Reinforcement Learning

Sunmook Choi felixchoi@korea.ac.kr

August 29, 2023

1 Introduction

We have learned a method, dynamic programming, to solve Markov decision processes. Markov decision processes provide a formal framework for modeling decision-making problems in which the optimal decision depends on the current state of the environment. Dynamic programming approximates optimal value functions and policies in iterative ways to find optimal decisions.

However, a huge assumption is required to use dynamic programming method. That is, the dynamics $p(s', r|s, a)$ must be known in advance. Since the dynamics define markov decision processes (MDPs) (a model of the environment), dynamic programming is called *model-based* programming. In most cases, however, the dynamics of an environment is not known. Therefore, to solve MDPs without the knowledge of the underlying MDPs, we instead use a method of 'learning', which is also called *model-free* learning. In this section, we will figure out reinforcement learning as a type of model-free learning.

Reinforcement learning (RL) is a method to solve MDPs without being known the complete information of an environment. Recall that we say MDPs are solved if an optimal value function is found. In dynamic programming method, an optimal value function v_* is estimated by V^* using iterative algorithms based on Bellman equations. With an optimal value function, an optimal policy is found by the following equation:

$$\pi^*(s) \approx \arg\max_a Q^*(s, a) \quad \text{and} \quad Q^*(s, a) = \sum_{s', r} p(s', r|s, a)[r + \gamma V^*(s')] \tag{1.1}$$

However, being unknown the dynamics $p(s', r|s, a)$ of an environment changes the whole algorithm. First, the exact Bellman equations cannot be implemented in such situations with incomplete information. Second, an optimal policy cannot be found from estimated $V^*(s)$ because it also requires the dynamics. Therefore, we should estimate optimal value functions or optimal policies in different perspectives.

As we do not know the complete information of an environment, RL methods only require

simulated experiences (samples or observations) from an environment to learn optimal policies. In the following methods, Monte Carlo method and Temporal Difference methods, an optimal action-value function is estimated from the samples. Such methods are under a principle of generalized policy iteration (GPI), which alternatively proceeds policy evaluation and policy improvement repeatedly.

Exploration and Exploitation Exploration and exploitation are two conflicting goals in reinforcement learning. Exploitation means taking the action that is currently believed to be the best based on the current knowledge. We have been discussed about exploitation by using greedy policy. On the other hand, exploration means taking a non-greedy action to gain more information about the environment. In other words, exploitation is the act of making the best decision based on current information, and exploration is the act of gathering more information to make better decisions in the future. A good balance between exploration and exploitation is required to find the optimal policy and value function. One of the most popular exploration strategies is the ϵ-greedy strategy, which selects the greedy action with probability $1 - \epsilon$ and selects a random action with probability ϵ for small $\epsilon \in (0, 1)$.

2 Monte Carlo Method

Monte Carlo method is a way of solving a given MDP, that is, finding the optimal action-value function of an MDP, based on averaging sample returns. To ensure the availability of well-defined returns, we define Monte Carlo methods only for episodic tasks. In the policy evaluation phase, the method samples experiences from an environment, and it averages returns to estimate the action-value function with the current policy. In the policy improvement phase, the method updates the policy based on the so far estimated action-value function.

One thing we should notice is that, in the policy evaluation phase, we try to estimate $Q^\pi(s, a)$ corresponding to a fixed policy π. It is necessary to estimate the action-value function $Q^\pi(s, a)$ instead of the state-value function $V^\pi(s)$ because we improve our policy π into π' as below:

$$\pi'(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V^\pi(s')] = \arg\max_a Q^\pi(s, a) \tag{2.1}$$

When we are using RL, we are in a situation without knowing the dynamics of given environment. Therefore, we cannot improve the policy by estimating state-value function $V^\pi(s)$. For this reason, action-value functions are estimated in various manners.

2.1 Policy Evaluation (Monte Carlo Prediction)

Recall the definition of the action-value function:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \,|\, S_t = s, A_t = a] \tag{2.2}$$

An obvious way to estimate $q_\pi(s,a)$ is to average the returns observed after visits to that state-action pair (s,a), following the idea of Monte Carlo methods. In particular, we will try to estimate $q_\pi(s,a)$ given a set of episodes obtained by following π and passing through (s,a). Each occurrence of state-action pair (s,a) in an episode is called a *visit* to (s,a). Possibly, the pair (s,a) may be visited more than one times in the same episode; let us call the first time it is visited in an episode the *first visit* to (s,a). The *first-visit MC method* estimates $q_\pi(s,a)$ as the average of the returns following first visits to (s,a), whereas the *every-visit MC method* averages the returns following all visits to (s,a). Practically, we can estimate $q_\pi(s,a)$ by following the steps below.

At the first (or every) time step t that the state-action pair (s,a) was visited in an episode,

1. increment count: $n(s,a) \leftarrow n(s,a) + 1$
2. increment total return: $S(s,a) \leftarrow S(s,a) + G_t$

By accumulating the count $n(s,a)$ and total return $S(s,a)$ for all episodes experienced, the action-value $q_\pi(s,a)$ is estimated by the mean return $Q(s,a) = \frac{S(s,a)}{n(s,a)}$.

Incremental Monte Carlo Updates Equivalently, we can use the moving average trick. That is, given a sequence $x_1, x_2, \ldots$, the partial mean μ_k can be computed incrementally:

$$\mu_k = \frac{1}{k}\sum_{i=1}^{k} x_i = \frac{1}{k}\Big(\sum_{i=1}^{k-1} x_i + x_j\Big) = \frac{1}{k}\Big((k-1)\mu_{k-1} + x_k\Big) = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1}).$$

With this idea, we can estimate $q_\pi(s,a)$ by updating $Q(s,a)$ incrementally after an episode. For each episode $S_0, A_0, R_1, S_1, \ldots S_{T-1}, A_{T-1}, R_T$ and for each state-action pair (S_t, A_t) with return G_t in the episode, we update $Q(s,a)$ as below:

$$\begin{aligned} n(S_t, A_t) &\leftarrow n(S_t, A_t) + 1 \\ Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \frac{1}{n(S_t, A_t)}\big[G_t - Q(S_t, A_t)\big] \end{aligned}$$

Constant-α Monte Carlo Updates In practice, instead of tracking the count parameter $n(s,a)$ for each pair, we may use a constant 'step-size' α instead of $\frac{1}{n(S_t,A_t)}$:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G_t - Q(S_t, A_t)].$$

The use of constant α would get a benefit during training. Along the learning process, the Q-table gets 'smarter', so recent samples should be more important than old ones. Because of the term $(1-\alpha)Q(S_t, A_t)$, old episodes will be exponentially forgotten.

2.2 Policy Improvement (Monte Carlo Control)

In Chapter 2, we have shown that the maximization of the action-value function over actions actually provides better policy using the policy improvement theorem. In this section, we define an ϵ-greedy policy, and we will show that it is better than the older one.

Algorithm 2.1 Monte Carlo method

Initialize $Q(s,a)$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, and $Q(terminal,\cdot) = 0$.

$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

$\pi \leftarrow$ arbitrarily ϵ-soft policy (non-empty probabilities)

for each episode **do**
  Generate an episode following (fixed) π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for $t = T-1, T-2, \ldots, 1, 0$ (each step in episode) **do**
    $G \leftarrow \gamma G + R_{t+1}$
    if (S_t, A_t) appears in $(S_0, A_0), \cdots, (S_{t-1}, A_{t-1})$ **then** ▷ First-visit MC Prediction
      break
    else
      Append G to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow$ **average**$(Returns(S_t, A_t))$ ▷ **average**: incremental or constant-α
    end if
  end for
  for each S_t in the episode **do** ▷ ϵ-greedy MC control
    $\pi(a|S_t) \leftarrow \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(S_t)|} & \text{if } a = \arg\max_a Q(S_t, a) \\ \frac{\epsilon}{|\mathcal{A}(S_t)|} & \text{otherwise} \end{cases}$
  end for
end for

Definition 1 (ϵ-greedy policy)**.** *For an action-value function q_π, the ϵ-greedy policy π' with respect to q_π is defined as below:*

$$\pi'(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \textit{if} \quad a = \arg\max_{a'} q_\pi(s, a'), \\ \frac{\epsilon}{|\mathcal{A}|} & \textit{otherwise} \end{cases}. \tag{2.3}$$

Proposition 2. *For any policy π, the ϵ-greedy policy π' with respect to q_π is always improved, i.e.,*

$$v_{\pi'}(s) \geq v_\pi(s)$$

for all $s \in \mathcal{S}$.

Proof. To show that $\pi' \geq \pi$, it suffices to show that $q_\pi(s, \pi') \geq v_\pi(s)$ for all $s \in \mathcal{S}$ by the policy

improvement theorem in Chapter 2. Recall that $q_\pi(s, \pi') = \sum_a \pi'(a|s) q_\pi(s, a)$. Let $m = |\mathcal{A}|$.

$$
\begin{aligned}
q_\pi(s, \pi'(s)) &= \sum_{a \in \mathcal{A}} \pi'(a|s) q_\pi(s, a) \\
&= \frac{\epsilon}{m} \sum_{a \in \mathcal{A}} q_\pi(s, a) + (1 - \epsilon) \max_{a'} q_\pi(s, a') \\
&\geq \frac{\epsilon}{m} \sum_{a \in \mathcal{A}} q_\pi(s, a) + (1 - \epsilon) \sum_{a \in \mathcal{A}} \frac{\pi(a|s) - \epsilon/m}{1 - \epsilon} q_\pi(s, a') \qquad (\because \textstyle\sum_a \frac{\pi(a|s) - \epsilon/m}{1-\epsilon} = 1) \\
&= \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a) \\
&= v_\pi(s)
\end{aligned}
$$

This completes the proof. □

3 Temporal Difference Method

Recall the update equation of constant-α Monte Carlo (MC) policy evaluation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \big[G_t - Q(S_t, A_t) \big]$$

The MC target is the return G_t, which could be computed after an episode is finished, that is, MC methods updates $Q(s, a)$ for the pair (s, a) that occurs in the trajectory of an episode. In particular, MC methods must wait until the end of the episode to update the table, which might be considered inefficient.

Temporal difference (TD) methods also update Q-table by obtaining samples from the environment. However, unlike MC methods, TD methods estimate the action-value function by using the estimate of the value function at the next time step to update the estimate of the value function at the current time step. This *bootstrapping method* allows us to update Q-table at each time step, which makes TD method more efficient. We describe two types of TD method, Sarsa and Q-learning. They both use bootstrapping, but the underlying theory of estimating Q-table is slightly different.

3.1 Sarsa

We first state the update equation of $Q(s, a)$:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \big[R_{t+1} + \gamma\, Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \big]$$

This update equation is indeed eligible to estimate $Q(s, a)$ because of Bellman expectation equation:

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

However, because $q_\pi(S_{t+1}, A_{t+1})$ is not known, we use the current estimate $Q(S_{t+1}, A_{t+1})$ instead. When its update is based on an existing estimate, we say that it is a *bootstrapping method*. Of course, the bootstrapping method makes the TD target biased. The algorithm of Sarsa is as follows:

Algorithm 3.1 Sarsa

Parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$.

for each episode **do**
    Initialize S
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    for each step in episode **do**
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha\,[\,R + \gamma\, Q(S', A') - Q(S, A)\,]$
        $S \leftarrow S'$; $A \leftarrow A'$
        if S is terminal **then**
            break
        end if
    end for
end for

Notice that the update rule uses every element of 5-tuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state-action pair to the next. This 5-tuple give rise to the name "SARSA" for the algorithm. Notice that the action A_{t+1} in the target $R_{t+1} + \gamma\, Q(S_{t+1}, A_{t+1})$ is chosen by the same policy π which also chose the action A_t (and we call it *on-policy method*).

The quantity $R_{t+1} + \gamma\, Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$ is a sort of error, measuring the difference between the estimated value $Q(S_t, A_t)$ and the better estimate $R_{t+1} + \gamma\, Q(S_{t+1}, A_{t+1})$. This quantity is called the TD error, and it arises in various forms throughout reinforcement learning.

Remark 1. *The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on Q. For example, one could use ϵ-greedy or ϵ-soft policies. Sarsa converges with probability 1 to an optimal policy and action-value function as long as all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (which can be arrange, for example, with ϵ-greedy policies by setting $\epsilon = 1/t$). (page 129, [1])*

3.2 Q-Learning

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning (Watkins, 1989), defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\,\big].$$

The update equation can be explained by Bellman optimality equation:

$$q_*(s, a) = \mathbb{E}\big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \,|\, S_t = s, A_t = a\big]$$

In this case, the Q-table directly approximates the optimal action-value function q_* independent of the policy being followed. Similar to Sarsa algorithm, since we have no access to the true optimal action-value function $q_*(S_{t+1}, a)$, the update equation uses the current estimate $Q(S_{t+1}, a)$ (bootstrapping), and hence, the TD target is again biased. The algorithm of Q-learning is as follows:

Algorithm 3.2 Q-learning

Parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$.

for each episode **do**
    Initialize S
    for each step in episode **do**
        Choose A from S using policy derived from Q (e.g., ϵ-greedy)
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha[\, R + \gamma \max_a Q(S', a) - Q(S, A)\,]$
        $S \leftarrow S'$
        if S is terminal **then**
            break
        end if
    end for
end for

In Q-learning, the TD target is $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$, where the chosen action a is the one that maximizes Q in the next state. The action a is different from the action that would be chosen by the policy that chooses the current action A_t (*off-policy method*). We describe the difference of on-policy and off-policy in the next subsection.

3.3 On-Policy v.s. Off-Policy

We first define two policies: target policy and behavior policy. The *target policy* is the policy that an agent is trying to learn. The *behavior policy* is the policy that is being used by an agent for choosing actions and generating experiences. We call an algorithm an *on-policy method* if the target policy is equal to the behavior policy of the algorithm. On the other hand, we call an algorithm an *off-policy method* if the target policy and the behavior policy are different.

As I mentioned before, Sarsa is an on-policy method while Q-learning is an off-policy method. Let's see more details. The update equations of Sarsa and Q-learning are as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[\, R_{t+1} + \gamma\, Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\,\big] \qquad \text{(Sarsa)}$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[\, R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\,\big] \qquad \text{(Q-learning)}$$

The behavior policy is the policy that is used by an agent so that it makes experiences from the environment. From the state S_t, the behavior policy chooses the action A_t, the agent takes the action A_t in the environment, and the environment brings the agent to the next state S_{t+1}.

Now we consider the target of the update equations. The target of $Q(S_t, A_t)$ in Sarsa is $R_{t+1} + \gamma\, Q(S_{t+1}, A_{t+1})$. Conceptually, we say that the action A_{t+1} is chosen by the *target policy*. However, the action A_{t+1} is actually chosen by the behavior policy of the agent. Therefore, it makes Sarsa algorithm an on-policy method. Likewise, the target of $Q(S_t, A_t)$ in Q-learning is $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$. We can see that the target policy of Q-learning is

$$\pi(s) = \arg\max_a Q(s, a),$$

which is different from the behavior policy of Q-learning. Hence, we call Q-learning algorithm an off-policy method. The difference in on-policy and off-policy methods makes the behavior of an optimal policy obtained from Sarsa and Q-learning different. We will discuss about the difference in the next section.

3.4 Sarsa vs. Q-learning

In this section, we will focus on the difference of Sarsa and Q-learning from the following example: Cliff Walking. The grid world is shown in Figure 3.1. This environment is introduced in Example 6.6 of [1]. This experiment can be also found in my GitHub[1].

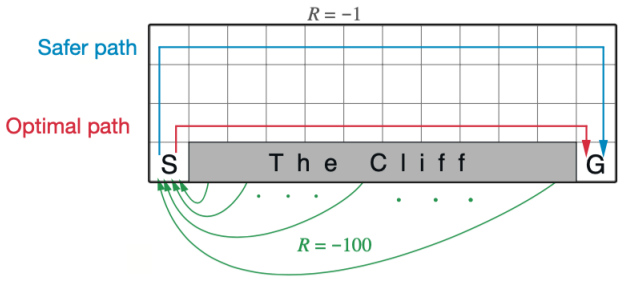

Figure 3.1: Cliff Walking Environment

This is a standard undiscounted, episodic task, with fixed start and goal states. The possible actions are up, down, right, and left. Reward is -1 on all transitions except those into the region marked "The Cliff". Stepping this region incurs a reward of -100 and sends the agent instantly back to the start (and the episode is still not terminated). As in Figure 3.1, the optimal path would give the largest reward, which is -13. Assume that the behavior policy is the ϵ-greedy policy with $\epsilon = 0.1$. We will compare the performance of Sarsa and Q-learning in the environment with the ϵ-greedy behavior policy.

Q-learning is an off-policy method that directly learns values for an optimal policy which travels right along the edge of the cliff. Unfortunately, since the behavior policy, which is different from the target policy, chooses actions according to ϵ-greedy policy, the agent occasionally falls off the cliff. The graph of an experiment using Q-learning is shown in Figure 3.2. From the figure, we can see

[1] `https://github.com/smfelixchoi/MATH-DRL-study/blob/main/2.Reinforcement_Learning/Temporal_Difference_methods/Temporal_Difference_methods.ipynb`

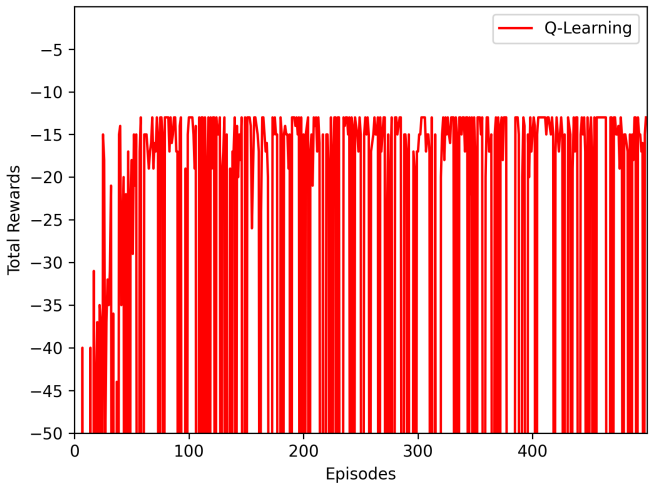

Figure 3.2: Sum of rewards during episode with Q-learning

that it reaches the optimal path with reward -13. However, it makes worse rewards too often due to falling off the cliff (resulting the reward of -100).

On the other hand, Sarsa is an on-policy method so that the action selection is directly considered on the target policy. Therefore, the information about cliff-falling would be in the target policy which is equal to the behavior policy. This makes the resulting policy from Sarsa the longer but safer path through the upper part of the grid. The graph of an experiment using Sarsa is shown in Figure 3.3. From the figure, we can see that the maximum of total rewards is -17, which corresponds to

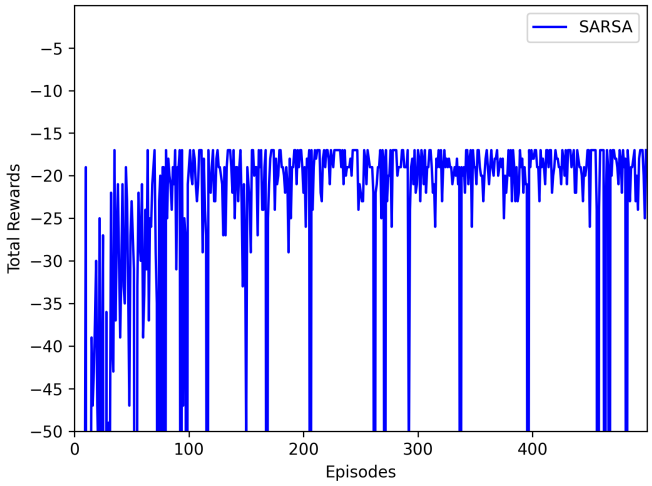

Figure 3.3: Sum of rewards during episode with Sarsa

the safer path in Figure 3.1. This safer path also results in less number of cliff-falling compared to

Q-learning. Therefore, if we run such experiments multiple times and average the rewards, we can see that the average reward of Q-learning will be smaller than that of Sarsa. This result can be seen in Figure 3.4. Of course, if ϵ were gradually reduced, then both methods would asymptotically converge to the optimal policy.

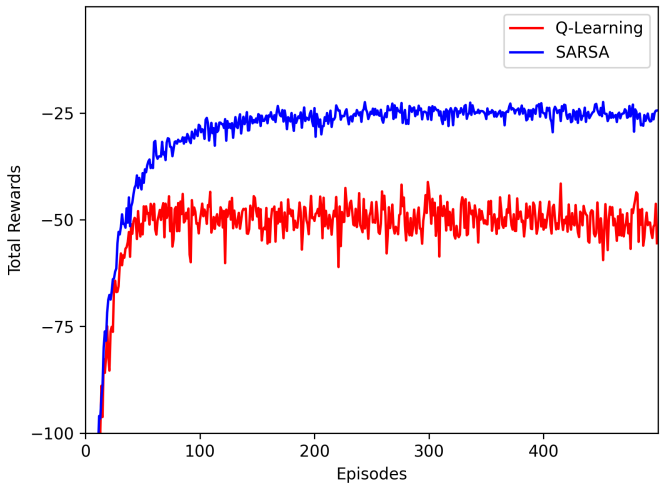

Figure 3.4: Comparison between Sarsa and Q-learning. The graph is plotted by repeating the experiments 500 times.

3.5 Expected Sarsa

Expected Sarsa like Q-learning algorithm that uses the expected value over next state-action pairs instead of the maximization.

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1}) \,|\, S_{t+1}] - Q(S_t, A_t)\big] \\ &\leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\big] \end{aligned}$$

Given the next state, S_{t+1}, this algorithm moves *deterministically* in the same direction as Sarsa moves in expectation. Expected Sarsa costs more than the original Sarsa, but it eliminates the variance due to the random selection of A_{t+1}. Hence, given the same amount of experience, we might expect it to perform slightly better than Sarsa.

Let's look at the experimental results using Cliff Walking. With the same setting in Section 3.4, the graph of total rewards using Expected Sarsa is shown in Figure 3.5. We can see that it reaches the reward of -15, which is lower than the optimal reward -13 (by Q-learning) and greater than the safe path -17 (by Sarsa). We can think of that this behavior is resulted by the expectation in the target of Expected Sarsa. Now we consider the average performance of Expected Sarsa. The comparison of performances between Sarsa, Q-learning, and Expected Sarsa is shown in Figure

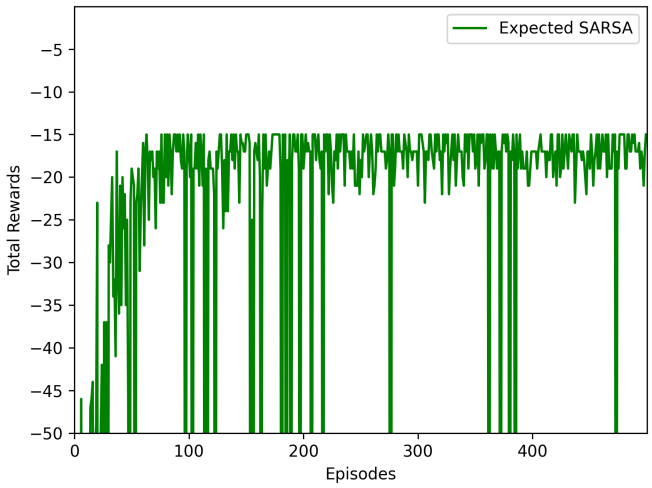

Figure 3.5: Expected Sarsa Performance

3.6. We can see that the average performance of Expected Sarsa is better than both Sarsa and Q-learning. In cliff walking, the state transitions are all deterministic and all randomness comes from the policy. In such cases, Expected Sarsa can take an advantage of expectation over Sarsa because the optimality and the randomness are both considered in the target.

3.6 Maximization Bias and Double Learning

We have discussed algorithms, Sarsa and Q-learning, that involve *maximization* in the construction of their target policies. In Q-learning, the target policy is the greedy policy, while in Sarsa, the target policy, which is the same as the behavior policy, is often ϵ-greedy policy, which also involves maximization. In these algorithms, a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias. In this sense, we call the bias *maximization bias* or overestimation bias. We will see how such a bias can be introduced by the following examples: Figure 6.5 and Example 6.7 of [1]. This experiment can be also found in my GitHub[2].

A simple MDP model is shown in Figure 3.7. There are two non-terminal states **A** and **B**. Episodes always start in **A** with a choice between two actions, left and right. The right action transitions immediately to the terminal state with a reward and return of zero. The left action transitions to **B**, also with a reward of zero. In the state **B**, there are many possible actions all of which cause immediate termination with a reward drawn from $\mathcal{N}(-0.1, 1)$. Then, the expected return for any trajectory starting from the left action is -0.1, which means that taking left in state **A** is always a mistake. However, because of the randomness of reward, the agent can be fooled by

[2] `https://github.com/smfelixchoi/MATH-DRL-study/blob/main/2.Reinforcement_Learning/Temporal_Difference_methods/Double_Learning/Double_Learning.ipynb`

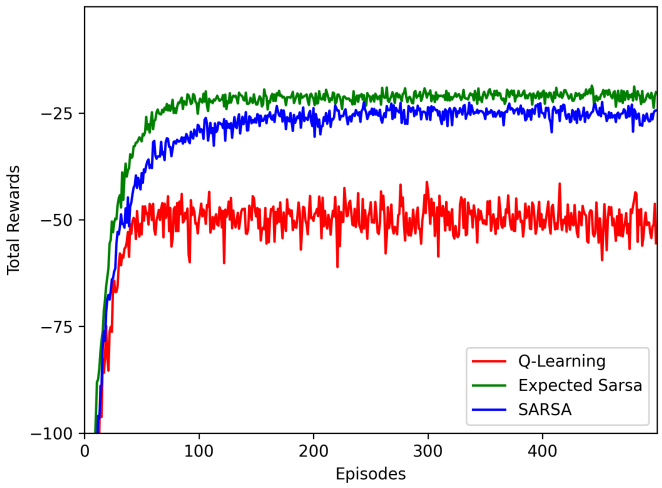

Figure 3.6: Comparison between Sarsa, Q-learning, and Expected Sarsa. The graph is plotted by repeating the experiments 500 times.

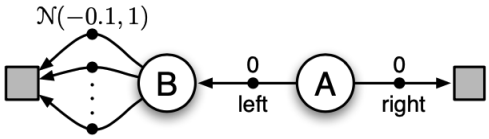

Figure 3.7: Simple MDP

some samples that obtained positive rewards. Hence, in the beginning of learning, the algorithms may favor the left action because of maximization bias making **B** appear to have a positive value. From Figure 3.8, the maximization bias indeed occurs in the MDP. Assumptions for the experiment to plot the figure are as follows: the behavior policy is the ϵ-greedy policy with $\epsilon = 0.1$; the data are averaged over 10,000 runs; the initial action-value estimates were zero; any ties in ϵ-greedy action selection were broken randomly. In the figure, we can see that agent chooses the left action from **A** very often, while the optimal percentage of choosing left actions is 5% due to ϵ-greedy policy. Notice that maximization bias would not occur in the environment with deterministic dynamics (deterministic reward and deterministic transitions). However, when the rewards are deterministic but the state transitions are stochastic, the overestimation bias can be happened.

Then, how can we avoid maximization bias? One way to view the problem is that the problem is due to using the same samples both to determine the maximizing action and to estimate its value. Assume that we construct two unbiased estimators $Q_1(s, a)$ and $Q_2(s, a)$ for the true value $q(s, a)$. At a state s, we can determine the maximizing action $A^* = \arg\max_a Q_1(s, a)$, and we can use Q_2 to provide the estimate of its value $Q_2(s, A^*) = Q_2(s, \arg\max_a Q_1(s, a))$. Then, by the assumption of unbiasedness, we have $\mathbb{E}[Q_2(s, A^*)] = q(s, A^*)$. Hence, the estimate is again unbiased. This idea is called *double learning*. Note that although we learn two estimates, only one estimate is updated

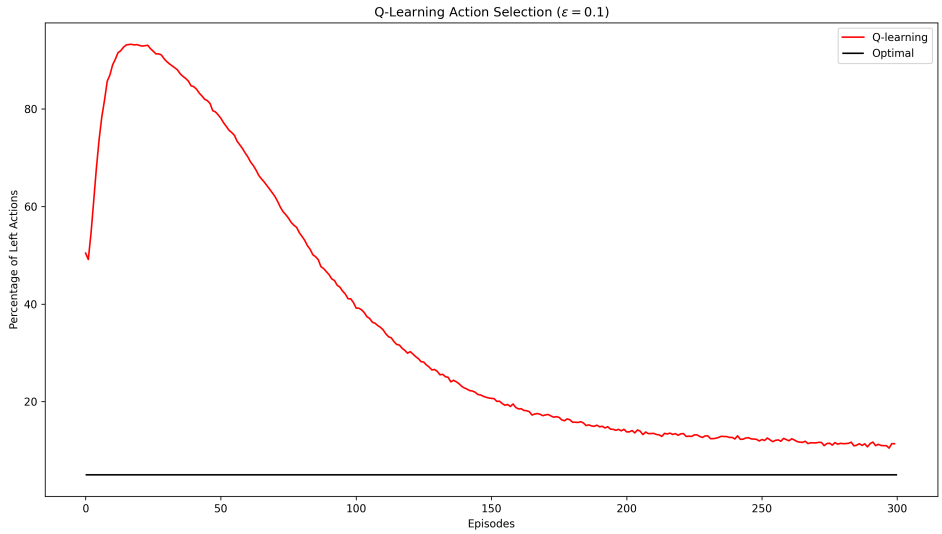

Figure 3.8: The percentage of choosing left action from the state **A**.

on each play. Hence, double learning doubles the memory requirements, but does not increase the amount of computation per step.

The double learning algorithm analogous to Q-learing, called Double Q-learning, is shown in Algorithm 3.3. The tables Q_1 and Q_2 are treated completely symmetrically. Also, there are also double versions of Sarsa and Expected Sarsa. The experimental result is shown in Figure 3.9. In the figure, we can see that Double Q-learning is essentially unaffected by maximization bias. From the plot, the percentage of choosing the left actions at the state **A** almost directly reduces as the learning proceeds. From the result, we can see that Double Q-learning indeed addresses the maximization bias.

References

[1] Andrew Barto and Richard S. Sutton, *Reinforcement Learning: An Introduction* (2nd ed.), The MIT Press, 2018.

Algorithm 3.3 Double Q-learning

Parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, such that $Q(terminal, \cdot) = 0$.

for each episode **do**
    Initialize S
    for each step in episode **do**
        Choose A from S using policy ϵ-greedy in $Q_1 + Q_2$
        Take action A, observe R, S'
        With 0.5 probability:
            $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha[\, R + \gamma Q_2(S', \arg\max_a Q_1(S', a)) - Q_1(S, A)\,]$
        else:
            $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha[\, R + \gamma Q_1(S', \arg\max_a Q_2(S', a)) - Q_2(S, A)\,]$
        $S \leftarrow S'$
        if S is terminal **then**
            break
        end if
    end for
end for

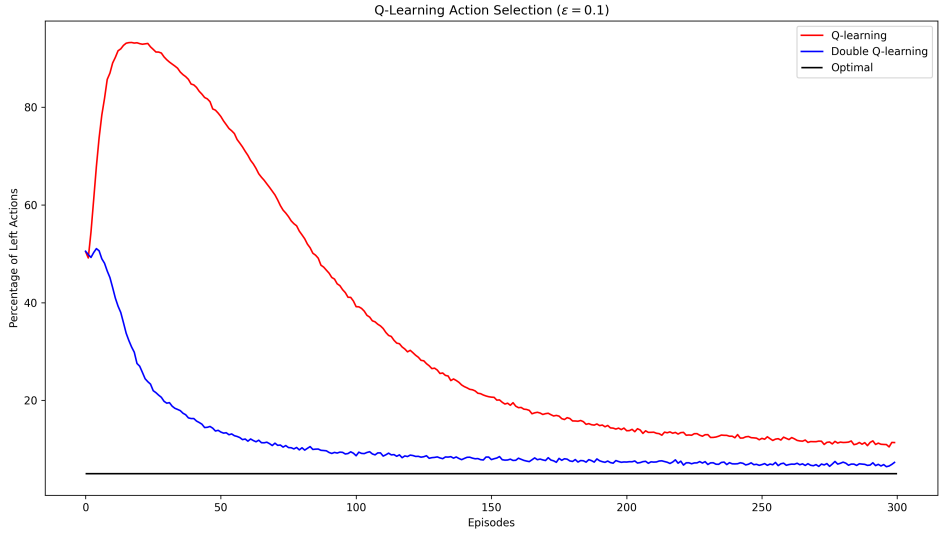

Figure 3.9: Comparison of Q-learning and Double Q-learning on a simple MDP.