

Temporal Difference Learning

Why wait for the end when you can learn from the now?

4.1 TEMPORAL DIFFERENCE LEARNING

Temporal Difference (TD) learning, introduced in 1988 [1], is a fundamental concept in Reinforcement Learning (RL) that incorporates ideas from dynamic programming. Monte Carlo methods, described in Chapter 5, require waiting until the end of an episode to update value estimates. In contrast, TD learning updates value estimates after each step, allowing for online and incremental learning. A step refers to a single agent-environment interaction. The simplest TD learning algorithm is **one-step TD**,¹ which updates the value of the current state $V(s)$ using the immediate reward r and the estimated value of the next state $V(s')$. One-step TD is the focus of this chapter.

Q-learning, one of several TD algorithms, was first introduced in a 1989 thesis [2], and later presented in a follow-up paper [3]. SARSA is another TD algorithm, described in this chapter as well. Although simpler to implement than Q-learning, it was developed later and first published in a technical report from 1994 [4]. Q-learning and SARSA will be explained in subsequent sections. Both algorithms have their own advantages and drawbacks, making it an open question which is the better choice for a specific problem. Their differences, along with guidelines for selecting between them, are highlighted in Worked example 4.7 and Appendix A.

4.2 Q-LEARNING

Q-learning is an off-policy algorithm. The concepts of on-policy and off-policy were introduced in Chapter 3. The algorithm updates the Q-value based on the assumption that the agent will act optimally in the future. The update rule uses the maximum Q-value of the next state, thereby assessing the best possible action. Optimal action selection is demonstrated in Worked example 4.1. This separation of action selection allows Q-learning to learn a policy independently of the actions taken during exploration, which is why it is considered off-policy. Alternatively, this can be phrased as the

target policy being different from the behavior policy. Worked example 4.2 is about behavior and target policy.

We begin by defining the key mathematical relations used in Q-learning. Eq. (4.1) specifies how the agent updates the value of a given state-action pair in its memory. The core idea is to update the Q-function for state s and action a by adding $\alpha \cdot \delta$. The learning rate α , introduced in Chapter 2, determines how malleable an agent is (how easily its memory is updated by new experiences). The term δ is the **TD error**, which indicates how the Q-value should be adjusted to better reflect the target value. The target value for each state-action pair is defined by Eqs. (4.2)–(4.4). The use of these equations is demonstrated in Worked example 4.5.

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot \delta \quad \boxed{1} \quad (4.1)$$

where the TD error δ is defined by the equations below.

$$\delta = q_{tar} - Q(s, a) \quad \boxed{1} \quad (4.2)$$

$$q_{tar} = \begin{cases} r + \gamma \cdot V(s') & (s' \text{ is not terminal}) \\ r & (else) \end{cases} \quad \boxed{1} \quad (4.3)$$

$$V(s') = \max_{a'} Q(s', a') \quad \boxed{1} \quad (4.4)$$

The idea of changing a function toward desired values was introduced in Chapter 2, where the function was called $f(x)$. The underlying concept remains the same: incrementally improving a function estimate based on observed data. However, there is a key difference between the function fitting in Chapter 2 and that in Q-learning: Q-learning is recursive. This recursive nature is highlighted in Worked example 4.4.

As outlined in the skeleton RL pseudocode in Chapter 3 (Section 3.10), the training process consists of an outer and an inner loop. The **training loop** manages the overall training process, iterating through multiple episodes to refine the policy. In contrast, the inner **episode loop** focuses on executing individual episodes, where the agent interacts with the environment. Table 4.1 presents the pseudocode for the Q-learning algorithm. Broadly speaking, the agent's memory Q is updated over a finite number of episodes, with the agent improving its policy during each episode.

The action selection takes place at line 5. The phrase “ ϵ -greedy policy” means that the action is selected according to:

$$a = \begin{cases} a_{random} & (p < \epsilon) \\ \operatorname{argmax}_a Q(s, a) & (else) \end{cases} \quad \boxed{1} \quad (4.5)$$

TABLE 4.1 Q-Learning Algorithm

1	initialize Q	Setup action-value function
2	while trainingCriterion	Training loop
3	$s \leftarrow \text{getStartState}()$	Get start state
4	while episodeCriterion	Episode loop
5	$a \leftarrow \text{chooseAction}(s)$	ϵ -greedy policy
6	$s', r \leftarrow \text{step}(s, a)$	Apply action a in state s
7	$q_{\text{Tar}} \leftarrow r + \gamma \cdot \max_{a'} Q(s', a')$	Target state-action value
8	$\delta \leftarrow q_{\text{Tar}} - Q(s, a)$	TD error
9	$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot \delta$	Update Q at s and a
10	$s \leftarrow s'$	Iterate to next state
11	end while	Episode loop ending
12	end while	Training loop ending

Note: The training loop (line 2) typically specifies the number of episodes to run. The Episode loop usually terminates when a terminal state is reached (criterion at line 4). For simplicity, the logic that prevents bootstrapping from terminal states is omitted (line 7).

This means that, with probability ϵ , the action is random; otherwise, it is selected according to the current policy. The current policy in Q-learning is to choose the action that yields the highest Q-value in state s . The reason for selecting a random action is to encourage the agent to be exploratory. Being exploratory increases the chance of visiting states that have not yet been explored. The action selection described in Eq. (4.5) is exemplified in Worked example 4.1.

Line 6 in Table 4.1 represents the agent-environment interaction. An agent's proposed action a is applied to the environment. The environment responds with a next state s' and a reward r . After line 6 has been executed, all mandatory information in an experience is present. In Chapter 3, it was stated that s, a, r, s' represents the minimal information needed in an experience. At lines 7–9, the information in an experience, in combination with Eqs. (4.2)–(4.4), is used to update the agent's memory Q .

What about the experience concept introduced in Chapter 2? Why is it not included in Table 4.1? Why learn about a concept and then not use it? It is a matter of prioritization. When the code in Table 4.1 was formulated, simplicity was prioritized. An experience object would not contribute significantly to explaining the logic of the code. However, when implementing the code in “whatever-your-favorite-language-is”, it is highly advisable to use an experience object. Think about the episode termination criteria, for example, the *is-terminal* flag in the most recent experience would be very handy.

Worked example 4.5 shows the collection of experienced state-actions when Q-learning is applied. After episode 3, all state-action combinations have been explored, and initial Q-values are established (Worked examples 4.1–4.5).

WORKED EXAMPLE 4.1

Demonstrate ϵ -greedy action selection in Q-learning.

Assume a $Q(s, a)$ function with values as shown in the table below. It includes two actions L, and R, and two states, 0 and 1.

$s \backslash a$	L	R
0	1	2
1	2	1

Approach: Apply Eq. (4.5)

Case	ϵ	s	Random Action	a
1	1	0	Yes	L or R with equal probability
2	0	0	No	R
3	0	1	No	L

Comment: In case 2 and 3, a is the action maximizing Q in s .

WORKED EXAMPLE 4.2

Explain target policy and behavior policy.

These concepts were introduced in Chapter 3 but are repeated here for convenience. In RL, the behavior policy selects actions during learning, while the target policy is the one being learned or improved. In on-policy methods, both policies are the same; in off-policy methods, they differ.

WORKED EXAMPLE 4.3

Explain the calculation of δ , the TD error used in Q-learning.

Eq. (4.4) expresses an optimization problem. The $Q(s, a)$ function is evaluated for all possible actions. For a specific state, the best action maximizes Q . The motivation behind Eq. (4.3) is the Bellman optimality principle introduced in Chapter 2. The agent assumes that it will take the best action in future, when entering the next s' . Updating Q at the current state-action value from this future state-value estimate is called bootstrapping. This technique, where a function recursively “uses itself” for updating, has already been presented in the book. In Chapter 3, iterative policy evaluation and value iteration were presented. Both these techniques rely on bootstrapping, which means that they update the value of state s using estimates of the values of the subsequent state s' .

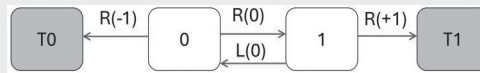
WORKED EXAMPLE 4.4

How does fitting the sigmoid function in Chapter 2 differ from function fitting in Q-learning?

When fitting the sigmoid function, the reference values are provided by an analytical expression, making it a form of supervised learning. In contrast, Q-learning is recursive: the target Q -value is not given externally but depends on the Q -function itself, evaluated at a different input. This makes Q-learning a form of bootstrapping, where the function is used to improve its own estimate over time.

WORKED EXAMPLE 4.5

Give an example of how the Q memory might change for a few episodes. Assume Q-learning and a Q-memory initialized with only zeros. In addition, the learning rate α is 0.1 and the discount factor γ is 1. Transitions are defined as shown in the figure and table below. As in Worked example 4.1, there are two actions L, and R, and two states, 0 and 1.



$s \backslash a$	L (Left)	R (Right)
0	$r = -1$, moving to terminal state	$r = 0$, moving to state 1
1	$r = 0$, moving to state 0	$r = +1$, moving to terminal state

Approach: apply Q-learning as described in Table 4.1

The tables below illustrate the execution of three episodes. The term “New Q” is the Q function after being updated according to Eq. (4.1), it has the structure:

$Q(s=0, a=L)$	$Q(s=0, a=R)$
$Q(s=1, a=L)$	$Q(s=1, a=R)$

Consequently, the columns represent actions and the rows represent states. For convenience, the definition of Q_{tar} is repeated: $Q_{tar} = \begin{cases} r + \gamma \cdot \max_a Q(s', a) & (s' \text{ is not terminal}) \\ r & (\text{else}) \end{cases}$

—First episode—

“Ep.” denotes the episode number, “T” is a terminal state, and “NA” stands for Not Applicable.

Ep.	Step	s	Random Action	a	r	s'	$V(s')$	Q_{tar}	δ	New Q
0	0	0	Yes	L	-1	T	NA	-1	$-1 - Q(0, L) = -1$	-0.1 0

Comment: Entered terminal state. No next state s' , so no bootstrapping (therefore NA). After this episode, the value for $Q(s=0, a=L)$ is -0.1.

—Second episode—

Ep.	Step	s	Random Action	a	r	s'	$V(s')$	Q_{tar}	δ	New Q
1	0	0	No	R	0	1	0	$0 + \gamma \cdot 0 = 0$	$0 - Q(0, R) = 0$	-0.1 0
1	1	1	Yes	R	+1	T	NA	1	$1 - Q(1, R) = 1$	-0.1 0

Comment for step 0: All Q-values are zero, hence bootstraps from zero.

Comment for step 1: value for s' not defined (terminal state), hence only reward r affects Q_{tar} .

Comment: after the second episode the optimal policy is in place; it recommends to take action R for both states.

—Third episode—

Ep.	Step	s	Random Action	a	r	s'	$V(s')$	Q_{tar}	δ	New Q
2	0	1	Yes	L	0	0	0	$0 + \gamma \cdot 0 = 0$	$0 - Q(1, L) = 0$	-0.1 0
2	1	0	Yes	L	-1	T	NA	-1	$-1 - Q(1, L) = -0.9$	-0.19 0

Comment for step 1: the policy persists from episode 2, always take action R.

4.3 SARSA

SARSA is, in contrast to Q-learning, an on-policy algorithm. It updates the Q-value using the actual action taken in the next state, meaning it learns the value of the policy it is currently following. The update rule incorporates the next action selected by the current behavior policy, such as the ϵ -greedy policy. Since SARSA learns directly from the sequence s, a, r, s', a' , including the next action chosen by the same policy, it is classified as on-policy learning. In this case, the target policy is identical to the behavior policy.

Instead of using the action that maximizes Q at the next state s' , SARSA uses the action selected according to the current policy. This reflects its on-policy nature—the Q-value is updated based on the policy the agent is following, not an assumed optimal one. Mathematically, the update rule is:

$$q_{tar} = \begin{cases} r + \gamma \cdot Q(s', a') & (s' \text{ is not terminal}) \\ r & (else) \end{cases} \quad \boxed{1} \quad (4.6)$$

The pseudocode for SARSA is presented in Table 4.2. The only difference compared to Q-learning appears in line 8: there is no max operator, and the term Q_{tar} explicitly includes the next action a' . This ties directly to the concept of experience, introduced in Chapter 3. For SARSA, the complete experience consists of five elements: s, a, r, s', a' , the inclusion of a' is essential, as it represents the action selected by the current policy in the next state. This sequence of elements is what gives the algorithm its name: SARSA.

Worked example 4.6 applies the SARSA algorithm to a simple problem, the same as in Worked example 4.5. Worked example 4.7 discusses the difference between Q-learning and SARSA.

TABLE 4.2 SARSA-Learning Algorithm

1	initialize Q	Setup
2	while trainingCriterion	Training loop
3	$s \leftarrow \text{getStartState}()$	Initialize start state
4	$a \leftarrow \text{chooseAction}(s)$	Choose first action
5	while episodeCriterion	Episode loop
6	$s', r \leftarrow \text{step}(s, a)$	Apply action a in state s
7	$a' \leftarrow \text{chooseAction}(s')$	Choose action in new state s'
8	$q_{Tar} \leftarrow r + \gamma \cdot Q(s', a')$	Target state-action value.
9	$\delta \leftarrow q_{Tar} - Q(s, a)$	TD error
10	$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot \delta$	Update Q at s and a
11	$s \leftarrow s'$	Iterate to next state
12	$a \leftarrow a'$	Chosen action will be used in next step
13	end while	Episode loop ending
14	end while	Training loop ending

Note: The training and episode loop criteria are the same as for q-learning (see Table 4.1). For simplicity, the logic that prevents bootstrapping from terminal states is omitted (line 8).

WORKED EXAMPLE 4.6

Use the same setup as in Worked example 4.5, but apply the SARSA learning algorithm instead.

Approach: Apply SARSA as described in Table 4.2

The tables below illustrate the execution of two episodes. The term “New Q” is the Q function after being updated according to Eq. (4.6). It has the structure:

$Q(s=0, a=L)$	$Q(s=0, a=R)$
$Q(s=1, a=L)$	$Q(s=1, a=R)$

—First episode—

“Ep.” denotes the episode number and “NA” stands for Not Applicable.

Ep.	Step	s	a	r	s'	Random Action	a'	Q_{tar}	δ	New Q	
0	0	0	R	0	1	Yes	R	$0 + \gamma \cdot Q(1, R) = 0$	$0 - Q(0, R) = 0$	0	0
										0	0
0	1	1	R	+1	T	NA	NA	1	$1 - Q(1, R) = 1$	0	0
										0	0.1

Comment for step 0: All Q-values are zero, hence bootstraps from zero.

Comment for step 1: Entered terminal state. No next state s' , so no bootstrapping. After this step, the value for $Q(s=1, a=R)$ is updated to 0.1.

—Second episode—

Ep.	Step	s	a	r	s'	Random Action	a'	Q_{tar}	δ	New Q	
1	0	1	L	0	0	Yes	L	$0 + \gamma \cdot Q(0, L) = 0$	$0 - Q(1, L) = 0$	0	0
										0	0.1
1	1	0	L	-1	T	NA	NA	-1	$-1 - Q(0, L) = -1$	-0.1	0
										0	0.1

Comment: Q reflects the optimal policy; it recommends always taking action R.

WORKED EXAMPLE 4.7

Elaborate on the differences between Q-learning and SARSA.

It's difficult to say which algorithm is generally better—Q-learning or SARSA—as both have their strengths and weaknesses, but some trends can be identified. Q-learning uses the max operator to estimate future rewards, which can make it computationally demanding when the action space is large and can lead to overly optimistic estimates in stochastic environments. These optimistic estimates occur because Q-learning always picks the highest Q-value, which may be inflated due to random fluctuations early in learning. SARSA, in contrast, updates based on the actual action taken, avoiding max computations and resulting in more cautious and stable learning. However, Q-learning tends to perform better in deterministic environments, where its optimistic updates align with predictable outcomes and can accelerate convergence to the optimal policy.