RL Ch6 Temporal-Difference Learning

Le-Rong HSU

June 2023

1 Introduction

If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be *temporal-difference* (TD) learning. TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).

In this chapter, we will introduce TD prediction and based on this idea, we develop three important algorithms, Sarsa prediction method (on-policy TD control), Q-learning (off-policy TD method), which has a transformation double Q-learning, and expected Sarsa method. We will also compare these algorithms in examples.

2 TD prediction

Both TD and Monte Carlo methods use experience to solve the prediction problem. Given some experience following a policy π, both methods update their estimate V of v_π for the nonterminal states S_t occurring in that experience. Roughly speaking, Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V(S_t)$. A simple every-visit Monte Carlo method suitable for nonstationary environments is

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]. \tag{6.1}$$

This is call *constant-α MC* method.

Remark. Monte Carlo methods perform an update for each state based on the entire sequence of observed rewards from that state until the end of the episode. The G_t is the cumulative discounted future reward, which counts the reward from the current state t up to time T.

In contrast, TD methods need to wait only until the next time step. The simplest TD method makes the update:

$$V(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{6.2}$$

Remark. Recall in chapter 2,

$$\text{Newestimate} \leftarrow \text{Newestimate} + \text{Stepsize}[\text{Target} - \text{Oldestimate}], \tag{2.4}$$

where the bracket is the error estimate that is what we wish to minimize.

The target for the TD update is $\gamma V(S_{t+1}) - V(S_t)$. This TD method is called *TD(0)*, or *one step TD*, because it is a special case of the $TD(\lambda)$ and n-step TD methods developed in Chapter 12 and Chapter 7. Its procedural form is presented in the following:

Algorithm 1 Tabular $TD(0)$ for estimating v_π

Input: policy π to be evaluated
Algorithm parameter: step size $\alpha \in [0, 1)$
Initialize $V(s)$ for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$
for each episode **do**
  Initialize S
  for each step of episode **do**
    $A \leftarrow$ action given by π
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\left[R + \gamma V(S') - V(S)\right]$
    $S \leftarrow S'$
  end for
  Until S is terminal
end for

Because TD(0) bases its update in part on an existing estimate, we say that it is a *bootstrapping* method (bootstrapping means using one or more estimated values in the update step for the *same* kind of estimated value).

Recall that

$$\begin{aligned} v_\pi(s) &= \mathbb{E}[G_t | S_t = s] && (6.3)\\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] && (6.4) \end{aligned}$$

- Monte Carlo method
  - goal: estimate (6.3)
  - is an estimate since we don't the true value of (6.3)
- DP method
  - goal: estimate (6.4)
  - is an estimate since $v_\pi(S_{t+1})$ is unknown

The TD(0) samples the expected values in (6.4) and it uses the current estimate V instead of the true v_π. Thus, TD methods combine the sampling of Monte

Carlo with the bootstrapping of DP. We refer to TD and Monte Carlo updates as *sample updates* because they involve looking ahead to a sample successor state (or state–action pair), using the value of the successor and the reward along the way to compute a backed-up value, and then updating the value of the original state (or state–action pair) accordingly.

Sample updates differ from the *expected* updates of DP methods in that they are based on a single sample successor rather than on a complete distribution of all possible successors.

Note that the quantity in brackets in the TD(0) update is a sort of error, measuring the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$. We call this the *TD error*:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{6.5}$$

Also note that

- *TD error* is *made at that time*
- δ_t is the error in $V(S_t)$, but available at time $t+1$
- if the array V does not change during the episode, then the Monte Carlo error can be written as a sum of TD errors:

$$\begin{aligned} G_t - V(S_t) &= R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) \qquad (3.9) \\ &= \delta_t + \gamma(G_{t+1} - V(S_{t+1})) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2(G_{t+2} - V(S_{t+2})) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(G_T - V(S_T)) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(0-0) \\ &= \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k \qquad (6.6) \end{aligned}$$

Example 6.1: Driving Home Each day as you drive home from work, you try to predict how long it will take to get home. When you leave your office, you note the time, the day of week, the weather, and anything else that might be relevant. Say on this Friday you are leaving at exactly 6 o'clock, and you estimate that it will take 30 minutes to get home. As you reach your car it is 6:05, and you notice it is starting to rain. Traffic is often slower in the rain, so you reestimate that it will take 35 minutes from then, or a total of 40 minutes. Fifteen minutes later you have completed the highway portion of your journey in good time. As you exit onto a secondary road you cut your estimate of total travel time to 35 minutes. Unfortunately, at this point you get stuck behind a slow truck, and the road is too narrow to pass. You end up having to follow the truck until you turn onto the side street where you live at 6:40. Three minutes later you are home. The sequence of states, times, and predictions is thus as follows:

State	*Elapsed Time (minutes)*	*Predicted Time to Go*	*Predicted Total Time*
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

The rewards in this example are the elapsed times on each leg of the journey (since we want to predict how long it will take to get home). We are not discounting ($\gamma = 1$), and thus the return for each state is the actual time to go from that state.

Now we compare the prediction from MC methods and TD methods:

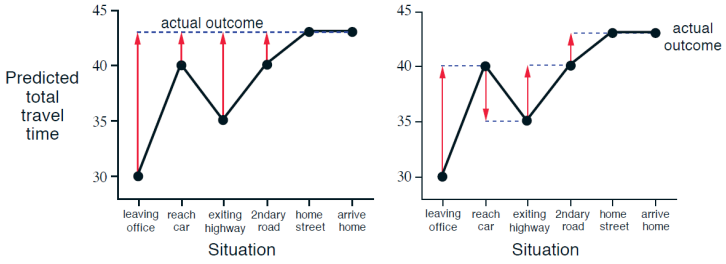


Figure 6.1: Changes recommended in the driving home example by Monte Carlo methods (left) and TD methods (right).

The red arrows indicate the difference between actual outcome and prediction. For the constant-α MC method (left), these are exactly the errors between the estimated value (predicted time to go) in each state and the actual return (actual time to go).

On the other hand, according to a TD approach, you would learn immedi-

ately, shifting your initial estimate from 30 minutes toward 50. In fact, each estimate would be shifted toward the estimate that immediately follows it. According to the figure, one can observe that each error is proportional to the change over time of the prediction, that is, to the *temporal differences* in predictions.

3 Advantages of TD Prediction Methods

Obviously, TD methods have an advantage over DP methods in that they do not require a model of the environment, of its reward and next-state probability distributions.

The next most obvious advantage of TD methods over Monte Carlo methods is that they are naturally implemented in an online, fully incremental fashion. With Monte Carlo methods one must wait until the end of an episode, because only then is the return known, whereas with TD methods one need wait only one time step. It turns out to be a critical consideration.

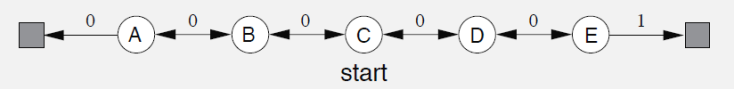
- Some applications have very long episodes, so that delaying all learning until the end of the episode is too slow
- Other applications are continuing tasks and have no episodes at all.

The TD methods do make sense. For any fixed policy π, TD(0) has been proved to converge to v_π, in the mean for a constant step-size parameter if it is sufficiently small, and with probability 1 if the step-size parameter decreases according to the usual stochastic approximation conditions (2.7).

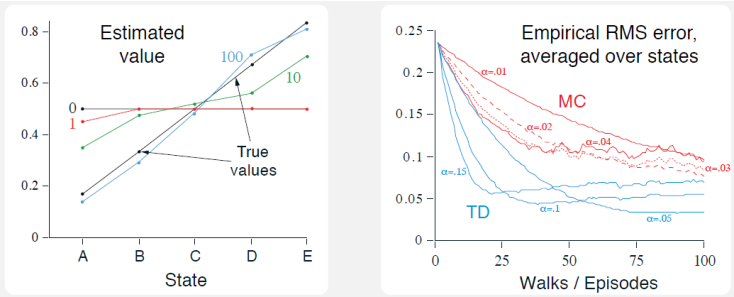
$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \text{ and } \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty \tag{2.7}$$

If both TD methods and MC methods are able to converge to correct predictions, the next question is "Which one is faster"? At the current time this is an open question in the sense that no one has been able to prove mathematically that one method converges faster than the other. In fact, it is not even clear what is the most appropriate formal way to phrase this question. **In practice, however, TD methods have usually been found to converge faster than constant-α MC methods on stochastic tasks**, as illustrated in Example 6.2.

Example 6.2: Random Walk A *Markov reward process*, or MRP, is a Markov decision process without actions. We will often use MRPs when focusing on the prediction problem, in which **there is no need to distinguish the dynamics due to the environment from those due to the agent**.



Consider the random walk, which is a Markov reward process, on these nodes. In this MRP, all episodes start in the center state, C, then proceed either left or right by one state on each step, with equal probability. Episodes terminate either on the extreme left or the extreme right. When an episode terminates on the right, a reward of +1 occurs; all other rewards are zero.



The left graph above shows the values learned after various numbers of episodes on a single run of TD(0). The estimates after 100 episodes are about as close as they ever come to the true values—with a constant step-size parameter ($\alpha = 0.1$ in this example). The performance measure shown is the root mean square (RMS) error between the value function learned and the true value function, averaged over the five states, then averaged over 100 runs.

Remark. Root mean square error:

$$M = \sqrt{\frac{\sum_{i=1}^{n} x_i^2}{n}}$$

4 Optimality of TD(0)

Suppose there is available only a finite amount of experience, say 10 episodes or 100 time steps. What strategy should be take to "make use of data optimally"?

Given an approximate value function, V , the increments specified by (6.1) or (6.2) are computed for every time step t at which a nonterminal state is visited, but the value function is changed only once, by the sum of all the increments. That is, we sum up $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ for all t and then update V.

Remark. We will introduce an example of batch updating: Sarsa. In fact, batch updating just a subtle re-ordering of how the experience and updates interact. You can use batch updates where experience is in short supply. Instead of taking each piece of experience once as it is observed, you can store and re-play the trajectories that you have seen so far.

When using batch updating, TD(0) converges deterministically to a single answer which is independent of the step-size parameter, α, as long as α is chosen to be sufficiently small. The constant-α MC method also converges deterministically under the same conditions, but to a different answer.

Example 6.3: Random walk under batch updating After each new episode, all episodes seen so far were treated as a batch. They were repeatedly presented to the algorithm, either TD(0) or constant-α MC, with α sufficiently small that the value function converged.

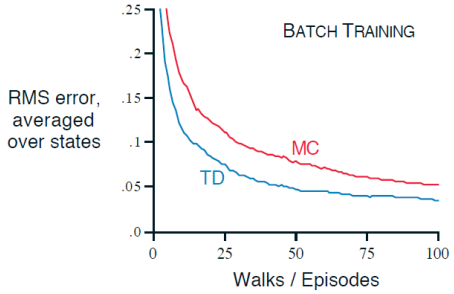


Figure 6.2 Performance of TD(0) and constant-α MC under batch training on the random walk task.

How is it that batch TD was able to perform better than this optimal method? The answer is that the **Monte Carlo method is optimal only in a limited way**, and that **TD is optimal in a way that is more relevant to predicting returns**.

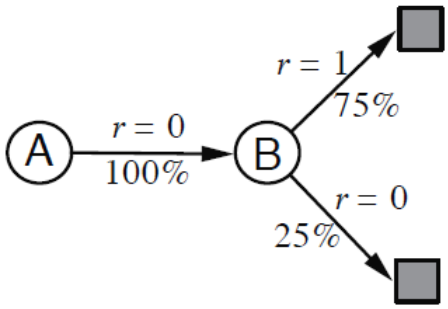
Example 6.4: You are the Predictor Place yourself now in the role of the predictor of returns for an unknown Markov reward process. Suppose you observe the following eight episodes:

A, 0, B, 0	B, 1
B, 1	B, 1
B, 1	B, 1
B, 1	B, 0

This means that the first episode started in state A, transitioned to B with a reward of 0, and then terminated from B with a reward of 0. The other seven episodes were even shorter, starting from B and terminating immediately. Given this batch of data, what would you say are the optimal predictions, the best values for the estimates V (A) and V (B)? Everyone would probably agree that

the optimal value for $V(B)$ is $\frac{3}{4}$, because six out of the eight times in state B the process terminated immediately with a return of 1, and the other two times in B the process terminated immediately with a return of 0.

What is the optimal value for the estimate V (A) given this data? Here there are two reasonable answers, and we will see the difference between MC methods and TD methods.



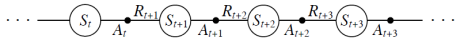
Example 6.4 illustrates a general difference between the estimates found by batch TD(0) and batch Monte Carlo methods.

- batch Monte Carlo methods: always find the estimates that minimize mean square error on the training set
- batch TD(0): always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process.

5 Sarsa: On-policy TD Control

We turn now to the use of TD prediction methods for the control problem. As usual, we follow the pattern of generalized policy iteration (GPI), only this time using TD methods for the evaluation or prediction part.

The first step is to learn an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate $q_\pi(s, a)$ for the current behavior policy π and for all states s and actions a. This can be done using essentially the same TD method described above for learning v_π. Recall that an episode consists of an alternating sequence of states and state–action pairs:



Now we consider transitions from state–action pair to state–action pair, and learn the values of state–action pairs. Formally they are identical: they are both Markov chains with a reward process:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \qquad (6.7)$$

Sarsa: state, action, reward, state', action'

This update is done after every transition from a nonterminal state S_t. If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero. The update uses every element of the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$.

As in all on-policy methods, we continually estimate q_π for the behavior policy π, and at the same time change π toward greediness with respect to q_π.

Sarsa converges with probability 1 to an optimal policy and action-value function as long as every state-action pair are visited infinitely many times and the policy converges in the limit to the greedy policy. 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. In this case, we could choose $\varepsilon = 1/t$ so that the policy converges.

Algorithm 2 Sarsa (on-policy TD control) for estimating $Q \approx q_*$

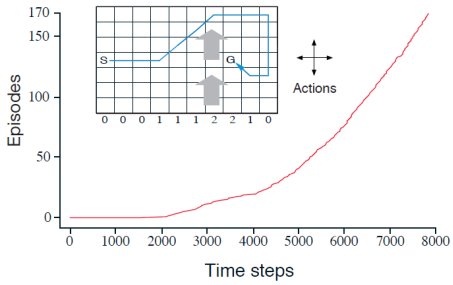
Algorithm parameters: step size $\alpha \in [0, 1)$, small ε
Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily except that Q(terminal,$\cdot$) = 0
for each episode **do**
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  for each step of episode **do**
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
    $S \leftarrow S'$, $A \leftarrow A'$
  end for
end for

Example 6.5: Windy Gridworld

- start, goal state
- up, down, left, right
- the resultant next states are shifted upward by a "wind", whose strength varies from column to column
- setting: $\varepsilon = 0.1$, $\alpha = 0.5$, $Q(s, a) = 0$ for all s, a

Conclusion: Monte Carlo methods cannot easily be used here because termination is not guaranteed for all policies. If a policy was ever found that caused the agent to stay in the same state, then the next episode would never end. Online learning methods such as Sarsa do not have this problem because they quickly learn during the episode that such policies are poor, and switch to something else.

Remark. **Online learning** is a way of data being given, meaning that the data, or samples, are given **sequentially**. It is helpful when we have to update the estimates or parameters frequently.



ε-greedy Sarsa

6 Q-learning: Off-policy TD Control

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

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{6.8}$$

Remark. Sarsa:

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

In this case, Q directly approximates q_*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. However, all that is required for correct convergence is that all pairs continue to be updated. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to q_*.

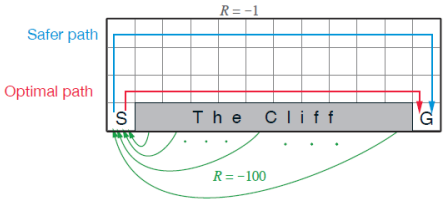
Remark. Notice that there is a subtle difference between Sarsa and Q-learning. In each episode of Sarsa, after we choose A from initialized S, for each step of episode, we first take action A and choose A' from S'. Then update Q and S, A. For Q-learning, we don't need to update A as we take maximum when updating Q.

Let's compare Sarsa (on-policy) and Q-learning (off-policy).

Example 6.6: Cliff Walking

Algorithm 3 Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in [0, 1)$, small ε
Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$
for each episode **do**
  Initialize S
  for each step of episode **do**
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$
  end for
  Until S is terminal
end for

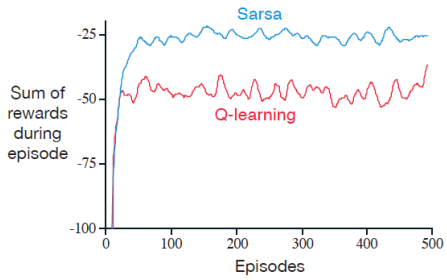


- undicounted
- episodic
- start and goal states
- up, down, left, right
- Reward is -1 on all transitions except those into region marked "The Cliff." Stepping into this region incurs a reward of -100 and sends the agent instantly back to the start.

The graph below shows the performance of Sarsa and Q-learning methods with ε-greedy selection, $\varepsilon = 0.1$.

After an initial transient, Q-learning learns values for the optimal policy that travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off because of ε-greedy selection. On the other hand, Sarsa takes the action selection into account and learns the longer but safer path through the upper part of the grid.

Remark. Why is Q-learning considered off-policy and Sarsa considered on-policy? This is my answer.



In on-policy learning, the $Q(s, a)$ function is learned from actions that we took using our current policy $\pi(a|s)$:

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$$

Both A and A' are chosen from π, so Sarsa is on-policy.

On the other hand, in off-policy learning, the $Q(s, a)$ function is learned from actions that we took using another policy (actually):

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$$

For Q-learning, it chooses action with ε-greedy policy and update Q with greedy policy. That is, we assumed **greedy policy** were followed during the update despite the fact that we actually don't follow a greedy policy.

7 Expected Sarsa

As its name suggests, is just like Q-learning except that instead of the maximum over next state–action pairs it uses the expected value, taking into account how likely each action is under the current policy. The update rule is:

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1})|S_{t+1}] - Q(S_t, A_t)] \quad (1) \\ &\leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)] \end{aligned} \tag{6.9}$$

but that otherwise follows the schema of Q-learning. Given the next sate S_{t+1}, this algorithm moves deterministically in the same direction as Sarsa moves in *expectation*, and accordingly it's called *expected Sarsa*.

Expected Sarsa is more complex computationally than Sarsa but, in return, it eliminates the variance due to the random selection of A_{t+1}. Given the same amount of experience, we might expect it to perform slightly better than Sarsa, and indeed it generally does.

Figure 6.3 shows summary results on the cliff-walking task with compared to Sarsa and Q-learning.

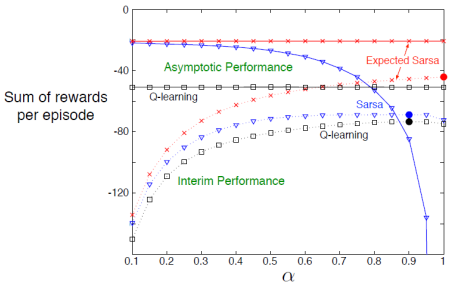


Figure 6.3: All algorithms used an ε-greedy policy with ε= 0.1. Asymptotic performance is an average over 100,000 episodes whereas interim performance is an average over the first 100 episodes. These data are averages of over 50,000 and 10 runs for the interim and asymptotic cases respectively. The solid circles mark the best interim performance of each method.

I summarize the key points of figure 6.3:

- Expected Sarsa can safely set $\alpha = 1$ without suffering any degradation of asymptotic performance
- Sarsa can only perform well in the long run at a small value of α, at which short-term performance is poor.

Remark. In general, we could use a policy different from the target policy π to generate behavior, in which case it becomes an off-policy algorithm. For example, π is the greedy policy while behavior is more exploratory; then Expected Sarsa is exactly Q-learning.

In this sense Expected Sarsa subsumes and generalizes Q-learning while reliably improving over Sarsa. Except for the small additional computational cost, Expected Sarsa may completely dominate both of the other more-well-known TD control algorithms.

8 Maximization Bias and Double Learning

Suppose the maximum of the true values is zero. If the maximum of the estimates is positive, a positive bias, then we say it the *maximization bias*.

The ε-greedy and greedy policy assumed in Sarsa and Q-learning involves a maximization operation. A maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias.

Example 6.7: Maximization Bias Example

- nonterminal state **A,B**
- always start in **A**
- **left** and **right**
- **right**$\rightarrow$ reward 0
- **left** $\rightarrow$ reward 0, and from which there are many possible actions which cause immediate termination with a reward drawn from $\mathcal{N}(-0.1, 1)$

Thus, any trajectory starting with **left** has expected return -0.1 and thus choosing **left** is always a mistake.

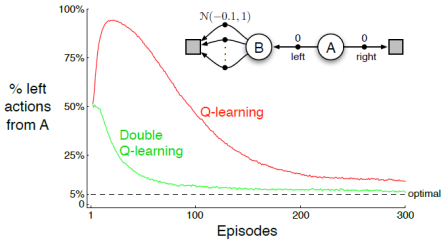


Figure 6.5: Comparison of Q-learning and Double Q-learning on a simple episodic MDP (shown inset). Q-learning initially learns to take the left action much more often than the right action, and always takes it significantly more often than the 5% minimum probability enforced by ε-greedy action selection with $\varepsilon = 0.1$. In contrast, Double Q-learning is essentially unaffected by maximization bias. These data are averaged over 10,000 runs. The initial action-value estimates were zero. Any ties in ε-greedy action selection were broken randomly.

Remark. parameters: $\varepsilon = 0.1, \alpha = 0.1$ and $\gamma = 1$

However, it's surprising that our control methods may favor **left** because of maximization bias making **B** appear to have a positive value. Even at asymptote, Q-learning takes the left action about 5% more often than is optimal at our parameter settings.

Remark. Although the mean is negative, it's still possible to draw a positive reward, which results in maximization bias.

Are there algorithms that avoid maximization bias? One way to view the problem is that it is due to using the same samples (plays) both to determine the maximizing action and to estimate its value. Suppose we divided the plays in two sets and used them to learn two independent estimates, call them $Q_1(a)$ and $Q_2(a)$, each an estimate of the true value $q(a)$, for all $a \in \mathcal{A}$. We could choose one estimate, say $Q_1(a)$, to determine the maximizing action $A^* = \arg\max_a Q_1(a)$, and the other, Q_2, to provide the estimate of its value, $Q_2(A^*) = Q_2(\arg\max_a Q_1(a))$. This estimate will then be unbiased in the sense that $\mathbb{E}[Q_2(A^*)] = q(A^*)$. This is the idea of *double learning*.

Note that although we learn two estimates, only one estimate is updated on each play; double learning doubles the memory requirements, but does not increase the amount of computation per step.

Double Q-learning divides the time steps in two, perhaps by flipping a coin on each step. If the coin comes up heads, the update is

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_2(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t)] \tag{6.10}$$

If it's tail, then

$$Q_2(S_t, A_t) \leftarrow Q_2(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_1(S_{t+1}, \arg\max_a Q_2(S_{t+1}, a)) - Q_2(S_t, A_t)]$$

The complete algorithm is:

Algorithm 4 Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in [0, 1)$, small ε
Initialize $Q_1(s, a), Q_2(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$
for each episode **do**
  Initialize S
  for each step of episode **do**
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probability:
    $Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_2(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t)]$
    else:
    $Q_2(S_t, A_t) \leftarrow Q_2(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_1(S_{t+1}, \arg\max_a Q_2(S_{t+1}, a)) - Q_2(S_t, A_t)]$
    $S \leftarrow S'$
  end for
  Until S is terminal
end for

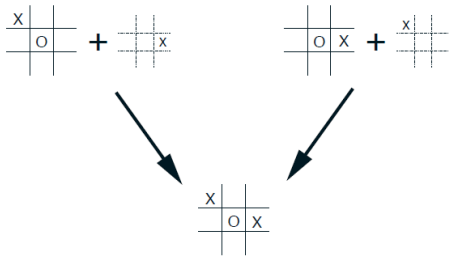
9 Games, Afterstates, and Other Special Cases

In chapter 1, the author use an example, tic-tac-toc. to illustrate the key ingredients of reinforcement learning. In that example, one method to estimate of the probabilities of winning is through the temporal difference with "value function. The update rule in that example is:

$$V(S_t) \leftarrow V(S_t) + \alpha[V(S_{t+1} - V(S_t)].$$

However, V is actually neither action value function nor state value function. Conventional state-value function evaluates states after the agent taking an action. In contrast, V in this example evaluates "board positions" *after* the agent has made its move. We call these *afterstates*, and the value functions over these are *afterstate value functions*. Afterstates are useful when we have knowledge of an initial part of the environment's dynamics but not necessarily of the full dynamics.

A conventional action-value function would map from positions and moves to an estimate of the value. But many position—move pairs produce the same resulting position, as in the example below:



In such cases the position–move pairs are different but produce the same "afterposition," and thus must have the same value. **A conventional action-value function would have to separately assess both pairs, whereas an afterstate value function would immediately assess both equally**.

10 Summary

TD methods are alternatives to Monte Carlo methods for solving the prediction problem. In both cases, the extension to the control problem is via the idea of generalized policy iteration (GPI).

GPI drives the value function to accurately predict returns for the current policy; this is the prediction problem. When the first process is based on experience, a complication arises concerning maintaining sufficient exploration. We can classify TD control methods according to whether they are **on-policy** or **off-policy**. There is a third way in which TD methods can be extended to: **actor-critic** methods, which is the method applied in **deep hedging**.

The methods presented in this chapter are today the most widely used reinforcement learning methods. This is probably due to their great simplicity: they can be applied online, with a minimal amount of computation.

All the new algorithms will retain the essence of those introduced here: they will be able to **process experience online**, with **relatively little computation**. In the next two chapters we extend them to n-step forms (a link to Monte Carlo methods) and forms that include a model of the environment (a link to planning and dynamic programming). Then, in the second part of the book we extend them to various forms of function approximation rather than tables (a link to deep learning and artificial neural networks).