

Chapter 6

Temporal-Difference Learning

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). The relationship between TD, DP, and Monte Carlo methods is a recurring theme in the theory of reinforcement learning. This chapter is the beginning of our exploration of it. Before we are done, we will see that these ideas and methods blend into each other and can be combined in many ways. In particular, in Chapter 7 we introduce the TD(λ) algorithm, which seamlessly integrates TD and Monte Carlo methods.

As usual, we start by focusing on the policy evaluation or *prediction* problem, that of estimating the value function v_π for a given policy π . For the *control* problem (finding an optimal policy), DP, TD, and Monte Carlo methods all use some variation of generalized policy iteration (GPI). The differences in the methods are primarily differences in their approaches to the prediction problem.

6.1 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

$$V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)], \quad (6.1)$$

where G_t is the actual return following time t , and α is a constant step-size parameter (c.f., Equation 2.4). Let us call this method *constant- α MC*. Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(S_t)$ (only then is G_t known), TD methods need wait only until the next time step. At time $t+1$ they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$. The simplest TD method, known as *TD(0)*, is

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

In effect, the target for the Monte Carlo update is G_t , whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$.

Because the TD method bases its update in part on an existing estimate, we say that it is a *bootstrapping* method, like DP. We know from Chapter 3 that

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \quad (6.3)$$

$$\begin{aligned} &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \mid S_t = s \right] \\ &= \mathbb{E}_\pi [R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]. \end{aligned} \quad (6.4)$$

Roughly speaking, Monte Carlo methods use an estimate of (6.3) as a target, whereas DP methods use an estimate of (6.4) as a target. The Monte Carlo target is an estimate because the expected value in (6.3) is not known; a sample return is used in place of the real expected return. The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because $v_\pi(S_{t+1})$ is not known and the current estimate, $V(S_{t+1})$, is used instead. The TD target is an estimate for both reasons: it 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. As we shall see, with care and imagination this can take us a long way toward obtaining the advantages of both Monte Carlo and DP methods.

Figure 6.1 specifies TD(0) completely in procedural form, and Figure 6.2 shows its backup diagram. The value estimate for the state node at the top of

```

Input: the policy  $\pi$  to be evaluated
Initialize  $V(s)$  arbitrarily (e.g.,  $V(s) = 0, \forall s \in \mathcal{S}^+$ )
Repeat (for each episode):
  Initialize  $S$ 
  Repeat (for each step of episode):
     $A \leftarrow$  action given by  $\pi$  for  $S$ 
    Take action  $A$ ; observe reward,  $R$ , and next state,  $S'$ 
     $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$ 
     $S \leftarrow S'$ 
  until  $S$  is terminal

```

Figure 6.1: Tabular TD(0) for estimating v_π .

Figure 6.2: The backup diagram for TD(0).

the backup diagram is updated on the basis of the one sample transition from it to the immediately following state. We refer to TD and Monte Carlo updates as *sample backups* 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 changing the value of the original state (or state–action pair) accordingly. *Sample* backups differ from the *full* backups of DP methods in that they are based on a single sample successor rather than on a complete distribution of all possible successors.

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, 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

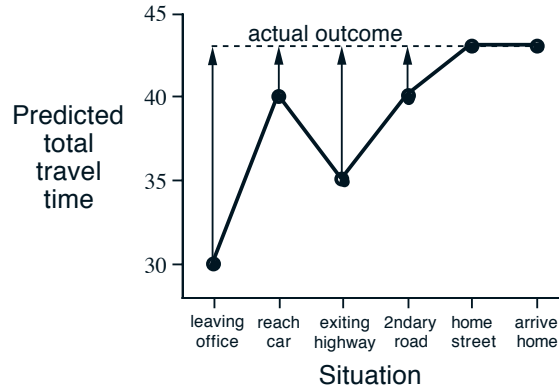


Figure 6.3: Changes recommended by Monte Carlo methods in the driving home example.

thus as follows:

<i>State</i>	<i>Elapsed Time (minutes)</i>	<i>Predicted Time to Go</i>	<i>Predicted Total Time</i>
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.¹ We are not discounting ($\gamma = 1$), and thus the return for each state is the actual time to go from that state. The value of each state is the *expected* time to go. The second column of numbers gives the current estimated value for each state encountered.

A simple way to view the operation of Monte Carlo methods is to plot the predicted total time (the last column) over the sequence, as in Figure 6.3. The arrows show the changes in predictions recommended by the constant- α MC method (6.1), for $\alpha = 1$. These are exactly the errors between the estimated value (predicted time to go) in each state and the actual return (actual time to go). For example, when you exited the highway you thought it would take only 15 minutes more to get home, but in fact it took 23 minutes. Equation 6.1 applies at this point and determines an increment in the estimate of time to go after exiting the highway. The error, $G_t - V(S_t)$, at this time is eight

¹If this were a control problem with the objective of minimizing travel time, then we would of course make the rewards the *negative* of the elapsed time. But since we are concerned here only with prediction (policy evaluation), we can keep things simple by using positive numbers.

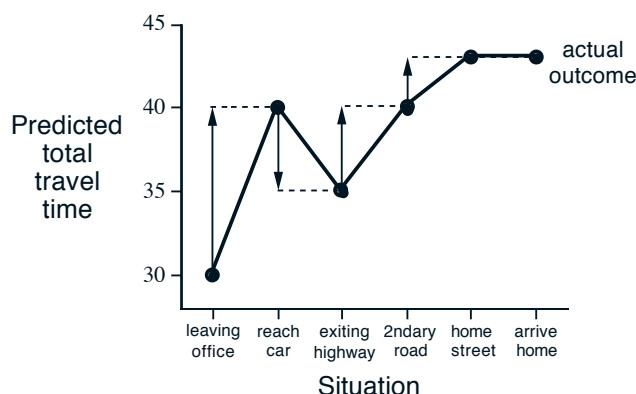


Figure 6.4: Changes recommended by TD methods in the driving home example.

minutes. Suppose the step-size parameter, α , is $1/2$. Then the predicted time to go after exiting the highway would be revised upward by four minutes as a result of this experience. This is probably too large a change in this case; the truck was probably just an unlucky break. In any event, the change can only be made off-line, that is, after you have reached home. Only at this point do you know any of the actual returns.

Is it necessary to wait until the final outcome is known before learning can begin? Suppose on another day you again estimate when leaving your office that it will take 30 minutes to drive home, but then you become stuck in a massive traffic jam. Twenty-five minutes after leaving the office you are still bumper-to-bumper on the highway. You now estimate that it will take another 25 minutes to get home, for a total of 50 minutes. As you wait in traffic, you already know that your initial estimate of 30 minutes was too optimistic. Must you wait until you get home before increasing your estimate for the initial state? According to the Monte Carlo approach you must, because you don't yet know the true return.

According to a TD approach, on the other hand, you would learn immediately, shifting your initial estimate from 30 minutes toward 50. In fact, each estimate would be shifted toward the estimate that immediately follows it. Returning to our first day of driving, Figure 6.4 shows the same predictions as Figure 6.3, except with the changes recommended by the TD rule (6.2) (these are the changes made by the rule if $\alpha = 1$). Each error is proportional to the change over time of the prediction, that is, to the *temporal differences* in predictions.

Besides giving you something to do while waiting in traffic, there are several computational reasons why it is advantageous to learn based on your current predictions rather than waiting until termination when you know the actual

return. We briefly discuss some of these next.

6.2 Advantages of TD Prediction Methods

TD methods learn their estimates in part on the basis of other estimates. They learn a guess from a guess—they *bootstrap*. Is this a good thing to do? What advantages do TD methods have over Monte Carlo and DP methods? Developing and answering such questions will take the rest of this book and more. In this section we briefly anticipate some of the answers.

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 on-line, 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. Surprisingly often this turns out to be a critical consideration. Some applications have very long episodes, so that delaying all learning until an episode's end is too slow. Other applications are continuing tasks and have no episodes at all. Finally, as we noted in the previous chapter, some Monte Carlo methods must ignore or discount episodes on which experimental actions are taken, which can greatly slow learning. TD methods are much less susceptible to these problems because they learn from each transition regardless of what subsequent actions are taken.

But are TD methods sound? Certainly it is convenient to learn one guess from the next, without waiting for an actual outcome, but can we still guarantee convergence to the correct answer? Happily, the answer is yes. For any fixed policy π , the TD algorithm described above 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). Most convergence proofs apply only to the table-based case of the algorithm presented above (6.2), but some also apply to the case of general linear function approximation. These results are discussed in a more general setting in the next two chapters.

If both TD and Monte Carlo methods converge asymptotically to the correct predictions, then a natural next question is “Which gets there first?” In other words, which method learns faster? Which makes the more efficient use of limited data? 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

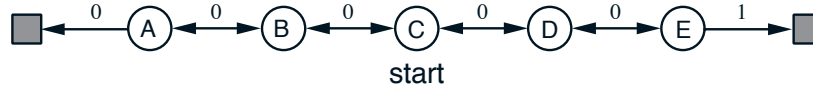


Figure 6.5: A small Markov process for generating random walks.

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 the following example.

Example 6.2: Random Walk In this example we empirically compare the prediction abilities of TD(0) and constant- α MC applied to the small Markov process shown in Figure 6.5. All episodes start in the center state, C, and proceed either left or right by one state on each step, with equal probability. This behavior is presumably due to the combined effect of a fixed policy and an environment's state-transition probabilities, but we do not care which; we are concerned only with predicting returns however they are generated. 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. For example, a typical walk might consist of the following state-and-reward sequence: C, 0, B, 0, C, 0, D, 0, E, 1. Because this task is undiscounted and episodic, the true value of each state is the probability of terminating on the right if starting from that state. Thus, the true value of the center state is $v_\pi(C) = 0.5$. The true values of all the states, A through E, are $\frac{1}{6}$, $\frac{2}{6}$, $\frac{3}{6}$, $\frac{4}{6}$, and $\frac{5}{6}$. Figure 6.6 shows the values learned by TD(0) approaching the true values as more episodes are experienced. Averaging over many episode sequences, Figure 6.7 shows the average error in the predictions found by TD(0) and constant- α MC, for a variety of values of α , as a function of number of episodes. In all cases the approximate value function was initialized to the intermediate value $V(s) = 0.5$, for all s . The TD method is consistently better than the MC method on this task over this number of episodes. ■

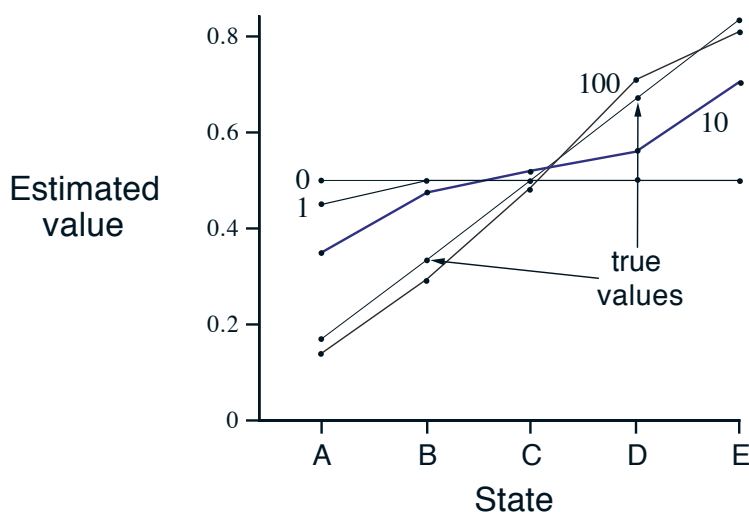


Figure 6.6: Values learned by TD(0) after various numbers of episodes. The final estimate is about as close as the estimates ever get to the true values. With a constant step-size parameter ($\alpha = 0.1$ in this example), the values fluctuate indefinitely in response to the outcomes of the most recent episodes.

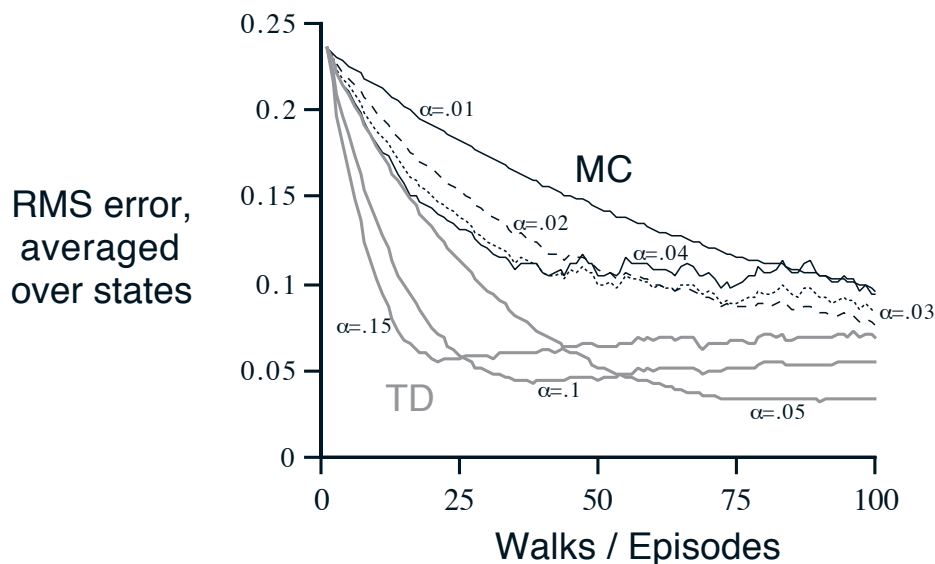


Figure 6.7: Learning curves for TD(0) and constant- α MC methods, for various values of α , on the prediction problem for the random walk. The performance measure shown is the root mean-squared (RMS) error between the value function learned and the true value function, averaged over the five states. These data are averages over 100 different sequences of episodes.

6.3 Optimality of TD(0)

Suppose there is available only a finite amount of experience, say 10 episodes or 100 time steps. In this case, a common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer. 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. Then all the available experience is processed again with the new value function to produce a new overall increment, and so on, until the value function converges. We call this *batch updating* because updates are made only after processing each complete *batch* of training data.

Under batch updating, TD(0) converges deterministically to a single answer 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. Understanding these two answers will help us understand the difference between the two methods. Under normal updating the methods do not move all the way to their respective batch answers, but in some sense they take steps in these directions. Before trying to understand the two answers in general, for all possible tasks, we first look at a few examples.

Example 6.3 *Random walk under batch updating.* Batch-updating versions of TD(0) and constant- α MC were applied as follows to the random walk prediction example (Example 6.2). 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. The resulting value function was then compared with v_π , and the average root mean-squared error across the five states (and across 100 independent repetitions of the whole experiment) was plotted to obtain the learning curves shown in Figure 6.8. Note that the batch TD method was consistently better than the batch Monte Carlo method. ■

Under batch training, constant- α MC converges to values, $V(s)$, that are sample averages of the actual returns experienced after visiting each state s . These are optimal estimates in the sense that they minimize the mean-squared error from the actual returns in the training set. In this sense it is surprising that the batch TD method was able to perform better according to the root mean-squared error measure shown in Figure 6.8. 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. But first let's develop our

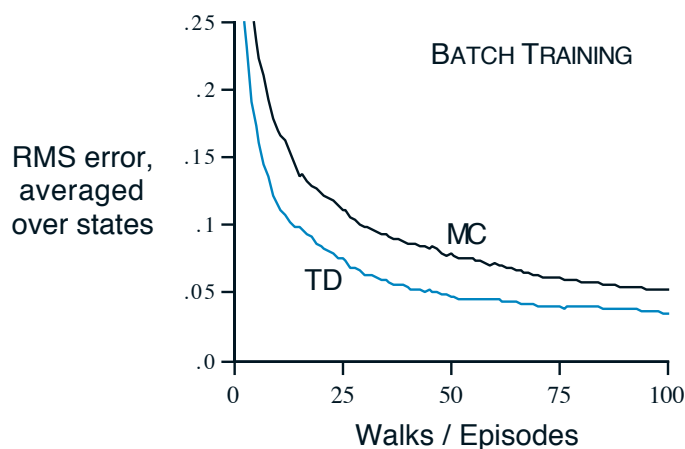


Figure 6.8: Performance of TD(0) and constant- α MC under batch training on the random walk task.

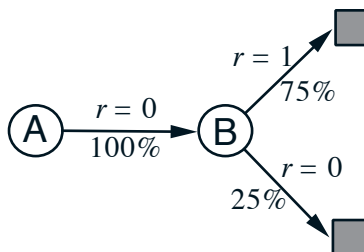
intuitions about different kinds of optimality through another example.

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 $V(\mathbf{A})$ and $V(\mathbf{B})$? Everyone would probably agree that the optimal value for $V(\mathbf{B})$ 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.

But what is the optimal value for the estimate $V(\mathbf{A})$ given this data? Here there are two reasonable answers. One is to observe that 100% of the times the process was in state **A** it traversed immediately to **B** (with a reward of 0); and since we have already decided that **B** has value $\frac{3}{4}$, therefore **A** must have value $\frac{3}{4}$ as well. One way of viewing this answer is that it is based on first modeling the Markov process, in this case as



and then computing the correct estimates given the model, which indeed in this case gives $V(A) = \frac{3}{4}$. This is also the answer that batch TD(0) gives.

The other reasonable answer is simply to observe that we have seen A once and the return that followed it was 0; we therefore estimate $V(A)$ as 0. This is the answer that batch Monte Carlo methods give. Notice that it is also the answer that gives minimum squared error on the training data. In fact, it gives zero error on the data. But still we expect the first answer to be better. If the process is Markov, we expect that the first answer will produce lower error on *future* data, even though the Monte Carlo answer is better on the existing data. ■

The above example 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-squared error on the training set, whereas batch TD(0) always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process. In general, the *maximum-likelihood estimate* of a parameter is the parameter value whose probability of generating the data is greatest. In this case, the maximum-likelihood estimate is the model of the Markov process formed in the obvious way from the observed episodes: the estimated transition probability from i to j is the fraction of observed transitions from i that went to j , and the associated expected reward is the average of the rewards observed on those transitions. Given this model, we can compute the estimate of the value function that would be exactly correct if the model were exactly correct. This is called the *certainty-equivalence estimate* because it is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated. In general, batch TD(0) converges to the certainty-equivalence estimate.

This helps explain why TD methods converge more quickly than Monte Carlo methods. In batch form, TD(0) is faster than Monte Carlo methods because it computes the true certainty-equivalence estimate. This explains the advantage of TD(0) shown in the batch results on the random walk task (Figure 6.8). The relationship to the certainty-equivalence estimate may also explain in part the speed advantage of nonbatch TD(0) (e.g., Figure 6.7). Although the nonbatch methods do not achieve either the certainty-equivalence

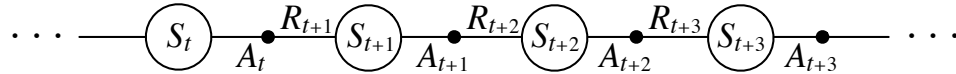
or the minimum squared-error estimates, they can be understood as moving roughly in these directions. Nonbatch TD(0) may be faster than constant- α MC because it is moving toward a better estimate, even though it is not getting all the way there. At the current time nothing more definite can be said about the relative efficiency of on-line TD and Monte Carlo methods.

Finally, it is worth noting that although the certainty-equivalence estimate is in some sense an optimal solution, it is almost never feasible to compute it directly. If N is the number of states, then just forming the maximum-likelihood estimate of the process may require N^2 memory, and computing the corresponding value function requires on the order of N^3 computational steps if done conventionally. In these terms it is indeed striking that TD methods can approximate the same solution using memory no more than N and repeated computations over the training set. On tasks with large state spaces, TD methods may be the only feasible way of approximating the certainty-equivalence solution.

6.4 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. As with Monte Carlo methods, we face the need to trade off exploration and exploitation, and again approaches fall into two main classes: on-policy and off-policy. In this section we present an on-policy TD control method.

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:



In the previous section we considered transitions from state to state and learned the values of states. Now we consider transitions from state-action pair to state-action pair, and learn the value of state-action pairs. Formally these cases are identical: they are both Markov chains with a reward process. The theorems assuring the convergence of state values under TD(0) also apply

```

Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$ 
Repeat (for each episode):
  Initialize  $S$ 
  Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
  Repeat (for each step of episode):
    Take action  $A$ , observe  $R, S'$ 
    Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
     $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$ 
     $S \leftarrow S'; A \leftarrow A'$ 
  until  $S$  is terminal

```

Figure 6.9: Sarsa: An on-policy TD control algorithm.

to the corresponding algorithm for action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \quad (6.5)$$

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. This rule uses every element of the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state–action pair to the next. This quintuple gives rise to the name *Sarsa* for the algorithm.

It is straightforward to design an on-policy control algorithm based on the Sarsa prediction method. 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 . The general form of the Sarsa control algorithm is given in Figure 6.9.

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. According to Satinder Singh (personal communication), 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 arranged, for example, with ϵ -greedy policies by setting $\epsilon = 1/t$), but this result has not yet been published in the literature.

Example 6.5: Windy Gridworld Figure 6.10 shows a standard gridworld, with start and goal states, but with one difference: there is a crosswind upward through the middle of the grid. The actions are the standard four—**up**, **down**, **right**, and **left**—but in the middle region the resultant next states are shifted upward by a “wind,” the strength of which varies from column to column. The strength of the wind is given below each column, in number of cells shifted

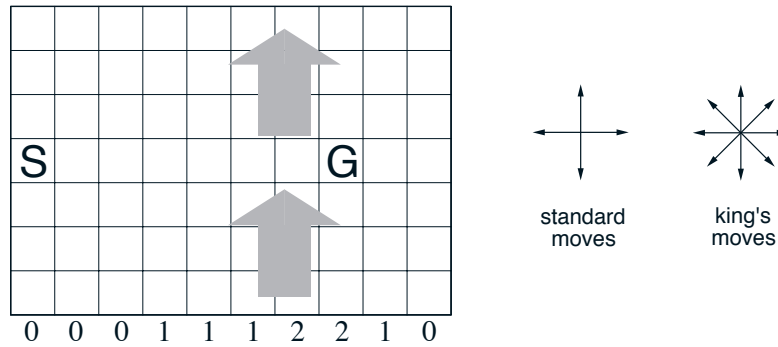


Figure 6.10: Gridworld in which movement is altered by a location-dependent, upward “wind.”

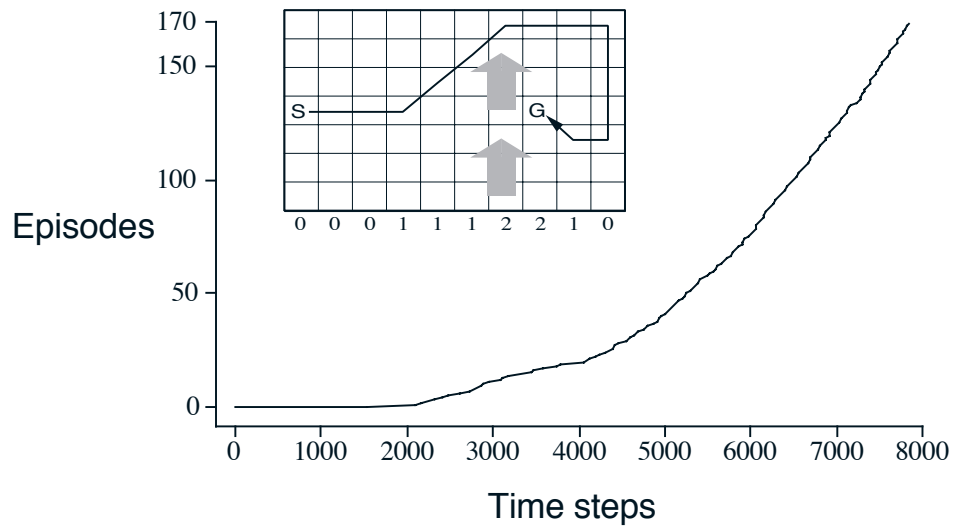


Figure 6.11: Results of Sarsa applied to the windy gridworld.

upward. For example, if you are one cell to the right of the goal, then the action `left` takes you to the cell just above the goal. Let us treat this as an undiscounted episodic task, with constant rewards of -1 until the goal state is reached. Figure 6.11 shows the result of applying ε -greedy Sarsa to this task, with $\varepsilon = 0.1$, $\alpha = 0.5$, and the initial values $Q(s, a) = 0$ for all s, a . The increasing slope of the graph shows that the goal is reached more and more quickly over time. By 8000 time steps, the greedy policy (shown inset) was long since optimal; continued ε -greedy exploration kept the average episode length at about 17 steps, two more than the minimum of 15. Note that Monte Carlo methods cannot easily be used on this task 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. Step-by-step 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. ■

6.5 Q-Learning: Off-Policy TD Control

One of the most important breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as *Q-learning* (Watkins, 1989). Its simplest form, *one-step Q-learning*, is defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]. \quad (6.6)$$

In this case, the learned action-value function, 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. The policy still has an effect in that it determines which state-action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. As we observed in Chapter 5, this is a minimal requirement in the sense that any method guaranteed to find optimal behavior in the general case must require it. 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_* . The Q-learning algorithm is shown in procedural form in Figure 6.12.

What is the backup diagram for Q-learning? The rule (6.6) updates a state-action pair, so the top node, the root of the backup, must be a small, filled action node. The backup is also *from* action nodes, maximizing over all those actions possible in the next state. Thus the bottom nodes of the backup diagram should be all these action nodes. Finally, remember that we indicate

```

Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$ 
Repeat (for each episode):
  Initialize  $S$ 
  Repeat (for each step of episode):
    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
    Take action  $A$ , observe  $R, S'$ 
     $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$ 
     $S \leftarrow S'$ ;
  until  $S$  is terminal

```

Figure 6.12: Q-learning: An off-policy TD control algorithm.

taking the maximum of these “next action” nodes with an arc across them (Figure 3.7). Can you guess now what the diagram is? If so, please do make a guess before turning to the answer in Figure 6.14.

Example 6.6: Cliff Walking This gridworld example compares Sarsa and Q-learning, highlighting the difference between on-policy (Sarsa) and off-policy (Q-learning) methods. Consider the gridworld shown in the upper part of Figure 6.13. This is a standard undiscounted, episodic task, with start and goal states, and the usual actions causing movement up, down, right, and left. Reward is -1 on all transitions except those into the the region marked “The Cliff.” Stepping into this region incurs a reward of -100 and sends the agent instantly back to the start. The lower part of the figure shows the performance of the Sarsa and Q-learning methods with ϵ -greedy action selection, $\epsilon = 0.1$. After an initial transient, Q-learning learns values for the optimal policy, that which travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off the cliff because of the ϵ -greedy action selection. Sarsa, on the other hand, takes the action selection into account and learns the longer but safer path through the upper part of the grid. Although Q-learning actually learns the values of the optimal policy, its on-line performance is worse than that of Sarsa, which learns the roundabout policy. Of course, if ϵ were gradually reduced, then both methods would asymptotically converge to the optimal policy. ■

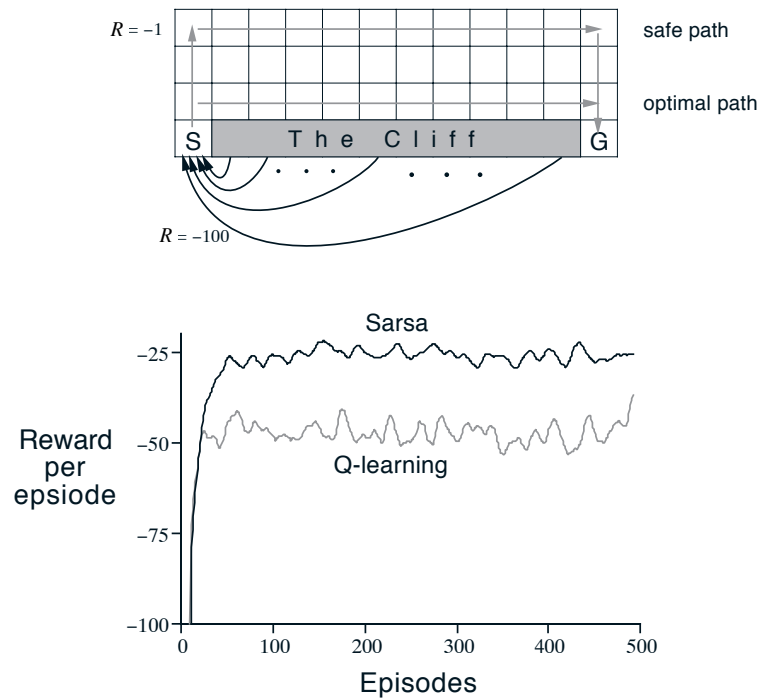


Figure 6.13: The cliff-walking task. The results are from a single run, but smoothed.

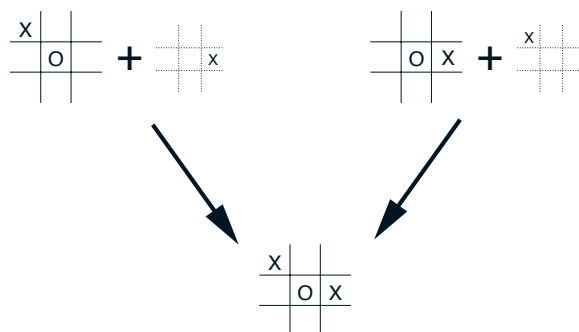


Figure 6.14: The backup diagram for Q-learning.

6.6 Games, Afterstates, and Other Special Cases

In this book we try to present a uniform approach to a wide class of tasks, but of course there are always exceptional tasks that are better treated in a specialized way. For example, our general approach involves learning an *action*-value function, but in Chapter 1 we presented a TD method for learning to play tic-tac-toe that learned something much more like a *state*-value function. If we look closely at that example, it becomes apparent that the function learned there is neither an action-value function nor a state-value function in the usual sense. A conventional state-value function evaluates states in which the agent has the option of selecting an action, but the state-value function used in tic-tac-toe evaluates board positions *after* the agent has made its move. Let us call these *afterstates*, and value functions over these, *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. For example, in games we typically know the immediate effects of our moves. We know for each possible chess move what the resulting position will be, but not how our opponent will reply. Afterstate value functions are a natural way to take advantage of this kind of knowledge and thereby produce a more efficient learning method.

The reason it is more efficient to design algorithms in terms of afterstates is apparent from the tic-tac-toe example. 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 this example:



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. Any learning about the position-move pair on the left would immediately transfer to the pair on the right.

Afterstates arise in many tasks, not just games. For example, in queuing

tasks there are actions such as assigning customers to servers, rejecting customers, or discarding information. In such cases the actions are in fact defined in terms of their immediate effects, which are completely known. For example, in the access-control queuing example described in the previous section, a more efficient learning method could be obtained by breaking the environment's dynamics into the immediate effect of the action, which is deterministic and completely known, and the unknown random processes having to do with the arrival and departure of customers. The afterstates would be the number of free servers after the action but before the random processes had produced the next conventional state. Learning an afterstate value function over the afterstates would enable all actions that produced the same number of free servers to share experience. This should result in a significant reduction in learning time.

It is impossible to describe all the possible kinds of specialized problems and corresponding specialized learning algorithms. However, the principles developed in this book should apply widely. For example, afterstate methods are still aptly described in terms of generalized policy iteration, with a policy and (afterstate) value function interacting in essentially the same way. In many cases one will still face the choice between on-policy and off-policy methods for managing the need for persistent exploration.

6.7 Summary

In this chapter we introduced a new kind of learning method, temporal-difference (TD) learning, and showed how it can be applied to the reinforcement learning problem. As usual, we divided the overall problem into a prediction problem and a control problem. 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) that we abstracted from dynamic programming. This is the idea that approximate policy and value functions should interact in such a way that they both move toward their optimal values.

One of the two processes making up GPI drives the value function to accurately predict returns for the current policy; this is the prediction problem. The other process drives the policy to improve locally (e.g., to be ε -greedy) with respect to the current value function. When the first process is based on experience, a complication arises concerning maintaining sufficient exploration. We have grouped the TD control methods according to whether they deal with this complication by using an on-policy or off-policy approach. Sarsa and actor-critic methods are on-policy methods, and Q-learning and R-learning

are off-policy methods.

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 on-line, with a minimal amount of computation, to experience generated from interaction with an environment; they can be expressed nearly completely by single equations that can be implemented with small computer programs. In the next few chapters we extend these algorithms, making them slightly more complicated and significantly more powerful. All the new algorithms will retain the essence of those introduced here: they will be able to process experience on-line, with relatively little computation, and they will be driven by TD errors. The special cases of TD methods introduced in the present chapter should rightly be called *one-step*, *tabular*, *model-free* TD methods. In the next three chapters we extend them to multistep forms (a link to Monte Carlo methods), forms using function approximation rather than tables (a link to artificial neural networks), and forms that include a model of the environment (a link to planning and dynamic programming).

Finally, in this chapter we have discussed TD methods entirely within the context of reinforcement learning problems, but TD methods are actually more general than this. They are general methods for learning to make long-term predictions about dynamical systems. For example, TD methods may be relevant to predicting financial data, life spans, election outcomes, weather patterns, animal behavior, demands on power stations, or customer purchases. It was only when TD methods were analyzed as pure prediction methods, independent of their use in reinforcement learning, that their theoretical properties first came to be well understood. Even so, these other potential applications of TD learning methods have not yet been extensively explored.

Bibliographical and Historical Remarks

As we outlined in Chapter 1, the idea of TD learning has its early roots in animal learning psychology and artificial intelligence, most notably the work of Samuel (1959) and Klopff (1972). Samuel's work is described as a case study in Section 15.2. Also related to TD learning are Holland's (1975, 1976) early ideas about consistency among value predictions. These influenced one of the authors (Barto), who was a graduate student from 1970 to 1975 at the University of Michigan, where Holland was teaching. Holland's ideas led to a number of TD-related systems, including the work of Booker (1982) and the bucket brigade of Holland (1986), which is related to Sarsa as discussed below.

6.1–2 Most of the specific material from these sections is from Sutton (1988),

including the TD(0) algorithm, the random walk example, and the term “temporal-difference learning.” The characterization of the relationship to dynamic programming and Monte Carlo methods was influenced by Watkins (1989), Werbos (1987), and others. The use of backup diagrams here and in other chapters is new to this book. Example 6.4 is due to Sutton, but has not been published before.

Tabular TD(0) was proved to converge in the mean by Sutton (1988) and with probability 1 by Dayan (1992), based on the work of Watkins and Dayan (1992). These results were extended and strengthened by Jaakkola, Jordan, and Singh (1994) and Tsitsiklis (1994) by using extensions of the powerful existing theory of stochastic approximation. Other extensions and generalizations are covered in the next two chapters.

6.3 The optimality of the TD algorithm under batch training was established by Sutton (1988). The term *certainty equivalence* is from the adaptive control literature (e.g., Goodwin and Sin, 1984). Illuminating this result is Barnard’s (1993) derivation of the TD algorithm as a combination of one step of an incremental method for learning a model of the Markov chain and one step of a method for computing predictions from the model.

6.4 The Sarsa algorithm was first explored by Rummerly and Niranjan (1994), who called it *modified Q-learning*. The name “Sarsa” was introduced by Sutton (1996). The convergence of one-step tabular Sarsa (the form treated in this chapter) has been proved by Satinder Singh (personal communication). The “windy gridworld” example was suggested by Tom Kalt.

Holland’s (1986) bucket brigade idea evolved into an algorithm closely related to Sarsa. The original idea of the bucket brigade involved chains of rules triggering each other; it focused on passing credit back from the current rule to the rules that triggered it. Over time, the bucket brigade came to be more like TD learning in passing credit back to any temporally preceding rule, not just to the ones that triggered the current rule. The modern form of the bucket brigade, when simplified in various natural ways, is nearly identical to one-step Sarsa, as detailed by Wilson (1994).

6.5 Q-learning was introduced by Watkins (1989), whose outline of a convergence proof was later made rigorous by Watkins and Dayan (1992). More general convergence results were proved by Jaakkola, Jordan, and Singh (1994) and Tsitsiklis (1994).

- 6.6** R-learning is due to Schwartz (1993). Mahadevan (1996), Tadepalli and Ok (1994), and Bertsekas and Tsitsiklis (1996) have studied reinforcement learning for undiscounted continuing tasks. In the literature, the undiscounted continuing case is often called the case of maximizing “average reward per time step” or the “average-reward case.” The name R-learning was probably meant to be the alphabetic successor to Q-learning, but we prefer to think of it as a reference to the learning of *relative* values. The access-control queuing example was suggested by the work of Carlström and Nordström (1997).

Exercises

Exercise 6.1 This is an exercise to help develop your intuition about why TD methods are often more efficient than Monte Carlo methods. Consider the driving home example and how it is addressed by TD and Monte Carlo methods. Can you imagine a scenario in which a TD update would be better on average than an Monte Carlo update? Give an example scenario—a description of past experience and a current state—in which you would expect the TD update to be better. Here’s a hint: Suppose you have lots of experience driving home from work. Then you move to a new building and a new parking lot (but you still enter the highway at the same place). Now you are starting to learn predictions for the new building. Can you see why TD updates are likely to be much better, at least initially, in this case? Might the same sort of thing happen in the original task?

Exercise 6.2 From Figure 6.6, it appears that the first episode results in a change in only $V(A)$. What does this tell you about what happened on the first episode? Why was only the estimate for this one state changed? By exactly how much was it changed?

Exercise 6.3 The specific results shown in Figure 6.7 are dependent on the value of the step-size parameter, α . Do you think the conclusions about which algorithm is better would be affected if a wider range of α values were used? Is there a different, fixed value of α at which either algorithm would have performed significantly better than shown? Why or why not?

Exercise 6.4 In Figure 6.7, the RMS error of the TD method seems to go down and then up again, particularly at high α ’s. What could have caused this? Do you think this always occurs, or might it be a function of how the approximate value function was initialized?

Exercise 6.5 Above we stated that the true values for the random walk task

are $\frac{1}{6}$, $\frac{2}{6}$, $\frac{3}{6}$, $\frac{4}{6}$, and $\frac{5}{6}$, for states A through E. Describe at least two different ways that these could have been computed. Which would you guess we actually used? Why?

Exercise 6.6: Windy Gridworld with King's Moves Re-solve the windy gridworld task assuming eight possible actions, including the diagonal moves, rather than the usual four. How much better can you do with the extra actions? Can you do even better by including a ninth action that causes no movement at all other than that caused by the wind?

Exercise 6.7: Stochastic Wind Resolve the windy gridworld task with King's moves, assuming that the effect of the wind, if there is any, is stochastic, sometimes varying by 1 from the mean values given for each column. That is, a third of the time you move exactly according to these values, as in the previous exercise, but also a third of the time you move one cell above that, and another third of the time you move one cell below that. For example, if you are one cell to the right of the goal and you move `left`, then one-third of the time you move one cell above the goal, one-third of the time you move two cells above the goal, and one-third of the time you move to the goal.

Exercise 6.8 What is the backup diagram for Sarsa?

Exercise 6.9 Why is Q-learning considered an *off-policy* control method?

Exercise 6.10 Consider the learning algorithm that 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. That is, consider the algorithm otherwise like Q-learning except with the update rule

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

Is this new method an on-policy or off-policy method? What is the backup diagram for this algorithm? Given the same amount of experience, would you expect this method to work better or worse than Sarsa? What other considerations might impact the comparison of this method with Sarsa?

Exercise 6.11 Describe how the task of Jack's Car Rental (Example 4.2) could be reformulated in terms of afterstates. Why, in terms of this specific task, would such a reformulation be likely to speed convergence?

