is large. For example, it may be impossible to store too many state or action values. Moreover, since the value of a state is updated only if it is visited, the values of unvisited states cannot be estimated. However, when the state space is large, it is difficult to ensure that all the states are visited. It is therefore favorable if the values of visited states can be generalized to unvisited states.

These problems can be solved if we approximate the state and action values using *parameterized functions*. More specifically, we can use $\hat{v}(s, w)$ as a parameterized function to approximate the true state value $v_\pi(s)$ of a policy π. Here, s is the state variable and $w \in \mathbb{R}^m$ is a parameter vector. The dimension of w may be much less than $|\mathcal{S}|$. In this way, we only need store the m-dimensional vector w (as well as the function structure) rather than the $|\mathcal{S}|$-dimensional state values $\{v_\pi(s)\}_{s\in\mathcal{S}}$. On the other hand, when a state s is visited, the parameter w is updated so that the values of some other unvisited states can also be updated. In this way, the learned values can generalize to unvisited states.

Another reason why the content of this chapter is important is that this is the venue where artificial neural networks are introduced into reinforcement learning (RL) as non-linear function approximators. In addition, the idea of *value function approximation* can be extended to *policy function approximation* as introduced in the next chapter.

8.1 Motivating examples: curve fitting

Now we use an example to demonstrate what function approximation is.

Suppose we have some states $s_1, \ldots, s_{|\mathcal{S}|}$. Their state values are $v_\pi(s_1), \ldots, v_\pi(s_{|\mathcal{S}|})$, where π is a given policy. If we plot the values, we get $|\mathcal{S}|$ dots as shown in Figure 8.1. Suppose $|\mathcal{S}|$ is very large and we hope to use a simple curve to approximate these dots to save storage. The simplest way is to use a straight line to fit the dots. Suppose the

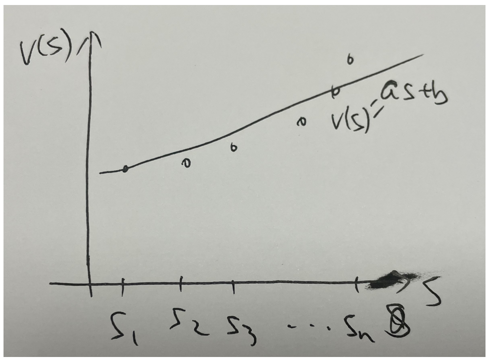


Figure 8.1: An illustration of function approximation of samples.

equation of the straight line is

$$v_\pi(s) \approx as + b,$$

where $a, b \in \mathbb{R}$. The equation can be rewritten as

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

where w is the *parameter vector* and $\phi(s)$ the *feature vector* of s. It is notable that $v_\pi(s)$ is *linear* in w.

The benefit of the approximation approach is obvious. While the tabular representation needs to store $|\mathcal{S}|$ state values, now we only need to store two parameters a and b. Every time we would like to access the value of a state, we only need to calculate the inner product between the parameter vector w and the feature vector $\phi(s)$ as indicated in (8.1). Such a benefit is, however, *not* free. It comes with a cost: the state values can not be represented accurately. This is not surprising because there does not exist a straight line that can perfectly fit all the points in Figure 8.1. That is why this method is called value *approximation*.

In addition to a straight line, we can approximate the points using a second-order polynomial curve as

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

In this case, the dimensions of w and $\phi(s)$ increase, but the values may be fitted more accurately. It must be noted that, although $v_\pi(s)$ is a *nonlinear* function in s, it is *linear* in w.

Of course, we can use even higher-order polynomial curves to fit the dots. It also needs more parameters. From a fundamental point of view, when we use a low-dimensional vector to represent a high-dimensional data set, some information will certainly be lost. Therefore, value function approximation enhances storage and computational efficiency by sacrificing accuracy.

In practice, which feature vector should we use to fit the data points? Should we fit the points as a first-order straight line or a second-order curve? The answer is nontrivial. That is because the selection of feature vector relies on certain domain knowledge. The better we understand a problem, the better feature vectors we can select. If we do not have domain knowledge, a popular solution is to use neural networks to let the network learn features and then fit the points automatically.

Another important problem is how to find the optimal parameter vector. This is a regression problem. We can find the optimal w by optimizing the following objective function:

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

where

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

This is a simple least-squares problem. The optimal value of w that can minimize J_1 can be obtained as

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

The curve fitting example presented in this section illustrates the basic idea of value function approximation. This idea will be formally and thoroughly studied in the rest of this chapter.

8.2 Objective function

Let $v_\pi(s)$ and $\hat{v}(s,w)$ be the true state value and approximated state value of $s \in \mathcal{S}$. Our goal is to find an *optimal* w so that $\hat{v}(s,w)$ can best approximate $v_\pi(s)$ for every s. To find the optimal w, we need *two steps*. The first step, as discussed in this section, is to define an objective function. The second step, as discussed in the next section, is to derive algorithms optimizing the objective function.

The objective function considered in value function approximation is

$$J(w) = \mathbb{E}[(v_\pi(S) - \hat{v}(S,w))^2]. \tag{8.2}$$

Our goal is to find the best w that can minimize $J(w)$. Here, the expectation is with respect to the random variable $S \in \mathcal{S}$. Thus, the objective function can be interpreted as the average approximation error over all states.

While S in (8.2) is a random variable, what is the probability distribution of S? This problem is often confusing to beginners because we have not discussed the probability distribution of states so far in this book. There are several ways to define the probability distribution of S.

– The first way is to use an *even distribution*. That is to treat all the states to be *equally important* by setting the probability of each state as $1/|\mathcal{S}|$. In this case, the objective function in (8.2) becomes

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

However, the importance of some states may be less than others. For example, some states may be rarely visited by a policy. Therefore, this way does not consider the real dynamics of the Markov process under the given policy.

– The second way, which is the focus of this chapter, is to use the *stationary distribution*. Stationary distribution is an important concept that will be frequently used in this book. In short, it describes the *long-run behavior* of a Markov process. Interested readers may see the details given in the shaded box.

Let $\{d_\pi(s)\}_{s \in \mathcal{S}}$ denote the stationary distribution of the Markov process under policy π. By definition, $d_\pi(s) \geq 0$ and $\sum_{s \in \mathcal{S}} d_\pi(s) = 1$. The objective function in (8.2) can be rewritten as

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

This function is a weighted squared error. Since more frequently visited states have higher values of $d_\pi(s)$, their weights in the objective function are also higher than those rarely visited states.

The value of $d_\pi(s)$ is nontrivial to obtain because it requires knowing the state transition probability matrix P_π (see the shaded box below). Fortunately, we do not need to calculate the specific value of $d_\pi(s)$ to minimize this objective function as shown in the next section.

Stationary distribution of a Markov process

Stationary distribution is also called *steady-state distribution*, or *limiting distribution*. It describes the long-run behavior of a Markov process. It is useful not only for value approximation as introduced in this chapter but also for policy approximation as introduced in the next chapter.

The key tool to analyze stationary distribution is $P_\pi \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}$, which is the probability transition matrix under the given policy π. If the states are indexed as $1,\dots,|\mathcal{S}|$, then $[P_\pi]_{ij}$ is the probability jumping from state i to state j.

1) What is the interpretation of P_π^k?

First of all, it is necessary to understand some properties of P_π^k $(k=1,2,3,\dots)$.

The probability of the state transiting from i to j using exactly k steps is denoted as

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

where t_0 and t_k are the initial and kth time steps, respectively. Apparently,

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

which means that $[P_\pi]_{ij}$ is the probability transiting from i to j using *a single step*. Now consider P_π^2. It can be verified that

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

Since $[P_\pi]_{iq}[P_\pi]_{qj}$ is the joint probability transiting from i to q and then from q to j, we know that $[P_\pi^2]_{ij}$ is the probability transiting from i to j using *exactly two steps*. That is

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

Similarly, we know that

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

which means that $[P_\pi^k]_{ij}$ is the probability transiting from i to j using *exactly k steps*.

2) Stationary probability distribution

Now we come back to the probability distribution of the states.

Let $d_0 \in \mathbb{R}^{|\mathcal{S}|}$ be the initial probability distribution vector of the states. For example, if s is always selected as the starting state, then $d_0(s)=1$ and the other entries of d_0 are 0. Let d_k be the probability distribution vector after exactly k steps starting from d_0. The relationship between d_k and d_0 is

$$d_k^T = d_0^T P_\pi^k.$$

Why multiply d_0 to the left of P_π^k? As aforementioned, the ith column of P_π^k is the probability reaching each state after k steps starting from state i.

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

$$P_\pi^k \to W = \mathbf{1}_n d_\pi^T, \quad \text{as} \quad k \to \infty, \tag{8.3}$$

where W is a constant matrix with all the rows equal to a vector denoted as d_π^T. This is an interesting result. The conditions under which (8.3) is valid will be discussed shortly. If (8.3) is valid, then we have

$$d_k^T = d_0^T P_\pi^k \to d_0^T W = d_0^T \mathbf{1}_n d_\pi^T = d_\pi^T \quad \text{as} \quad k \to \infty.$$

That means the state distribution converges to a constant value d_π, which is called the *stationary distribution*. The stationary distribution depends on the dynamics of the Markov process and hence the policy π. Interestingly, the stationary distribution is independent of the initial distribution d_0. That is, no matter which state the agent starts from, the state where the agent is located after a sufficiently long period can always be described by the stationary distribution.

What is the value of d_π? Since $d_k^T = d_{k-1}^T P_\pi$ and $d_k, d_{k-1} \to d_\pi$, we know

$$d_\pi^T = d_\pi^T P_\pi.$$

As a result, d_π is the left eigenvector of P_π associated with the eigenvalue of 1.

3) Under what conditions do stationary distributions exist?

A general class of Markov processes that have unique stationary distributions is *regular* Markov processes. To define "regular", we need first introduce some basic concepts.

– State j is said to be *accessible* from state i if $[P_\pi]_{ij}^k > 0$ for certain finite integer k, which means the agent starting from i can always reach j after a finite number of transitions.

– If two states i and j are mutually accessible, then the two states are said to *communicate*.

– A Markov process is called *irreducible* if all the states communicate with each other. In other words, the agent starting from an arbitrary state can always reach any other states within a finite number of steps.

– A Markov process is called *regular* if it is irreducible and, in the meantime, there exists a state i such that $[P_\pi]_{ii} > 0$. Here, $[P_\pi]_{ii} > 0$ means the agent may transit to i starting from i with one step.

For a regular Markov process, there exists a unique stationary distribution d_π such that $d_\pi^T = d_\pi^T P_\pi$. More details can be found in [18, Chapter IV].

4) What kind of policies can lead to stationary distributions?

Once the policy is determined, a Markov decision process becomes a Markov process, whose long-run behavior is jointly determined by the given policy and the system model. Then, an important question is what kind of policies can lead to regular Markov processes? In general, the answer is *exploration policies*. For example, ϵ-greedy policies introduced in the last chapter are exploratory and can often lead to regular Markov processes. That is because the ϵ-greedy policies are exploratory in the sense that each action has a positive probability to be taken. As a result, the states can communicate with each other under an ϵ-greedy when the system model allows.

5) Illustrative examples

We next show some examples to illustrate stationary distribution. We estimate the stationary distribution by using the fraction of the times each state is visited. In particular, suppose the number of times that the agent visits s in an episode generated following π is $n_\pi(s)$. Then $d_\pi(s)$ can be approximated by $d_\pi(s) \approx n_\pi(s)/\sum_{s'\in\mathcal{S}} n_\pi(s')$.

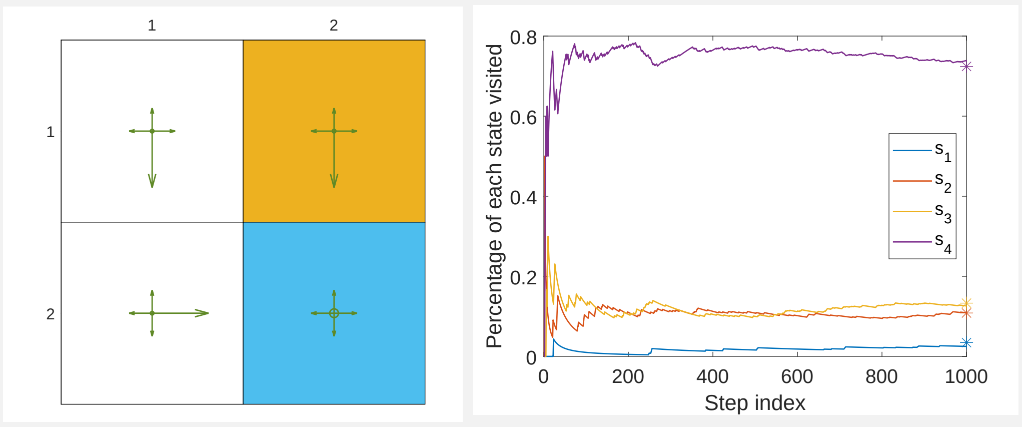


Figure 8.2: Long-run behavior of an ϵ-greedy policy with $\epsilon = 0.5$. The asterisks in the right figure represent the theoretical values of the elements in d_π.

First, consider the example in Figure 8.2. The policy in this example is ϵ-greedy with $\epsilon = 0.5$. The states are indexed as s_1, s_2, s_3, s_4, which respectively correspond to the top-left, top-right, bottom-left, bottom-right cells in the grid. It can be verified that the Markov process induced by the policy is regular. That is due to the following reasons. First, since all the states communicate, the resulting Markov process is irreducible. Second, since every state can transit to itself, the resulting Markov process is regular. Mathematically, the matrix P_π^T is

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

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

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

The ith element of d_π corresponds to the probability for the agent to be located on s_i in the long run.

We next verify the above theoretical value of d_π by executing the policy for sufficiently many steps in simulation and observing the long-run behavior. Specifically, we randomly select a starting state. For example, suppose the starting state is s_1. Following the policy, we run 1,000 steps. The number of each state visited during the process is shown in Figure 8.2. As can be seen, after hundreds of steps, the times each state visited converge to the theoretical values of d_π.

Second, consider the example in Figure 8.3. The policy in this example is ϵ-greedy with $\epsilon = 0.3$. It can also be verified that the Markov process induced by the policy is regular. The corresponding stationary distribution is $d_\pi = [0.0099, 0.0633, 0.0717, 0.8551]^T$. As shown in the right subfigure, the long-run behavior gradually converges to d_π. Compared to the first example, it can be seen in the second example that the probability of visiting s_4 is higher in the long run. That is simply because the policy is less exploratory due to a small value of ϵ and the target state s_4 is visited more frequently.

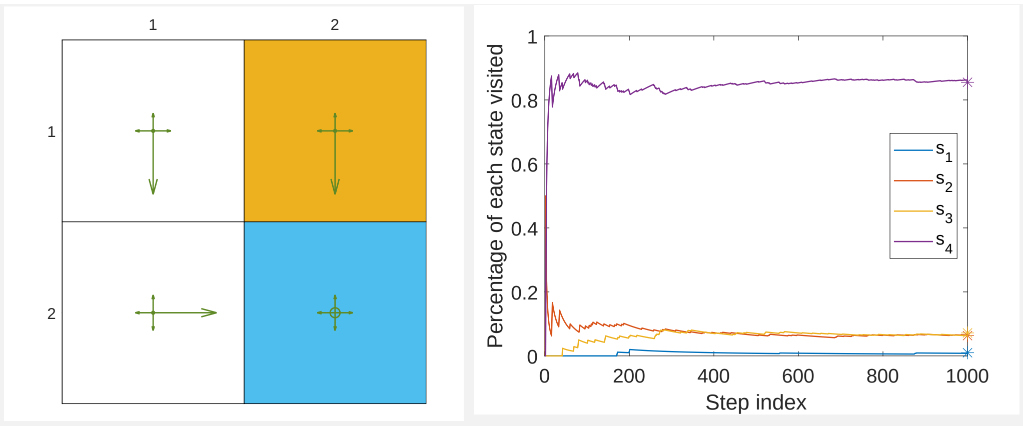


Figure 8.3: Long-run behavior of an ϵ-greedy policy with $\epsilon = 0.3$. The asterisks in the right figure represent the theoretical values of the elements in d_π.

Third, consider the example in Figure 8.4. The policy in this example is greedy with $\epsilon = 0$. The resulting Markov process is not irreducible because no state can communicate with each other. However, it can be verified that P_π^T in this case still has a single eigenvalue equal to 1 and the associated eigenvector is $d_\pi = [0, 0, 0, 1]^T$. The long-run behavior illustrated in Figure 8.4 is consistent with d_π. Compared to the previous two examples, the probability to visit the target state in the long run is as high as one since the policy here is greedy. This example also demonstrates that the condition that the Markov process should be regular is merely sufficient but not necessary to ensure a unique stationary distribution.

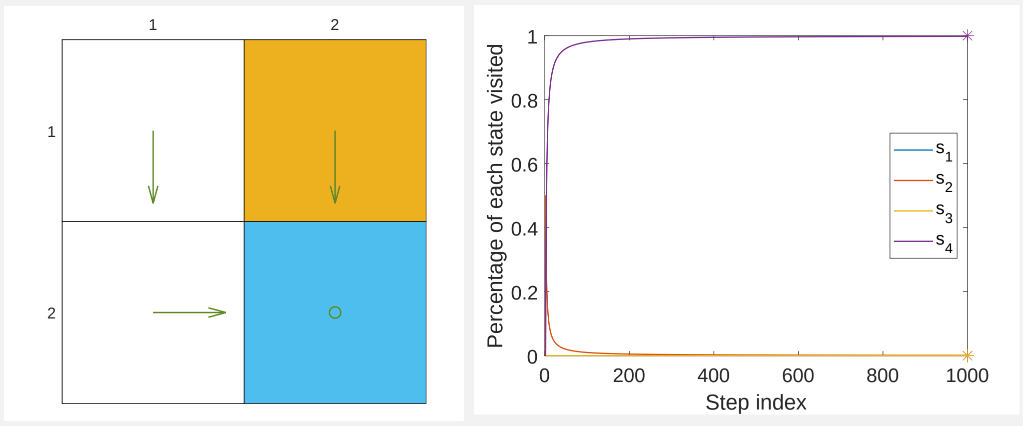


Figure 8.4: Long-run behavior of a greedy policy. The asterisks in the right figure represent the theoretical values of the elements in d_π.

8.3 Optimization algorithms

To minimize the objective function $J(w)$ in (8.2), we can use the gradient-descent algorithm:

$$\begin{aligned} w_{k+1} &= w_k - \alpha_k \nabla_w \mathbb{E}[(v_\pi(S) - \hat{v}(S,w))^2] \\ &= w_k - \alpha_k \mathbb{E}[\nabla_w (v_\pi(S) - \hat{v}(S,w))^2] \\ &= w_k - 2\alpha_k \mathbb{E}[(v_\pi(S) - \hat{v}(S,w))(-\nabla_w \hat{v}(S,w))] \\ &= w_k + 2\alpha_k \mathbb{E}[(v_\pi(S) - \hat{v}(S,w))\nabla_w \hat{v}(S,w)], \end{aligned} \tag{8.4}$$

where the coefficient 2 before α_k can be dropped without loss of generality. The above gradient-descent algorithm requires calculating the expectation. By the spirit of stochastic gradient descendent, we can remove the expectation operation from (8.4) to obtain the following algorithm:

$$w_{t+1} = w_t + \alpha_t (v_\pi(s_t) - \hat{v}(s_t, w_t))\nabla_w \hat{v}(s_t, w_t), \tag{8.5}$$

where s_t is a sample of S.

It must be noted that (8.5) is *not* implementable because it requires the true state value v_π, which is the unknown to be estimated. We can replace $v_\pi(s_t)$ with an approximation so that the algorithm is implementable. In particular, suppose we have an episode $(s_0, r_1, s_1, r_2, \dots)$.

– First, let g_t be the discounted return calculated starting from s_t in the episode. Then, g_t can be used as an approximation of $v_\pi(s_t)$. The algorithm in (8.5) hence becomes

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

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

– Second, by the spirit of TD learning, $r_{t+1} + \gamma \hat{v}(s_{t+1}, w_t)$ can be viewed as an approximation of $v_\pi(s_t)$. Then, the algorithm in (8.5) becomes

$$w_{t+1} = w_t + \alpha_t \left[r_{t+1} + \gamma \hat{v}(s_{t+1}, w_t) - \hat{v}(s_t, w_t) \right] \nabla_w \hat{v}(s_t, w_t). \tag{8.6}$$

This is the algorithm of TD learning with function approximation.

The TD algorithm in (8.6) is one of the focuses of this chapter. In the next two sections, we will continue to study this algorithm from the perspectives of function structure and convergence analysis. Notably, (8.6) can only approximate the *state values* of a given policy. We will extend this algorithm to Sarsa and Q-learning with function approximation that can approximate *action values* and then used to search for optimal policies in Sections 8.7 and 8.8.

Pseudocode: TD learning with function approximation

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

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

8.4 Linear function approximation

An important problem about the TD algorithm in (8.6) is how to select the function $\hat{v}(s,w)$. There are two potential approaches. The first approach, which is popular nowadays, is to use a neural network as a *nonlinear* approximator, where the input is the state, the output is $\hat{v}(s,w)$, and the network parameter is w. The second approach, which is the focus of this chapter, is to use a *linear* function

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

where $\phi(s) \in \mathbb{R}^m$ is the feature vector associated with s. The dimensions of the two vectors are the same as m, which is usually much smaller than $|\mathcal{S}|$. In the linear case, we have the gradient

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

Substituting the gradient into (8.6) yields

$$w_{t+1} = w_t + \alpha_t \big[r_{t+1} + \gamma\phi^T(s_{t+1})w_t - \phi^T(s_t)w_t\big]\phi(s_t), \tag{8.7}$$

which is the algorithm of TD learning with linear function approximation. It is called *TD-Linear* in short. Here we discuss some important problems with it.

– Why do we need to study linear function approximators while nonlinear neural networks are widely used nowadays? That is because the theoretical properties of the TD algorithm in the linear case can be much better understood than in the nonlinear case. For example, the TD algorithm guarantees to converge when linear functions are used, but may diverge when inappropriate nonlinear functions are used [19]. Fully understanding the linear case is necessary for us to appropriately use nonlinear approximators.

– Is linear function approximation powerful? Linear function approximation does have some limitations due to its simple linear form. However, it is still powerful. One piece of evidence is that the tabular representation is merely a special case of linear function approximation. To see that, consider the special feature vector for state s:

$$\phi(s) = e_{i(s)} \in \mathbb{R}^{|\mathcal{S}|},$$

where the subscript $i(s)$ is the index of state s, and the vector $e_{i(s)}$ is the vector with the $i(s)$th entry as 1 and the others as 0. In this case,

$$\hat{v}(s, w) = e_{i(s)}^T w = [w]_{i(s)},$$

where $[w]_{i(s)}$ is the $i(s)$th entry of w. In this case, w is exactly the true state value to be estimated and the TD algorithm becomes the normal tabular TD algorithm. Specifically, (8.7) in this case becomes

$$w_{t+1} = w_t + \alpha_t \left(r_{t+1} + \gamma [w_t]_{i(s_{t+1})} - [w_t]_{i(s_t)} \right) e_{i(s_t)}.$$

The above equation merely updates the $i(s_t)$th entry of w_t. Multiplying $e_{i(s_t)}^T$ on both sides of the equation gives

$$[w_{t+1}]_{i(s_t)} = [w_t]_{i(s_t)} + \alpha_t \left(r_{t+1} + \gamma [w_t]_{i(s_{t+1})} - [w_t]_{i(s_t)} \right).$$

By rewriting $[w_t]_{i(s_t)} = w_t(s_t)$, the above equation becomes

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

which is exactly the tabular TD algorithm.

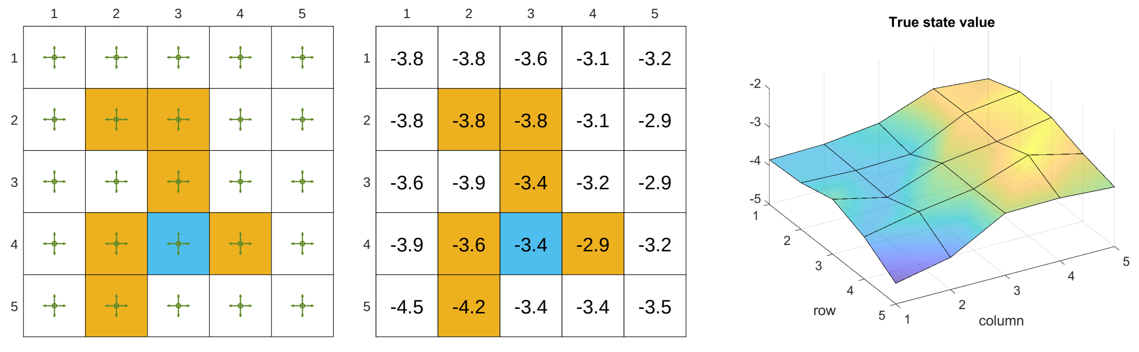
It is notable that, to achieve the lookup table representation, the dimension of w must be $|\mathcal{S}|$. When the dimension is reduced, some information about the state values will be lost and the linear form may not approximate the tabular case accurately.

8.5 Illustrative examples

We next present some examples to demonstrate how to use the TD-Linear algorithm in (8.7) to estimate the state values of a given policy. In the meantime, we illustrate how to select feature vectors.

Consider the grid-world example shown in Figure 8.5. The given policy takes any action at any state with the probability of 0.2. Our aim is to estimate the state values under this policy. There are 25 state values in total. We next show that we can use less than 25 parameters to approximate these state values. Set $r_{\text{forbidden}} = r_{\text{boundary}} = -1$, $r_{\text{target}} = 1$, and $\gamma = 0.9$. The true state values are given in Figure 8.5(b). The true state values are further visualized as a three-dimensional surface in Figure 8.5(c).

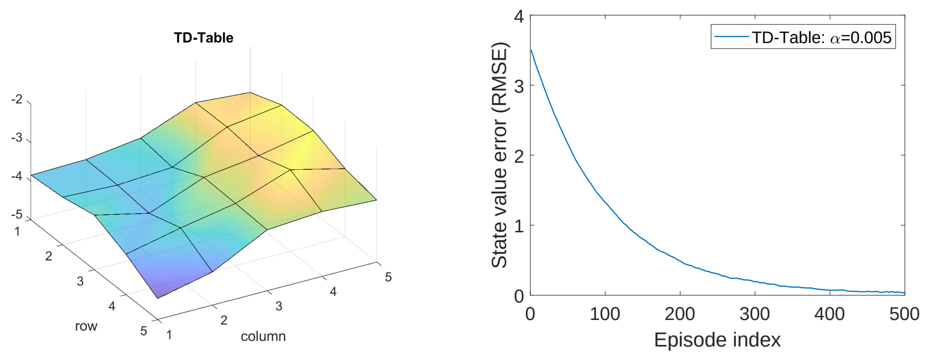
To estimate the state values, 500 episodes were generated following the given policy.



	1	2	3	4	5
1	-3.8	-3.8	-3.6	-3.1	-3.2
2	-3.8	-3.8	-3.8	-3.1	-2.9
3	-3.6	-3.9	-3.4	-3.2	-2.9
4	-3.9	-3.6	-3.4	-2.9	-3.2
5	-4.5	-4.2	-3.4	-3.4	-3.5

(a) (b) (c)

Figure 8.5: A policy and its true state values. (a) The policy to be evaluated. (b) The true state values are represented in a table. (c) The true state values are visualized as a 3D surface.



(a) (b)

Figure 8.6: TD-Table estimation results. (a) The estimated state values are visualized as a 3D surface. (b) The evolution of the state estimation error, which is the root-mean-squared error (RMSE).

Each episode has 500 steps and starts from a randomly selected state-action pair following a uniform distribution. For comparison, we first estimate the state values using the tabular TD algorithm introduced in Chapter 7 (TD-Table in short). The results are shown in Figure 8.6. As can be seen, the TD-Table algorithm can accurately estimate the state values.

To implement the TD-Linear algorithm, it is necessary to select the feature vector $\phi(s)$ first. Our goal here is to use $\phi^T(s)w$ to approximate the three-dimensional (3D) surface representing the true state values as shown in Figure 8.5(c). When implementing the TD-Linear algorithm, we initialize w in the way that each element is randomly drawn from a standard normal distribution where the mean is 0 and the standard deviation is 1.

– The first type of feature vector is based on polynomials. In the grid-world example, a state s corresponds to a 2D location. Let x and y denote the column and row indexes of s. In the implementation, we normalize x and y so that their values are within the interval of [-1,+1]. With abused notations, the normalized values are also represented by x and y. Then, the simplest feature vector is

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

In this case, we have

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

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

$$\phi(s) = \begin{bmatrix} 1 \\ x \\ y \end{bmatrix} \in \mathbb{R}^3. \tag{8.8}$$

In this case, the approximated state value is

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

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

The estimation result when we use the feature vector in (8.8) is given in Figure 8.7(a). As can be seen, all the estimated state values form a 2D plane. Although the estimation error decreases as more episodes are used, the error cannot converge to zero due to the limited approximation ability of $\hat{v}(s,w)$ in this case.

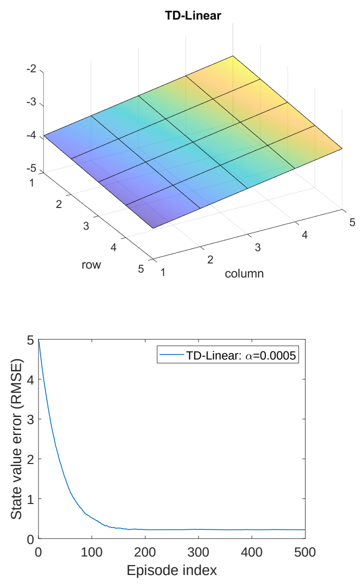
To enhance the approximation ability, we can increase the dimension of the feature vector. To that end, we can consider

$$\phi(s) = [1, x, y, x^2, y^2, xy]^T \in \mathbb{R}^6. \tag{8.9}$$

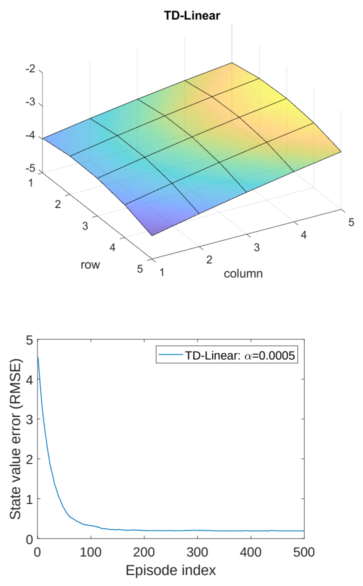
In this case, $\hat{v}(s,w) = \phi^T(s)w = w_1 + w_2 x + w_3 y + w_4 x^2 + w_5 y^2 + w_6 xy$ corresponds to a quadratic 3D surface. We can further increase the dimension of the feature vector to use

$$\phi(s) = [1, x, y, x^2, y^2, xy, x^3, y^3, x^2 y, xy^2]^T \in \mathbb{R}^{10}. \tag{8.10}$$

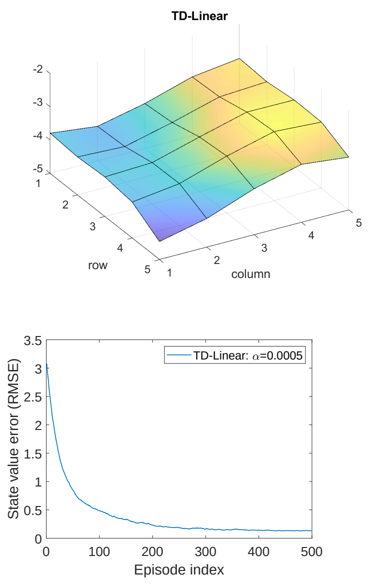
The estimation results when we use the feature vectors in (8.9) and (8.10) are given in Figure 8.7(b)-(c). As can be seen, the higher the dimension of the feature vector is, the more accurate the state values can be approximated. However, for all three cases, the estimation error cannot converge to zero because these linear approximators still have limited approximation ability.



(a) $\phi(s) \in \mathbb{R}^3$

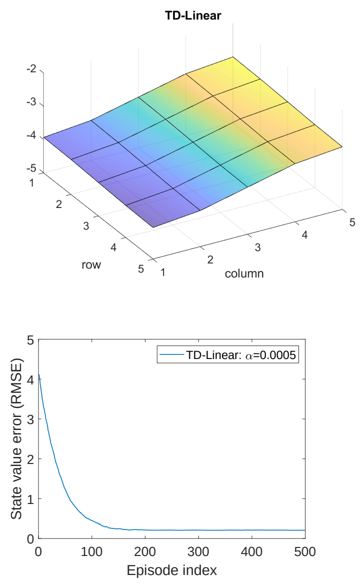


(b) $\phi(s) \in \mathbb{R}^6$

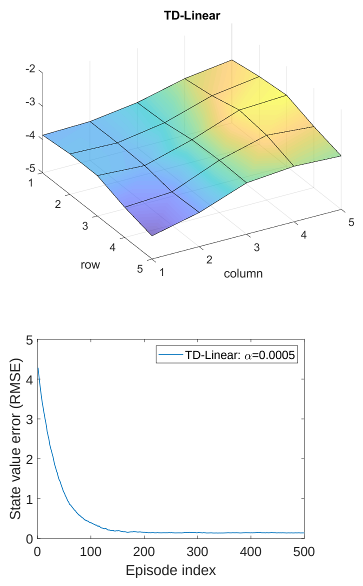


(c) $\phi(s) \in \mathbb{R}^{10}$

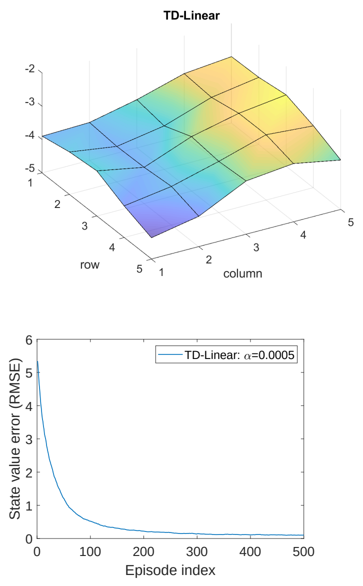
Figure 8.7: TD-Linear estimation results with polynomial features given in (8.8), (8.9), and (8.10).



(a) $q = 1$ and $\phi(s) \in \mathbb{R}^4$



(b) $q = 2$ and $\phi(s) \in \mathbb{R}^9$



(c) $q = 3$ and $\phi(s) \in \mathbb{R}^{16}$

Figure 8.8: TD-Linear estimation results with the Fourier features given in (8.11).

– In addition to polynomial feature vectors, there are many other types of features such as Fourier basis and tile coding. An excellent introduction can be found in [3, Chapter 9]. We next demonstrate how to apply Fourier features to the grid-world example. Fourier features show good performance when applied in Sarsa for searching optimal policies, which will be shown in Section 8.7 later. We normalize x and y of each state to the interval of $[0,1]$. The feature vector is

$$\phi(s)=\begin{bmatrix}\vdots\\ \cos\big(\pi(c_1x+c_2y)\big)\\ \vdots\end{bmatrix}\in\mathbb{R}^{(q+1)^2}. \tag{8.11}$$

Here, π denotes the circumference ratio, which is $3.14\dots$, instead of a policy. More importantly, c_1, c_2 can take any integer in $\{0,1,\dots,q\}$, where q is a user-selected integer. As a result, there are $(q+1)^2$ possible ways for c_1, c_2 taking values. Hence, the dimension of $\phi(s)$ is $(q+1)^2$. For example, consider the case of $q=1$. Then, c_1, c_2 take values in $\{0,1\}$. The feature vector in this case is

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

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

8.6 Theoretical analysis of TD learning

Up to now, we finished the story for introducing TD learning with function approximation. This story started from the objective function in (8.2). To optimize this objective function, we introduced the stochastic algorithm in (8.5). The true value function, which is unknown, in the algorithm is replaced by an approximation, leading to the TD algorithm in (8.6).

Although this story is helpful to understand the basic idea of value function approximation, it is not mathematically rigorous. For example, the algorithm in (8.6) actually does not minimizes the objective function in (8.2) as will be shown in this section.

This section presents a theoretical analysis of the TD algorithm in (8.6) to reveal why the algorithm works effectively and what mathematical problems it solves. Since general nonlinear approximators are difficult to analyze, this section only focuses on the linear cases.

8.6.1 Convergence analysis

To study the convergence of (8.6), we first consider the following deterministic algorithm:

$$w_{t+1} = w_t + \alpha_t \mathbb{E}\Big[\big(r_{t+1} + \gamma \phi^T(s_{t+1})w_t - \phi^T(s_t)w_t\big)\phi(s_t)\Big], \tag{8.12}$$

where the expectation is with respect to the random variables s_t, s_{t+1}, r_{t+1}. The distribution of s_t is assumed to the stationary distribution d_π. The algorithm in (8.12) is deterministic because the random variables s_t, s_{t+1}, r_{t+1} will all disappear after calculating the expectation.

Why would we consider this deterministic algorithm? First, the convergence of this deterministic algorithm is relatively easier (though nontrivial) to analyze as shown later. Second and more importantly, the convergence of this deterministic algorithm implies the convergence of the stochastic TD algorithm in (8.6). That is because (8.6) can be viewed as a stochastic gradient-descent (SGD) implementation of (8.12). Therefore, we only focus on the convergence of the deterministic algorithm in the rest of this section.

The expression of (8.12) may look complex at first glance. It can be written in a concise expression. To do that, define

$$\Phi = \begin{bmatrix} \vdots \\ \phi^T(s) \\ \vdots \end{bmatrix} \in \mathbb{R}^{|S|\times m}, \quad D = \begin{bmatrix} \ddots & & \\ & d_\pi(s) & \\ & & \ddots \end{bmatrix} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}, \tag{8.13}$$

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

Lemma 8.1. *The expectation in* (8.12) *can be written as*

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

where

$$\begin{aligned} A &= \Phi^T D(I - \gamma P_\pi)\Phi \in \mathbb{R}^{m\times m}, \\ b &= \Phi^T D r_\pi \in \mathbb{R}^m. \end{aligned} \tag{8.14}$$

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

The proof can be found in a shaded box given later.

With the expression in Lemma 8.1, the deterministic algorithm in (8.12) can be rewritten as

$$w_{t+1} = w_t + \alpha_t(b - Aw_t), \tag{8.15}$$

which is a simple iterative algorithm. The convergence properties of this algorithm are analyzed below.

First, what is the eventually converged value of w_t? Hypothetically, if w_t converges to a constant value w^* as $t \to \infty$, then (8.15) implies $w^* = w^* + \alpha_\infty(b - Aw^*)$, which suggests that $b - Aw^* = 0$ and hence

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

Several remarks about this steady-state value are given below.

- Is A invertible? The answer is yes. In fact, A is not only invertible but also positive definite. That is, for any nonzero vector x of appropriate dimension, $x^T Ax > 0$. This property is proven in detail in the shaded box given later.
- What is the meaning of $w^* = A^{-1}b$? It is actually the optimal solution for minimizing the *projected Bellman error*, which will be introduced in the next section. One interesting conclusion is that, when the dimension of w is the same as $|\mathcal{S}|$ and $\phi(s_i) = [0, \ldots, 1, \ldots, 0]^T$ where the entry corresponding to s_i is 1, we have

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

  To see that, we have $\Phi = I$ in this case and hence

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

  Thus

  $$w^* = A^{-1}b = (I - \gamma P_\pi)^{-1} D^{-1} D r_\pi = (I - \gamma P_\pi)^{-1} r_\pi = v_\pi.$$

  Although it is a special case, the fact that w^* becomes v_π in this case gives us more confidence about the algorithm.

Second, we next that $w_t \to w^* = A^{-1}b$ as $t \to \infty$. In fact, since $w_{t+1} = w_t + \alpha_t(b - Aw_t)$ is a relatively simple algorithm, it can be proved in many ways.

- The first way is to consider $g(w) \doteq b - Aw$. Since w^* is the root of $g(w) = 0$, it is actually a root-finding problem. Such a problem can be solved by the Robbins-Monro (RM) algorithm introduced in Chapter 6. The algorithm $w_{t+1} = w_t + \alpha_t(b - Aw_t)$ is in fact an RM algorithm. The convergence of RM algorithms can shed light on the convergence of $w_{t+1} = w_t + \alpha_t(b - Aw_t)$. That is w_t converges to w^* when $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$.

– The second way is to study the convergence error $\delta_t = w_t - w^*$. We only need to show that δ_t converges to zero. To do that, substituting $w_t = \delta_t + w^*$ into $w_{t+1} = w_t + \alpha_t(b - Aw_t)$ gives

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

and hence

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

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

$$\|\delta_{t+1}\| \le \|I - \alpha A\|^{t+1}\|\delta_0\|,$$

where $\|\cdot\|$ denotes the 2-norm of a vector or matrix. When $\alpha > 0$ is sufficiently small, the spectral radius of $I - \alpha A$ is strictly less than one and hence δ_t converges to zero.

Proof of Lemma 8.1

By using conditional expectation, we have

$$\begin{aligned}
&\mathbb{E}\Big[r_{t+1}\phi(s_t) + \phi(s_t)\big(\gamma\phi^T(s_{t+1}) - \phi^T(s_t)\big)w_t\Big] \\
&= \sum_{s\in\mathcal{S}} d_\pi(s)\mathbb{E}\Big[r_{t+1}\phi(s_t) + \phi(s_t)\big(\gamma\phi^T(s_{t+1}) - \phi^T(s_t)\big)w_t \big| s_t = s\Big] \\
&= \sum_{s\in\mathcal{S}} d_\pi(s)\mathbb{E}\Big[r_{t+1}\phi(s_t)\big|s_t = s\Big] + \sum_{s\in\mathcal{S}} d_\pi(s)\mathbb{E}\Big[\phi(s_t)\big(\gamma\phi^T(s_{t+1}) - \phi^T(s_t)\big)w_t\big|s_t = s\Big],
\end{aligned} \tag{8.16}$$

where d_π is the stationary distribution under policy π. Therefore, there is an implicit assumption that s_t obeys the stationary distribution, which requires that 1) the process has run for a sufficiently long time so that stationary distribution has been achieved, and 2) the experience sample used for training is on-policy, which means the sampling must follow the given policy π.

First, consider the first term in (8.16). Since

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

where $r_\pi(s) = \sum_a \pi(a|s)\sum_r rp(r|s,a)$, the first term in (8.16) can be written as

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

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

Second, consider the second term in (8.16). Since

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

the second term in (8.16) becomes

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

Combining the above two terms we have

$$\mathbb{E}\Big[\big(r_{t+1}+\gamma\phi^T(s_{t+1})w_t-\phi^T(s_t)w_t\big)\phi(s_t)\Big]=\Phi^TDr_\pi-\Phi^TD(I-\gamma P_\pi)\Phi w_t\doteq b-Aw_t \tag{8.17}$$

where $b=\Phi^TDr_\pi$ and $A=\Phi^TD(I-\gamma P_\pi)\Phi$.

Prove that $A=\Phi^TD(I-\gamma P_\pi)\Phi$ is positive definite.

The matrix A is positive definite if $x^TAx>0$ for any nonzero vector x of appropriate dimension. If A is positive (or negative) definite, then it is denoted as $A\succ 0$ (or $A\prec 0$). Here, $\succ$ and $\prec$ should be differentiated from $>$ and $<$, which indicates an elementwise comparison. Note that A may not be symmetric since P_π may not be symmetric. While positive definite matrices often refer to symmetric matrices, non-symmetric ones can also be positive definite.

To show $A\succ 0$, we first show that

$$D(I-\gamma P_\pi)\doteq M\succ 0.$$

It is clear that $M \succ 0$ implies $A \succ 0$ since Φ is a tall matrix with full column rank (if the feature vectors are independent). Note that

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

Since $M - M^T$ is skew-symmetric and hence $x^T(M - M^T)x = 0$ for any x, we know that $M \succ 0$ if and only if $M + M^T \succ 0$.

To show $M + M^T \succ 0$, we apply the fact that strictly diagonal dominant matrices are positive definite [20].

First, we show

$$(M + M^T)\mathbf{1} > 0, \tag{8.18}$$

where $\mathbf{1} = [1, \dots, 1]^T \in \mathbb{R}^{|\mathcal{S}|}$. The proof of (8.18) is given below. Since $P_\pi \mathbf{1} = \mathbf{1}$, we have $M\mathbf{1} = D(I - \gamma P_\pi)\mathbf{1} = D(\mathbf{1} - \gamma \mathbf{1}) = (1 - \gamma) d_\pi$, where $d_\pi = [d_\pi(s_1), \dots, d_\pi(s_n)]^T$. On the other hand, $M^T\mathbf{1} = (I - \gamma P_\pi^T) D \mathbf{1} = (I - \gamma P_\pi^T) d_\pi = (1 - \gamma) d_\pi$, where the last equality is because $P_\pi^T d_\pi = d_\pi$. In summary,

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

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

Second, the element-wise form of (8.18) is

$$\sum_{j=1}^{|\mathcal{S}|} [M + M^T]_{ij} > 0, \quad \text{for all } i$$

which can be further written as

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

It can be verified that the diagonal entries of M are positive and the off-diagonal entries of M are non-positive. Therefore, the above inequality can be rewritten as

$$\left|[M + M^T]_{ii}\right| > \sum_{j \neq i} \left|[M + M^T]_{ij}\right|, \quad i = 1, \dots, |\mathcal{S}|.$$

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

8.6.2 TD learning minimizes the projected Bellman error

While we have shown in the last subsection that the TD algorithm will converge to $w^* = A^{-1}b$, it is still unclear what the interpretation of this solution is. In this section, we show that w^* is the optimal solution minimizing the *projected Bellman error*.

First of all, we revisit the idea of value function approximation as an optimization problem. The objective function introduced in (8.2) is

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

In a matrix-vector form, $J_E(w)$ can be expressed as

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

where v_π is the true state value vector, $\hat{v}(w)$ the approximated one, and $\|\cdot\|_D^2$ is the 2-norm weighted by a diagonal matrix D. In particular, $\|x\|_D^2 = x^T D x = \|D^{1/2}x\|_2^2$. Here, D is the matrix with the equilibrium state distribution on the diagonal.

This is one of the first objective functions that we can imagine when talking about function approximation. However, it relies on the true state, which is unknown. To obtain an implementable algorithm, we have to consider other objective functions such as the *Bellman error* and *projected Bellman error* [21–25].

The idea of Bellman error minimization is as follows. Since v_π satisfies the Bellman equation $v_\pi = r_\pi + \gamma P_\pi v_\pi$, it is expected that the approximated one $\hat{v}(w)$ should satisfy this equation as much as possible. Thus, the Bellman error is

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

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

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

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

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

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

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

In fact, TD learning does *not* optimize this objective function. Instead, it minimizes a *projected Bellman error* as shown below.

While the orthogonal projection may be complex for general nonlinear cases, it is easy to analyze in the linear case where $\hat{v}(w) = \Phi w$. Here, Φ is defined in (8.13). The range space of Φ is the set of all possible linear approximations. Then,

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

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

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

The proof is given in the shaded box below. Therefore, the TD-Linear algorithm actually minimizes the projected Bellman error rather than (8.2).

Show that $w^* = A^{-1}b$ minimizes $J_{PBE}(w)$

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

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

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

$$\Phi w = \Phi(\Phi^T D \Phi)^{-1}\Phi^T D(r_\pi + \gamma P_\pi \Phi w). \tag{8.19}$$

Since Φ has full column rank, we have $\Phi x = \Phi y \Leftrightarrow x = y$ for any x, y. Therefore, (8.19) implies

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

where A, b are given in (8.14).

How far is the solution minimizing the projected Bellman error from the true state value v_π? If we use a linear approximator, the approximated value that minimizes the projected Bellman error is Φw^*. Its difference from the true state value v_π is

$$\|\Phi w^* - v_\pi\|_D \le \frac{1}{1-\gamma}\min_w \|\hat{v}(w) - v_\pi\|_D = \frac{1}{1-\gamma}\min_w \sqrt{J_E(w)}. \tag{8.20}$$

That is, the difference between Φw^* and v_π is bounded from above by the minimum value of $J_E(w)$. This bound provides more confidence about the TD algorithm with linear value approximation. However, this bound is loose, especially when γ is close to one. It is thus mainly of the theoretical value.

Proof of (8.20)

Note that

$$\begin{aligned}
\|\Phi w^* - v_\pi\|_D &= \|\Phi w^* - Mv_\pi + Mv_\pi - v_\pi\|_D \\
&\le \|\Phi w^* - Mv_\pi\|_D + \|Mv_\pi - v_\pi\|_D \\
&= \|MT_\pi(\Phi w^*) - MT_\pi(v_\pi)\|_D + \|Mv_\pi - v_\pi\|_D,
\end{aligned}$$

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

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

we have

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

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

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

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

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

– Properties of matrix weighted norms. By definition, $\|x\|_D = \sqrt{x^T D x} = \|D^{1/2}x\|_2$.

The induced matrix norm is $\|A\|_D = \max_{x\neq 0}\|Ax\|_D/\|x\|_D = \|D^{1/2}AD^{-1/2}\|_2$. For matrices A, B of appropriate dimensions, we have $\|ABx\|_D \leq \|A\|_D\|B\|_D\|x\|_D$. To see that, $\|ABx\|_D = \|D^{1/2}ABx\|_2 = \|D^{1/2}AD^{-1/2}D^{1/2}BD^{-1/2}D^{1/2}x\|_2 \leq \|D^{1/2}AD^{-1/2}\|_2\|D^{1/2}BD^{-1/2}\|_2\|D^{1/2}x\|_2 = \|A\|_D\|B\|_D\|x\|_D$.

– Proof of $\|M\|_D = 1$. That is because $\|M\|_D = \|\Phi(\Phi^TD\Phi)^{-1}\Phi^TD\|_D = \|D^{1/2}\Phi(\Phi^TD\Phi)^{-1}\Phi^TDD^{-1/2}\|_2 = 1$, where the last equality is because the matrix in the 2-norm is an orthogonal projection matrix and the 2-norm of any orthogonal projection matrices is equal to one.

– Proof of $\|P_\pi x\|_D \leq \|x\|_D$ any $x \in \mathbb{R}^{|\mathcal{S}|}$. First,

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

Reorganizing the above equation gives

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

8.6.3 Least-squares TD

This section introduces an algorithm called *least-squares TD* (LSTD) [26]. It is another algorithm that minimizes the projected Bellman error.

Recall that the optimal parameter to minimize the projected Bellman error is $w^* = A^{-1}b$, where $A = \Phi^TD(I-\gamma P_\pi)\Phi$ and $b = \Phi^TDr_\pi$. In fact, it follows from (8.17) that A and b can also be written as

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

The above two equations show that A and b are expectations of some random variables. The *idea* of LSTD is simple: if we can use random samples to directly obtain the estimates of A and b, denoted as $\hat{A}$ and $\hat{b}$, then the optimal parameter can be directly estimated as

$w^* \approx \hat{A}^{-1}\hat{b}$.

In particular, suppose that $(s_0, r_1, s_1, \dots, r_t, s_t, \dots)$ is a trajectory obtained by following the policy. Let $\hat{A}_t$ and $\hat{b}_t$ be the estimates of A and b, respectively, at time t. They are calculated as

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

Then, the estimated parameter is

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

Since $\hat{A}_t$ is required to be invertible, $\hat{A}_t$ is usually biased by a small constant matrix δI where I is the identity matrix and δ is a small positive number.

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

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

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

The above expression decomposes $\hat{A}_{t+1}$ into the sum of two matrices. Its inverse can be calculated as [27]

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

Therefore, we can directly store and update $\hat{A}_t^{-1}$ to avoid the need to calculate the matrix inverse. The initial value of such recursive algorithm can be selected as $\hat{A}_0^{-1} = \sigma I$, where σ is a positive number. This recursive algorithm does not require setting the step size. However, it requires setting the initial value of $\hat{A}_0^{-1}$. A good tutorial on recursive least squares can be found in [28].

8.7 Sarsa with function approximation

So far in this chapter, we merely considered the problem of state value estimation. To search for optimal policies, we need to estimate action values. This section introduces how to estimate action values using Sarsa in the presence of value function approximation.

There are two steps in every iteration of Sarsa with function approximation. The first is policy evaluation, which is to estimate the action values. The second is policy improvement, which is to find the soft greedy policy based on the current action values. Its difference from tabular Sarsa lies only in the first step for action value estimation. In particular, the action value $q_\pi(s,a)$ is described by a function $\hat{q}(s,a,w)$ parameterized by w. Replacing $\hat{v}(s,w)$ by $\hat{q}(s,a,w)$ in (8.6) gives the following update rule of Sarsa learning with furcation approximation:

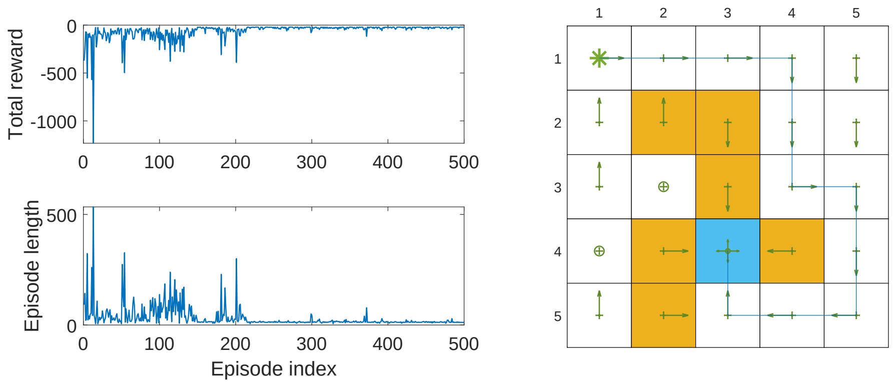
$$w_{t+1} = w_t + \alpha_t\Big[r_{t+1} + \gamma\hat{q}(s_{t+1},a_{t+1},w_t) - \hat{q}(s_t,a_t,w_t)\Big]\nabla_w\hat{q}(s_t,a_t,w_t). \tag{8.21}$$

The analysis of (8.21) is also similar to (8.6) and omitted here. When linear functions are used, we have

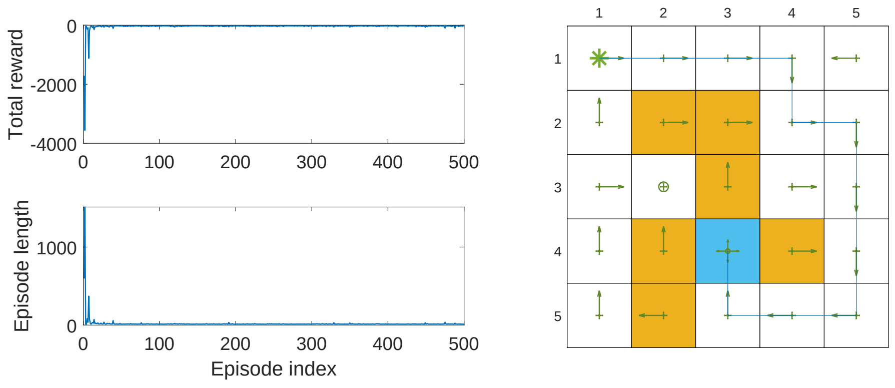
$$\hat{q}(s,a,w) = w^T\phi(s,a),$$

where $\phi(s,a)$ is a feature vector. In this case, $\nabla_w\hat{q}(s,a,w) = \phi(s,a)$. The implementation details can be found in the pseudocode of Sarsa with function approximation. It should be noted that accurately estimating the action values of a given policy needs to run (8.21) sufficiently many times. However, (8.21) is merely executed once before switching to the policy improvement step. This follows the exactly same idea as the tabular Sarsa algorithm. Moreover, the pseudocode aims to solve a task where the policy should lead the agent to the target state starting from a specific state. As a result, the algorithm may not be able to find the optimal policy for every state. Of course, if more experience data is available, the optimal policy for every state can also be obtained.

An illustrative example is given in Figure 8.9(a). In this example, the task is to find a good policy that can lead the agent to the target starting from the top-left state. As can be seen, both the total reward and length of the trajectory gradually converge to steady values. In this example, the linear Fourier function bases of order 5 are used.



(a) Sarsa with linear function approximation.



(b) Q-learning with linear function approximation

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

8.8 Q-learning with function approximation

Similar to Sarsa, tabular Q-learning can also be extended to the case of value function approximation. The update rule is

$$w_{t+1} = w_t + \alpha_t \Big[r_{t+1} + \gamma \max_{a \in \mathcal{A}(s_{t+1})} \hat{q}(s_{t+1}, a, w_t) - \hat{q}(s_t, a_t, w_t) \Big] \nabla_w \hat{q}(s_t, a_t, w_t), \tag{8.22}$$

which is the same as Sarsa in (8.21) except that $\hat{q}(s_{t+1}, a_{t+1}, w_t)$ in (8.21) is replaced by $\max_{a \in \mathcal{A}(s_{t+1})} \hat{q}(s_{t+1}, a, w_t)$.

Similar to the tabular case, (8.22) can be implemented in either on-policy or off-policy fashion (see the pseudocode). An example demonstrating the on-policy version is given in Figure 8.9(b). In this example, the task is to find a good policy that can lead the agent

Pseudocode: Sarsa with function approximation

```
Initialization: Initial parameter vector $w_0$. Initial policy $\pi_0$.
Aim: Search a policy that can lead the agent to the target from an initial state-action pair $(s_0, a_0)$.

For each episode, do
    If the current $s_t$ is not the target state, do
        Take action $a_t$ following $\pi_t(s_t)$, generate $r_{t+1}, s_{t+1}$, and then take action $a_{t+1}$ following $\pi_t(s_{t+1})$
        Update parameter:
        $w_{t+1} = w_t + \alpha_t\Big[r_{t+1} + \gamma\hat{q}(s_{t+1}, a_{t+1}, w_t) - \hat{q}(s_t, a_t, w_t)\Big]\nabla_w\hat{q}(s_t, a_t, w_t)$
        Update policy:
        $\pi_{t+1}(a|s_t) = 1 - \frac{\varepsilon}{|\mathcal{A}(s)|}(|\mathcal{A}(s)| - 1)$ if $a = \arg\max_{a\in\mathcal{A}(s_t)} \hat{q}(s_t, a, w_{t+1})$
        $\pi_{t+1}(a|s_t) = \frac{\varepsilon}{|\mathcal{A}(s)|}$ otherwise
```

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

```
Initialization: Initial parameter vector $w_0$. Initial policy $\pi_0$. Small $\varepsilon > 0$.
Aim: Search a good policy that can lead the agent to the target from an initial state-action pair $(s_0, a_0)$.

For each episode, do
    If the current $s_t$ is not the target state, do
        Take action $a_t$ following $\pi_t(s_t)$, and generate $r_{t+1}, s_{t+1}$
        Update parameter:
        $w_{t+1} = w_t + \alpha_t\Big[r_{t+1} + \gamma \max_{a\in\mathcal{A}(s_{t+1})} \hat{q}(s_{t+1}, a, w_t) - \hat{q}(s_t, a_t, w_t)\Big]\nabla_w\hat{q}(s_t, a_t, w_t)$
        Update policy:
        $\pi_{t+1}(a|s_t) = 1 - \frac{\varepsilon}{|\mathcal{A}(s)|}(|\mathcal{A}(s)| - 1)$ if $a = \arg\max_{a\in\mathcal{A}(s_t)} \hat{q}(s_t, a, w_{t+1})$
        $\pi_{t+1}(a|s_t) = \frac{\varepsilon}{|\mathcal{A}(s)|}$ otherwise
```

to the target state from the top-left state. As can be seen, Q-learning with linear function approximation can successfully complete the task. Here, linear Fourier basis functions of order 5 are used. The off-policy version will be demonstrated when we introduce deep Q-learning.

8.9 Deep Q-learning

We can introduce deep neural networks into Q-learning to obtain *deep Q-learning* or *deep Q-network* (DQN) [29–31]. Deep Q-learning is one of the earliest algorithms that introduce deep neural networks into RL. Of course, the neural network does not have to be deep. For many tasks including our grid-world examples, a shallow network with one hidden layer is already sufficient. We, however, always refer to the algorithm as deep Q-learning regardless of the depth of the network.

The idea of deep Q-learning is similar to (8.22). However, the mathematical formulation and implementation are substantially different.

8.9.1 Algorithm description

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

$$J = \mathbb{E}\left[\left(R + \gamma \max_{a\in\mathcal{A}(S')} \hat{q}(S', a, w) - \hat{q}(S, A, w)\right)^2\right], \tag{8.23}$$

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

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

is the Bellman optimality equation. The proof was given in the last chapter when we introduce the Tabular Q-learning algorithm. Therefore, the value of $R+\gamma \max_{a\in\mathcal{A}(S')} \hat{q}(S', a, w)-\hat{q}(S, A, w)$ should be zero in the expectation sense if $\hat{q}$ can accurately approximate the optimal action values.

To minimize the objective function in (8.23), we need to calculate its gradient with respect to w. It is noted that the parameter w not only appears in $\hat{q}(S, A, w)$ but also in $y \doteq R + \gamma \max_{a\in\mathcal{A}(S')} \hat{q}(S', a, w)$. For the sake of simplicity, we can assume that w in y is fixed (at least for a while) when we calculate the gradient. To do that, we can introduce two networks. One is a *main network* representing $\hat{q}(s, a, w)$ and the other is a *target* *network* $\hat{q}(s, a, w_T)$. The objective function in this case degenerates to

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

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

$$\nabla_w J = \mathbb{E}\left[\left(R + \gamma \max_{a \in \mathcal{A}(S')} \hat{q}(S', a, w_T) - \hat{q}(S, A, w)\right) \nabla_w \hat{q}(S, A, w)\right]. \tag{8.24}$$

The basic idea of deep Q-learning is to use the gradient in (8.24) to minimize the objective function in (8.23). However, such an optimization process evolves some important techniques that deserve special attention.

1) One technique is *experience replay* [29, 30, 32]. That is, after we have collected some experience samples, we do not use these samples in the order they were collected. Instead, we store them in a data set, called *replay buffer*, and draw a batch of samples randomly to train the neural network. In particular, let (s, a, r, s') be an experience sample and $\mathcal{B} \doteq \{(s, a, r, s')\}$ be the replay buffer. Every time we train the neural network, we can draw a mini-batch of random samples from the reply buffer. The draw of samples, or called *experience replay*, should follow a *uniform distribution*.

   Why is experience replay necessary in deep Q-learning and why does the reply must follow a uniform distribution? The answers lie in the objective function in (8.23).

   First, to well define the objective function, we must specify the probability distributions for S, A, R, S'. The distributions of R and S' are determined by the system model. The tricky part is the distributions of S and A. Suppose the experience samples are collected by following a behavior policy π_b. Then, the distribution of A is $A \sim \pi_b(S)$. However, if we treat the state-action pair (S, A) as a single instead of two random variables, then we can *remove the dependence* of the sample (S, A, R, S) on π_b. That is because, once (S, A) is given, R and S are purely determined by the system model. The distribution of the state-action pair (S, A) is assumed to be *uniform*.

   Second, although the state-action samples are assumed to be uniformly distributed, they are *not* in practice because they are generated consequently by the behavior policy. To break the correlation between consequent samples to satisfy the assumption of uniform distribution, we can use the experience replay technique by uniformly drawing samples from the reply buffer. This is the mathematical reason why experience replay is necessary and why the experience reply must be uniform. A benefit of random sampling is that each experience sample may be used multiple times, which can increase the data efficiency. This is especially important when we have a limited amount of data.

2) Another technique is to use two networks, a main network and a target network. The

Pseudocode: Deep Q-learning (off-policy version)

```
Initialization: A main network and a target network with the same initial parameter.
Aim: Learn an optimal target network to approximate the optimal action values from the
experience samples generated by a behavior policy $\pi_b$.

Store the experience samples generated by $\pi_b$ in a reply buffer $\mathcal{B} = \{(s, a, r, s')\}$
    For each iteration, do
        Uniformly draw a mini-batch of samples from $\mathcal{B}$
        For each sample $(s, a, r, s')$, calculate the target value as $y_T = r + \gamma \max_{a \in \mathcal{A}(s')} \hat{q}(s', a, w_T)$, where $w_T$ is the parameter of the target network
        Update the main network to minimize $(y_T - \hat{q}(s, a, w))^2$ using the mini-batch $\{(s, a, y_T)\}$
    Set $w_T = w$ every $C$ iterations
```

reason has been explained when we calculate the gradient in (8.24). The implementation details are clarified below. Let w and w_T denote the parameters of the main and target networks, respectively. They are set to be the same initially.

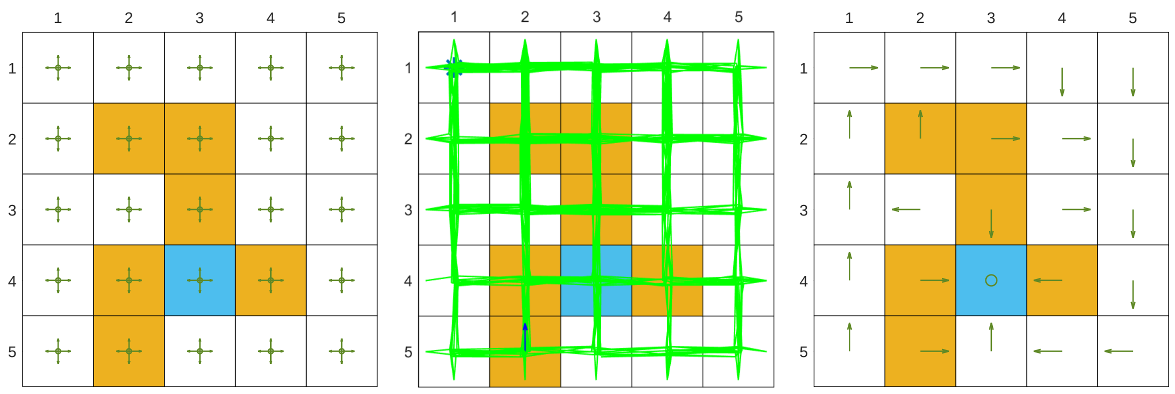
In every iteration, we draw a mini-batch of samples $\{(s, a, r, s')\}$ from the reply buffer. The inputs of the networks include state s and action a. The target output is $y_T \doteq r + \gamma \max_{a \in \mathcal{A}(s')} \hat{q}(s', a, w_T)$. Then, we directly minimize the TD error or called loss function $(y - \hat{q}(s, a, w))^2$ over the mini-batch $\{(s, a, y_T)\}$ instead of a single sample to improve efficiency and stability. Note that, when updating the main network parameter, the target output is calculated based on the target network.

The parameter of the main network is updated in every iteration. By contrast, the target network is set to be the same as the main network every a certain number of iterations to meet the assumption that w_T is fixed when calculating the gradient in (8.24). In addition, we do not bother to explicitly use the gradient in (8.24) to update the main network. Instead, we rely on the built-in gradient calculation methods of neural network software packages. Finally, more theoretical discussion on deep Q-learning can be found in [31].

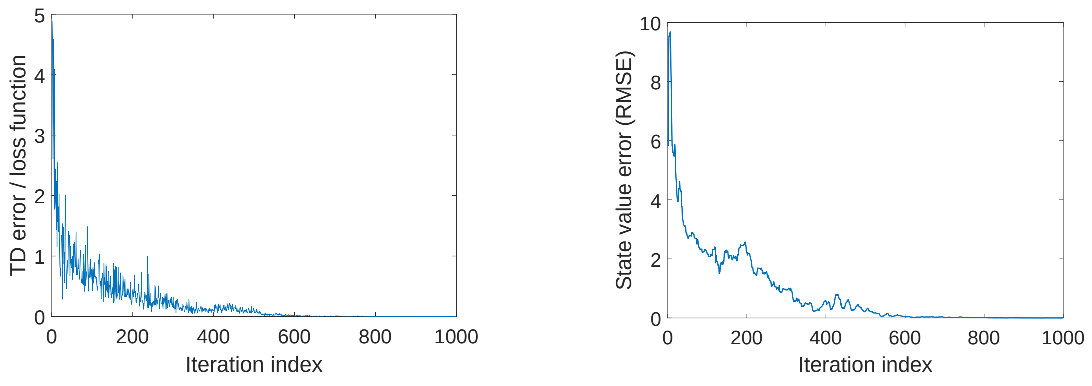
8.9.2 Illustrative examples

An example is given in Figure 8.10 to illustrate deep Q-learning (see the implementation in the pseudocode). This example aims to learn optimal action values for every state-action pair. Once the optimal action values are obtained, the optimal greedy policy can be obtained immediately. One single episode is used to train the network. This episode is generated by a behavior policy shown in Figure 8.10(a). This behavior policy is exploratory in the sense that it takes any action at any state with the same probability.

If we use the tabular Q-learning algorithm, we need a very long episode of, for example, 100,000 steps. If we use deep Q-learning, we only need a short episode of, for example, 1,000 steps. This short episode is visualized in Figure 8.10(b). Although there are only 1,000 steps, almost all the state action pairs are visited in this episode due to the strong exploration ability of the behavior policy. Nevertheless, some state-action pairs are visited only a few times.



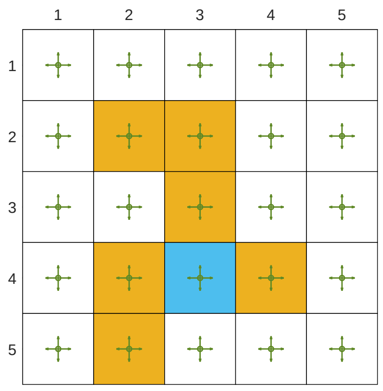
(a) The behavior policy. (b) An episode of 1,000 steps. (c) The finally searched policy.



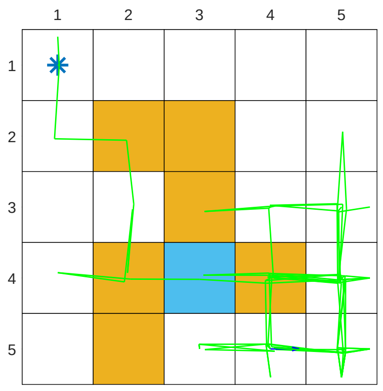
(d) The TD error converges to zero. (e) The state estimation error converges to zero.

Figure 8.10: Optimal policy searching by deep Q-learning. Here, $\gamma = 0.9$, $r_{\text{boundary}} = r_{\text{forbidden}} = -10$, $r_{\text{target}} = 1$. The batch size is 100.

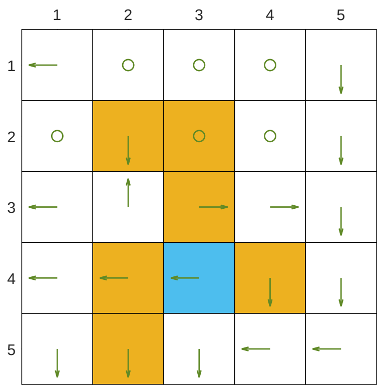
In this example, a shallow neural network with one single hidden layer is used as a nonlinear approximator of $\hat{q}(s, a, w)$. The hidden layer has 100 neurons. The neural network has three inputs and one output. The first two inputs are the normalized row and normalized column of a state. The third input is the normalized action value. Here, the normalization means converting any value to the interval of [0,1]. The output of the network is the predicted action value. The reason that we input the row and column of a state into the network rather than its index is that we know that a state corresponds to a two-dimensional location in the grid. The more information about the state we use when designing the network, the better the network can perform. Of course, the neural network can also be designed in other ways. For example, it can have two inputs and five outputs, where the two inputs are the normalized row and column of a state and the outputs are the five approximated action values for the input state.



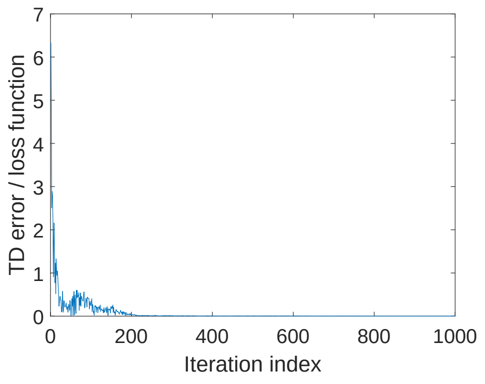
(a) The behavior policy.



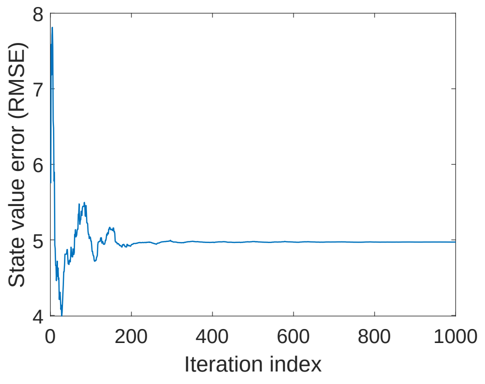
(b) An episode of 100 steps.



(c) The finally searched policy.



(d) The TD error converges to zero.



(e) The state estimation error converges to zero.

Figure 8.11: Optimal policy searching by deep Q-learning. Here, $\gamma = 0.9$, $r_{\text{boundary}} = r_{\text{forbidden}} = -10$, $r_{\text{target}} = 1$. The batch size is 50.

The mini-batch size is 100 samples. The training process is show in Figure 8.10(d)-(e). As can be seen, the loss function, defined as the average TD error of each batch, converges to zero, meaning the network is well-trained eventually. The state value estimation error also converges to zero, indicating that the action values are well approximated. The finally obtained policy is optimal.

This example validates the high efficiency of deep Q-learning. Although we only have a short episode of 1,000 steps, the sample efficiency is high. One reason is that the experience samples can be repeatedly used. Another reason is the method based on function approximation has a strong generalization ability.

We can challenge the algorithm by deliberately considering even fewer experience samples. For example, Figure 8.11 shows the results when we are given an episode of merely 100 steps. In this case, although the network can still be well trained in the sense that the loss function converges to zero, the state estimation error cannot converge to zero. It is not surprising because 100 experience samples are too few to obtain optimal action values.

8.10 Summary

This chapter introduces the method of RL based on value function approximation. The key to understanding this method is to understand the objective functions and optimization algorithm. Unlike the tabular cases, here a policy is optimal if it can minimize certain scalar objective functions. The simplest objective function is the squared error between true state values and the estimated ones. There are also other objective functions such as the Bellman error and the projected Bellman error. We showed that the TD-Linear algorithm actually minimizes the projected Bellman error. Several optimization algorithms such as Sarsa and Q-learning with value approximation were introduced and demonstrated by examples.

Value function approximation is important because it allows artificial neural networks to enter the field of RL. Although neural networks are widely used nowadays as nonlinear function approximators, this chapter mainly focused on the case of linear functions to study fundamental theoretical properties. Fully understanding the linear case is necessary for us to appropriately use nonlinear approximators. Interested readers may refer to [19] for a thorough analysis of algorithms of TD learning with function approximation.

An important concept named stationary distribution emerges in this chapter when we try to define objective functions for optimization. This distribution plays an important role in the convergence analysis. The convergence analysis reveals the importance of on-policy sampling. In particular, it is required that the states are sampled following the stationary distribution of the given policy. If this condition is not satisfied, convergence may not be guaranteed [19]. The stationary distribution also plays an important role in the next chapter where we will use functions to approximate policies instead of values. Stationary distribution is a fundamental property of Markov processes. An excellent introduction to this topic can be found in [18, Chapter IV]. The contents of this chapter heavily rely on matrix analysis. Many results were simply used without detailed explanation. Excellent references on matrix analysis and linear algebra include [20,33].

8.11 Q&A

– Q: What is the fundamental difference between the tabular representation of values and function approximation?

  A: The fundamental difference is how the optimal policies are defined. For the tabular representation, a policy is optimal if it has greater state values than any other policy. For the value function approximation method, the optimal policy is to minimize a scalar objective function.

– Q: Why do we need to study stationary distribution?

  A: That is because it is a necessary concept to define a valid objective function such as (8.2). In particular, the random variable in the objective function is the state S, which must obey a probability distribution. This probability distribution is usually set as the stationary distribution of the given policy because it can reflect the frequencies the states are visited under the given policy.

– Q: Why do we need to study linear function approximators while nonlinear approximators such as artificial neural networks are widely used nowadays?

  A: Linear function approximation is the simplest case. The theoretical properties in this case can be thoroughly analyzed. By contrast, the nonlinear cases are difficult to analyze. Analyzing the linear cases can certainly help us better understand and use nonlinear approximators.

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

  A: The fundamental reason is due to the definition of the scalar objective function, which assumes the state-action samples obey a uniform distribution. By contrast, the tabular Q-learning algorithm does not require experience replay. It is, therefore, important to understand the mathematics behind an algorithm in order to use it properly.