Chapter 7

Temporal-Difference Learning

This chapter introduces temporal-difference (TD) learning, which is one of the most well-known methods in reinforcement learning (RL). Similar to Monte Carlo learning, TD learning is also model-free, but it has some advantages as we will see. With the preparation in the last chapter, we will see in this chapter that TD learning algorithms can be viewed as special Robbins-Monro (RM) algorithms solving the Bellman or Bellman optimality equation.

7.1 Motivating examples: stochastic algorithms

We next consider some stochastic problems and show how to use the RM algorithm to solve them.

1) First, consider the simple mean estimation problem. That is to calculate

$$w = \mathbb{E}[X],$$

based on some iid samples $\{x\}$ of the random variable X. By writing $g(w) = w - \mathbb{E}[X]$, we can reformulate the problem to a root-finding problem

$$g(w) = 0.$$

Since we can only obtain a sample x of X, the noisy observation we can get is

$$\tilde{g}(w, \eta) = w - x = (w - \mathbb{E}[X]) + (\mathbb{E}[X] - x) \doteq g(w) + \eta.$$

Then, according to the last chapter, we know the RM algorithm for solving $g(w) = 0$ is

$$\begin{aligned} w_{k+1} &= w_k - \alpha_k \tilde{g}(w_k, \eta_k) \\ &= w_k - \alpha_k (w_k - x_k). \end{aligned}$$

It has been proven in the last chapter that w_k converges to $\mathbb{E}[X]$ with probability 1 if $\sum_{k=1}^{\infty}\alpha_k=\infty$ and $\sum_{k=1}^{\infty}\alpha_k^2<\infty$.

2) Second, we consider a little more complex problem. That is to estimate the mean of a function $v(X)$,

$$w=\mathbb{E}[v(X)],$$

based on some iid random samples $\{x\}$ of X. To solve this problem, we define

$$\begin{aligned} g(w) &= w-\mathbb{E}[v(X)] \\ \tilde{g}(w,\eta) &= w-v(x)=(w-\mathbb{E}[v(X)])+(\mathbb{E}[v(X)]-v(x))\doteq g(w)+\eta. \end{aligned}$$

Then, the problem becomes a root-finding problem: $g(w)=0$. The corresponding RM algorithm is

$$\begin{aligned} w_{k+1} &= w_k-\alpha_k\tilde{g}(w_k,\eta_k) \\ &= w_k-\alpha_k[w_k-v(x_k)]. \end{aligned}$$

3) Third, we consider a more complex problem. That is to calculate

$$w=\mathbb{E}[R+\gamma v(X)],$$

where R,X are random variables, γ is a constant, and $v(\cdot)$ is a function. This problem is more complex than the first two because it involves multiple random variables. We can still apply the RM algorithm in this case. Suppose we can obtain samples $\{x\}$ and $\{r\}$ of X and R. we define

$$\begin{aligned} g(w) &= w-\mathbb{E}[R+\gamma v(X)], \\ \tilde{g}(w,\eta) &= w-[r+\gamma v(x)] \\ &= (w-\mathbb{E}[R+\gamma v(X)])+(\mathbb{E}[R+\gamma v(X)]-[r+\gamma v(x)]) \\ &\doteq g(w)+\eta. \end{aligned}$$

Then, the problem becomes a root-finding problem: $g(w)=0$. The corresponding RM algorithm is

$$\begin{aligned} w_{k+1} &= w_k-\alpha_k\tilde{g}(w_k,\eta_k) \\ &= w_k-\alpha_k[w_k-(r_k+\gamma v(x_k))]. \end{aligned}$$

The reader may have noticed that the above three examples become more and more complex. However, they all can be solved by the RM algorithm. In fact, the RM algorithm of the third example already has a similar expression as the TD learning algorithms introduced in the following sections.

7.2 TD learning of state values

TD learning often refers to a broad class of RL algorithms. For example, all the algorithms introduced in this chapter fall into the scope of TD learning. However, in this section TD learning specifically refers to a classic algorithm for estimating state values [3, 13].

7.2.1 Algorithm

Suppose π is a given policy. Our aim is to calculate the state values under π. Recall that the definition of state value is

$$v_\pi(s) = \mathbb{E}\big[R + \gamma G | S = s\big], \quad s \in \mathcal{S} \tag{7.1}$$

where S, R, and G are the random variables representing the current state, immediate reward, and discounted return, respectively. Since

$$\mathbb{E}[G|S = s] = \sum_a \pi(a|s) \sum_{s'} p(s'|s, a) v_\pi(s') = \mathbb{E}[v_\pi(S')|S = s],$$

where S' is the random variable representing the next state, we can rewrite (7.1) as

$$v_\pi(s) = \mathbb{E}\big[R + \gamma v_\pi(S') | S = s\big], \quad s \in \mathcal{S}. \tag{7.2}$$

Equation (7.2) is another expression of the Bellman equation. It is sometimes called the *Bellman expectation equation*, an important tool to design and analyze TD algorithms.

To obtain the state values, we need to solve (7.2). What we have is an episode $(s_0, r_1, s_1, \dots, s_t, r_{t+1}, s_{t+1}, \dots)$ generated following the given policy π. Here, t denotes the time step. To solve (7.2) using the episode, the TD learning algorithm is

$$v_{t+1}(s_t) = v_t(s_t) - \alpha_t(s_t)\Big[v_t(s_t) - [r_{t+1} + \gamma v_t(s_{t+1})]\Big], \tag{7.3}$$

$$v_{t+1}(s) = v_t(s), \quad \text{for all } s \neq s_t, \tag{7.4}$$

where $t = 0, 1, 2, \dots$. Here, $v_t(s_t)$ is the estimated state value of $v_\pi(s_t)$; $\alpha_t(s_t)$ is the learning rate of s_t at time t.

It is notable that, at time t, only the value of the visited state s_t is updated whereas the values of the unvisited states $s \neq s_t$ remain unchanged. The update in (7.4) will be omitted in all the algorithms introduced in this chapter without causing any confusion. Nevertheless, this equation should be kept in mind because the algorithm would not be mathematically complete without this equation.

Why does the TD learning algorithm look like this? One may be confused the first time seeing the algorithm. Notably, the TD algorithm in (7.3) has a similar expression to those RM algorithms introduced in the previous section. In fact, the TD algorithm can be derived by applying the RM algorithm to solve the Bellman equation. The details are given in the shaded box below. From this point of view, readers can also better understand the essence of the TD algorithm.

A derivation of the TD algorithm

We next show how the TD algorithm can be obtained by applying the RM algorithm to solve the Bellman equation in (7.2).

In particular, by defining

$$g(v_\pi(s)) = v_\pi(s) - \mathbb{E}\big[R + \gamma v_\pi(S')|s\big],$$

we can rewrite (7.2) as

$$g(v_\pi(s)) = 0.$$

Since we can only obtain the samples r and s' of R and S', the noisy observation we have is

$$\begin{aligned}\tilde{g}(v_\pi(s)) &= v_\pi(s) - \big[r + \gamma v_\pi(s)\big]\\ &= \underbrace{\Big(v_\pi(s) - \mathbb{E}\big[R + \gamma v_\pi(S')|s\big]\Big)}_{g(v_\pi(s))} + \underbrace{\Big(\mathbb{E}\big[R + \gamma v_\pi(S')|s\big] - \big[r + \gamma v_\pi(s)\big]\Big)}_{\eta}.\end{aligned}$$

Therefore, the RM algorithm for solving $g(v_\pi(s)) = 0$ is

$$\begin{aligned}v_{k+1}(s) &= v_k(s) - \alpha_k \tilde{g}(v_k(s))\\ &= v_k(s) - \alpha_k\Big(v_k(s) - \big[r_k + \gamma v_\pi(s'_k)\big]\Big), \qquad k = 1, 2, 3, \ldots \end{aligned} \tag{7.5}$$

where $v_k(s)$ is the estimate of $v_\pi(s)$ at the kth step; r_k, s'_k are the samples of R, S' obtained at the kth step.

The RM algorithm in (7.5) has two assumptions that deserve special attention.

1) In order to implement this algorithm, we must repeatedly start from s to obtain the experience set $\{(s, r_k, s'_k)\}$ for $k = 1, 2, 3, \ldots$.
2) It is notable that $v_\pi(s'_k)$ on the right-hand side of (7.5) is not $v_k(s'_k)$ because we only estimate $v_\pi(s)$ and assume that $v_\pi(s')$ is already known for any other state s'. Only in this way can the above RM algorithm be strictly valid.

These two assumptions are usually invalid in practice. In particular, we may not be able to sample experience repeatedly starting from s in practice. As a result, we cannot obtain the set $\{(s, r_k, s'_k)\}$. What we usually have are episodes of sequential experiences, where the state s may be visited or sampled once, and then the agent moves to other states. Moreover, we cannot assume that $v_\pi(s')$ is already known for any other state s'. Instead, we need to estimate $v_\pi(s)$ for every $s \in \mathcal{S}$.

To remove the two assumptions in the RM algorithm, we can modify it and then obtain the TD algorithm in (7.3). One modification is that $\{(s, r_k, s'_k)\}$ is changed to $\{(s_t, r_{t+1}, s_{t+1})\}$ so that the algorithm can utilize the sequential samples in an episode rather than fixing on a specific state s. Another modification is that the update of $v_t(s_t)$ depends on $v_t(s_{t+1})$ rather than $v_\pi(s_{t+1})$ because $v_\pi(s_{t+1})$ is also to be estimated. Whether such an update can ensure convergence? The answer is yes and will be analyzed later.

7.2.2 Properties

We next discuss some important properties of the TD algorithm.

First, we examine the expression of the TD algorithm more closely. In particular, (7.3) can be annotated as

$$\underbrace{v_{t+1}(s_t)}_{\text{new estimate}} = \underbrace{v_t(s_t)}_{\text{current estimate}} - \alpha_t(s_t)\Big[\overbrace{v_t(s_t) - [\underbrace{r_{t+1} + \gamma v_t(s_{t+1})}_{\text{TD target } \bar{v}_t}]}^{\text{TD error } \delta_t}\Big], \tag{7.6}$$

where

$$\bar{v}_t \doteq r_{t+1} + \gamma v(s_{t+1})$$

is called the *TD target* and

$$\delta_t \doteq v(s_t) - [r_{t+1} + \gamma v(s_{t+1})] = v(s_t) - \bar{v}_t$$

is called the *TD error*. It is clear that the new estimate $v_{t+1}(s_t)$ is a combination of the current estimate $v_t(s_t)$ and the TD error.

– What is the interpretation of the TD error? The TD error should be zero in the expectation sense. That is simply because

$$\begin{aligned}
\mathbb{E}[\delta_t | S_t = s_t] &= \mathbb{E}\big[v_\pi(S_t) - [R_{t+1} + \gamma v_\pi(S_{t+1})] | S_t = s_t\big] \\
&= v_\pi(s_t) - \mathbb{E}\big[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s_t\big] \\
&= 0.
\end{aligned}$$

Therefore, the TD error reflects the deficiency between the current estimate v_t and the true state value v_π.

In a more abstract level, the TD error can be interpreted as *innovation*, which means new information obtained from the experience (s_t, r_{t+1}, s_{t+1}). The fundamental idea of TD learning is to correct our current understanding of the state value based on the new information obtained. Innovation is a fundamental concept in many estimation problems such as Kalman filtering.

– Why is $\bar{v}_t$ called the TD target? That is because $\bar{v}_t$ is a *target value* that the algorithm attempts to drive $v(s_t)$ to approach. To see that, we consider a simple case where α and $\bar{v}$ are both constant. Then, (7.6) becomes

$$v_{t+1} = v_t - \alpha(v_t - \bar{v}) = (1-\alpha)v_t + \alpha\bar{v}.$$

It can be shown that $v_t \to \bar{v}$ as $t \to \infty$ if $0 < \alpha < 1$. To prove that, let $\delta_t = v_t - \bar{v}$. Then the above equation becomes $\delta_{t+1} = (1-\alpha)\delta_t$. Hence, we have $\delta_{t+1} = (1-\alpha)\delta_t = (1-\alpha)^2\delta_{t-1} = \cdots = (1-\alpha)^t\delta_1$. Since $(1-\alpha)^t \to 0$ as $t \to \infty$, we know $\delta_t \to 0$ and hence $v_t \to \bar{v}$ as $t \to \infty$. Therefore, $\bar{v}$ is the target value that v_t will converge to.

Second, the TD algorithm in (7.3) can only estimate the state value of a given policy. To find optimal policies, we still need to further calculate the action values and then do policy improvement. In fact, when the system model is unavailable, we should directly estimate action values. Nevertheless, the TD algorithm introduced in this section is very basic and important for understanding other algorithms introduced in the rest of the chapter.

Third, while TD learning and MC learning are both model-free, what are the advantages and disadvantages of TD learning compared to MC learning? The answers are summarized in Table (7.1).

7.2.3 Convergence

Although the TD algorithm can be viewed as an RM algorithm solving the Bellman equation, it is still necessary to give a rigorous convergence analysis.

Theorem 7.1 (Convergence of TD Learning). *By the TD algorithm* (7.3), *$v_t(s)$ converges with probability 1 to $v_\pi(s)$ for all $s \in \mathcal{S}$ as $t \to \infty$ if $\sum_t \alpha_t(s) = \infty$ and $\sum_t \alpha_t^2(s) < \infty$ for all $s \in \mathcal{S}$.*

It should be noted that the condition of $\sum_t \alpha_t(s) = \infty$ and $\sum_t \alpha_t^2(s) < \infty$ should be valid for all $s \in \mathcal{S}$. That requires every state must be visited an infinite (or sufficiently many) number of times. At time step t, if $s = s_t$ which means that s is visited at time t, then $\alpha_t(s) > 0$; otherwise, $\alpha_t(s) = 0$ for all the other $s \neq s_t$.

In addition, the learning rate α is often selected as a small constant. In this case, the condition that $\sum_t \alpha_t^2(s) < \infty$ is invalid anymore. When α is constant, it can still be shown that the algorithm converges in the sense of expectation sense.

TD/Sarsa learning	MC learning
Online: TD learning is online. It can update the state/action values immediately after receiving a reward.	**Offline:** MC learning is offline. It has to wait until an episode has been completely collected. That is because it must calculate the discounted return from a state-action pair to the end of the episode.
Continuing tasks: Since TD learning is online, it can handle both episodic and continuing tasks. Continuing tasks may not have a terminal state.	**Episodic tasks:** Since MC learning is offline, it can only handle episodic tasks where the episodes terminate after a finite number of steps.
Bootstrapping: TD bootstraps because the update of a state/action value relies on the previous estimate of this value. As a result, TD requires an initial guess of the values.	**Non-bootstrapping:** MC is not bootstrapping, because it can directly estimate state/action values without any initial guess.
Low estimation variance: The estimation variance of TD is lower than MC because there are fewer random variables. For instance, when updating an action value, Sarsa merely requires samples of three random variables: $R_{t+1}, S_{t+1}, A_{t+1}$.	**High estimation variance:** The estimation variance of MC is higher since many random variables are involved. For example, to estimate $q_\pi(s_t, a_t)$, we need samples of $R_{t+1}+\gamma R_{t+2}+\gamma^2 R_{t+3}+\ldots$. Suppose the length of each episode is L. Assume each state has the same number of actions as $\lvert\mathcal{A}\rvert$. Then there are $\lvert\mathcal{A}\rvert^L$ possible episodes in total following a soft policy. If we merely use a few episodes to estimate, it is not surprising that the estimation variance is high though the bias is zero.

Table 7.1: Comparison between TD learning and MC learning.

Proof of Theorem 7.1

We prove the convergence based on Theorem 6.3 given in the last chapter. To do that, we need first to construct a process like the one in Theorem 6.3.

Consider an arbitrary state $s \in \mathcal{S}$. Let k be the times s has been visited. Note that k is different from t. Here, t is the time step when sampling the experiences (s_t, r_{t+1}, s_{t+1}). It represents the total times that all the states are visited, whereas k is the times that s is visited. Every time s is visited, $k \to k+1$.

The algorithm (7.3) implies the following process:

$$\begin{aligned} v_{k+1}(s) &= v_k(s) - \alpha_k(s)[v_k(s) - (r_{k+1} + \gamma v_k(s'))] \\ &= (1-\alpha_k(s))v_k(s) + \alpha_k(s)(r_{k+1} + \gamma v_k(s')), \qquad k = 1,2,\ldots \end{aligned} \tag{7.7}$$

where s' is the next state transited from s. We next show that the three conditions in Theorem 6.3 are satisfied.

First, the convergence of (7.7) requires $\sum_k \alpha_k(s) = \infty$ and $\sum_k \alpha_k^2(s) < \infty$ according to Theorem 6.3. This condition is tricky in the context of multiple states. It must be noted that this condition must be satisfied for each state, which means every state must be visited an infinite number of times.

Second, define the estimation error as $\Delta_k(s) = v_k(s) - v_\pi(s)$, where $v_\pi(s)$ is the state value of s under policy π. Then, deducting $v_\pi(s)$ on both sides of (7.7) gives

$$\Delta_{k+1}(s) = (1-\alpha_k(s))\Delta_k(s) + \alpha_k(s)[\underbrace{r_{k+1} + \gamma v_k(s_{k+1}) - v_\pi(s)}_{e_k}], \qquad k = 1,2,\ldots$$

To apply Theorem 6.3, we need to show that $\|\mathbb{E}[e_k(s)|\mathcal{H}_k]\|_\infty \le \gamma \|\Delta_k(s)\|_\infty$, where $\mathcal{H}_k = \{\Delta_k, \Delta_{k-1}, \ldots, e_{k-1}, \ldots, \alpha_{k-1}, \ldots\}$. To see that, we have

$$\begin{aligned} \|\mathbb{E}[e_k(s)|\mathcal{H}_k]\|_\infty = \left\| \begin{matrix} \vdots \\ \mathbb{E}[e_k(s)|\mathcal{H}_k] \\ \vdots \end{matrix} \right\|_\infty &= \|r_\pi + \gamma P_\pi v_k - v_\pi\|_\infty \\ &= \|r_\pi + \gamma P_\pi v_k - (r_\pi + \gamma P_\pi v_\pi)\|_\infty \\ &= \gamma \, \|P_\pi v_k - P_\pi v_\pi\|_\infty \\ &\le \gamma \, \|v_k - v_\pi\|_\infty \\ &= \gamma \left\| \begin{matrix} \vdots \\ \Delta_k(s) \\ \vdots \end{matrix} \right\|_\infty = \gamma \|\Delta_k(s)\|_\infty. \end{aligned}$$

The symbol $\|\cdot\|_\infty$ is abused a little here: It stands for the maximum norm calculated over all states and, in the meantime, the L_∞ norm of vectors. Since the discount rate $\gamma \in (0,1)$, condition 3 in the theorem is satisfied.

Third, regarding the variance, we have $\text{Var}[e_k|\mathcal{H}_k] = \text{Var}[r_{k+1} + \gamma v_k(s_{k+1}) - v_\pi(s)|\mathcal{H}_k] = \text{Var}[r_{k+1} + \gamma v_k(s_{k+1})|\mathcal{H}_k]$. Since r_{k+1} is bounded, condition 4 can be proved without difficulties.

The above proof is essentially the same as [11].

7.3 TD learning of action values: Sarsa

The TD algorithm introduced in the last section can only estimate state values. In this section, we introduce, Sarsa, an algorithm that can directly estimate action values. Estimating action values is important because the policy can be improved based on action values.

7.3.1 Algorithm

Given a policy π, our aim is to estimate the action values of π. Suppose we have an episode of experiences generated following π: $(s_0, a_0, r_1, s_1, a_1, \dots, s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, \dots)$. Based on the episode, we can use the following *Sarsa* algorithm to estimate the action values:

$$\begin{aligned} q_{t+1}(s_t, a_t) &= q_t(s_t, a_t) - \alpha_t(s_t, a_t)\Big[q_t(s_t, a_t) - [r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1})]\Big], \qquad (7.8) \\ q_{t+1}(s, a) &= q_t(s, a), \quad \text{for all } (s, a) \neq (s_t, a_t), \end{aligned}$$

where $t = 0, 1, 2, \dots$. Here, $q_t(s_t, a_t)$ is the estimated action value of (s_t, a_t); $\alpha_t(s_t, a_t)$ is the learning rate depending on s_t, a_t. At time step t, only the state-action (s_t, a_t) which is being visited is updated. The estimates of all the other state-action pairs remain the same.

Some properties of Sarsa are discussed as follows.

- Why is this algorithm called Sarsa? That is because each step of the algorithm involves $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. Sarsa is the abbreviation of state-action-reward-state-action. The algorithm of Sarsa was first proposed in [14] and the name of Sarsa was due to [3].
- Why is Sarsa designed in this way? One may have noticed that the expression of Sarsa is very similar to the TD algorithm introduced in the last section. We can obtain Sarsa by replacing the state value estimate $v(s)$ in the TD algorithm with the action value estimate $q(s, a)$. As a result, Sarsa is nothing but an action-value version of the TD algorithm.
- What does the Sarsa algorithm do mathematically? Similar to the analysis of the TD learning in the last section, we know that Sarsa is a stochastic approximation algorithm solving the following equation:

$$q_\pi(s, a) = \mathbb{E}\left[R + \gamma q_\pi(S', A') | s, a\right], \quad \text{for all } s, a. \qquad (7.9)$$

Equation (7.13) is another expression of the Bellman equation expressed in terms of action values. The proof is given in the shaded box below.

Show that (7.13) **is the Bellman equation**

First of all, the Bellman equation expressed in terms of action values is

$$
\begin{aligned}
q_\pi(s,a) &= \sum_r rp(r|s,a) + \gamma \sum_{s'} \sum_{a'} q_\pi(s',a') p(s'|s,a)\pi(a'|s') && (7.10)\\
&= \sum_r rp(r|s,a) + \gamma \sum_{s'} p(s'|s,a) \sum_{a'} q_\pi(s',a')\pi(a'|s'). && (7.11)
\end{aligned}
$$

This expression has been given in Section 2.6.2. It describes the relationship among the values of all the actions. Since

$$
\begin{aligned}
p(s',a'|s,a) &= p(s'|s,a)p(a'|s',s,a)\\
&= p(s'|s,a)p(a'|s') \quad \text{(due to conditional independence)}\\
&\doteq p(s'|s,a)\pi(a'|s'),
\end{aligned}
$$

(7.10) can be rewritten as

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

By the definition of expected value, the above equation is equivalent to (7.13). Hence, (7.13) is the Bellman equation.

– Since Sarsa is the action-value version of the TD algorithm in the last section, its convergence also naturally follows. The convergence analysis is similar to Theorem 7.1 and omitted here. The convergence result is summarized below.

Theorem 7.2 (Convergence of Sarsa learning)**.** *By the Sarsa algorithm in* (7.8), $q_t(s,a)$ *converges with probability 1 to the action value* $q_\pi(s,a)$ *as* $t \to \infty$ *for all* (s,a) *if* $\sum_t \alpha_t(s,a) = \infty$ *and* $\sum_t \alpha_t^2(s,a) < \infty$ *for all* (s,a).

It should be noted that the condition of $\sum_t \alpha_t(s,a) = \infty$ and $\sum_t \alpha_t^2(s,a) < \infty$ should be valid for all (s,a). That requires every state-action pair must be visited an infinite (or sufficiently many) number of times. At time step t, if $(s,a) = (s_t,a_t)$, then $\alpha_t(s,a) > 0$; otherwise, $\alpha_t(s,a) = 0$ for all the other $(s,a) \neq (s_t,a_t)$.

7.3.2 Implementation

The Sarsa algorithm in (7.8) can only estimate the action values of a given policy. The ultimate goal of RL is to find optimal policies. To do that, inspired by the model-based policy iteration algorithm, we can combine Sarsa with a policy improvement step to search for optimal policies. Such a combination for optimal policy search is also often called Sarsa when the context is clear. As shown in the pseudocode, each iteration has two steps. The first is to update the q-value of the visited state-action pair. The second is to update the policy as an ϵ-greedy one. Two points deserve special attention.

In the q-value update step, unlike the model-based policy iteration or value iteration algorithm where the values of all states are updated in each iteration, Sarsa only updates a single state-action pair that is visited at time step t. After that, the policy of s_t is updated immediately. This is based on the idea of generalized policy iteration. That is, although the q-value estimated based on the experience obtained in a single step is not sufficiently accurate, we do not need to wait until a sufficiently accurate q-value can be estimated before updating the policy. Moreover, after the policy is updated, the policy is immediately used to generate the next experience. To ensure all the state-action pairs can be visited, the policy is ϵ-greedy instead of greedy.

A simulation example is shown in Figure 7.1 to demonstrate the Sarsa algorithm. It should be noted that the task here is not to find the optimal policies for all states. Instead, it aims to find a good path from a specific starting state to a target state. This task is often encountered in practice because in many cases both the starting state and target state are fixed and we only need to find a good path connecting them. This task is also relatively simple because we do not need to explore all the states. However, it is still necessary in theory to explore all states to ensure the searched path is optimal. Nevertheless, without exploring all the states, the algorithm can usually generate a good path even though its optimality is not ensured.

The simulation setup and results shown in Figure 7.1 are elaborated below.

- In this example, all the episodes start from the top left state and end in the target state. The reward setting is $r_{\text{target}} = 0$, $r_{\text{forbidden}} = r_{\text{boundary}} = -10$, and $r_{\text{other}} = -1$. The learning rate is $\alpha = 0.1$ and the value of ϵ is 0.1. The initial guess of the action values are selected as $q_0(s, a) = 0$ for all s, a. The initial policy obeys a uniform distribution: $\pi_0(a|s) = 0.2$ for all s, a.
- The left figure in Figure 7.1(a) shows the final policy obtained by Sarsa. As can be seen, this policy can successfully reach the target state from the starting state. However, the policies for other states may not be optimal. That is because the other states are not of interest and every episode ends when the target state is reached. If we need to find the optimal policies for all states, we should ensure all the states are visited sufficiently many times by, for example, starting episodes from different states.
- The right-top subfigure in Figure 7.1(a) shows the total reward collected by each episode. Here, the total reward is the non-discounted sum of all immediate rewards obtained along an episode. As can be seen, the total reward of each episode increases gradually. That is simply because the initial policy is not good and hence negative rewards are frequently obtained. As the policy becomes better, the rewards increase

Pseudocode: Policy searching by Sarsa

```
Initialization: Initial $q_0(s,a)$ and $\pi_0(a|s)$ for all $s,a$. Learning rate $\alpha_t(s_t,a_t)$ is selected as a small positive constant for all $t$. Small $\epsilon > 0$.
Aim: Search a good policy that can lead the agent to a target state from an initial state-action pair $(s_0,a_0)$.

For each episode, do
    If the current $s_t$ is not the target state, do
        Collect the experience $(s_t,a_t,r_{t+1},s_{t+1},a_{t+1})$: In particular, take action $a_t$ following $\pi_t(s_t)$, generate $r_{t+1},s_{t+1}$, and then take action $a_{t+1}$ following $\pi_t(s_{t+1})$.
        Update q-value:
            $q_{t+1}(s_t,a_t) = q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - [r_{t+1} + \gamma q_t(s_{t+1},a_{t+1})]\Big]$
        Update policy:
            $\pi_{t+1}(a|s_t) = 1 - \frac{\epsilon}{|\mathcal{A}(s)|}(|\mathcal{A}(s)| - 1)$ if $a = \arg\max_a q_{t+1}(s_t,a)$
            $\pi_{t+1}(a|s_t) = \frac{\epsilon}{|\mathcal{A}(s)|}$ otherwise
```

accordingly.

– The right-bottom subfigure in Figure 7.1(a) shows that the length of each episode drops gradually. That is because the initial policy is not good and may take many detours before reaching the target. As the policy becomes better, the length of the trajectory becomes shorter.

7.4 TD learning of action values: Expected Sarsa

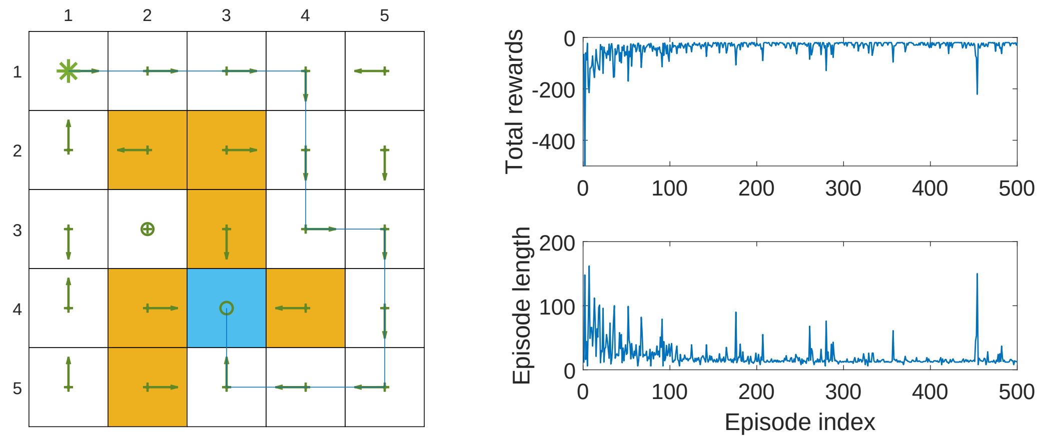
A variant of Sarsa is the *Expected Sarsa* algorithm:

$$\begin{aligned} q_{t+1}(s_t,a_t) &= q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - (r_{t+1} + \gamma\mathbb{E}[q_t(s_{t+1},A)|s_{t+1}])\Big], \\ q_{t+1}(s,a) &= q_t(s,a), \quad \text{for all } (s,a) \neq (s_t,a_t), \end{aligned} \tag{7.12}$$

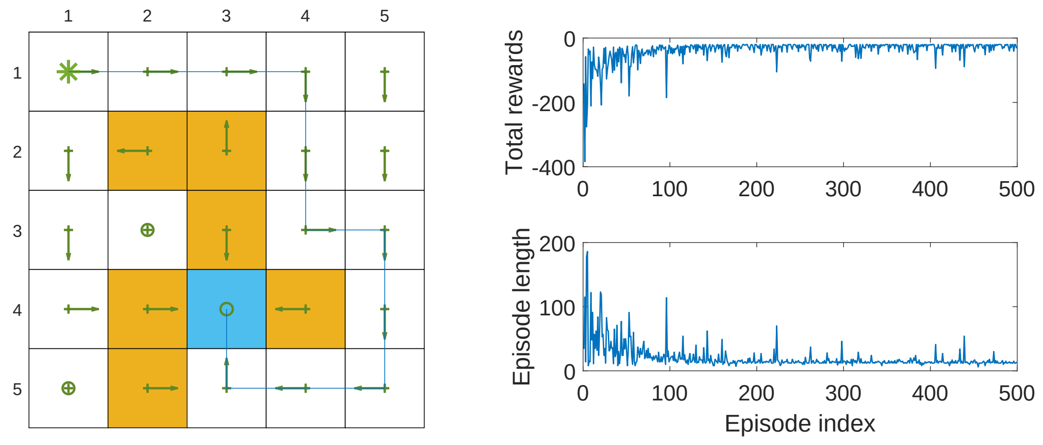
where

$$\mathbb{E}[q_t(s_{t+1},A)|s_{t+1}]) = \sum_a \pi_t(a|s_{t+1}) q_t(s_{t+1},a) \doteq v_t(s_{t+1})$$

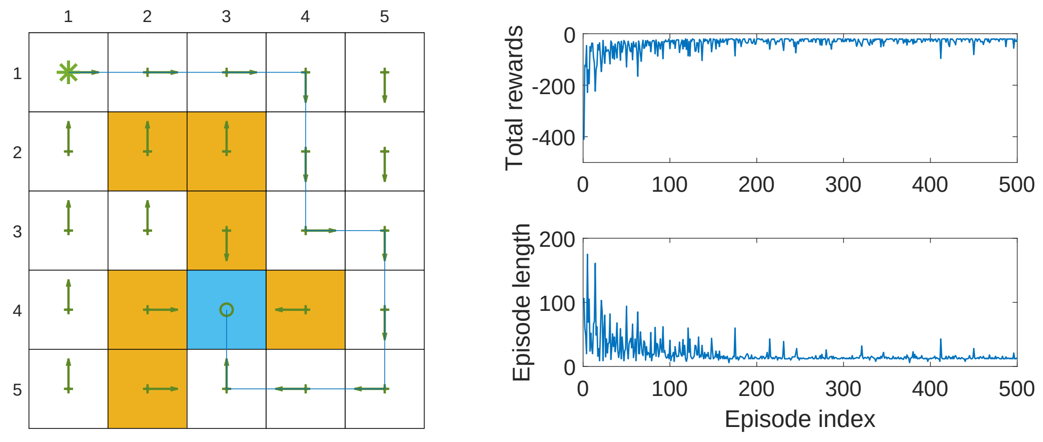
is the expected value of $q_t(s_{t+1},a)$ under policy π_t. The expression of the Expected Sarsa algorithm is very similar to that of Sarsa. They are different only in terms of the TD target. In particular, the TD target in Expected Sarsa is $r_{t+1} + \gamma\mathbb{E}[q_t(s_{t+1},A)|s_{t+1}]$ while that of Sarsa is $r_{t+1} + \gamma q_t(s_{t+1},a_{t+1})$. Since the algorithm involves an expected value, it is named Expected Sarsa. Although calculating the expected value may increase the computational complexity a little, it is beneficial in the sense that it reduces the estimation



(a) Sarsa



(b) Expected Sarsa



(c) Q-learning

Figure 7.1: Examples to demonstrate Sarsa, Expected Sarsa, and Q-learning. In each example, the episodes start from the top-left state and end in the target state. The aim is to find a good path from the starting state to the target state. The reward setting is $r_{\text{target}} = 0$, $r_{\text{forbidden}} = r_{\text{boundary}} = -10$, and $r_{\text{other}} = -1$. The learning rate is $\alpha = 0.1$ and the value of ϵ is 0.1. The left figures above show the final policy obtained by different algorithms. The right figures show the total reward and length of every episode.

variances because it reduces random variables in Sarsa from $\{s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}\}$ to $\{s_t, a_t, r_{t+1}, s_{t+1}\}$. As a result, Expected Sarsa performs generally better than Sarsa. The simulation result of Expected Sarsa is given in Figure 7.1(b).

Similar to the analysis of the TD learning algorithm in Section 7.2.1, it can be seen that Expected Sarsa is a stochastic approximation algorithm for solving the following equation:

$$q_\pi(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma \mathbb{E}[q_\pi(S_{t+1}, A_{t+1})|S_{t+1}]\Big|S_t = s, A_t = a\Big], \quad \text{for all } s, a. \tag{7.13}$$

The above equation may look strange at the first glance. In fact, it is another expression of the Bellman equation. To see that, since

$$\mathbb{E}[q_\pi(S_{t+1}, A_{t+1})|S_{t+1}] = \sum_{A'} q_\pi(S_{t+1}, A')\pi(A'|S_{t+1}) = v_\pi(S_{t+1}),$$

substituting the above equation into (7.13) gives

$$q_\pi(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a\Big],$$

which is clearly the Bellman equation.

The implementation of Expected Sarsa is the same as Sarsa, except for the q-value update step. Details are omitted here.

7.5 TD learning of action values: n-step Sarsa

This section introduces n-step Sarsa, another extension of Sarsa. It will be shown that Sarsa and MC learning are two extreme cases of n-step Sarsa.

First, the definition of action value is

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

The discounted return G_t can be written in different forms as

$$\begin{aligned}
\text{Sarsa} \longleftarrow \quad G_t^{(1)} &= R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}),\\
G_t^{(2)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, A_{t+2}),\\
&\ \vdots\\
n\text{-step Sarsa} \longleftarrow \quad G_t^{(n)} &= R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^n q_\pi(S_{t+n}, A_{t+n}),\\
&\ \vdots\\
\text{MC} \longleftarrow \quad G_t^{(\infty)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots
\end{aligned}$$

It should be noted that $G_t = G_t^{(1)} = G_t^{(2)} = G_t^{(n)} = G_t^{(\infty)}$, where the superscripts merely indicate the different decomposition structures of G_t.

Sarsa aims to solve

$$q_\pi(s,a) = \mathbb{E}[G_t^{(1)}|s,a] = \mathbb{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|s,a].$$

MC learning aims to solve

$$q_\pi(s,a) = \mathbb{E}[G_t^{(\infty)}|s,a] = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots |s,a].$$

An intermediate algorithm called *n-step Sarsa* aims to solve

$$q_\pi(s,a) = \mathbb{E}[G_t^{(n)}|s,a] = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^n q_\pi(S_{t+n}, A_{t+n})|s,a].$$

The algorithm of n-step Sarsa is

$$\begin{aligned} q_{t+1}(s_t,a_t) &= q_t(s_t,a_t) \\ &\quad - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - [r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^n q_t(s_{t+n}, a_{t+n})]\Big]. \end{aligned} \tag{7.14}$$

n-step Sarsa is more general because it becomes the (one-step) Sarsa algorithm when $n = 1$ and the MC learning algorithm when $n = \infty$.

To implement (7.14), we need the experience $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, \dots, r_{t+n}, s_{t+n}, a_{t+n})$. Since $(r_{t+n}, s_{t+n}, a_{t+n})$ has not been collected at time t, we are not able to implement (7.14) directly in practice. However, we can wait until time $t+n$ to update the q-value of (s_t, a_t). To that end, (7.14) can be modified to

$$\begin{aligned} q_{t+n}(s_t,a_t) &= q_{t+n-1}(s_t,a_t) \\ &\quad - \alpha_{t+n-1}(s_t,a_t)\Big[q_{t+n-1}(s_t,a_t) - [r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^n q_{t+n-1}(s_{t+n}, a_{t+n})]\Big], \end{aligned}$$

where $q_{t+n}(s_t,a_t)$ is the estimate of $q_\pi(s_t,a_t)$ at time $t+n$.

Since n-step Sarsa includes Sarsa and MC learning as two extreme cases, it is not surprising that the performance of n-step Sarsa is a blend of Sarsa and MC learning as well. Specifically, if n is selected to be a large number, its performance is close to MC learning and hence has a large variance but a small bias. If n is selected to be small, its performance is close to Sarsa and hence has a relatively large bias due to the initial guess and relatively low variance. A complete treatment of multi-step temporal-difference learning can be found in [3, Chapter 7]. Finally, n-step Sarsa is also for policy evaluation. It can be combined with the policy improvement step to search for optimal policies. The implementation is similar to Sarsa and omitted here.

7.6 TD learning of optimal action values: Q-learning

In this section, we introduce the Q-learning algorithm [15, 16], one of the most widely used RL algorithms. It should be noted that Sarsa can only estimate the action values of a given policy. It must be combined with a policy improvement step to find optimal policies and hence their optimal action values. By contrast, Q-learning can directly estimate optimal action values.

7.6.1 Algorithm

We first give the expression of the Q-learning algorithm and then analyze it in detail. The Q-learning algorithm is

$$\begin{aligned} q_{t+1}(s_t,a_t) &= q_t(s_t,a_t) - \alpha_t(s_t,a_t)\left[q_t(s_t,a_t) - [r_{t+1} + \gamma \max_{a\in\mathcal{A}(s_{t+1})} q_t(s_{t+1},a)]\right], \\ q_{t+1}(s,a) &= q_t(s,a), \quad \text{for all } (s,a) \neq (s_t,a_t), \end{aligned} \tag{7.15}$$

where $t = 0, 1, 2, \ldots$. Here, $q_t(s_t,a_t)$ is the estimated action value of (s_t,a_t) and $\alpha_t(s_t,a_t)$ is the learning rate depending on s_t, a_t.

The expression of Q-learning is very similar to Sarsa. They are different only in terms of the TD target: The TD target in Q-learning is $r_{t+1} + \gamma \max_{a\in\mathcal{A}(s_{t+1})} q_t(s_{t+1},a)$ while that of Sarsa is $r_{t+1} + \gamma q_t(s_{t+1},a_{t+1})$.

Why is Q-learning designed in its form and what does it do mathematically? Similar to the analysis of the TD learning algorithm in Section 7.2.1, it can be seen from the expression of Q-learning that it is a stochastic approximation algorithm for solving the action values from the following equation:

$$q(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_a q(S_{t+1},a) \Big| S_t = s, A_t = a\right], \quad \text{for all } s,a. \tag{7.16}$$

This equation actually is the Bellman optimality equation expressed in terms of action values. Interested readers can check the proof given in the shaded box below. The convergence analysis of Q-learning is similar to Theorem 7.1 and omitted here. Detailed proofs can be found in [11, 16].

Show that (7.16) is the Bellman optimality equation

By the definition of expectation, (7.16) can be rewritten as

$$q(s,a) = \sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) \max_{a\in\mathcal{A}(s')} q(s',a).$$

Taking maximum on both sides of the equation gives

$$\max_{a\in\mathcal{A}(s)} q(s,a) = \max_{a\in\mathcal{A}(s)} \left[\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) \max_{a\in\mathcal{A}(s')} q(s',a)\right].$$

Denote $v(s) \doteq \max_{a\in\mathcal{A}(s)} q(s,a)$. Then the above equation becomes

$$\begin{aligned} v(s) &= \max_{a\in\mathcal{A}(s)} \left[\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a)v(s')\right] \\ &= \max_{\pi} \sum_{a\in\mathcal{A}(s)} \pi(a|s) \left[\sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a)v(s')\right], \end{aligned}$$

which is clearly the Bellman optimality equation.

7.6.2 Off-policy vs on-policy

Before further discussing the properties and implementation of Q-learning, we first introduce two important concepts: *on-policy learning* and *off-policy learning.*

There exist two policies in a TD learning task: *behavior policy* and *target policy.* The behavior policy is used to generate experience samples. The target policy is constantly updated toward an optimal policy. When the behavior policy is the same as the target policy, such a kind of learning is called on-policy. Otherwise, when they are different, the learning is called off-policy. Off-policy learning is more general than on-policy because off-policy becomes on-policy when the behavior policy and target policy are the same.

The advantage of off-policy learning compared to on-policy learning is that it can search for optimal policies based on the experiences generated by any other policies, which may be a policy designed by other non-learning approaches or a policy executed by a human operator. As an important special case, the behavior policy can be selected to be *exploratory.* For example, if we would like to estimate the action values of all state-action pairs, we must generate episodes visiting every state-action pair sufficiently many times. In this case, exploratory behavior policies are favorable. By contract, although Sarsa uses ϵ-greedy policies to maintain certain exploration ability, the value of ϵ is usually small and hence the exploration ability is limited. In this case, a large number of episodes are required to ensure all the state-action pairs are visited, lowering the sample efficiency. To overcome this limitation, we can use policies with a strong exploration ability to generate episodes and then use off-policy learning to learn optimal policies. A representative explanatory policy is the uniform policy that selects any action at a state with the same probability.

Another concept that may be confused with on-policy/off-policy is *online/offline* *learning.* Online learning refers to the case where the value and policy can be updated once an experience sample is obtained. Offline learning refers to the case that the update can only be done after all experience samples have been collected. For example, TD learning is online, whereas MC learning is offline. An on-policy learning algorithm such as Sarsa must work online because the updated policy must be used to generate new experience samples. An off-policy learning algorithm such as Q-learning can work either online or offline. It can either update the value and policy upon receiving an experience sample or update after collecting all experience samples.

7.6.3 Q-learning is off-policy

All the TD learning algorithms introduced in this chapter are on-policy, except Q-learning.

1) Sarsa is on-policy. That is because the target policy is used to generate new experience samples once it has been updated. Hence, the behavior policy is the same as the target policy. The on-policy attribute of Sarsa can be clearly seen from the mathematical expression of the algorithm. In particular, the TD target in Sarsa is

$$\bar{q}_t = r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1}).$$

Here, a_{t+1} is generated following $\pi_t(s_{t+1})$, which is both the target and behavior policy.

2) Expected Sarsa is also on-policy. It can be clearly seen from its TD target

$$\bar{q}_t = r_{t+1} + \gamma \sum_a \pi_t(a|s_{t+1}) q_t(s_{t+1}, a),$$

where $\pi_t(s_{t+1})$ is used in the calculation.

3) MC learning is also on-policy because it aims to estimate the action values of the policy used to generate the episodes.

4) Q-learning is off-policy. That means the behavior policy can be different from the target policy. This property can also be seen from the TD target of Q-learning:

$$\bar{q}_t = r_{t+1} + \gamma \max_a q_t(s_{t+1}, a).$$

The calculation of this TD target does not require any policy.

Whether an algorithm is on-policy or off-policy is determined by the fundamental properties of the algorithm rather than how it is implemented. The fundamental reason why Q-learning is off-policy is that it aims to solve the *Bellman optimality equation.* Since solving the Bellman optimality equation does not depend on any specific policy, it can use experience samples generated by any other policies. By contrast, the fundamental reason why other TD algorithms are on-policy is that they aim to solve the *Bellman equation of* *a given policy.* Since the Bellman equation depends on a given policy, solving it requires the episodes generated by the given policy.

7.6.4 Implementation

Since Q-learning is off-policy, it can be implemented in an off-policy or on-policy fashion. The pseudocode of both the on-policy version and the off-policy version of Q-learning is given in the following boxes.

The on-policy version is the same as Sarsa except for the TD target in the q-value update step. In this case, the behavior policy is the same as the target policy, which is an ϵ-greedy policy.

In the off-policy version, the episode is generated by the behavior policy π_b which can be any policy. If we would like to sample experiences for every state-action pair, π_b should be selected to be exploratory. Here, the target policy π_T is greedy rather than ϵ-greedy. That is because the target policy is not used to generate episodes and hence is not required to be exploratory. Moreover, the off-policy version of Q-learning presented here is offline. That is because all the experience samples are collected first and then processed to estimate an optimal target policy. Of course, it can be modified to become online. Once a sample is collected, the sample can be used to update the q-value and target policy immediately. Nevertheless, the updated target policy is not used to generate new samples.

Pseudocode: Policy searching by Q-learning (on-policy version)

```
Initialization: Initial $q_0(s,a)$ and $\pi_0(a|s)$ for all $s, a$. Learning rate $\alpha_t(s_t,a_t)$ is selected as a small positive constant for all $t$. Small $\epsilon > 0$.
Aim: Search a good policy that can lead the agent to a target state from an initial state-action pair $(s_0, a_0)$.

For each episode, do
    If the current $s_t$ is not the target state, do
        Collect the experience $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$: In particular, take action $a_t$ following $\pi_t(s_t)$, generate $r_{t+1}, s_{t+1}$, and then take action $a_{t+1}$ following $\pi_t(s_{t+1})$.
        Update q-value:
            $q_{t+1}(s_t,a_t) = q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - [r_{t+1} + \gamma \max_a q_t(s_{t+1},a)]\Big]$
        Update policy:
            $\pi_{t+1}(a|s_t) = 1 - \frac{\epsilon}{|\mathcal{A}(s)|}(|\mathcal{A}(s)| - 1)$ if $a = \arg\max_a q_{t+1}(s_t,a)$
            $\pi_{t+1}(a|s_t) = \frac{\epsilon}{|\mathcal{A}(s)|}$ otherwise
```

Pseudocode: Optimal policy search by Q-learning (off-policy version)

```
Initialization: Initial guess $q_0(s,a)$ for all $s,a$. Learning rate $\alpha_t(s_t,a_t)$ is selected as a small positive constant for all $t$. Behavior policy $\pi_b$.
Aim: Learn an optimal target policy $\pi_T$ and optimal action values from some episodes generated by $\pi_b$.

For each episode $\{s_0, a_0, r_1, s_1, a_1, r_2, \dots\}$ generated by $\pi_b$, do
    For each step $t = 0, 1, 2, \dots$ of the episode, do
        Update q-value:
            $q_{t+1}(s_t,a_t) = q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q(s_t,a_t) - [r_{t+1} + \gamma \max_a q_t(s_{t+1},a)]\Big]$
        Update target policy:
            $\pi_{T,t+1}(a|s_t) = 1$ if $a = \arg\max_a q_{t+1}(s_t,a)$
            $\pi_{T,t+1}(a|s_t) = 0$ otherwise
```

7.6.5 Illustrative examples

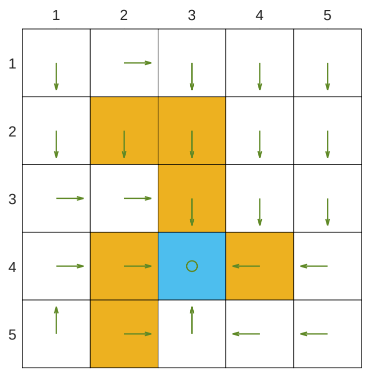
We next present some examples to demonstrate off-policy Q-learning. The task in these examples is to find an optimal policy for all the states. The reward setting is $r_{\text{boundary}} = r_{\text{forbidden}} = -1$, and $r_{\text{target}} = 1$. The discount rate is $\gamma = 0.9$. The learning rate is $\alpha = 0.1$.

1) We first calculate the ground truth by using the model-based policy iteration algorithm. In particular, an optimal policy and the corresponding optimal state values are shown in Figure 7.2(a) and (b), respectively.

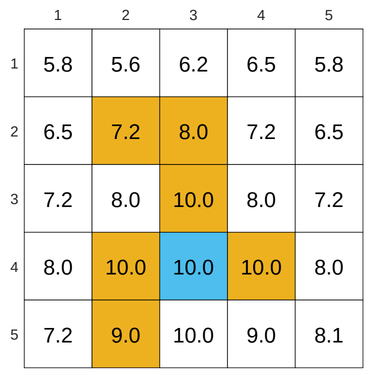
2) We use a uniform behavior policy, where the probability of taking any action at any state is 0.2, to generate a single episode of 100,000 steps. The behavior policy is shown in Figure 7.2(c) and the episode collected is shown in Figure 7.2(d). Due to the good exploration ability of the behavior policy, the episode visits every state-action pair many times.

3) Based on the episode generated by the behavior policy, the final target policy learned by Q-learning is shown in Figure 7.2(e). This policy is optimal because the estimated state value error (root-mean-squared error) converges to zero as shown in Figure 7.2(f). Here, the estimated state value can be calculated from the estimated action values In addition, one may have noticed that the learned optimal policy is not the same as the one in Figure 7.2(a). For example, the policies for (row=3,column=4) are different. That is because there are multiple optimal policies that have the same optimal state values.

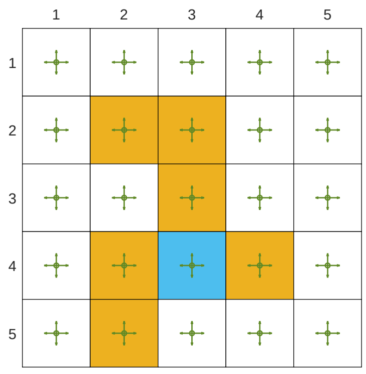
4) Since Q-learning bootstraps, the performance of the algorithm depends on the initial guess of the action values. As shown in Figure 7.2(g), when the initial guess is close to the true value, the estimate converges within about 10,000 steps. Otherwise, the convergence requires more steps (Figure 7.2(h)). Nevertheless, these figures demon-



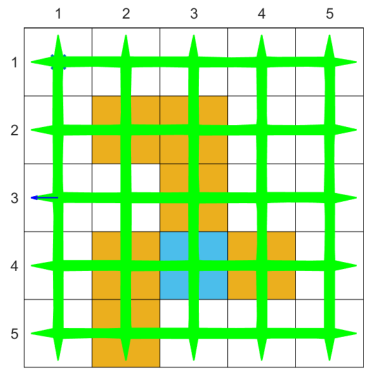
(a) Optimal policy



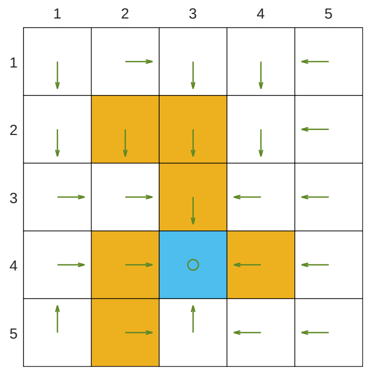
(b) Optimal state value



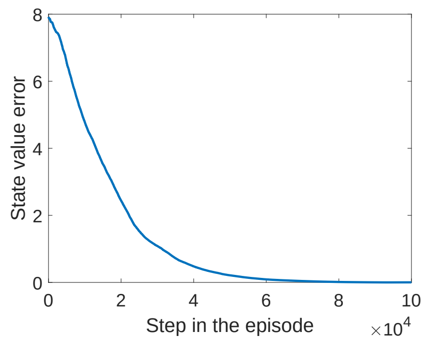
(c) Behavior policy



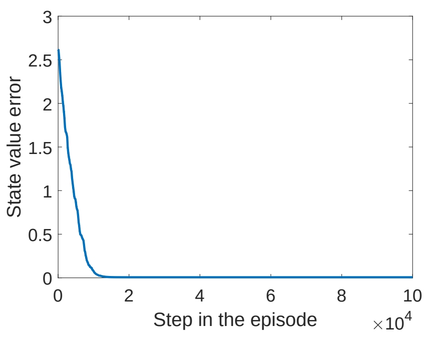
(d) Generated episode



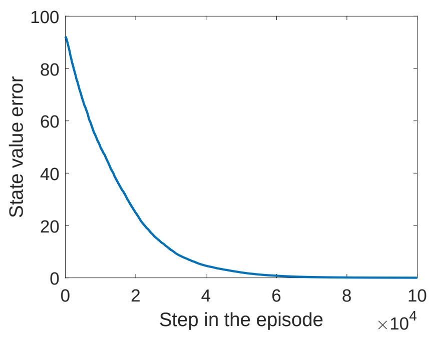
(e) Estimated policy



(f) State value error: $q_0(s, a) = 0$

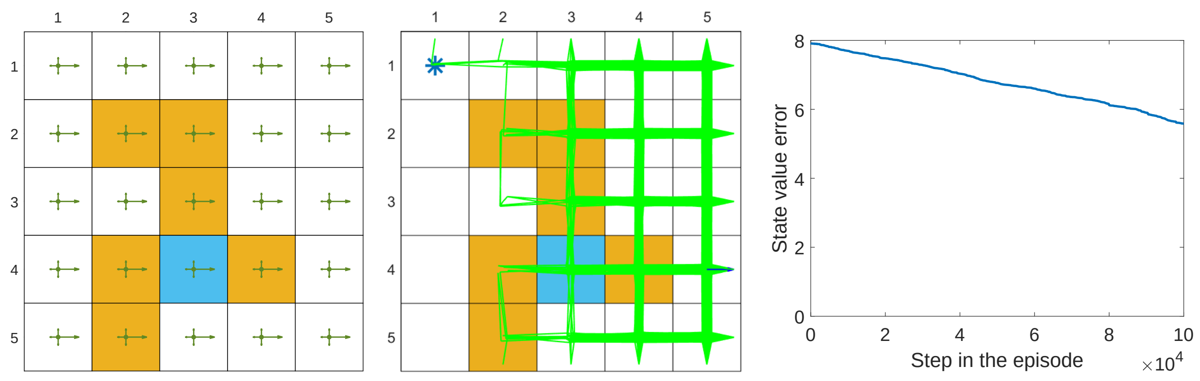


(g) State value error: $q_0(s, a) = 10$

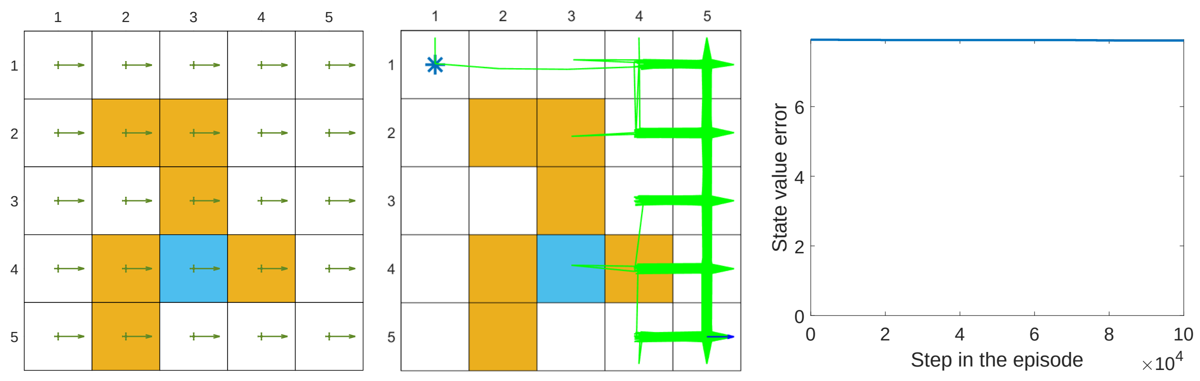


(h) State value error: $q_0(s, a) = 100$

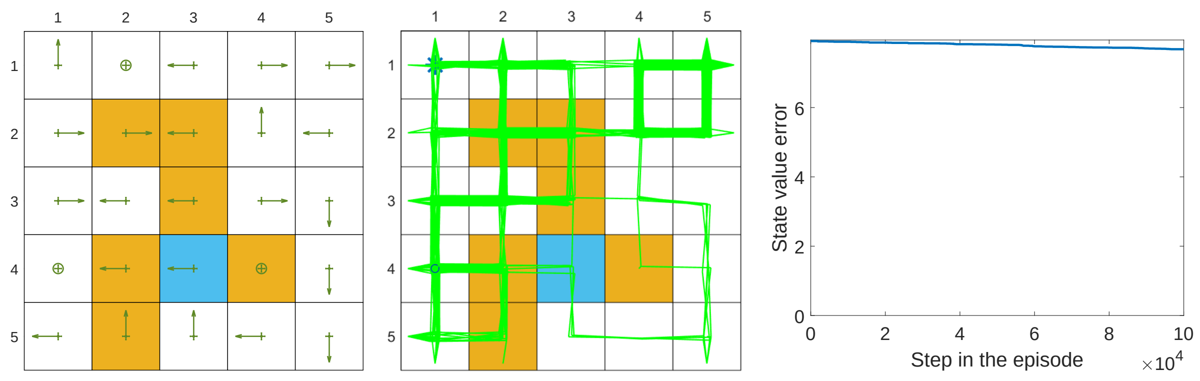
Figure 7.2: Examples to demonstrate off-policy learning by Q-learning. The optimal policy and optimal state values are shown in (a) and (b), respectively. The behavior policy and the generated single episode are shown in (c) and (d), respectively. The estimated policy and the estimation error evolution are shown in (e) and (f), respectively. The cases with different initial q-value are shown in (g) and (h), respectively.



(a) $\epsilon = 0.5$



(b) $\epsilon = 0.1$



(c) $\epsilon = 0.1$

Figure 7.3: Performance of Q-learning given different non-exploratory behavior policies. The figures in the left column show the behavior policies. The figures in the middle column show the generated episodes following the corresponding behavior policies. The episode in every example has 100,000 steps. The figures in the right column show the evolution of the root-mean-squared error of the estimated state values, which are calculated from the estimated action values.

strate that, even though the initial value is not sufficiently inaccurate, Q-learning can still learn satisfactorily fast.

When the behavior policy is not exploratory, the learning performance drops significantly. For example, consider the behavior policies shown in Figure 7.3. They are ϵ-greedy policies with $\epsilon = 0.5$ or 0.1. By the way, the uniform exploratory policy can be viewed as ϵ-greedy with $\epsilon = 1$. It can be clearly seen that, when ϵ decreases from 1 to 0.5 and then to 0.1, the learning rate drops significantly. As a result, although ϵ-greedy policies have certain exploration abilities, their exploration ability becomes weak when ϵ is small.

7.7 A unified viewpoint

Up to now, we have introduced different TD algorithms including Sarsa, Expected Sarsa, n-step Sarsa, and Q-learning. In this section, we introduce a unified framework to incorporate all these algorithms as well as MC learning.

These TD algorithms can all be interpreted as stochastic approximation algorithms solving the Bellman equation or Bellman optimality equation. In particular, all the algorithms can also be expressed in a unified expression:

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t)[q_t(s_t, a_t) - \bar{q}_t], \tag{7.17}$$

where $\bar{q}_t$ is the *TD target*. Different TD algorithms have different $\bar{q}_t$. See Table 7.2 for a list. The MC method can also be expressed as (7.17) by setting $\alpha_t(s_t, a_t) = 1$ and hence $q_{t+1}(s_t, a_t) = \bar{q}_t$.

Similar to the analysis of the TD learning algorithm in Section 7.2.1, it can be seen that (7.17) aims to solve a unified equation: $q(s, a) = \mathbb{E}[\bar{q}_t|s, a]$. This equitation has different expressions for different algorithms. These expressions are summarized in Table 7.2. As can be seen, all of them expect Q-learning aim to solve the Bellman equation with different expressions. Q-learning aims to solve the Bellman optimality equation, which is the fundamental reason why Q-learning is off-policy whereas the others are on-policy.

7.8 Summary

This chapter introduces an important class of RL algorithms called TD learning. The specific algorithms that we introduced include Sarsa, Expected Sarsa, n-step Sarsa, and Q-learning. From a mathematical point of view, all these algorithms are stochastic approximation algorithms solving the Bellman (optimality) equation.

The TD algorithms introduced in this chapter except Q-learning are to evaluate a given policy. That is to estimate the state/action values from experience samples. Together with policy improvement, they can be used to search for optimal policies based on the

Algorithm	**Expression of $\bar{q}_t$ in** (7.17)
Sarsa	$\bar{q}_t = r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1})$
n-step Sarsa	$\bar{q}_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^n q_t(s_{t+n}, a_{t+n})$
Expected Sarsa	$\bar{q}_t = r_{t+1} + \gamma \sum_a \pi_t(a\|s_{t+1}) q_t(s_{t+1}, a)$
Q-learning	$\bar{q}_t = r_{t+1} + \gamma \max_a q_t(s_{t+1}, a)$
Monte Carlo	$\bar{q}_t = r_{t+1} + \gamma r_{t+2} + \dots$

Algorithm	**Equation aimed to solve**
Sarsa	BE: $q_\pi(s,a) = \mathbb{E}\left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \| S_t = s, A_t = a\right]$
n-step Sarsa	BE: $q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^n G_{t+n} \| S_t = s, A_t = a]$
Expected Sarsa	BE: $q_\pi(s,a) = \mathbb{E}\big[R_{t+1} + \gamma \mathbb{E}[q_\pi(S_{t+1}, A_{t+1}) \| S_{t+1}] \big\| S_t = s, A_t = a\big]$
Q-learning	BOE: $q(s,a) = \mathbb{E}\left[R_{t+1} + \max_a q(S_{t+1}, a) \big\| S_t = s, A_t = a\right]$
Monte Carlo	BE: $q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \dots \| S_t = s, A_t = a]$

Table 7.2: A unified point of view of TD algorithms. Here, BE and BOE stand for the Bellman equation and Bellman optimality equation, respectively.

idea of generalized policy iteration. Moreover, these TD algorithms are also on-policy. That is, the target policy is used as the behavior policy to generate experience samples. In addition, compared to the MC learning method, TD learning is online and iterative. That is they can incrementally update the value and policy based on the experience sample obtained in each step.

Compared to other TD algorithms, Q-learning is special in the sense that it is off-policy. In particular, the target policy can be different from the behavior policy in Q-learning. As a result, the behavior policy can be chosen to be an exploratory policy to increase sampling efficiency. The fundamental reason why Q-learning is off-policy is that Q-learning aims to solve the Bellman optimality equation, which is independent of any specific policy. By contrast, the other TD algorithms introduced in this chapter are on-policy because they aim to solve the Bellman equation of a given policy.

It is worth mentioning that there are some methods to convert an on-policy learning algorithm to off-policy. Importance sampling is a widely used method [3, 17]. Moreover, there are various extensions of the TD algorithms introduced in this chapter. For example, the TD(λ) method provides a more general and unified framework for TD learning. The TD algorithm in Section 7.6 is simply TD(0), a special case of TD(λ). Details of TD(λ) can be found in [3, 13].

7.9 Q&A

can off-policy, can the behavior policy be time-varying?

– Q: From a mathematical point of view, what does TD learning do?

  A: TD algorithms are stochastic approximation algorithms for solving state/action values from the Bellman equation.

– Q: What does the term "TD" in TD learning mean?

  A: Every TD algorithm has a TD error, which represents the deficiency between the new sample and the current estimates. This deficiency is between two different time steps and hence called temporal difference.

– Q: What does the term "learning" in TD learning mean?

  A: Up to now, it has become clear that the term learning is mathematically estimating some values from experience samples. In particular, all the RL algorithms introduced so far contain value estimation and policy estimation. Policy estimation depends on value estimation. Value estimation is to estimate state/action values from experience samples, which is referred to as a learning process.

– Q: What are online learning and offline learning?

  A: Offline learning means the learning process can only start after a complete episode has been collected. Online learning means the value and policy can be updated instantly when a sample is received. MC learning is offline. TD learning is online.

– Q: What is the fundamental difference between TD learning and MC learning?

  A: The expression of a TD learning algorithm looks quite different from MC learning. That is because MC learning is offline whereas TD learning is online, iterative, and bootstrapping.

– Q: What is the fundamental common property between TD learning and MC learning?

  A: They both are algorithms estimating state/action values from experience samples.

– Q: While TD learning such as Sarsa aims to estimate action values, how can it be used for optimal policy searching?

  A: To obtain an optimal policy, the value estimation process should interact with the policy update process. That is, after a value is updated, the corresponding policy should be updated. Then, a new sample generated by the updated policy is used to update values again. This is the idea of generalized policy iteration.

– Q: When we use Sarsa for policy searching, why do we update policies as ϵ-greedy?

  A: That is because the policy is also used to generate samples for value estimation and hence we hope it is exploratory.

- Q: While the convergence of TD learning requires that the learning rate converges to zero gradually, why is the learning rate often chosen to be constant in practice?

  A: The fundamental reason is the policy to be evaluated is nonstationary. A TD learning algorithm like Sarsa aims to estimate the action values of a *given fixed policy*. In other words, it aims to evaluate the given policy. If the policy is stationary, the learning rate can decrease to zero gradually to ensure convergence with probability 1 as stated in Theorem 7.1.

  However, when we put Sarsa in the context of optimal policy searching, the value estimation process keeps interacting with the policy update process. Therefore, the policy that Sarsa aims to evaluate keeps *changing*. In this case, we need to use a constant learning rate because, every time we have a new policy to evaluate, the learning rate cannot be too small. The drawback of a constant learning rate is that the value estimate may still fluctuate eventually. However, as long as the constant learning rate is sufficiently small, the fluctuation would not jeopardize the convergence.

- Q: Should we estimate the optimal policies for all states or a subset of states?

  A: It depends on the task. One may have noticed that some tasks in the illustrative examples in this chapter are not to find the optimal policies for all states. Instead, they only need to find a good path from a specific starting state to the target state. Such a task is not data demanding because we do not need to visit every state-action pair sufficiently many times. As shown in the examples in Figure 7.1, even if some states are not visited, we can still obtain a good path.

  It, however, must be noted that the path is only good but not guaranteed to be optimal. That is because not all state-action pairs have been explored and there may be better paths unexplored. Nevertheless, given sufficient data, there is a high probability for us to find a good or *locally* optimal path. By locally optimal, we mean the path is optimal within a neighborhood of the path.

- Q: What are behavior policy and target policy?

  A: There exist two policies in a TD learning task: *behavior policy* and *target policy*. The behavior policy is used to generate experience samples. The target policy is the one constantly updated based on the experience samples.

- Q: What are off-policy learning and on-policy learning?

  A: When the behavior policy is the same as the target policy, such kind of learning is called on-policy. When they are different, the learning is called off-policy.

- Q: What are the advantages of off-policy learning compared to on-policy learning?

  A: Since on-policy learning is a special case of off-policy learning when the behavior and target policies are the same, off-policy learning is more flexible than on-policy learning. For example, off-policy learning can estimate optimal policies based on some experience samples generated by any other behavior policies.

– Q: What is the fundamental reason for Q-learning to be off-policy while all the other TD algorithms in this chapter are on-policy?

  A: The fundamental reason why Q-learning is off-policy is that Q-learning aims to solve the *Bellman optimality equation*. The other TD algorithms are on-policy because they aim to solve the *Bellman equation of a given policy*. Since the Bellman optimality equation does not depend on any specific policy, it can use the experience samples generated by any other policies to estimate the optimal policy. However, the Bellman equation depends on a given policy, solving it of course requires the experience samples generated by the given policy.

– Q: Why does the off-policy version of Q-learning update policies as greedy instead of ϵ-greedy?

  A: Since the target policy is not required to generate experience samples, it is not required to be exploratory.