Chapter 7

Temporal-Difference Methods

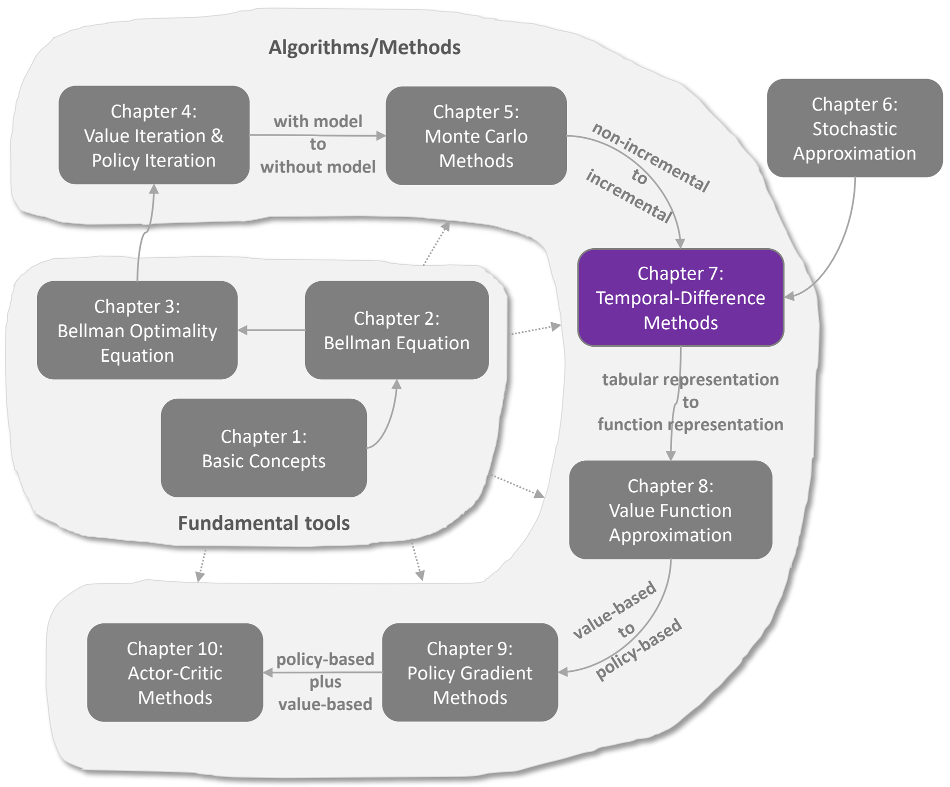

Algorithms/Methods
Chapter 4: Value Iteration & Policy Iteration
with model to without model
Chapter 5: Monte Carlo Methods
non-incremental to incremental
Chapter 6: Stochastic Approximation
Chapter 7: Temporal-Difference Methods
Chapter 3: Bellman Optimality Equation
Chapter 2: Bellman Equation
Chapter 1: Basic Concepts
Fundamental tools
tabular representation to function representation
Chapter 8: Value Function Approximation
value-based to policy-based
Chapter 10: Actor-Critic Methods
policy-based plus value-based
Chapter 9: Policy Gradient Methods

Figure 7.1: Where we are in this book.

This chapter introduces temporal-difference (TD) methods for reinforcement learning. Similar to Monte Carlo (MC) learning, TD learning is also model-free, but it has some advantages due to its incremental form. With the preparation in Chapter 6, readers will not feel alarmed when seeing TD learning algorithms. In fact, TD learning algorithms can be viewed as special stochastic algorithms for solving the Bellman or Bellman optimality equations.

Since this chapter introduces quite a few TD algorithms, we first overview these algorithms and clarify the relationships between them.

◇ Section 7.1 introduces the most basic TD algorithm, which can estimate the *state* *values* of a given policy. It is important to understand this basic algorithm first before studying the other TD algorithms.

◇ Section 7.2 introduces the Sarsa algorithm, which can estimate the *action values* of a given policy. This algorithm can be combined with a policy improvement step to find optimal policies. The Sarsa algorithm can be easily obtained from the TD algorithm in Section 7.1 by replacing state value estimation with action value estimation.

◇ Section 7.3 introduces the n-step Sarsa algorithm, which is a generalization of the Sarsa algorithm. It will be shown that Sarsa and MC learning are two special cases of n-step Sarsa.

◇ Section 7.4 introduces the Q-learning algorithm, which is one of the most classic reinforcement learning algorithms. While the other TD algorithms aim to solve the Bellman equation of a given policy, Q-learning aims to directly solve the Bellman optimality equation to obtain optimal policies.

◇ Section 7.5 compares the TD algorithms introduced in this chapter and provides a unified point of view.

7.1 TD learning of state values

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

7.1.1 Algorithm description

Given a policy π, our goal is to estimate $v_\pi(s)$ for all $s \in \mathcal{S}$. Suppose that we have some experience samples $(s_0, r_1, s_1, \dots, s_t, r_{t+1}, s_{t+1}, \dots)$ generated following π. Here, t denotes the time step. The following TD algorithm can estimate the state values using these samples:

$$v_{t+1}(s_t) = v_t(s_t) - \alpha_t(s_t)\Big[v_t(s_t) - \big(r_{t+1} + \gamma v_t(s_{t+1})\big)\Big], \tag{7.1}$$

$$v_{t+1}(s) = v_t(s), \quad \text{for all } s \neq s_t, \tag{7.2}$$

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

It should be noted that, at time t, only the value of the visited state s_t is updated. The values of the unvisited states $s \neq s_t$ remain unchanged as shown in (7.2). Equation (7.2) is often omitted for simplicity, but it should be kept in mind because the algorithm would be mathematically incomplete without this equation.

Readers who see the TD learning algorithm for the first time may wonder why it is designed like this. In fact, it can be viewed as a special stochastic approximation algorithm for solving the Bellman equation. To see that, first recall that the definition of the state value is

$$v_\pi(s) = \mathbb{E}\big[R_{t+1} + \gamma G_{t+1} | S_t = s\big], \quad s \in \mathcal{S}. \tag{7.3}$$

We can rewrite (7.3) as

$$v_\pi(s) = \mathbb{E}\big[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s\big], \quad s \in \mathcal{S}. \tag{7.4}$$

That is because $\mathbb{E}[G_{t+1}|S_t = s] = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) v_\pi(s') = \mathbb{E}[v_\pi(S_{t+1})|S_t = s]$. Equation (7.4) is another expression of the Bellman equation. It is sometimes called the *Bellman expectation equation*.

The TD algorithm can be derived by applying the Robbins-Monro algorithm (Chapter 6) to solve the Bellman equation in (7.4). Interested readers can check the details in Box 7.1.

Box 7.1: Derivation of the TD algorithm

We next show that the TD algorithm in (7.1) can be obtained by applying the Robbins-Monro algorithm to solve (7.4).

For state s_t, we define a function as

$$g(v_\pi(s_t)) \doteq v_\pi(s_t) - \mathbb{E}\big[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s_t\big].$$

Then, (7.4) is equivalent to

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

Our goal is to solve the above equation to obtain $v_\pi(s_t)$ using the Robbins-Monro algorithm. Since we can obtain r_{t+1} and s_{t+1}, which are the samples of R_{t+1} and S_{t+1}, the noisy observation of $g(v_\pi(s_t))$ that we can obtain is

$$\begin{aligned}\tilde{g}(v_\pi(s_t)) &= v_\pi(s_t) - \big[r_{t+1} + \gamma v_\pi(s_{t+1})\big] \\ &= \underbrace{\Big(v_\pi(s_t) - \mathbb{E}\big[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s_t\big]\Big)}_{g(v_\pi(s_t))} \\ &\qquad + \underbrace{\Big(\mathbb{E}\big[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s_t\big] - \big[r_{t+1} + \gamma v_\pi(s_{t+1})\big]\Big)}_{\eta}.\end{aligned}$$

Therefore, the Robbins-Monro algorithm (Section 6.2) for solving $g(v_\pi(s_t)) = 0$ is

$$\begin{aligned} v_{t+1}(s_t) &= v_t(s_t) - \alpha_t(s_t)\tilde{g}(v_t(s_t)) \\ &= v_t(s_t) - \alpha_t(s_t)\Big(v_t(s_t) - \big[r_{t+1} + \gamma v_\pi(s_{t+1})\big]\Big), \end{aligned} \tag{7.5}$$

where $v_t(s_t)$ is the estimate of $v_\pi(s_t)$ at time t, and $\alpha_t(s_t)$ is the learning rate.

The algorithm in (7.5) has a similar expression to that of the TD algorithm in (7.1). The only difference is that the right-hand side of (7.5) contains $v_\pi(s_{t+1})$, whereas (7.1) contains $v_t(s_{t+1})$. That is because (7.5) is designed to merely estimate the state value of s_t by assuming that the state values of the other states are already known. If we would like to estimate the state values of all the states, then $v_\pi(s_{t+1})$ on the right-hand side should be replaced with $v_t(s_{t+1})$. Then, (7.5) is exactly the same as (7.1). However, can such a replacement still ensure convergence? The answer is yes, and it will be proven later in Theorem 7.1.

7.1.2 Property analysis

Some important properties of the TD algorithm are discussed as follows.

First, we examine the expression of the TD algorithm more closely. In particular, (7.1) can be described as

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

where

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

is called the *TD target* and

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

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

◇ Why is $\bar{v}_t$ called the TD target?

This is because $\bar{v}_t$ is the *target value* that the algorithm attempts to drive $v(s_t)$ to. To see that, subtracting $\bar{v}_t$ from both sides of (7.6) gives

$$\begin{aligned} v_{t+1}(s_t) - \bar{v}_t &= \big[v_t(s_t) - \bar{v}_t\big] - \alpha_t(s_t)\big[v_t(s_t) - \bar{v}_t\big] \\ &= [1 - \alpha_t(s_t)]\big[v_t(s_t) - \bar{v}_t\big]. \end{aligned}$$

Taking the absolute values of both sides of the above equation gives

$$|v_{t+1}(s_t) - \bar{v}_t| = |1 - \alpha_t(s_t)||v_t(s_t) - \bar{v}_t|.$$

Since $\alpha_t(s_t)$ is a small positive number, we have $0 < 1 - \alpha_t(s_t) < 1$. It then follows that

$$|v_{t+1}(s_t) - \bar{v}_t| < |v_t(s_t) - \bar{v}_t|.$$

The above inequality is important because it indicates that the new value $v_{t+1}(s_t)$ is closer to $\bar{v}_t$ than the old value $v_t(s_t)$. Therefore, this algorithm mathematically drives $v_t(s_t)$ toward $\bar{v}_t$. This is why $\bar{v}_t$ is called the TD target.

◇ What is the interpretation of the TD error?

First, this error is called *temporal-difference* because $\delta_t = v_t(s_t) - (r_{t+1} + \gamma v_t(s_{t+1}))$ reflects the discrepancy between two time steps t and $t+1$. Second, the TD error is zero in the expectation sense when the state value estimate is accurate. To see that, when $v_t = v_\pi$, the expected value of the TD error is

$$\begin{aligned}\mathbb{E}[\delta_t|S_t = s_t] &= \mathbb{E}\big[v_\pi(S_t) - (R_{t+1} + \gamma v_\pi(S_{t+1}))|S_t = s_t\big] \\ &= v_\pi(s_t) - \mathbb{E}\big[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s_t\big] \\ &= 0. \qquad\qquad \text{(due to (7.3))}\end{aligned}$$

Therefore, the TD error reflects not only the discrepancy between two time steps but also, more importantly, the discrepancy between the estimate v_t and the true state value v_π.

On a more abstract level, the TD error can be interpreted as the *innovation*, which indicates new information obtained from the experience sample (s_t, r_{t+1}, s_{t+1}). The fundamental idea of TD learning is to correct our current estimate of the state value based on the newly obtained information. Innovation is fundamental in many estimation problems such as Kalman filtering [33, 34].

Second, the TD algorithm in (7.1) can only estimate the state values of a given policy. To find optimal policies, we still need to further calculate the action values and then conduct policy improvement. This will be introduced in Section 7.2. Nevertheless, the TD algorithm introduced in this section is very basic and important for understanding the other algorithms in this chapter.

Third, while both TD learning and MC learning are model-free, what are their advantages and disadvantages? The answers are summarized in Table 7.1.

TD learning	**MC learning**
Incremental: TD learning is incremental. It can update the state/action values immediately after receiving an experience sample.	*Non-incremental:* MC learning is non-incremental. It must wait until an episode has been completely collected. That is because it must calculate the discounted return of the episode.
Continuing tasks: Since TD learning is incremental, it can handle both episodic and continuing tasks. Continuing tasks may not have terminal states.	*Episodic tasks:* Since MC learning is non-incremental, it can only handle episodic tasks where the episodes terminate after a finite number of steps.
Bootstrapping: TD learning bootstraps because the update of a state/action value relies on the previous estimate of this value. As a result, TD learning requires an initial guess of the values.	*Non-bootstrapping:* MC is not bootstrapping because it can directly estimate state/action values without initial guesses.
Low estimation variance: The estimation variance of TD is lower than that of MC because it involves fewer random variables. For instance, to estimate an action value $q_\pi(s_t, a_t)$, Sarsa merely requires the 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 $R_{t+1}+\gamma R_{t+2}+\gamma^2 R_{t+3}+\dots$. Suppose that the length of each episode is L. Assume that each state has the same number of actions as $\lvert\mathcal{A}\rvert$. Then, there are $\lvert\mathcal{A}\rvert^L$ possible episodes following a soft policy. If we merely use a few episodes to estimate, it is not surprising that the estimation variance is high.

Table 7.1: A comparison between TD learning and MC learning.

7.1.3 Convergence analysis

The convergence analysis of the TD algorithm in (7.1) is given below.

Theorem 7.1 (Convergence of TD learning). *Given a policy π, by the TD algorithm in (7.1), $v_t(s)$ converges almost surely to $v_\pi(s)$ as $t \to \infty$ for all $s \in \mathcal{S}$ if $\sum_t \alpha_t(s) = \infty$ and $\sum_t \alpha_t^2(s) < \infty$ for all $s \in \mathcal{S}$.*

Some remarks about α_t are given below. First, the condition of $\sum_t \alpha_t(s) = \infty$ and $\sum_t \alpha_t^2(s) < \infty$ must be valid for all $s \in \mathcal{S}$. Note that, at time t, $\alpha_t(s) > 0$ if s is being visited and $\alpha_t(s) = 0$ otherwise. The condition $\sum_t \alpha_t(s) = \infty$ requires the state s to be visited an infinite (or sufficiently many) number of times. This requires either the condition of exploring starts or an exploratory policy so that every state-action pair can possibly be visited many times. Second, the learning rate α_t is often selected as a small positive constant in practice. In this case, the condition that $\sum_t \alpha_t^2(s) < \infty$ is no longer valid. When α is constant, it can still be shown that the algorithm converges in the sense of expectation [24, Section 1.5].

Box 7.2: Proof of Theorem 7.1

We prove the convergence based on Theorem 6.3 in Chapter 6. To do that, we need first to construct a stochastic process as that in Theorem 6.3. Consider an arbitrary state $s \in \mathcal{S}$. At time t, it follows from the TD algorithm in (7.1) that

$$v_{t+1}(s) = v_t(s) - \alpha_t(s)\Big(v_t(s) - (r_{t+1} + \gamma v_t(s_{t+1}))\Big), \quad \text{if } s = s_t, \tag{7.7}$$

or

$$v_{t+1}(s) = v_t(s), \quad \text{if } s \neq s_t. \tag{7.8}$$

The estimation error is defined as

$$\Delta_t(s) \doteq v_t(s) - v_\pi(s),$$

where $v_\pi(s)$ is the state value of s under policy π. Deducting $v_\pi(s)$ from both sides of (7.7) gives

$$\begin{aligned}\Delta_{t+1}(s) &= (1-\alpha_t(s))\Delta_t(s) + \alpha_t(s)(\underbrace{r_{t+1} + \gamma v_t(s_{t+1}) - v_\pi(s)}_{\eta_t(s)}) \\ &= (1-\alpha_t(s))\Delta_t(s) + \alpha_t(s)\eta_t(s), \qquad s = s_t. \end{aligned} \tag{7.9}$$

Deducting $v_\pi(s)$ from both sides of (7.8) gives

$$\Delta_{t+1}(s) = \Delta_t(s) = (1-\alpha_t(s))\Delta_t(s) + \alpha_t(s)\eta_t(s), \qquad s \neq s_t,$$

whose expression is the same as that of (7.9) except that $\alpha_t(s) = 0$ and $\eta_t(s) = 0$. Therefore, regardless of whether $s = s_t$, we obtain the following unified expression:

$$\Delta_{t+1}(s) = (1-\alpha_t(s))\Delta_t(s) + \alpha_t(s)\eta_t(s).$$

This is the process in Theorem 6.3. Our goal is to show that the three conditions in Theorem 6.3 are satisfied and hence the process converges.

The first condition is valid as assumed in Theorem 7.1. We next show that the second condition is valid. That is, $\|\mathbb{E}[\eta_t(s)|\mathcal{H}_t]\|_\infty \leq \gamma\|\Delta_t(s)\|_\infty$ for all $s \in \mathcal{S}$. Here, $\mathcal{H}_t$ represents the historical information (see the definition in Theorem 6.3). Due to the Markovian property, $\eta_t(s) = r_{t+1} + \gamma v_t(s_{t+1}) - v_\pi(s)$ or $\eta_t(s) = 0$ does not depend on the historical information once s is given. As a result, we have $\mathbb{E}[\eta_t(s)|\mathcal{H}_t] = \mathbb{E}[\eta_t(s)]$. For $s \neq s_t$, we have $\eta_t(s) = 0$. Then, it is trivial to see that

$$|\mathbb{E}[\eta_t(s)]| = 0 \leq \gamma \|\Delta_t(s)\|_\infty. \tag{7.10}$$

For $s = s_t$, we have

$$\begin{aligned}
\mathbb{E}[\eta_t(s)] &= \mathbb{E}[\eta_t(s_t)] \\
&= \mathbb{E}[r_{t+1} + \gamma v_t(s_{t+1}) - v_\pi(s_t)|s_t] \\
&= \mathbb{E}[r_{t+1} + \gamma v_t(s_{t+1})|s_t] - v_\pi(s_t).
\end{aligned}$$

Since $v_\pi(s_t) = \mathbb{E}[r_{t+1} + \gamma v_\pi(s_{t+1})|s_t]$, the above equation implies that

$$\begin{aligned}
\mathbb{E}[\eta_t(s)] &= \gamma \mathbb{E}[v_t(s_{t+1}) - v_\pi(s_{t+1})|s_t] \\
&= \gamma \sum_{s' \in \mathcal{S}} p(s'|s_t)[v_t(s') - v_\pi(s')].
\end{aligned}$$

It follows that

$$\begin{aligned}
|\mathbb{E}[\eta_t(s)]| &= \gamma \left| \sum_{s' \in \mathcal{S}} p(s'|s_t)[v_t(s') - v_\pi(s')] \right| \\
&\leq \gamma \sum_{s' \in \mathcal{S}} p(s'|s_t) \max_{s' \in \mathcal{S}} |v_t(s') - v_\pi(s')| \\
&= \gamma \max_{s' \in \mathcal{S}} |v_t(s') - v_\pi(s')| \\
&= \gamma \|v_t(s') - v_\pi(s')\|_\infty \\
&= \gamma \|\Delta_t(s)\|_\infty.
\end{aligned} \tag{7.11}$$

Therefore, at time t, we know from (7.10) and (7.11) that $|\mathbb{E}[\eta_t(s)]| \leq \gamma \|\Delta_t(s)\|_\infty$ for all $s \in \mathcal{S}$ regardless of whether $s = s_t$. Thus,

$$\|\mathbb{E}[\eta_t(s)]\|_\infty \leq \gamma \|\Delta_t(s)\|_\infty,$$

which is the second condition in Theorem 6.3. Finally, regarding the third condition, we have $\text{var}[\eta_t(s)|\mathcal{H}_t] = \text{var}[r_{t+1} + \gamma v_t(s_{t+1}) - v_\pi(s_t)|s_t] = \text{var}[r_{t+1} + \gamma v_t(s_{t+1})|s_t]$ for $s = s_t$ and $\text{var}[\eta_t(s)|\mathcal{H}_t] = 0$ for $s \neq s_t$. Since r_{t+1} is bounded, the third condition can be proven without difficulty.

The above proof is inspired by [32].

7.2 TD learning of action values: Sarsa

The TD algorithm introduced in Section 7.1 can only estimate *state values*. This section introduces another TD algorithm called Sarsa that can directly estimate *action values*. Estimating action values is important because it can be combined with a policy improvement step to learn optimal policies.

7.2.1 Algorithm description

Given a policy π, our goal is to estimate the action values. Suppose that we have some experience samples generated following π: $(s_0, a_0, r_1, s_1, a_1, \dots, s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, \dots)$. We can use the following *Sarsa* algorithm to estimate the action values:

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

where $t = 0, 1, 2, \dots$ and $\alpha_t(s_t, a_t)$ is the learning rate. Here, $q_t(s_t, a_t)$ is the estimate of $q_\pi(s_t, a_t)$. At time t, only the q-value of (s_t, a_t) is updated, whereas the q-values of the others remain the same.

Some important properties of the Sarsa algorithm are discussed as follows.

◇ Why is this algorithm called "Sarsa"? That is because each iteration of the algorithm requires $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. Sarsa is an abbreviation for state-action-reward-state-action. The Sarsa algorithm was first proposed in [35] and its name was coined by [3].

◇ Why is Sarsa designed in this way? One may have noticed that Sarsa is similar to the TD algorithm in (7.1). In fact, Sarsa can be easily obtained from the TD algorithm by replacing state value estimation with action value estimation.

◇ What does Sarsa do mathematically? Similar to the TD algorithm in (7.1), Sarsa is a stochastic approximation algorithm for solving the Bellman equation of a given policy:

$$q_\pi(s, a) = \mathbb{E}\left[R + \gamma q_\pi(S', A')|s, a\right], \quad \text{for all } (s, a). \qquad (7.13)$$

Equation (7.13) is the Bellman equation expressed in terms of action values. A proof is given in Box 7.3.

Box 7.3: Showing that (7.13) is the Bellman equation

As introduced in Section 2.8.2, the Bellman equation expressed in terms of action values is

$$\begin{aligned} q_\pi(s,a) &= \sum_r rp(r|s,a) + \gamma \sum_{s'} \sum_{a'} q_\pi(s',a')p(s'|s,a)\pi(a'|s') \\ &= \sum_r rp(r|s,a) + \gamma \sum_{s'} p(s'|s,a) \sum_{a'} q_\pi(s',a')\pi(a'|s'). \end{aligned} \tag{7.14}$$

This equation establishes the relationships among the action values. Since

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

(7.14) 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 the expected value, the above equation is equivalent to (7.13). Hence, (7.13) is the Bellman equation.

◇ Is Sarsa convergent? Since Sarsa is the action-value version of the TD algorithm in (7.1), the convergence result is similar to Theorem 7.1 and given below.

Theorem 7.2 (Convergence of Sarsa)**.** *Given a policy π, by the Sarsa algorithm in (7.12), $q_t(s,a)$ converges almost surely 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).*

The proof is similar to that of Theorem 7.1 and is omitted here. 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). In particular, $\sum_t \alpha_t(s,a) = \infty$ requires that every state-action pair must be visited an infinite (or sufficiently many) number of times. At time t, if $(s,a) = (s_t,a_t)$, then $\alpha_t(s,a) > 0$; otherwise, $\alpha_t(s,a) = 0$.

7.2.2 Optimal policy learning via Sarsa

The Sarsa algorithm in (7.12) can only estimate the action values of a given policy. To find optimal policies, we can combine it with a policy improvement step. The combination is also often called Sarsa, and its implementation procedure is given in Algorithm 7.1.

As shown in Algorithm 7.1, each iteration has two steps. The first step is to update the q-value of the visited state-action pair. The second step is to update the policy to an ϵ-greedy one. The q-value update step only updates the single state-action pair visited

Algorithm 7.1: Optimal policy learning by Sarsa

```
Initialization: $\alpha_t(s,a)=\alpha>0$ for all $(s,a)$ and all $t$. $\epsilon\in(0,1)$. Initial $q_0(s,a)$ for all $(s,a)$. Initial $\epsilon$-greedy policy $\pi_0$ derived from $q_0$.
Goal: Learn an optimal policy that can lead the agent to the target state from an initial state $s_0$.

For each episode, do
    Generate $a_0$ at $s_0$ following $\pi_0(s_0)$
    If $s_t$ $(t=0,1,2,\dots)$ is not the target state, do
        Collect an experience sample $(r_{t+1},s_{t+1},a_{t+1})$ given $(s_t,a_t)$: generate $r_{t+1},s_{t+1}$ by interacting with the environment; generate $a_{t+1}$ following $\pi_t(s_{t+1})$.
        Update q-value for $(s_t,a_t)$:
            $q_{t+1}(s_t,a_t)=q_t(s_t,a_t)-\alpha_t(s_t,a_t)\Big[q_t(s_t,a_t)-(r_{t+1}+\gamma q_t(s_{t+1},a_{t+1}))\Big]$
        Update policy for $s_t$:
            $\pi_{t+1}(a|s_t)=1-\frac{\epsilon}{|\mathcal{A}(s_t)|}(|\mathcal{A}(s_t)|-1)$ if $a=\arg\max_a q_{t+1}(s_t,a)$
            $\pi_{t+1}(a|s_t)=\frac{\epsilon}{|\mathcal{A}(s_t)|}$ otherwise
        $s_t\leftarrow s_{t+1}$, $a_t\leftarrow a_{t+1}$
```

at time t. Afterward, the policy of s_t is immediately updated. Therefore, we do *not* evaluate a given policy sufficiently well before updating the policy. This is based on the idea of generalized policy iteration. Moreover, after the policy is updated, the policy is immediately used to generate the next experience sample. The policy here is ϵ-greedy so that it is exploratory.

A simulation example is shown in Figure 7.2 to demonstrate the Sarsa algorithm. Unlike all the tasks we have seen in this book, the task here aims to find an optimal path from a specific starting state to a target state. It does *not* aim to find the optimal policies for all states. This task is often encountered in practice where the starting state (e.g., home) and the target state (e.g., workplace) are fixed, and we only need to find an optimal path connecting them. This task is relatively simple because we only need to explore the states that are close to the path and do not need to explore all the states. However, if we do not explore all the states, the final path may be *locally* optimal rather than globally optimal.

The simulation setup and simulation results are discussed below.

◇ *Simulation setup:* In this example, all the episodes start from the top-left state and terminate at the target state. The reward settings are $r_{\text{target}}=0$, $r_{\text{forbidden}}=r_{\text{boundary}}=-10$, and $r_{\text{other}}=-1$. Moreover, $\alpha_t(s,a)=0.1$ for all t and $\epsilon=0.1$. The initial guesses of the action values are $q_0(s,a)=0$ for all (s,a). The initial policy has a uniform distribution: $\pi_0(a|s)=0.2$ for all s,a.

◇ *Learned policy:* The left figure in Figure 7.2 shows the final policy learned by Sarsa. As can be seen, this policy can successfully lead to the target state from the starting

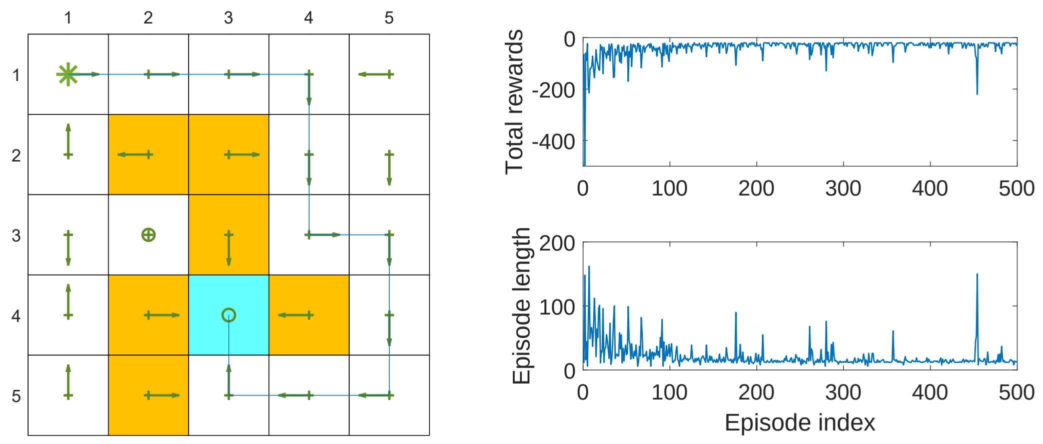


Figure 7.2: An example for demonstrating Sarsa. All the episodes start from the top-left state and terminate when reaching the target state (the blue cell). The goal is to find an optimal path from the starting state to the target state. The reward settings are $r_{\text{target}} = 0$, $r_{\text{forbidden}} = r_{\text{boundary}} = -10$, and $r_{\text{other}} = -1$. The learning rate is $\alpha = 0.1$ and the value of ϵ is 0.1. The left figure shows the final policy obtained by the algorithm. The right figures show the total reward and length of every episode.

state. However, the policies of some other states may not be optimal. That is because the other states are not well explored.

◇ *Total reward of each episode:* The top-right subfigure in Figure 7.2 shows the total reward of each episode. Here, the total reward is the non-discounted sum of all immediate rewards. As can be seen, the total reward of each episode increases gradually. That is because the initial policy is not good and hence negative rewards are frequently obtained. As the policy becomes better, the total reward increases.

◇ *Length of each episode:* The bottom-right subfigure in Figure 7.2 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. Notably, the length of an episode may increase abruptly (e.g., the 460th episode) and the corresponding total reward also drops sharply. That is because the policy is ϵ-greedy, and there is a chance for it to take non-optimal actions. One way to resolve this problem is to use decaying ϵ whose value converges to zero gradually.

Finally, Sarsa also has some variants such as Expected Sarsa. Interested readers may check Box 7.4.

Box 7.4: Expected Sarsa

Given a policy π, its action values can be evaluated by Expected Sarsa, which is a variant of Sarsa. The Expected Sarsa algorithm is

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

where

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

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 their TD targets. In particular, the TD target in Expected Sarsa is $r_{t+1} + \gamma \mathbb{E}[q_t(s_{t+1},A)]$, 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 called Expected Sarsa. Although calculating the expected value may increase the computational complexity slightly, it is beneficial in the sense that it reduces the estimation variances because it reduces the random variables in Sarsa from $\{s_t,a_t,r_{t+1},s_{t+1},a_{t+1}\}$ to $\{s_t,a_t,r_{t+1},s_{t+1}\}$.

Similar to the TD learning algorithm in (7.1), Expected Sarsa can be viewed as a stochastic approximation algorithm for solving the following equation:

$$q_\pi(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma \mathbb{E}[q_\pi(S_{t+1},A_{t+1})|S_{t+1}]\Big|S_t = s, A_t = a\Big], \quad \text{for all } s,a. \tag{7.15}$$

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

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

into (7.15) 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 similar to that of Sarsa. More details can be found in [3,36,37].

7.3 TD learning of action values: n-step Sarsa

This section introduces *n-step Sarsa*, an extension of Sarsa. We will see that Sarsa and MC learning are two extreme cases of n-step Sarsa.

Recall that the definition of the action value is

$$q_\pi(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a], \tag{7.16}$$

where G_t is the discounted return satisfying

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

In fact, G_t can also be decomposed into different forms:

$$\begin{aligned}
\text{Sarsa} \longleftarrow \quad G_t^{(1)} &= R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}),\\
G_t^{(2)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, A_{t+2}),\\
&\vdots\\
n\text{-step Sarsa} \longleftarrow \quad G_t^{(n)} &= R_{t+1} + \gamma R_{t+2} + \dots + \gamma^n q_\pi(S_{t+n}, A_{t+n}),\\
&\vdots\\
\text{MC} \longleftarrow \quad G_t^{(\infty)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} \dots
\end{aligned}$$

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.

Substituting different decompositions of $G_t^{(n)}$ into $q_\pi(s,a)$ in (7.16) results in different algorithms.

◇ When $n = 1$, we have

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

The corresponding stochastic approximation algorithm for solving this equation is

$$q_{t+1}(s_t,a_t) = q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - (r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1}))\Big],$$

which is the Sarsa algorithm in (7.12).

◇ When $n = \infty$, we have

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

The corresponding algorithm for solving this equation is

$$q_{t+1}(s_t, a_t) = g_t \doteq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots,$$

where g_t is a sample of G_t. In fact, this is the MC learning algorithm, which approximates the action value of (s_t, a_t) using the discounted return of an episode starting from (s_t, a_t).

◇ For a general value of n, we have

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

The corresponding algorithm for solving the above equation is

$$\begin{aligned} q_{t+1}(s_t, a_t) &= q_t(s_t, a_t) \\ &\quad - \alpha_t(s_t, a_t)\Big[q_t(s_t, a_t) - \big(r_{t+1} + \gamma r_{t+2} + \dots + \gamma^n q_t(s_{t+n}, a_{t+n})\big)\Big]. \end{aligned} \tag{7.17}$$

This algorithm is called *n-step Sarsa*.

In summary, n-step Sarsa is a more general algorithm because it becomes the (one-step) Sarsa algorithm when $n = 1$ and the MC learning algorithm when $n = \infty$ (by setting $\alpha_t = 1$).

To implement the n-step Sarsa algorithm in (7.17), we need the experience samples $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, \dots, r_{t+n}, s_{t+n}, a_{t+n})$. Since $(r_{t+n}, s_{t+n}, a_{t+n})$ has not been collected at time t, we have to wait until time $t + n$ to update the q-value of (s_t, a_t). To that end, (7.17) can be rewritten as

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

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 between that of Sarsa and MC learning. In particular, if n is selected as a large number, n-step Sarsa is close to MC learning: the estimate has a relatively high variance but a small bias. If n is selected to be small, n-step Sarsa is close to Sarsa: the estimate has a relatively large bias but a low variance. Finally, the n-step Sarsa algorithm presented here is merely used for policy evaluation. It must be combined with a policy improvement step to learn optimal policies. The implementation is similar to that of Sarsa and is omitted here. Interested readers may check [3, Chapter 7] for a detailed analysis of multi-step TD learning.

7.4 TD learning of optimal action values: Q-learning

In this section, we introduce the Q-learning algorithm, one of the most classic reinforcement learning algorithms [38,39]. Recall that Sarsa can only estimate the action values of a given policy, and it must be combined with a policy improvement step to find optimal policies. By contrast, Q-learning can directly estimate optimal action values and find optimal policies.

7.4.1 Algorithm description

The Q-learning algorithm is

$$\begin{aligned} q_{t+1}(s_t,a_t) &= q_t(s_t,a_t) - \alpha_t(s_t,a_t)\left[q_t(s_t,a_t) - \big(r_{t+1} + \gamma \max_{a\in\mathcal{A}(s_{t+1})} q_t(s_{t+1},a)\big)\right], \qquad (7.18) \\ q_{t+1}(s,a) &= q_t(s,a), \quad \text{for all } (s,a)\neq(s_t,a_t), \end{aligned}$$

where $t = 0, 1, 2, \ldots$. Here, $q_t(s_t, a_t)$ is the estimate of the *optimal* action value of (s_t, a_t) and $\alpha_t(s_t, a_t)$ is the learning rate for (s_t, a_t).

The expression of Q-learning is similar to that of Sarsa. They are different only in terms of their TD targets: the TD target of Q-learning is $r_{t+1} + \gamma \max_a q_t(s_{t+1}, a)$, whereas that of Sarsa is $r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1})$. Moreover, given (s_t, a_t), Sarsa requires $(r_{t+1}, s_{t+1}, a_{t+1})$ in every iteration, whereas Q-learning merely requires (r_{t+1}, s_{t+1}).

Why is Q-learning designed as the expression in (7.18), and what does it do mathematically? Q-learning is a stochastic approximation algorithm for solving the following equation:

$$q(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_a q(S_{t+1},a)\middle| S_t = s, A_t = a\right]. \qquad (7.19)$$

This is the Bellman optimality equation expressed in terms of action values. The proof is given in Box 7.5. The convergence analysis of Q-learning is similar to Theorem 7.1 and omitted here. More information can be found in [32, 39].

Box 7.5: Showing that (7.19) is the Bellman optimality equation

By the definition of expectation, (7.19) 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 the maximum of both sides of the equation gives

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

By denoting $v(s) \doteq \max_{a\in\mathcal{A}(s)} q(s,a)$, we can rewrite the above equation as

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

which is clearly the Bellman optimality equation in terms of state values as introduced in Chapter 3.

7.4.2 Off-policy vs on-policy

We next introduce two important concepts: *on-policy learning* and *off-policy learning.* What makes Q-learning slightly special compared to the other TD algorithms is that Q-learning is off-policy while the others are on-policy.

Two policies exist in any reinforcement learning task: a *behavior policy* and a *target policy.* The behavior policy is the one used to generate experience samples. The target policy is the one that is constantly updated to converge to an optimal policy. When the behavior policy is the same as the target policy, such a learning process is called *on-policy.* Otherwise, when they are different, the learning process is called *off-policy.*

The advantage of off-policy learning is that it can learn optimal policies based on the experience samples generated by other policies, which may be, for example, a policy executed by a human operator. As an important 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. Although Sarsa uses ϵ-greedy policies to maintain certain exploration abilities, the value of ϵ is usually small and hence the exploration ability is limited. By contrast, if we can use a policy with a strong exploration ability to generate episodes and then use off-policy learning to learn optimal policies, the learning efficiency would be significantly increased.

To determine if an algorithm is on-policy or off-policy, we can examine two aspects. The first is the mathematical problem that the algorithm aims to solve. The second is the experience samples required by the algorithm.

◇ Sarsa is on-policy.

The reason is as follows. Sarsa has two steps in every iteration. The first step is to evaluate a policy π by solving its Bellman equation. To do that, we need samples generated by π. Therefore, π is the behavior policy. The second step is to obtain an improved policy based on the estimated values of π. As a result, π is the target policy that is constantly updated and eventually converges to an optimal policy. Therefore, the behavior policy and the target policy are the same.

From another point of view, we can examine the samples required by the algorithm. The samples required by Sarsa in every iteration include $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. How these samples are generated is illustrated below:

$$s_t \xrightarrow{\pi_b} a_t \xrightarrow{\text{model}} r_{t+1}, s_{t+1} \xrightarrow{\pi_b} a_{t+1}$$

As can be seen, the behavior policy π_b is the one that generates a_t at s_t and a_{t+1} at s_{t+1}. The Sarsa algorithm aims to estimate the *action value* of (s_t, a_t) of a policy denoted as π_T, which is the target policy because it is improved in every iteration based on the estimated values. In fact, π_T is the same as π_b because the evaluation of π_T relies on the samples $(r_{t+1}, s_{t+1}, a_{t+1})$, where a_{t+1} is generated following π_b. In other words, the policy that Sarsa evaluates is the policy used to generate samples.

◇ Q-learning is off-policy.

The fundamental reason is that Q-learning is an algorithm for solving the *Bellman optimality equation*, whereas Sarsa is for solving the *Bellman equation* of a given policy. While solving the Bellman equation can evaluate the associated policy, solving the Bellman optimality equation can directly generate the optimal values and optimal policies.

In particular, the samples required by Q-learning in every iteration is $(s_t, a_t, r_{t+1}, s_{t+1})$. How these samples are generated is illustrated below:

$$s_t \xrightarrow{\pi_b} a_t \xrightarrow{\text{model}} r_{t+1}, s_{t+1}$$

As can be seen, the behavior policy π_b is the one that generates a_t at s_t. The Q-learning algorithm aims to estimate the *optimal action value* of (s_t, a_t). This estimation process relies on the samples (r_{t+1}, s_{t+1}). The process of generating (r_{t+1}, s_{t+1}) does not involve π_b because it is governed by the system model (or by interacting with the environment). Therefore, the estimation of the optimal action value of (s_t, a_t) does not involve π_b and we can use any π_b to generate a_t at s_t. Moreover, the target policy π_T here is the greedy policy obtained based on the estimated optimal values (Algorithm 7.3). The behavior policy does not have to be the same as π_T.

◇ MC learning is on-policy. The reason is similar to that of Sarsa. The target policy to be evaluated and improved is the same as the behavior policy that generates samples.

Another concept that may be confused with on-policy/off-policy is *online/offline*. Online learning refers to the case where the agent updates the values and policies while interacting with the environment. Offline learning refers to the case where the agent updates the values and policies using pre-collected experience data without interacting with the environment. If an algorithm is on-policy, then it can be implemented in an online fashion, but cannot use pre-collected data generated by other policies. If an algorithm is off-policy, then it can be implemented in either an online or offline fashion.

Algorithm 7.2: Optimal policy learning via Q-learning (on-policy version)

Initialization: $\alpha_t(s,a) = \alpha > 0$ for all (s,a) and all t. $\epsilon \in (0,1)$. Initial $q_0(s,a)$ for all (s,a). Initial ϵ-greedy policy π_0 derived from q_0.
Goal: Learn an optimal path that can lead the agent to the target state from an initial state s_0.

```
For each episode, do
    If $s_t$ ($t = 0,1,2,\dots$) is not the target state, do
        Collect the experience sample $(a_t, r_{t+1}, s_{t+1})$ given $s_t$: generate $a_t$ following $\pi_t(s_t)$; generate $r_{t+1}, s_{t+1}$ by interacting with the environment.
        *Update q-value for* $(s_t, a_t)$:
            $q_{t+1}(s_t,a_t) = q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q_t(s_t,a_t) - (r_{t+1} + \gamma \max_a q_t(s_{t+1},a))\Big]$
        *Update policy for* $s_t$:
            $\pi_{t+1}(a|s_t) = 1 - \frac{\epsilon}{|\mathcal{A}(s_t)|}(|\mathcal{A}(s_t)| - 1)$ if $a = \arg\max_a q_{t+1}(s_t,a)$
            $\pi_{t+1}(a|s_t) = \frac{\epsilon}{|\mathcal{A}(s_t)|}$ otherwise
```

Algorithm 7.3: Optimal policy learning via Q-learning (off-policy version)

Initialization: Initial guess $q_0(s,a)$ for all (s,a). Behavior policy $\pi_b(a|s)$ for all (s,a). $\alpha_t(s,a) = \alpha > 0$ for all (s,a) and all t.
Goal: Learn an optimal target policy π_T for all states from the experience samples generated by π_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 for* $(s_t, a_t)$:
            $q_{t+1}(s_t,a_t) = q_t(s_t,a_t) - \alpha_t(s_t,a_t)\Big[q(s_t,a_t) - (r_{t+1} + \gamma \max_a q_t(s_{t+1},a))\Big]$
        *Update target policy for* $s_t$:
            $\pi_{T,t+1}(a|s_t) = 1$ if $a = \arg\max_a q_{t+1}(s_t,a)$
            $\pi_{T,t+1}(a|s_t) = 0$ otherwise
```

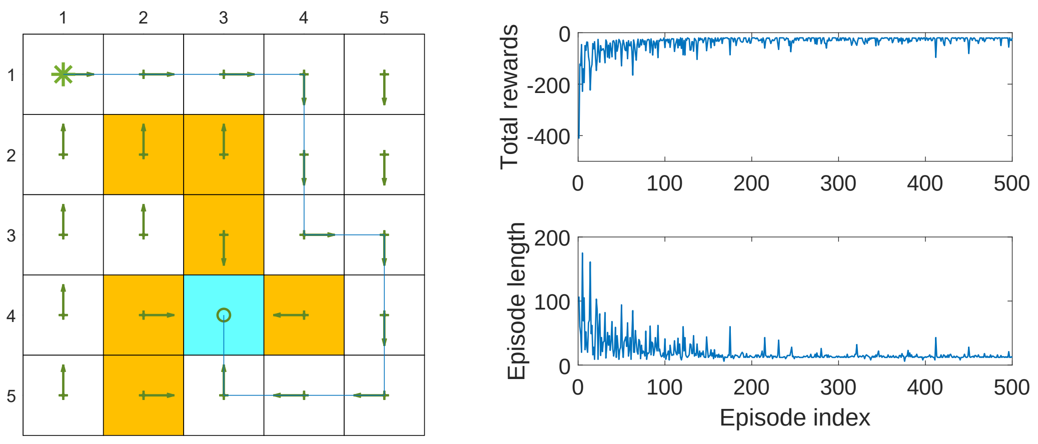


Figure 7.3: An example for demonstrating Q-learning. All the episodes start from the top-left state and terminate after reaching the target state. The aim is to find an optimal path from the starting state to the target state. The reward settings are $r_{\text{target}} = 0$, $r_{\text{forbidden}} = r_{\text{boundary}} = -10$, and $r_{\text{other}} = -1$. The learning rate is $\alpha = 0.1$ and the value of ϵ is 0.1. The left figure shows the final policy obtained by the algorithm. The right figure shows the total reward and length of every episode.

7.4.3 Implementation

Since Q-learning is off-policy, it can be implemented in either an on-policy or off-policy fashion.

The on-policy version of Q-learning is shown in Algorithm 7.2. This implementation is similar to the Sarsa one in Algorithm 7.1. Here, the behavior policy is the same as the target policy, which is an ϵ-greedy policy.

The off-policy version is shown in Algorithm 7.3. The behavior policy π_b can be any policy as long as it can generate sufficient experience samples. It is usually favorable when π_b is exploratory. Here, the target policy π_T is greedy rather than ϵ-greedy since it is not used to generate samples and hence is not required to be exploratory. Moreover, the off-policy version of Q-learning presented here is implemented offline: all the experience samples are collected first and then processed.

7.4.4 Illustrative examples

We next present examples to demonstrate Q-learning.

The first example is shown in Figure 7.3. It demonstrates on-policy Q-learning. The *goal* here is to find an optimal path from a starting state to the target state. The setup is given in the caption of Figure 7.3. As can be seen, Q-learning can eventually find an optimal path. During the learning process, the length of each episode decreases, whereas the total reward of each episode increases.

The second set of examples is shown in Figure 7.4 and Figure 7.5. They demonstrate off-policy Q-learning. The *goal* here 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 $\alpha = 0.1$.

◇ *Ground truth:* To verify the effectiveness of Q-learning, we first need to know the ground truth of the optimal policies and optimal state values. Here, the ground truth is obtained by the model-based policy iteration algorithm. The ground truth is given in Figures 7.4(a) and (b).

◇ *Experience samples:* The behavior policy has a uniform distribution: the probability of taking any action at any state is 0.2 (Figure 7.4(c)). A single episode with 100,000 steps is generated (Figure 7.4(d)). Due to the good exploration ability of the behavior policy, the episode visits every state-action pair many times.

◇ *Learned results:* Based on the episode generated by the behavior policy, the final target policy learned by Q-learning is shown in Figure 7.4(e). This policy is optimal because the estimated state value error (root-mean-square error) converges to zero as shown in Figure 7.4(f). In addition, one may notice that the learned optimal policy is not exactly the same as that in Figure 7.4(a). In fact, there exist multiple optimal policies that have the same optimal state values.

◇ *Different initial values:* Since Q-learning bootstraps, the performance of the algorithm depends on the initial guess for the action values. As shown in Figure 7.4(g), when the initial guess is close to the true value, the estimate converges within approximately 10,000 steps. Otherwise, the convergence requires more steps (Figure 7.4(h)). Nevertheless, these figures demonstrate that Q-learning can still converge rapidly even though the initial value is not accurate.

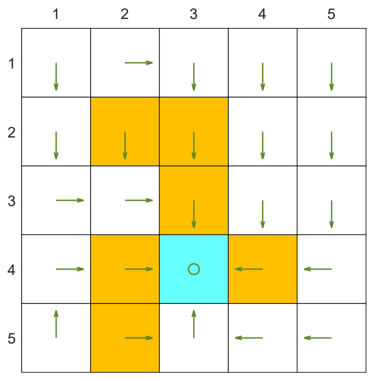
◇ *Different behavior policies:* When the behavior policy is not exploratory, the learning performance drops significantly. For example, consider the behavior policies shown in Figure 7.5. They are ϵ-greedy policies with $\epsilon = 0.5$ or 0.1 (the uniform policy in Figure 7.4(c) can be viewed as ϵ-greedy with $\epsilon = 1$). It is shown that, when ϵ decreases from 1 to 0.5 and then to 0.1, the learning speed drops significantly. That is because the exploration ability of the policy is weak and hence the experience samples are insufficient.

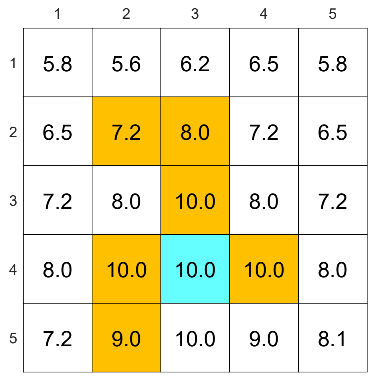
7.5 A unified viewpoint

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

In particular, the TD algorithms (for action value estimation) can be expressed in a unified expression:

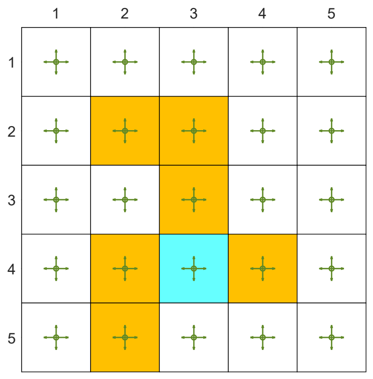
$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t)[q_t(s_t, a_t) - \bar{q}_t], \tag{7.20}$$

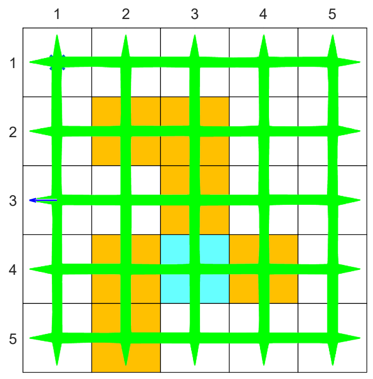

(a) Optimal policy

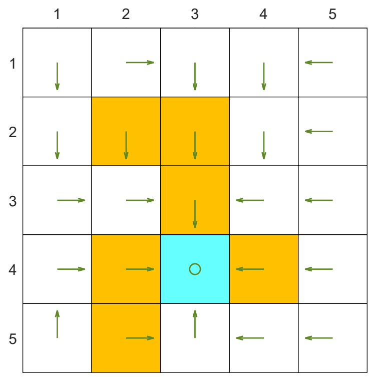


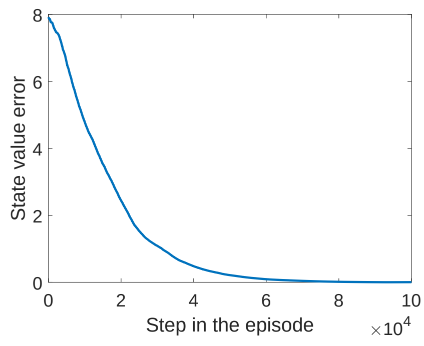
	1	2	3	4	5
1	5.8	5.6	6.2	6.5	5.8
2	6.5	7.2	8.0	7.2	6.5
3	7.2	8.0	10.0	8.0	7.2
4	8.0	10.0	10.0	10.0	8.0
5	7.2	9.0	10.0	9.0	8.1

(b) Optimal state value

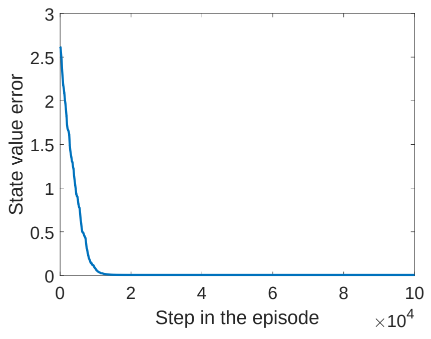

(c) Behavior policy


(d) Generated episode

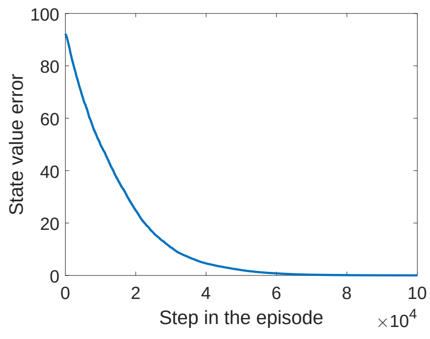

(e) Learned policy



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

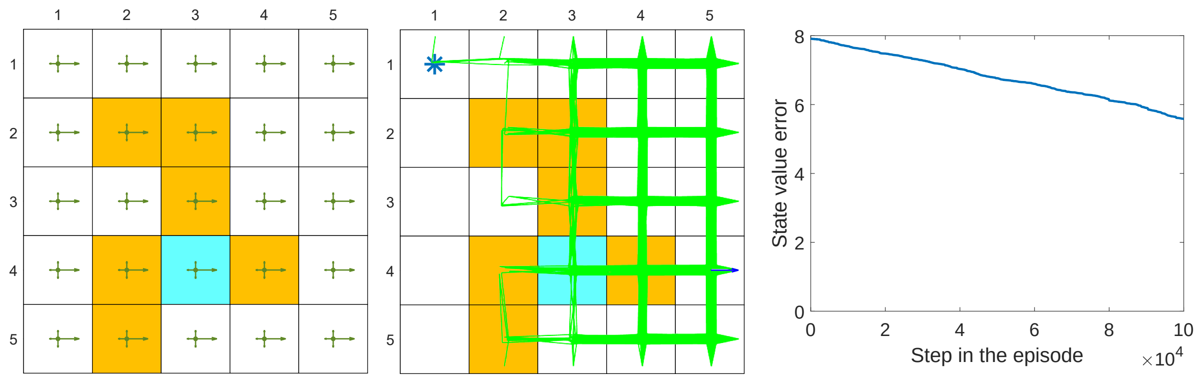


(g) State value error when $q_0(s,a) = 10$

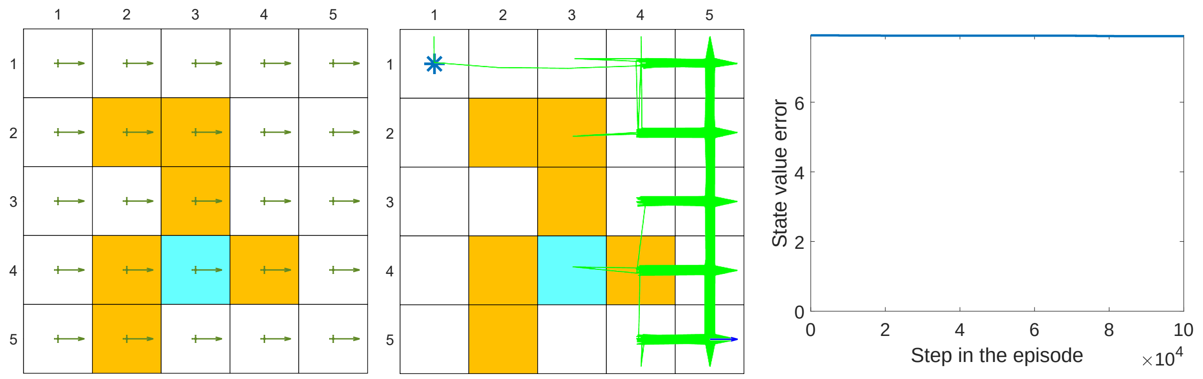


(h) State value error when $q_0(s,a) = 100$

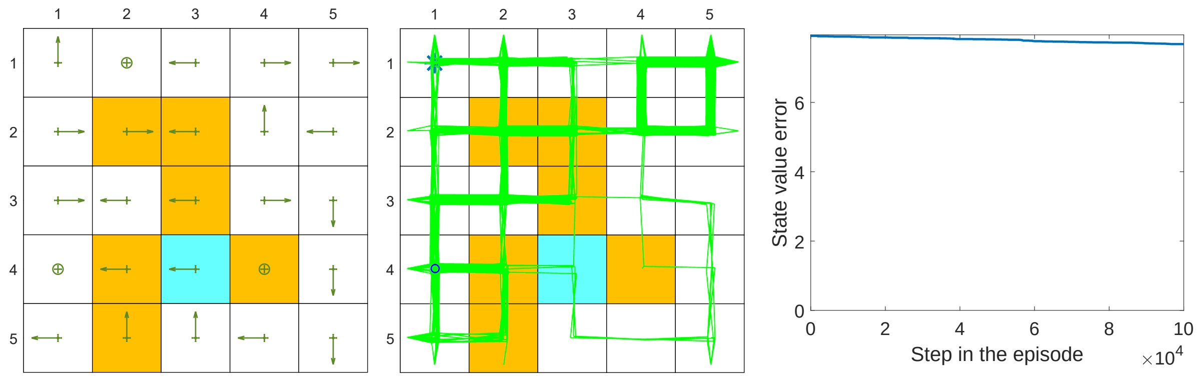
Figure 7.4: Examples for demonstrating off-policy learning via Q-learning. The optimal policy and optimal state values are shown in (a) and (b), respectively. The behavior policy and the generated episode are shown in (c) and (d), respectively. The estimated policy and the estimation error evolution are shown in (e) and (f), respectively. The cases with different initial values are shown in (g) and (h).



(a) $\epsilon = 0.5$



(b) $\epsilon = 0.1$



(c) $\epsilon = 0.1$

Figure 7.5: The performance of Q-learning drops when the behavior policy is not exploratory. 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 each example has 100,000 steps. The figures in the right column show the evolution of the root-mean-square error of the estimated state values.

Algorithm	**Expression of the TD target $\bar{q}_t$ in (7.20)**
Sarsa	$\bar{q}_t = r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1})$
n-step Sarsa	$\bar{q}_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^n q_t(s_{t+n}, a_{t+n})$
Q-learning	$\bar{q}_t = r_{t+1} + \gamma \max_a q_t(s_{t+1}, a)$
Monte Carlo	$\bar{q}_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots$

Algorithm	**Equation to be solved**
Sarsa	BE: $q_\pi(s,a) = \mathbb{E}\left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \vert S_t = s, A_t = a\right]$
n-step Sarsa	BE: $q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^n q_\pi(S_{t+n}, A_{t+n}) \vert S_t = s, A_t = a]$
Q-learning	BOE: $q(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_a q(S_{t+1}, a) \big\vert S_t = s, A_t = a\right]$
Monte Carlo	BE: $q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \vert S_t = s, A_t = a]$

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

where $\bar{q}_t$ is the *TD target*. Different TD algorithms have different $\bar{q}_t$. See Table 7.2 for a summary. The MC learning algorithm can be viewed as a special case of (7.20): we can set $\alpha_t(s_t, a_t) = 1$ and then (7.20) becomes $q_{t+1}(s_t, a_t) = \bar{q}_t$.

Algorithm (7.20) can be viewed as a stochastic approximation algorithm for solving a unified equation: $q(s,a) = \mathbb{E}[\bar{q}_t | s, a]$. This equation has different expressions with different $\bar{q}_t$. These expressions are summarized in Table 7.2. As can be seen, all of the algorithms aim to solve the Bellman equation except Q-learning, which aims to solve the Bellman optimality equation.

7.6 Summary

This chapter introduced an important class of reinforcement learning algorithms called TD learning. The specific algorithms that we introduced include Sarsa, n-step Sarsa, and Q-learning. All these algorithms can be viewed as stochastic approximation algorithms for solving Bellman or Bellman optimality equations.

The TD algorithms introduced in this chapter, except Q-learning, are used to evaluate a given policy. That is to estimate a given policy's state/action values from some experience samples. Together with policy improvement, they can be used to learn optimal policies. Moreover, these algorithms are on-policy: the target policy is used as the behavior policy to generate experience samples.

Q-learning is slightly special compared to the other TD algorithms in the sense that it is off-policy. The target policy can be different from the behavior policy in Q-learning. The fundamental reason why Q-learning is off-policy is that Q-learning aims to solve the Bellman optimality equation rather than the Bellman equation of a given policy.

It is worth mentioning that there are some methods that can convert an on-policy algorithm to be off-policy. Importance sampling is a widely used one [3, 40] and will be introduced in Chapter 10. Finally, there are some variants and extensions of the TD algorithms introduced in this chapter [41–45]. For example, the TD(λ) method provides a more general and unified framework for TD learning. More information can be found in [3, 20, 46].

7.7 Q&A

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

A: Every TD algorithm has a TD error, which represents the discrepancy between the new sample and the current estimate. Since this discrepancy is calculated between different time steps, it is called temporal-difference.

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

A: From a mathematical point of view, "learning" simply means "estimation". That is to estimate state/action values from some samples and then obtain policies based on the estimated values.

◇ Q: While Sarsa can estimate the action values of a given policy, how can it be used to learn optimal policies?

A: To obtain an optimal policy, the value estimation process should interact with the policy improvement process. That is, after a value is updated, the corresponding policy should be updated. Then, the updated policy generates new samples that can be used to estimate values again. This is the idea of generalized policy iteration.

◇ Q: Why does Sarsa update policies to be ϵ-greedy?

A: That is because the policy is also used to generate samples for value estimation. Hence, it should be exploratory to generate sufficient experience samples.

◇ Q: While Theorems 7.1 and 7.2 require that the learning rate α_t converges to zero gradually, why is it often set to be a small constant in practice?

A: The fundamental reason is that the policy to be evaluated keeps changing (or called nonstationary). In particular, a TD learning algorithm like Sarsa aims to estimate the action values of a given policy. If the policy is fixed, using a decaying learning rate is acceptable. However, in the optimal policy learning process, the policy that Sarsa aims to evaluate keeps *changing* after every iteration. We need a constant learning rate in this case; otherwise, a decaying learning rate may be too small to effectively evaluate policies. Although a drawback of constant learning rates is that the value estimate may fluctuate eventually, the fluctuation is neglectable as long as the constant learning rate is sufficiently small.

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

  A: It depends on the task. One may notice that some tasks considered in this chapter (e.g., Figure 7.2) do *not* require finding the optimal policies for all states. Instead, they only need to find an optimal path from a given starting state to the target state. Such tasks are not demanding in terms of data because the agent does not need to visit every state-action pair sufficiently many times. It, however, must be noted that the obtained path is not guaranteed to be optimal. That is because better paths may be missed if not all state-action pairs are well explored. Nevertheless, given sufficient data, we can still find a good or locally optimal path.

- ◇ Q: Why is Q-learning off-policy while all the other TD algorithms in this chapter are on-policy?

  A: The fundamental reason is that Q-learning aims to solve the Bellman optimality equation, whereas the other TD algorithms aim to solve the Bellman equation of a given policy. Details can be found in Section 7.4.2.

- ◇ Q: Why does the off-policy version of Q-learning update policies to be greedy instead of ϵ-greedy?

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