Example 8.6: RTDP on the Racetrack The racetrack problem of Exercise 5.12 (page 111) is a stochastic optimal path problem. Comparing RTDP and the conventional DP value iteration algorithm on an example racetrack problem illustrates some of the advantages of on-policy trajectory sampling.

Recall from the exercise that an agent has to learn how to drive a car around a turn like those shown in Figure 5.5 and cross the finish line as quickly as possible while staying on the track. Start states are all the zero-speed states on the starting line; the goal states are all the states that can be reached in one time step by crossing the finish line from inside the track. Unlike Exercise 5.12, here there is no limit on the car's speed, so the state set is potentially infinite. However, the set of states that can be reached from the set of start states via any policy is finite and can be considered to be the state set of the problem. Each episode begins in a randomly selected start state and ends when the car crosses the finish line. The rewards are -1 for each step until the car crosses the finish line. If the car hits the track boundary, it is moved back to a random start state, and the episode continues.

A racetrack similar to the small racetrack on the left of Figure 5.5 has 9,115 states reachable from start states by any policy, only 599 of which are relevant, meaning that they are reachable from some start state via some optimal policy. (The number of relevant states was estimated by counting the states visited while executing optimal actions for 10^7 episodes.)

The table below compares solving this task by conventional DP and by RTDP. These results are averages over 25 runs, each begun with a different random number seed. Conventional DP in this case is value iteration using exhaustive sweeps of the state set, with values updated one state at a time in place, meaning that the update for each state uses the most recent values of the other states (This is the Gauss-Seidel version of value iteration, which was found to be approximately twice as fast as the Jacobi version on this problem. See Section 4.8.) No special attention was paid to the ordering of the updates; other orderings could have produced faster convergence. Initial values were all zero for each run of both methods. DP was judged to have converged when the maximum change in a state value over a sweep was less than 10^{-4}, and RTDP was judged to have converged when the average time to cross the finish line over 20 episodes appeared to stabilize at an asymptotic number of steps. This version of RTDP updated only the value of the current state on each step.

	DP	RTDP
Average computation to convergence	28 sweeps	4000 episodes
Average number of updates to convergence	252,784	127,600
Average number of updates per episode	—	31.9
% of states updated $\leq$ 100 times	—	98.45
% of states updated $\leq$ 10 times	—	80.51
% of states updated 0 times	—	3.18

Both methods produced policies averaging between 14 and 15 steps to cross the finish line, but RTDP required only roughly half of the updates that DP did. This is the result of RTDP's on-policy trajectory sampling. Whereas the value of every state was updated

in each sweep of DP, RTDP focused updates on fewer states. In an average run, RTDP updated the values of 98.45% of the states no more than 100 times and 80.51% of the states no more than 10 times; the values of about 290 states were not updated at all in an average run. ■

Another advantage of RTDP is that as the value function approaches the optimal value function v_*, the policy used by the agent to generate trajectories approaches an optimal policy because it is always greedy with respect to the current value function. This is in contrast to the situation in conventional value iteration. In practice, value iteration terminates when the value function changes by only a small amount in a sweep, which is how we terminated it to obtain the results in the table above. At this point, the value function closely approximates v_*, and a greedy policy is close to an optimal policy. However, it is possible that policies that are greedy with respect to the latest value function were optimal, or nearly so, long before value iteration terminates. (Recall from Chapter 4 that optimal policies can be greedy with respect to many different value functions, not just v_*.) Checking for the emergence of an optimal policy before value iteration converges is not a part of the conventional DP algorithm and requires considerable additional computation.

In the racetrack example, by running many test episodes after each DP sweep, with actions selected greedily according to the result of that sweep, it was possible to estimate the earliest point in the DP computation at which the approximated optimal evaluation function was good enough so that the corresponding greedy policy was nearly optimal. For this racetrack, a close-to-optimal policy emerged after 15 sweeps of value iteration, or after 136,725 value-iteration updates. This is considerably less than the 252,784 updates DP needed to converge to v_*, but still more than the 127,600 updates RTDP required.

Although these simulations are certainly not definitive comparisons of the RTDP with conventional sweep-based value iteration, they illustrate some of advantages of on-policy trajectory sampling. Whereas conventional value iteration continued to update the value of all the states, RTDP strongly focused on subsets of the states that were relevant to the problem's objective. This focus became increasingly narrow as learning continued. Because the convergence theorem for RTDP applies to the simulations, we know that RTDP eventually would have focused only on relevant states, i.e., on states making up optimal paths. RTDP achieved nearly optimal control with about 50% of the computation required by sweep-based value iteration.

8.8 Planning at Decision Time

Planning can be used in at least two ways. The one we have considered so far in this chapter, typified by dynamic programming and Dyna, is to use planning to gradually improve a policy or value function on the basis of simulated experience obtained from a model (either a sample or a distribution model). Selecting actions is then a matter of comparing the current state's action values obtained from a table in the tabular case we have thus far considered, or by evaluating a mathematical expression in the approximate methods we consider in Part II below. Well before an action is selected for any current state S_t, planning has played a part in improving the table entries, or the function

approximation parameters, needed to select actions for many states, including S_t. Used this way, planning is not focused on the current state. We call planning used in this way *background planning*.

The other way to use planning is to begin and complete it *after* encountering each new state S_t, as a computation whose output is the selection of a single action A_t; on the next step planning begins anew with S_{t+1} to produce A_{t+1}, and so on. The simplest, and almost degenerate, example of this use of planning is when only state values are available, and an action is selected by comparing the values of model-predicted next states for each action (or by comparing the values of afterstates as in the tic-tac-toe example in Chapter 1). More generally, planning used in this way can look much deeper than one-step-ahead and evaluate action choices leading to many different predicted state and reward trajectories. Unlike the first use of planning, here planning focuses on a particular state. We call this *decision-time planning*.

These two ways of thinking about planning—using simulated experience to gradually improve a policy or value function, or using simulated experience to select an action for the current state—can blend together in natural and interesting ways, but they have tended to be studied separately, and that is a good way to first understand them. Let us now take a closer look at decision-time planning.

Even when planning is only done at decision time, we can still view it, as we did in Section 8.1, as proceeding from simulated experience to updates and values, and ultimately to a policy. It is just that now the values and policy are specific to the current state and the action choices available there, so much so that the values and policy created by the planning process are typically discarded after being used to select the current action. In many applications this is not a great loss because there are very many states and we are unlikely to return to the same state for a long time. In general, one may want to do a mix of both: focus planning on the current state *and* store the results of planning so as to be that much farther along should one return to the same state later. Decision-time planning is most useful in applications in which fast responses are not required. In chess playing programs, for example, one may be permitted seconds or minutes of computation for each move, and strong programs may plan dozens of moves ahead within this time. On the other hand, if low latency action selection is the priority, then one is generally better off doing planning in the background to compute a policy that can then be rapidly applied to each newly encountered state.

8.9 Heuristic Search

The classical state-space planning methods in artificial intelligence are decision-time planning methods collectively known as *heuristic search*. In heuristic search, for each state encountered, a large tree of possible continuations is considered. The approximate value function is applied to the leaf nodes and then backed up toward the current state at the root. The backing up within the search tree is just the same as in the expected updates with maxes (those for v_* and q_*) discussed throughout this book. The backing up stops at the state–action nodes for the current state. Once the backed-up values of these nodes are computed, the best of them is chosen as the current action, and then all backed-up values are discarded.

In conventional heuristic search no effort is made to save the backed-up values by changing the approximate value function. In fact, the value function is generally designed by people and never changed as a result of search. However, it is natural to consider allowing the value function to be improved over time, using either the backed-up values computed during heuristic search or any of the other methods presented throughout this book. In a sense we have taken this approach all along. Our greedy, ε-greedy, and UCB (Section 2.7) action-selection methods are not unlike heuristic search, albeit on a smaller scale. For example, to compute the greedy action given a model and a state-value function, we must look ahead from each possible action to each possible next state, take into account the rewards and estimated values, and then pick the best action. Just as in conventional heuristic search, this process computes backed-up values of the possible actions, but does not attempt to save them. Thus, heuristic search can be viewed as an extension of the idea of a greedy policy beyond a single step.

The point of searching deeper than one step is to obtain better action selections. If one has a perfect model and an imperfect action-value function, then in fact deeper search will usually yield better policies.[2] Certainly, if the search is all the way to the end of the episode, then the effect of the imperfect value function is eliminated, and the action determined in this way must be optimal. If the search is of sufficient depth k such that γ^k is very small, then the actions will be correspondingly near optimal. On the other hand, the deeper the search, the more computation is required, usually resulting in a slower response time. A good example is provided by Tesauro's grandmaster-level backgammon player, TD-Gammon (Section 16.1). This system used TD learning to learn an afterstate value function through many games of self-play, using a form of heuristic search to make its moves. As a model, TD-Gammon used a priori knowledge of the probabilities of dice rolls and the assumption that the opponent always selected the actions that TD-Gammon rated as best for it. Tesauro found that the deeper the heuristic search, the better the moves made by TD-Gammon, but the longer it took to make each move. Backgammon has a large branching factor, yet moves must be made within a few seconds. It was only feasible to search ahead selectively a few steps, but even so the search resulted in significantly better action selections.

We should not overlook the most obvious way in which heuristic search focuses updates: on the current state. Much of the effectiveness of heuristic search is due to its search tree being tightly focused on the states and actions that might immediately follow the current state. You may spend more of your life playing chess than checkers, but when you play checkers, it pays to think about checkers and about your particular checkers position, your likely next moves, and successor positions. No matter how you select actions, it is these states and actions that are of highest priority for updates and where you most urgently want your approximate value function to be accurate. Not only should your computation be preferentially devoted to imminent events, but so should your limited memory resources. In chess, for example, there are far too many possible positions to store distinct value estimates for each of them, but chess programs based on heuristic search can easily store distinct estimates for the millions of positions they encounter

[2]There are interesting exceptions to this (see Pearl, 1984).

looking ahead from a single position. This great focusing of memory and computational resources on the current decision is presumably the reason why heuristic search can be so effective.

The distribution of updates can be altered in similar ways to focus on the current state and its likely successors. As a limiting case we might use exactly the methods of heuristic search to construct a search tree, and then perform the individual, one-step updates from bottom up, as suggested by Figure 8.9. If the updates are ordered in this way and a tabular representation is used, then exactly the same overall update would be achieved as in depth-first heuristic search. Any state-space search can be viewed in this way as the piecing together of a large number of individual one-step updates. Thus, the performance improvement observed with deeper searches is not due to the use of multistep updates as such. Instead, it is due to the focus and concentration of updates on states and actions immediately downstream from the current state. By devoting a large amount of computation specifically relevant to the candidate actions, decision-time planning can produce better decisions than can be produced by relying on unfocused updates.

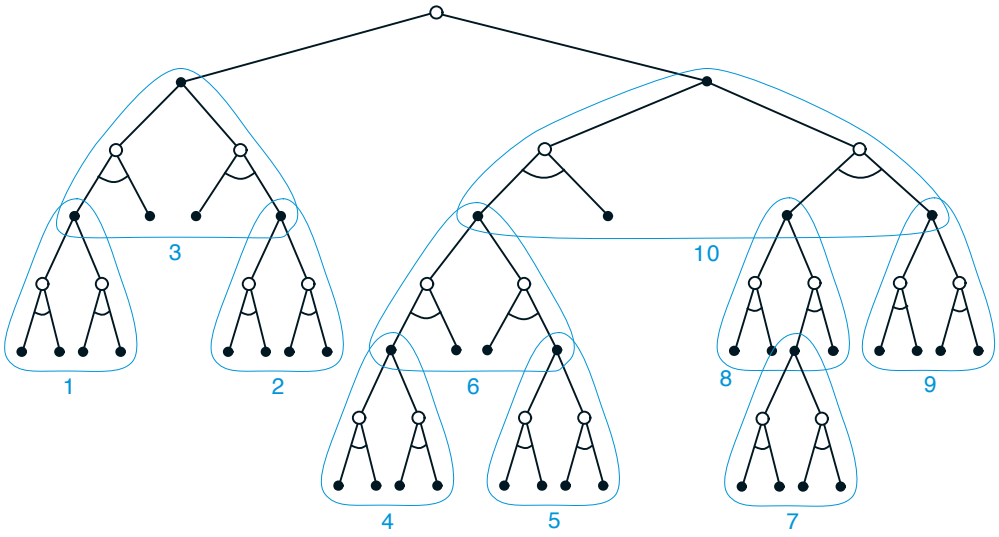


Figure 8.9: Heuristic search can be implemented as a sequence of one-step updates (shown here outlined in blue) backing up values from the leaf nodes toward the root. The ordering shown here is for a selective depth-first search.

8.10 Rollout Algorithms

Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to simulated trajectories that all begin at the current environment state. They estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible action and then follow the given policy. When the action-value estimates are considered to be accurate enough, the action (or one of the

actions) having the highest estimated value is executed, after which the process is carried out anew from the resulting next state. As explained by Tesauro and Galperin (1997), who experimented with rollout algorithms for playing backgammon, the term "rollout" comes from estimating the value of a backgammon position by playing out, i.e., "rolling out," the position many times to the game's end with randomly generated sequences of dice rolls, where the moves of both players are made by some fixed policy.

Unlike the Monte Carlo control algorithms described in Chapter 5, the goal of a rollout algorithm is not to estimate a complete optimal action-value function, q_*, or a complete action-value function, q_π, for a given policy π. Instead, they produce Monte Carlo estimates of action values only for each current state and for a given policy usually called the *rollout policy*. As decision-time planning algorithms, rollout algorithms make immediate use of these action-value estimates, then discard them. This makes rollout algorithms relatively simple to implement because there is no need to sample outcomes for every state-action pair, and there is no need to approximate a function over either the state space or the state-action space.

What then do rollout algorithms accomplish? The policy improvement theorem described in Section 4.2 tells us that given any two policies π and π' that are identical except that $\pi'(s) = a \neq \pi(s)$ for some state s, if $q_\pi(s, a) \geq v_\pi(s)$, then policy π' is as good as, or better, than π. Moreover, if the inequality is strict, then π' is in fact better than π. This applies to rollout algorithms where s is the current state and π is the rollout policy. Averaging the returns of the simulated trajectories produces estimates of $q_\pi(s, a')$ for each action $a' \in \mathcal{A}(s)$. Then the policy that selects an action in s that maximizes these estimates and thereafter follows π is a good candidate for a policy that improves over π. The result is like one step of the policy-iteration algorithm of dynamic programming discussed in Section 4.3 (though it is more like one step of *asynchronous* value iteration described in Section 4.5 because it changes the action for just the current state).

In other words, the aim of a rollout algorithm is to improve upon the rollout policy; not to find an optimal policy. Experience has shown that rollout algorithms can be surprisingly effective. For example, Tesauro and Galperin (1997) were surprised by the dramatic improvements in backgammon playing ability produced by the rollout method. In some applications, a rollout algorithm can produce good performance even if the rollout policy is completely random. But the performance of the improved policy depends on properties of the rollout policy and the ranking of actions produced by the Monte Carlo value estimates. Intuition suggests that the better the rollout policy and the more accurate the value estimates, the better the policy produced by a rollout algorithm is likely be (but see Gelly and Silver, 2007).

This involves important tradeoffs because better rollout policies typically mean that more time is needed to simulate enough trajectories to obtain good value estimates. As decision-time planning methods, rollout algorithms usually have to meet strict time constraints. The computation time needed by a rollout algorithm depends on the number of actions that have to be evaluated for each decision, the number of time steps in the simulated trajectories needed to obtain useful sample returns, the time it takes the rollout policy to make decisions, and the number of simulated trajectories needed to obtain good Monte Carlo action-value estimates.

Balancing these factors is important in any application of rollout methods, though there are several ways to ease the challenge. Because the Monte Carlo trials are independent of one another, it is possible to run many trials in parallel on separate processors. Another approach is to truncate the simulated trajectories short of complete episodes, correcting the truncated returns by means of a stored evaluation function (which brings into play all that we have said about truncated returns and updates in the preceding chapters). It is also possible, as Tesauro and Galperin (1997) suggest, to monitor the Monte Carlo simulations and prune away candidate actions that are unlikely to turn out to be the best, or whose values are close enough to that of the current best that choosing them instead would make no real difference (though Tesauro and Galperin point out that this would complicate a parallel implementation).

We do not ordinarily think of rollout algorithms as *learning* algorithms because they do not maintain long-term memories of values or policies. However, these algorithms take advantage of some of the features of reinforcement learning that we have emphasized in this book. As instances of Monte Carlo control, they estimate action values by averaging the returns of a collection of sample trajectories, in this case trajectories of simulated interactions with a sample model of the environment. In this way they are like reinforcement learning algorithms in avoiding the exhaustive sweeps of dynamic programming by trajectory sampling, and in avoiding the need for distribution models by relying on sample, instead of expected, updates. Finally, rollout algorithms take advantage of the policy improvement property by acting greedily with respect to the estimated action values.

8.11 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a recent and strikingly successful example of decision-time planning. At its base, MCTS is a rollout algorithm as described above, but enhanced by the addition of a means for accumulating value estimates obtained from the Monte Carlo simulations in order to successively direct simulations toward more highly-rewarding trajectories. MCTS is largely responsible for the improvement in computer Go from a weak amateur level in 2005 to a grandmaster level (6 dan or more) in 2015. Many variations of the basic algorithm have been developed, including a variant that we discuss in Section 16.6 that was critical for the stunning 2016 victories of the program AlphaGo over an 18-time world champion Go player. MCTS has proved to be effective in a wide variety of competitive settings, including general game playing (e.g., see Finnsson and Björnsson, 2008; Genesereth and Thielscher, 2014), but it is not limited to games; it can be effective for single-agent sequential decision problems if there is an environment model simple enough for fast multistep simulation.

MCTS is executed after encountering each new state to select the agent's action for that state; it is executed again to select the action for the next state, and so on. As in a rollout algorithm, each execution is an iterative process that simulates many trajectories starting from the current state and running to a terminal state (or until discounting makes any further reward negligible as a contribution to the return). The core idea of MCTS is to successively focus multiple simulations starting at the current state by

extending the initial portions of trajectories that have received high evaluations from earlier simulations. MCTS does not have to retain approximate value functions or policies from one action selection to the next, though in many implementations it retains selected action values likely to be useful for its next execution.

For the most part, the actions in the simulated trajectories are generated using a simple policy, usually called a rollout policy as it is for simpler rollout algorithms. When both the rollout policy and the model do not require a lot of computation, many simulated trajectories can be generated in a short period of time. As in any tabular Monte Carlo method, the value of a state–action pair is estimated as the average of the (simulated) returns from that pair. Monte Carlo value estimates are maintained only for the subset of state–action pairs that are most likely to be reached in a few steps, which form a tree rooted at the current state, as illustrated in Figure 8.10. MCTS incrementally extends the tree by adding nodes representing states that look promising based on the results of the simulated trajectories. Any simulated trajectory will pass through the tree and then exit it at some leaf node. Outside the tree and at the leaf nodes the rollout policy is used for action selections, but at the states inside the tree something better is possible. For these states we have value estimates for at least some of the actions, so we can pick among them using an informed policy, called the *tree policy*, that balances exploration

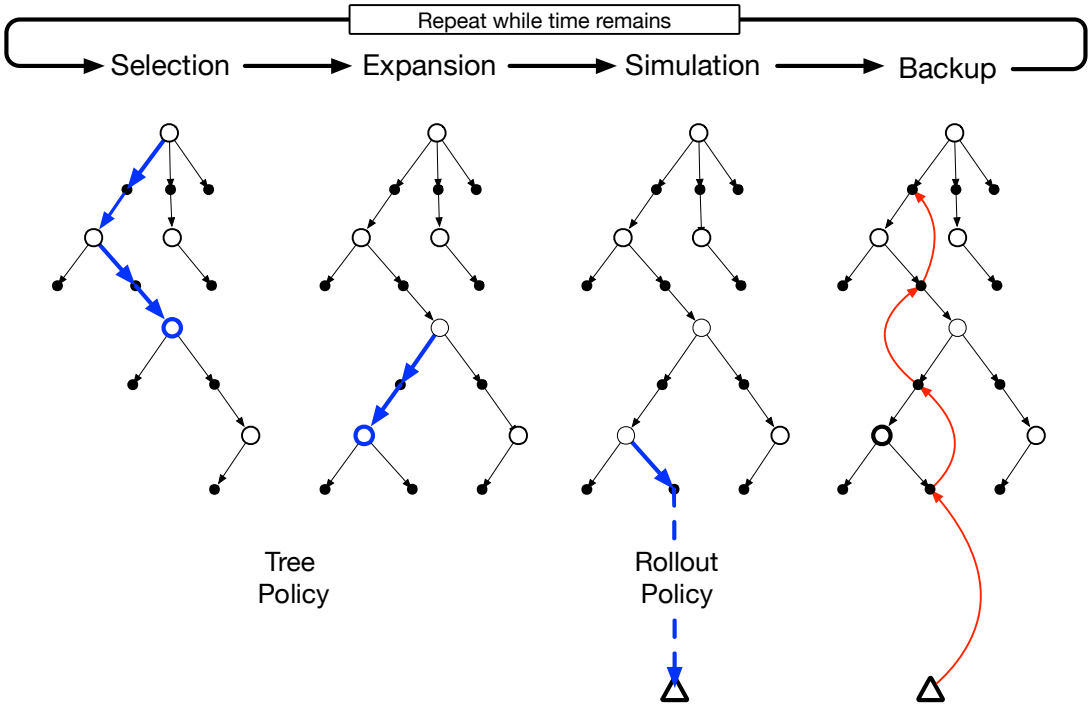


Figure 8.10: Monte Carlo Tree Search. When the environment changes to a new state, MCTS executes as many iterations as possible before an action needs to be selected, incrementally building a tree whose root node represents the current state. Each iteration consists of the four operations **Selection**, **Expansion** (though possibly skipped on some iterations), **Simulation**, and **Backup**, as explained in the text and illustrated by the bold arrows in the trees. Adapted from Chaslot, Bakkes, Szita, and Spronck (2008).

and exploitation. For example, the tree policy could select actions using an ε-greedy or UCB selection rule (Chapter 2).

In more detail, each iteration of a basic version of MCTS consists of the following four steps as illustrated in Figure 8.10:

1. **Selection.** Starting at the root node, a *tree policy* based on the action values attached to the edges of the tree traverses the tree to select a leaf node.

2. **Expansion.** On some iterations (depending on details of the application), the tree is expanded from the selected leaf node by adding one or more child nodes reached from the selected node via unexplored actions.

3. **Simulation.** From the selected node, or from one of its newly-added child nodes (if any), simulation of a complete episode is run with actions selected by the rollout policy. The result is a Monte Carlo trial with actions selected first by the tree policy and beyond the tree by the rollout policy.

4. **Backup.** The return generated by the simulated episode is backed up to update, or to initialize, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS. No values are saved for the states and actions visited by the rollout policy beyond the tree. Figure 8.10 illustrates this by showing a backup from the terminal state of the simulated trajectory directly to the state–action node in the tree where the rollout policy began (though in general, the entire return over the simulated trajectory is backed up to this state–action node).

MCTS continues executing these four steps, starting each time at the tree's root node, until no more time is left, or some other computational resource is exhausted. Then, finally, an action from the root node (which still represents the current state of the environment) is selected according to some mechanism that depends on the accumulated statistics in the tree; for example, it may be an action having the largest action value of all the actions available from the root state, or perhaps the action with the largest visit count to avoid selecting outliers. This is the action MCTS actually selects. After the environment transitions to a new state, MCTS is run again, sometimes starting with a tree of a single root node representing the new state, but often starting with a tree containing any descendants of this node left over from the tree constructed by the previous execution of MCTS; all the remaining nodes are discarded, along with the action values associated with them.

MCTS was first proposed to select moves in programs playing two-person competitive games, such as Go. For game playing, each simulated episode is one complete play of the game in which both players select actions by the tree and rollout policies. Section 16.6 describes an extension of MCTS used in the AlphaGo program that combines the Monte Carlo evaluations of MCTS with action values learned by a deep artificial neural network via self-play reinforcement learning.

Relating MCTS to the reinforcement learning principles we describe in this book provides some insight into how it achieves such impressive results. At its base, MCTS is a decision-time planning algorithm based on Monte Carlo control applied to simulations

that start from the root state; that is, it is a kind of rollout algorithm as described in the previous section. It therefore benefits from online, incremental, sample-based value estimation and policy improvement. Beyond this, it saves action-value estimates attached to the tree edges and updates them using reinforcement learning's sample updates. This has the effect of focusing the Monte Carlo trials on trajectories whose initial segments are common to high-return trajectories previously simulated. Further, by incrementally expanding the tree, MCTS effectively grows a lookup table to store a partial action-value function, with memory allocated to the estimated values of state–action pairs visited in the initial segments of high-yielding sample trajectories. MCTS thus avoids the problem of globally approximating an action-value function while it retains the benefit of using past experience to guide exploration.

The striking success of decision-time planning by MCTS has deeply influenced artificial intelligence, and many researchers are studying modifications and extensions of the basic procedure for use in both games and single-agent applications.

8.12 Summary of the Chapter

Planning requires a model of the environment. A *distribution model* consists of the probabilities of next states and rewards for possible actions; a sample model produces single transitions and rewards generated according to these probabilities. Dynamic programming requires a distribution model because it uses *expected updates*, which involve computing expectations over all the possible next states and rewards. A *sample model*, on the other hand, is what is needed to simulate interacting with the environment during which *sample updates*, like those used by many reinforcement learning algorithms, can be used. Sample models are generally much easier to obtain than distribution models.

We have presented a perspective emphasizing the surprisingly close relationships between planning optimal behavior and learning optimal behavior. Both involve estimating the same value functions, and in both cases it is natural to update the estimates incrementally, in a long series of small backing-up operations. This makes it straightforward to integrate learning and planning processes simply by allowing both to update the same estimated value function. In addition, any of the learning methods can be converted into planning methods simply by applying them to simulated (model-generated) experience rather than to real experience. In this case learning and planning become even more similar; they are possibly identical algorithms operating on two different sources of experience.

It is straightforward to integrate incremental planning methods with acting and model-learning. Planning, acting, and model-learning interact in a circular fashion (as in the diagram on page 162), each producing what the other needs to improve; no other interaction among them is either required or prohibited. The most natural approach is for all processes to proceed asynchronously and in parallel. If the processes must share computational resources, then the division can be handled almost arbitrarily—by whatever organization is most convenient and efficient for the task at hand.

In this chapter we have touched upon a number of dimensions of variation among state-space planning methods. One dimension is the variation in the size of updates. The

smaller the updates, the more incremental the planning methods can be. Among the smallest updates are one-step sample updates, as in Dyna. Another important dimension is the distribution of updates, that is, of the focus of search. Prioritized sweeping focuses backward on the predecessors of states whose values have recently changed. On-policy trajectory sampling focuses on states or state–action pairs that the agent is likely to encounter when controlling its environment. This can allow computation to skip over parts of the state space that are irrelevant to the prediction or control problem. Real-time dynamic programming, an on-policy trajectory sampling version of value iteration, illustrates some of the advantages this strategy has over conventional sweep-based policy iteration.

Planning can also focus forward from pertinent states, such as states actually encountered during an agent-environment interaction. The most important form of this is when planning is done at decision time, that is, as part of the action-selection process. Classical heuristic search as studied in artificial intelligence is an example of this. Other examples are rollout algorithms and Monte Carlo Tree Search that benefit from online, incremental, sample-based value estimation and policy improvement.

8.13 Summary of Part I: Dimensions

This chapter concludes Part I of this book. In it we have tried to present reinforcement learning not as a collection of individual methods, but as a coherent set of ideas cutting across methods. Each idea can be viewed as a dimension along which methods vary. The set of such dimensions spans a large space of possible methods. By exploring this space at the level of dimensions we hope to obtain the broadest and most lasting understanding. In this section we use the concept of dimensions in method space to recapitulate the view of reinforcement learning developed so far in this book.

All of the methods we have explored so far in this book have three key ideas in common: first, they all seek to estimate value functions; second, they all operate by backing up values along actual or possible state trajectories; and third, they all follow the general strategy of generalized policy iteration (GPI), meaning that they maintain an approximate value function and an approximate policy, and they continually try to improve each on the basis of the other. These three ideas are central to the subjects covered in this book. We suggest that value functions, backing up value updates, and GPI are powerful organizing principles potentially relevant to any model of intelligence, whether artificial or natural.

Two of the most important dimensions along which the methods vary are shown in Figure 8.11. These dimensions have to do with the kind of update used to improve the value function. The horizontal dimension is whether they are sample updates (based on a sample trajectory) or expected updates (based on a distribution of possible trajectories). Expected updates require a distribution model, whereas sample updates need only a sample model, or can be done from actual experience with no model at all (another dimension of variation). The vertical dimension of Figure 8.11 corresponds to the depth of updates, that is, to the degree of bootstrapping. At three of the four corners of the space are the three primary methods for estimating values: dynamic programming, TD, and Monte Carlo. Along the left edge of the space are the sample-update methods,

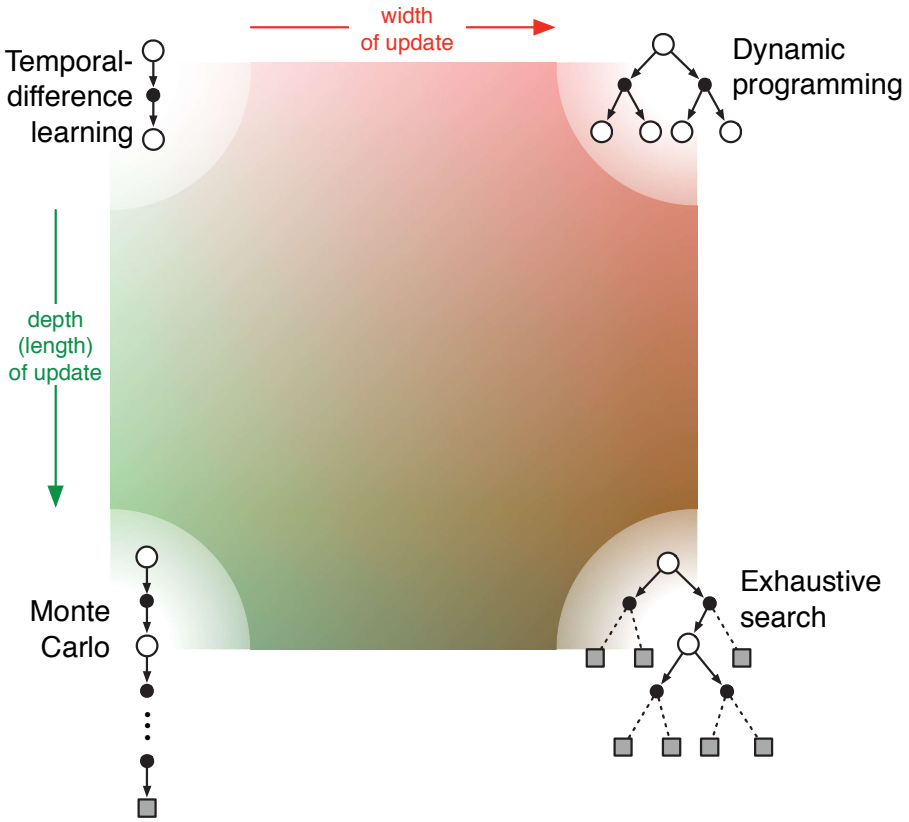


Figure 8.11: A slice through the space of reinforcement learning methods, highlighting the two of the most important dimensions explored in Part I of this book: the depth and width of the updates.

ranging from one-step TD updates to full-return Monte Carlo updates. Between these is a spectrum including methods based on n-step updates (and in Chapter 12 we will extend this to mixtures of n-step updates such as the λ-updates implemented by eligibility traces).

Dynamic programming methods are shown in the extreme upper-right corner of the space because they involve one-step expected updates. The lower-right corner is the extreme case of expected updates so deep that they run all the way to terminal states (or, in a continuing task, until discounting has reduced the contribution of any further rewards to a negligible level). This is the case of exhaustive search. Intermediate methods along this dimension include heuristic search and related methods that search and update up to a limited depth, perhaps selectively. There are also methods that are intermediate along the horizontal dimension. These include methods that mix expected and sample updates, as well as the possibility of methods that mix samples and distributions within a single update. The interior of the square is filled in to represent the space of all such intermediate methods.

A third dimension that we have emphasized in this book is the binary distinction between on-policy and off-policy methods. In the former case, the agent learns the value function for the policy it is currently following, whereas in the latter case it learns the

value function for the policy for a different policy, often the one that the agent currently thinks is best. The policy generating behavior is typically different from what is currently thought best because of the need to explore. This third dimension might be visualized as perpendicular to the plane of the page in Figure 8.11.

In addition to the three dimensions just discussed, we have identified a number of others throughout the book:

Definition of return Is the task episodic or continuing, discounted or undiscounted?

Action values vs. state values vs. afterstate values What kind of values should be estimated? If only state values are estimated, then either a model or a separate policy (as in actor–critic methods) is required for action selection.

Action selection/exploration How are actions selected to ensure a suitable trade-off between exploration and exploitation? We have considered only the simplest ways to do this: ε-greedy, optimistic initialization of values, soft-max, and upper confidence bound.

Synchronous vs. asynchronous Are the updates for all states performed simultaneously or one by one in some order?

Real vs. simulated Should one update based on real experience or simulated experience? If both, how much of each?

Location of updates What states or state–action pairs should be updated? Model-free methods can choose only among the states and state–action pairs actually encountered, but model-based methods can choose arbitrarily. There are many possibilities here.

Timing of updates Should updates be done as part of selecting actions, or only afterward?

Memory for updates How long should updated values be retained? Should they be retained permanently, or only while computing an action selection, as in heuristic search?

Of course, these dimensions are neither exhaustive nor mutually exclusive. Individual algorithms differ in many other ways as well, and many algorithms lie in several places along several dimensions. For example, Dyna methods use both real and simulated experience to affect the same value function. It is also perfectly sensible to maintain multiple value functions computed in different ways or over different state and action representations. These dimensions do, however, constitute a coherent set of ideas for describing and exploring a wide space of possible methods.

The most important dimension not mentioned here, and not covered in Part I of this book, is that of function approximation. Function approximation can be viewed as an orthogonal spectrum of possibilities ranging from tabular methods at one extreme through state aggregation, a variety of linear methods, and then a diverse set of nonlinear methods. This dimension is explored in Part II.

Bibliographical and Historical Remarks

8.1 The overall view of planning and learning presented here has developed gradually over a number of years, in part by the authors (Sutton, 1990, 1991a, 1991b; Barto, Bradtke, and Singh, 1991, 1995; Sutton and Pinette, 1985; Sutton and Barto, 1981b); it has been strongly influenced by Agre and Chapman (1990; Agre 1988), Bertsekas and Tsitsiklis (1989), Singh (1993), and others. The authors were also strongly influenced by psychological studies of latent learning (Tolman, 1932) and by psychological views of the nature of thought (e.g., Galanter and Gerstenhaber, 1956; Craik, 1943; Campbell, 1960; Dennett, 1978). In Part III of the book, Section 14.6 relates model-based and model-free methods to psychological theories of learning and behavior, and Section 15.11 discusses ideas about how the brain might implement these types of methods.

8.2 The terms *direct* and *indirect*, which we use to describe different kinds of reinforcement learning, are from the adaptive control literature (e.g., Goodwin and Sin, 1984), where they are used to make the same kind of distinction. The term *system identification* is used in adaptive control for what we call *model-learning* (e.g., Goodwin and Sin, 1984; Ljung and Söderstrom, 1983; Young, 1984). The Dyna architecture is due to Sutton (1990), and the results in this and the next section are based on results reported there. Barto and Singh (1990) consider some of the issues in comparing direct and indirect reinforcement learning methods. Early work extending Dyna to linear function approximation was done by Sutton, Szepesvári, Geramifard, and Bowling (2008) and by Parr, Li, Taylor, Painter-Wakefield, and Littman (2008).

8.3 There have been several works with model-based reinforcement learning that take the idea of exploration bonuses and optimistic initialization to its logical extreme, in which all incompletely explored choices are assumed maximally rewarding and optimal paths are computed to test them. The E^3 algorithm of Kearns and Singh (2002) and the R-max algorithm of Brafman and Tennenholtz (2003) are guaranteed to find a near-optimal solution in time polynomial in the number of states and actions. This is usually too slow for practical algorithms but is probably the best that can be done in the worst case.

8.4 Prioritized sweeping was developed simultaneously and independently by Moore and Atkeson (1993) and Peng and Williams (1993). The results in the box on page 170 are due to Peng and Williams (1993). The results in the box on page 171 are due to Moore and Atkeson. Key subsequent work in this area includes that by McMahan and Gordon (2005) and by van Seijen and Sutton (2013).

8.5 This section was strongly influenced by the experiments of Singh (1993).

8.6–7 Trajectory sampling has implicitly been a part of reinforcement learning from the outset, but it was most explicitly emphasized by Barto, Bradtke, and Singh (1995) in their introduction of RTDP. They recognized that Korf's (1990) *learning*

*real-time A** (LRTA*) algorithm is an asynchronous DP algorithm that applies to stochastic problems as well as the deterministic problems on which Korf focused. Beyond LRTA*, RTDP includes the option of updating the values of many states in the time intervals between the execution of actions. Barto et al. (1995) proved the convergence result described here by combining Korf's (1990) convergence proof for LRTA* with the result of Bertsekas (1982) (also Bertsekas and Tsitsiklis, 1989) ensuring convergence of asynchronous DP for stochastic shortest path problems in the undiscounted case. Combining model-learning with RTDP is called *Adaptive* RTDP, also presented by Barto et al. (1995) and discussed by Barto (2011).

8.9 For further reading on heuristic search, the reader is encouraged to consult texts and surveys such as those by Russell and Norvig (2009) and Korf (1988). Peng and Williams (1993) explored a forward focusing of updates much as is suggested in this section.

8.10 Abramson's (1990) expected-outcome model is a rollout algorithm applied to two-person games in which the play of both simulated players is random. He argued that even with random play, it is a "powerful heuristic" that is "precise, accurate, easily estimable, efficiently calculable, and domain-independent." Tesauro and Galperin (1997) demonstrated the effectiveness of rollout algorithms for improving the play of backgammon programs, adopting the term "rollout" from its use in evaluating backgammon positions by playing out positions with different randomly generating sequences of dice rolls. Bertsekas, Tsitsiklis, and Wu (1997) examine rollout algorithms applied to combinatorial optimization problems, and Bertsekas (2013) surveys their use in discrete deterministic optimization problems, remarking that they are "often surprisingly effective."

8.11 The central ideas of MCTS were introduced by Coulom (2006) and by Kocsis and Szepesvári (2006). They built upon previous research with Monte Carlo planning algorithms as reviewed by these authors. Browne, Powley, Whitehouse, Lucas, Cowling, Rohlfshagen, Tavener, Perez, Samothrakis, and Colton (2012) is an excellent survey of MCTS methods and their applications. David Silver contributed to the ideas and presentation in this section.

Part II: Approximate Solution Methods

In the second part of the book we extend the tabular methods presented in the first part to apply to problems with arbitrarily large state spaces. In many of the tasks to which we would like to apply reinforcement learning the state space is combinatorial and enormous; the number of possible camera images, for example, is much larger than the number of atoms in the universe. In such cases we cannot expect to find an optimal policy or the optimal value function even in the limit of infinite time and data; our goal instead is to find a good approximate solution using limited computational resources. In this part of the book we explore such approximate solution methods.

The problem with large state spaces is not just the memory needed for large tables, but the time and data needed to fill them accurately. In many of our target tasks, almost every state encountered will never have been seen before. To make sensible decisions in such states it is necessary to generalize from previous encounters with different states that are in some sense similar to the current one. In other words, the key issue is that of *generalization*. How can experience with a limited subset of the state space be usefully generalized to produce a good approximation over a much larger subset?

Fortunately, generalization from examples has already been extensively studied, and we do not need to invent totally new methods for use in reinforcement learning. To some extent we need only combine reinforcement learning methods with existing generalization methods. The kind of generalization we require is often called *function approximation* because it takes examples from a desired function (e.g., a value function) and attempts to generalize from them to construct an approximation of the entire function. Function approximation is an instance of *supervised learning*, the primary topic studied in machine learning, artificial neural networks, pattern recognition, and statistical curve fitting. In theory, any of the methods studied in these fields can be used in the role of function approximator within reinforcement learning algorithms, although in practice some fit more easily into this role than others.

Reinforcement learning with function approximation involves a number of new issues that do not normally arise in conventional supervised learning, such as nonstationarity, bootstrapping, and delayed targets. We introduce these and other issues successively over the five chapters of this part. Initially we restrict attention to on-policy training, treating in Chapter 9 the prediction case, in which the policy is given and only its value function is approximated, and then in Chapter 10 the control case, in which an approximation to the optimal policy is found. The challenging problem of off-policy learning with function approximation is treated in Chapter 11. In each of these three chapters we will have

to return to first principles and re-examine the objectives of the learning to take into account function approximation. Chapter 12 introduces and analyzes the algorithmic mechanism of *eligibility traces*, which dramatically improves the computational properties of multi-step reinforcement learning methods in many cases. The final chapter of this part explores a different approach to control, *policy-gradient methods*, which approximate the optimal policy directly and need never form an approximate value function (although they may be much more efficient if they do approximate a value function as well the policy).

Chapter 9

On-policy Prediction with Approximation

In this chapter, we begin our study of function approximation in reinforcement learning by considering its use in estimating the state-value function from on-policy data, that is, in approximating v_π from experience generated using a known policy π. The novelty in this chapter is that the approximate value function is represented not as a table but as a parameterized functional form with weight vector $\mathbf{w} \in \mathbb{R}^d$. We will write $\hat{v}(s,\mathbf{w}) \approx v_\pi(s)$ for the approximate value of state s given weight vector $\mathbf{w}$. For example, $\hat{v}$ might be a linear function in features of the state, with $\mathbf{w}$ the vector of feature weights. More generally, $\hat{v}$ might be the function computed by a multi-layer artificial neural network, with $\mathbf{w}$ the vector of connection weights in all the layers. By adjusting the weights, any of a wide range of different functions can be implemented by the network. Or $\hat{v}$ might be the function computed by a decision tree, where $\mathbf{w}$ is all the numbers defining the split points and leaf values of the tree. Typically, the number of weights (the dimensionality of $\mathbf{w}$) is much less than the number of states ($d \ll |\mathcal{S}|$), and changing one weight changes the estimated value of many states. Consequently, when a single state is updated, the change generalizes from that state to affect the values of many other states. Such *generalization* makes the learning potentially more powerful but also potentially more difficult to manage and understand.

Perhaps surprisingly, extending reinforcement learning to function approximation also makes it applicable to partially observable problems, in which the full state is not available to the agent. If the parameterized function form for $\hat{v}$ does not allow the estimated value to depend on certain aspects of the state, then it is just as if those aspects are unobservable. In fact, all the theoretical results for methods using function approximation presented in this part of the book apply equally well to cases of partial observability. What function approximation can't do, however, is augment the state representation with memories of past observations. Some such possible further extensions are discussed briefly in Section 17.3.

9.1 Value-function Approximation

All of the prediction methods covered in this book have been described as updates to an estimated value function that shift its value at particular states toward a "backed-up value," or *update target*, for that state. Let us refer to an individual update by the notation $s \mapsto u$, where s is the state updated and u is the update target that s's estimated value is shifted toward. For example, the Monte Carlo update for value prediction is $S_t \mapsto G_t$, the TD(0) update is $S_t \mapsto R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t)$, and the n-step TD update is $S_t \mapsto G_{t:t+n}$. In the DP (dynamic programming) policy-evaluation update, $s \mapsto \mathbb{E}_\pi[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) \mid S_t = s]$, an arbitrary state s is updated, whereas in the other cases the state encountered in actual experience, S_t, is updated.

It is natural to interpret each update as specifying an example of the desired input–output behavior of the value function. In a sense, the update $s \mapsto u$ means that the estimated value for state s should be more like the update target u. Up to now, the actual update has been trivial: the table entry for s's estimated value has simply been shifted a fraction of the way toward u, and the estimated values of all other states were left unchanged. Now we permit arbitrarily complex and sophisticated methods to implement the update, and updating at s generalizes so that the estimated values of many other states are changed as well. Machine learning methods that learn to mimic input–output examples in this way are called *supervised learning* methods, and when the outputs are numbers, like u, the process is often called *function approximation*. Function approximation methods expect to receive examples of the desired input–output behavior of the function they are trying to approximate. We use these methods for value prediction simply by passing to them the $s \mapsto u$ of each update as a training example. We then interpret the approximate function they produce as an estimated value function.

Viewing each update as a conventional training example in this way enables us to use any of a wide range of existing function approximation methods for value prediction. In principle, we can use any method for supervised learning from examples, including artificial neural networks, decision trees, and various kinds of multivariate regression. However, not all function approximation methods are equally well suited for use in reinforcement learning. The most sophisticated artificial neural network and statistical methods all assume a static training set over which multiple passes are made. In reinforcement learning, however, it is important that learning be able to occur online, while the agent interacts with its environment or with a model of its environment. To do this requires methods that are able to learn efficiently from incrementally acquired data. In addition, reinforcement learning generally requires function approximation methods able to handle nonstationary target functions (target functions that change over time). For example, in control methods based on GPI (generalized policy iteration) we often seek to learn q_π while π changes. Even if the policy remains the same, the target values of training examples are nonstationary if they are generated by bootstrapping methods (DP and TD learning). Methods that cannot easily handle such nonstationarity are less suitable for reinforcement learning.

9.2 The Prediction Objective ($\overline{\text{VE}}$)

Up to now we have not specified an explicit objective for prediction. In the tabular case a continuous measure of prediction quality was not necessary because the learned value function could come to equal the true value function exactly. Moreover, the learned values at each state were decoupled—an update at one state affected no other. But with genuine approximation, an update at one state affects many others, and it is not possible to get the values of all states exactly correct. By assumption we have far more states than weights, so making one state's estimate more accurate invariably means making others' less accurate. We are obligated then to say which states we care most about. We must specify a state distribution $\mu(s) \geq 0$, $\sum_s \mu(s) = 1$, representing how much we care about the error in each state s. By the error in a state s we mean the square of the difference between the approximate value $\hat{v}(s,\mathbf{w})$ and the true value $v_\pi(s)$. Weighting this over the state space by μ, we obtain a natural objective function, the *mean square value error*, denoted $\overline{\text{VE}}$:

$$\overline{\text{VE}}(\mathbf{w}) \doteq \sum_{s\in\mathcal{S}} \mu(s)\Big[v_\pi(s) - \hat{v}(s,\mathbf{w})\Big]^2. \tag{9.1}$$

The square root of this measure, the root $\overline{\text{VE}}$, gives a rough measure of how much the approximate values differ from the true values and is often used in plots. Often $\mu(s)$ is chosen to be the fraction of time spent in s. Under on-policy training this is called the *on-policy distribution*; we focus entirely on this case in this chapter. In continuing tasks, the on-policy distribution is the stationary distribution under π.

The on-policy distribution in episodic tasks

In an episodic task, the on-policy distribution is a little different in that it depends on how the initial states of episodes are chosen. Let $h(s)$ denote the probability that an episode begins in each state s, and let $\eta(s)$ denote the number of time steps spent, on average, in state s in a single episode. Time is spent in a state s if episodes start in s, or if transitions are made into s from a preceding state $\bar{s}$ in which time is spent:

$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(\bar{s}) \sum_a \pi(a|\bar{s}) p(s\,|\,\bar{s}, a), \quad \text{for all } s \in \mathcal{S}. \tag{9.2}$$

This system of equations can be solved for the expected number of visits $\eta(s)$. The on-policy distribution is then the fraction of time spent in each state normalized to sum to one:

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}, \quad \text{for all } s \in \mathcal{S}. \tag{9.3}$$

This is the natural choice without discounting. If there is discounting ($\gamma < 1$) it should be treated as a form of termination, which can be done simply by including a factor of γ in the second term of (9.2).

The two cases, continuing and episodic, behave similarly, but with approximation they must be treated separately in formal analyses, as we will see repeatedly in this part of the book. This completes the specification of the learning objective.

It is not completely clear that the $\overline{\text{VE}}$ is the right performance objective for reinforcement learning. Remember that our ultimate purpose—the reason we are learning a value function—is to find a better policy. The best value function for this purpose is not necessarily the best for minimizing $\overline{\text{VE}}$. Nevertheless, it is not yet clear what a more useful alternative goal for value prediction might be. For now, we will focus on $\overline{\text{VE}}$.

An ideal goal in terms of $\overline{\text{VE}}$ would be to find a *global optimum*, a weight vector $\mathbf{w}^*$ for which $\overline{\text{VE}}(\mathbf{w}^*) \leq \overline{\text{VE}}(\mathbf{w})$ for all possible $\mathbf{w}$. Reaching this goal is sometimes possible for simple function approximators such as linear ones, but is rarely possible for complex function approximators such as artificial neural networks and decision trees. Short of this, complex function approximators may seek to converge instead to a *local optimum*, a weight vector $\mathbf{w}^*$ for which $\overline{\text{VE}}(\mathbf{w}^*) \leq \overline{\text{VE}}(\mathbf{w})$ for all $\mathbf{w}$ in some neighborhood of $\mathbf{w}^*$. Although this guarantee is only slightly reassuring, it is typically the best that can be said for nonlinear function approximators, and often it is enough. Still, for many cases of interest in reinforcement learning there is no guarantee of convergence to an optimum, or even to within a bounded distance of an optimum. Some methods may in fact diverge, with their $\overline{\text{VE}}$ approaching infinity in the limit.

In the last two sections we outlined a framework for combining a wide range of reinforcement learning methods for value prediction with a wide range of function approximation methods, using the updates of the former to generate training examples for the latter. We also described a $\overline{\text{VE}}$ performance measure which these methods may aspire to minimize. The range of possible function approximation methods is far too large to cover all, and anyway too little is known about most of them to make a reliable evaluation or recommendation. Of necessity, we consider only a few possibilities. In the rest of this chapter we focus on function approximation methods based on gradient principles, and on linear gradient-descent methods in particular. We focus on these methods in part because we consider them to be particularly promising and because they reveal key theoretical issues, but also because they are simple and our space is limited.

9.3 Stochastic-gradient and Semi-gradient Methods

We now develop in detail one class of learning methods for function approximation in value prediction, those based on stochastic gradient descent (SGD). SGD methods are among the most widely used of all function approximation methods and are particularly well suited to online reinforcement learning.

In gradient-descent methods, the weight vector is a column vector with a fixed number of real valued components, $\mathbf{w} \doteq (w_1, w_2, \ldots, w_d)^\top$,[1] and the approximate value function $\hat{v}(s,\mathbf{w})$ is a differentiable function of $\mathbf{w}$ for all $s \in \mathcal{S}$. We will be updating $\mathbf{w}$ at each of a series of discrete time steps, $t = 0, 1, 2, 3, \ldots$, so we will need a notation $\mathbf{w}_t$ for the

[1] The $^\top$ denotes transpose, needed here to turn the horizontal row vector in the text into a vertical column vector; in this book vectors are generally taken to be column vectors unless explicitly written out horizontally or transposed.

weight vector at each step. For now, let us assume that, on each step, we observe a new example $S_t \mapsto v_\pi(S_t)$ consisting of a (possibly randomly selected) state S_t and its true value under the policy. These states might be successive states from an interaction with the environment, but for now we do not assume so. Even though we are given the exact, correct values, $v_\pi(S_t)$ for each S_t, there is still a difficult problem because our function approximator has limited resources and thus limited resolution. In particular, there is generally no $\mathbf{w}$ that gets all the states, or even all the examples, exactly correct. In addition, we must generalize to all the other states that have not appeared in examples.

We assume that states appear in examples with the same distribution, μ, over which we are trying to minimize the $\overline{\text{VE}}$ as given by (9.1). A good strategy in this case is to try to minimize error on the observed examples. *Stochastic gradient-descent* (SGD) methods do this by adjusting the weight vector after each example by a small amount in the direction that would most reduce the error on that example:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t - \frac{1}{2}\alpha\nabla\Big[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)\Big]^2 \tag{9.4}$$

$$= \mathbf{w}_t + \alpha\Big[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)\Big]\nabla\hat{v}(S_t,\mathbf{w}_t), \tag{9.5}$$

where α is a positive step-size parameter, and $\nabla f(\mathbf{w})$, for any scalar expression $f(\mathbf{w})$ that is a function of a vector (here $\mathbf{w}$), denotes the column vector of partial derivatives of the expression with respect to the components of the vector:

$$\nabla f(\mathbf{w}) \doteq \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \ldots, \frac{\partial f(\mathbf{w})}{\partial w_d}\right)^\top. \tag{9.6}$$

This derivative vector is the *gradient* of f with respect to $\mathbf{w}$. SGD methods are "gradient descent" methods because the overall step in $\mathbf{w}_t$ is proportional to the negative gradient of the example's squared error (9.4). This is the direction in which the error falls most rapidly. Gradient descent methods are called "stochastic" when the update is done, as here, on only a single example, which might have been selected stochastically. Over many examples, making small steps, the overall effect is to minimize an average performance measure such as the $\overline{\text{VE}}$.

It may not be immediately apparent why SGD takes only a small step in the direction of the gradient. Could we not move all the way in this direction and completely eliminate the error on the example? In many cases this could be done, but usually it is not desirable. Remember that we do not seek or expect to find a value function that has zero error for all states, but only an approximation that balances the errors in different states. If we completely corrected each example in one step, then we would not find such a balance. In fact, the convergence results for SGD methods assume that α decreases over time. If it decreases in such a way as to satisfy the standard stochastic approximation conditions (2.7), then the SGD method (9.5) is guaranteed to converge to a local optimum.

We turn now to the case in which the target output, here denoted $U_t \in \mathbb{R}$, of the tth training example, $S_t \mapsto U_t$, is not the true value, $v_\pi(S_t)$, but some, possibly random, approximation to it. For example, U_t might be a noise-corrupted version of $v_\pi(S_t)$, or it might be one of the bootstrapping targets using $\hat{v}$ mentioned in the previous section. In

these cases we cannot perform the exact update (9.5) because $v_\pi(S_t)$ is unknown, but we can approximate it by substituting U_t in place of $v_\pi(S_t)$. This yields the following general SGD method for state-value prediction:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[U_t - \hat{v}(S_t,\mathbf{w}_t)\Big]\nabla\hat{v}(S_t,\mathbf{w}_t). \tag{9.7}$$

If U_t is an *unbiased* estimate, that is, if $\mathbb{E}[U_t|S_t=s] = v_\pi(s)$, for each t, then $\mathbf{w}_t$ is guaranteed to converge to a local optimum under the usual stochastic approximation conditions (2.7) for decreasing α.

For example, suppose the states in the examples are the states generated by interaction (or simulated interaction) with the environment using policy π. Because the true value of a state is the expected value of the return following it, the Monte Carlo target $U_t \doteq G_t$ is by definition an unbiased estimate of $v_\pi(S_t)$. With this choice, the general SGD method (9.7) converges to a locally optimal approximation to $v_\pi(S_t)$. Thus, the gradient-descent version of Monte Carlo state-value prediction is guaranteed to find a locally optimal solution. Pseudocode for a complete algorithm is shown in the box below.

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

One does not obtain the same guarantees if a bootstrapping estimate of $v_\pi(S_t)$ is used as the target U_t in (9.7). Bootstrapping targets such as n-step returns $G_{t:t+n}$ or the DP target $\sum_{a,s',r}\pi(a|S_t)p(s',r\,|\,S_t,a)[r+\gamma\hat{v}(s',\mathbf{w}_t)]$ all depend on the current value of the weight vector $\mathbf{w}_t$, which implies that they will be biased and that they will not produce a true gradient-descent method. One way to look at this is that the key step from (9.4) to (9.5) relies on the target being independent of $\mathbf{w}_t$. This step would not be valid if a bootstrapping estimate were used in place of $v_\pi(S_t)$. Bootstrapping methods are not in fact instances of true gradient descent (Barnard, 1993). They take into account the effect of changing the weight vector $\mathbf{w}_t$ on the estimate, but ignore its effect on the target. They include only a part of the gradient and, accordingly, we call them *semi-gradient methods*.

Although semi-gradient (bootstrapping) methods do not converge as robustly as gradient methods, they do converge reliably in important cases such as the linear case discussed in the next section. Moreover, they offer important advantages that make them often clearly preferred. One reason for this is that they typically enable significantly faster learning, as we have seen in Chapters 6 and 7. Another is that they enable learning to

be continual and online, without waiting for the end of an episode. This enables them to be used on continuing problems and provides computational advantages. A prototypical semi-gradient method is semi-gradient TD(0), which uses $U_t \doteq R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w})$ as its target. Complete pseudocode for this method is given in the box below.

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S is terminal

State aggregation is a simple form of generalizing function approximation in which states are grouped together, with one estimated value (one component of the weight vector $\mathbf{w}$) for each group. The value of a state is estimated as its group's component, and when the state is updated, that component alone is updated. State aggregation is a special case of SGD (9.7) in which the gradient, $\nabla\hat{v}(S_t,\mathbf{w}_t)$, is 1 for S_t's group's component and 0 for the other components.

Example 9.1: State Aggregation on the 1000-state Random Walk Consider a 1000-state version of the random walk task (Examples 6.2 and 7.1 on pages 125 and 144). The states are numbered from 1 to 1000, left to right, and all episodes begin near the center, in state 500. State transitions are from the current state to one of the 100 neighboring states to its left, or to one of the 100 neighboring states to its right, all with equal probability. Of course, if the current state is near an edge, then there may be fewer than 100 neighbors on that side of it. In this case, all the probability that would have gone into those missing neighbors goes into the probability of terminating on that side (thus, state 1 has a 0.5 chance of terminating on the left, and state 950 has a 0.25 chance of terminating on the right). As usual, termination on the left produces a reward of -1, and termination on the right produces a reward of $+1$. All other transitions have a reward of zero. We use this task as a running example throughout this section.

Figure 9.1 shows the true value function v_π for this task. It is nearly a straight line, curving very slightly toward the horizontal for the last 100 states at each end. Also shown is the final approximate value function learned by the gradient Monte-Carlo algorithm with state aggregation after 100,000 episodes with a step size of $\alpha = 2 \times 10^{-5}$. For the state aggregation, the 1000 states were partitioned into 10 groups of 100 states each (i.e., states 1–100 were one group, states 101–200 were another, and so on). The staircase effect

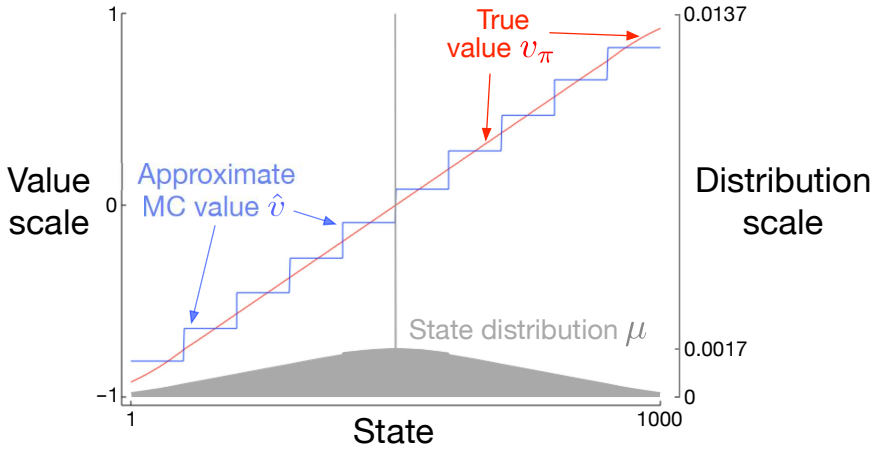


Figure 9.1: Function approximation by state aggregation on the 1000-state random walk task, using the gradient Monte Carlo algorithm (page 202).

shown in the figure is typical of state aggregation; within each group, the approximate value is constant, and it changes abruptly from one group to the next. These approximate values are close to the global minimum of the $\overline{\text{VE}}$ (9.1).

Some of the details of the approximate values are best appreciated by reference to the state distribution μ for this task, shown in the lower portion of the figure with a right-side scale. State 500, in the center, is the first state of every episode, but is rarely visited again. On average, about 1.37% of the time steps are spent in the start state. The states reachable in one step from the start state are the second most visited, with about 0.17% of the time steps being spent in each of them. From there μ falls off almost linearly, reaching about 0.0147% at the extreme states 1 and 1000. The most visible effect of the distribution is on the leftmost groups, whose values are clearly shifted higher than the unweighted average of the true values of states within the group, and on the rightmost groups, whose values are clearly shifted lower. This is due to the states in these areas having the greatest asymmetry in their weightings by μ. For example, in the leftmost group, state 100 is weighted more than 3 times more strongly than state 1. Thus the estimate for the group is biased toward the true value of state 100, which is higher than the true value of state 1. ■

9.4 Linear Methods

One of the most important special cases of function approximation is that in which the approximate function, $\hat{v}(\cdot,\mathbf{w})$, is a linear function of the weight vector, $\mathbf{w}$. Corresponding to every state s, there is a real-valued vector $\mathbf{x}(s) \doteq (x_1(s), x_2(s), \ldots, x_d(s))^\top$, with the same number of components as $\mathbf{w}$. Linear methods approximate the state-value function

by the inner product between $\mathbf{w}$ and $\mathbf{x}(s)$:

$$\hat{v}(s,\mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) \doteq \sum_{i=1}^{d} w_i x_i(s). \tag{9.8}$$

In this case the approximate value function is said to be *linear in the weights*, or simply *linear*.

The vector $\mathbf{x}(s)$ is called a *feature vector* representing state s. Each component $x_i(s)$ of $\mathbf{x}(s)$ is the value of a function $x_i : \mathcal{S} \to \mathbb{R}$. We think of a *feature* as the entirety of one of these functions, and we call its value for a state s a *feature of* s. For linear methods, features are *basis functions* because they form a linear basis for the set of approximate functions. Constructing d-dimensional feature vectors to represent states is the same as selecting a set of d basis functions. Features may be defined in many different ways; we cover a few possibilities in the next sections.

It is natural to use SGD updates with linear function approximation. The gradient of the approximate value function with respect to $\mathbf{w}$ in this case is

$$\nabla \hat{v}(s,\mathbf{w}) = \mathbf{x}(s).$$

Thus, in the linear case the general SGD update (9.7) reduces to a particularly simple form:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[U_t - \hat{v}(S_t,\mathbf{w}_t)\Big]\mathbf{x}(S_t).$$

Because it is so simple, the linear SGD case is one of the most favorable for mathematical analysis. Almost all useful convergence results for learning systems of all kinds are for linear (or simpler) function approximation methods.

In particular, in the linear case there is only one optimum (or, in degenerate cases, one set of equally good optima), and thus any method that is guaranteed to converge to or near a local optimum is automatically guaranteed to converge to or near the global optimum. For example, the gradient Monte Carlo algorithm presented in the previous section converges to the global optimum of the $\overline{\text{VE}}$ under linear function approximation if α is reduced over time according to the usual conditions.

The semi-gradient TD(0) algorithm presented in the previous section also converges under linear function approximation, but this does not follow from general results on SGD; a separate theorem is necessary. The weight vector converged to is also not the global optimum, but rather a point near the local optimum. It is useful to consider this important case in more detail, specifically for the continuing case. The update at each time t is

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha\Big(R_{t+1} + \gamma \mathbf{w}_t^\top \mathbf{x}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t\Big)\mathbf{x}_t \\ &= \mathbf{w}_t + \alpha\Big(R_{t+1}\mathbf{x}_t - \mathbf{x}_t\big(\mathbf{x}_t - \gamma \mathbf{x}_{t+1}\big)^\top \mathbf{w}_t\Big),\end{aligned} \tag{9.9}$$

where here we have used the notational shorthand $\mathbf{x}_t = \mathbf{x}(S_t)$. Once the system has reached steady state, for any given $\mathbf{w}_t$, the expected next weight vector can be written

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t), \tag{9.10}$$

where

$$\mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t] \in \mathbb{R}^d \quad \text{and} \quad \mathbf{A} \doteq \mathbb{E}\Big[\mathbf{x}_t\big(\mathbf{x}_t - \gamma\mathbf{x}_{t+1}\big)^\top\Big] \in \mathbb{R}^{d\times d} \tag{9.11}$$

From (9.10) it is clear that, if the system converges, it must converge to the weight vector $\mathbf{w}_{\text{TD}}$ at which

$$\begin{aligned} & \mathbf{b} - \mathbf{A}\mathbf{w}_{\text{TD}} = \mathbf{0} \\ \Rightarrow \quad & \mathbf{b} = \mathbf{A}\mathbf{w}_{\text{TD}} \\ \Rightarrow \quad & \mathbf{w}_{\text{TD}} \doteq \mathbf{A}^{-1}\mathbf{b}. \end{aligned} \tag{9.12}$$

This quantity is called the *TD fixed point*. In fact linear semi-gradient TD(0) converges to this point. Some of the theory proving its convergence, and the existence of the inverse above, is given in the box.

Proof of Convergence of Linear TD(0)

What properties assure convergence of the linear TD(0) algorithm (9.9)? Some insight can be gained by rewriting (9.10) as

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = (\mathbf{I} - \alpha\mathbf{A})\mathbf{w}_t + \alpha\mathbf{b}. \tag{9.13}$$

Note that the matrix $\mathbf{A}$ multiplies the weight vector $\mathbf{w}_t$ and not $\mathbf{b}$; only $\mathbf{A}$ is important to convergence. To develop intuition, consider the special case in which $\mathbf{A}$ is a diagonal matrix. If any of the diagonal elements are negative, then the corresponding diagonal element of $\mathbf{I} - \alpha\mathbf{A}$ will be greater than one, and the corresponding component of $\mathbf{w}_t$ will be amplified, which will lead to divergence if continued. On the other hand, if the diagonal elements of $\mathbf{A}$ are all positive, then α can be chosen smaller than one over the largest of them, such that $\mathbf{I} - \alpha\mathbf{A}$ is diagonal with all diagonal elements between 0 and 1. In this case the first term of the update tends to shrink $\mathbf{w}_t$, and stability is assured. In general, $\mathbf{w}_t$ will be reduced toward zero whenever $\mathbf{A}$ is *positive definite*, meaning $y^\top\mathbf{A}y > 0$ for any real vector $y \neq 0$. Positive definiteness also ensures that the inverse $\mathbf{A}^{-1}$ exists.

For linear TD(0), in the continuing case with $\gamma < 1$, the $\mathbf{A}$ matrix (9.11) can be written

$$\begin{aligned} \mathbf{A} &= \sum_s \mu(s) \sum_a \pi(a|s) \sum_{r,s'} p(r, s' \,|\, s, a)\mathbf{x}(s)\big(\mathbf{x}(s) - \gamma\mathbf{x}(s')\big)^\top \\ &= \sum_s \mu(s) \sum_{s'} p(s' \,|\, s)\mathbf{x}(s)\big(\mathbf{x}(s) - \gamma\mathbf{x}(s')\big)^\top \\ &= \sum_s \mu(s)\mathbf{x}(s)\Big(\mathbf{x}(s) - \gamma\sum_{s'} p(s' \,|\, s)\mathbf{x}(s')\Big)^\top \\ &= \mathbf{X}^\top\mathbf{D}(\mathbf{I} - \gamma\mathbf{P})\mathbf{X}, \end{aligned}$$

where $\mu(s)$ is the stationary distribution under π, $p(s' \,|\, s)$ is the probability of transition from s to s' under policy π, $\mathbf{P}$ is the $|\mathcal{S}| \times |\mathcal{S}|$ matrix of these probabilities,

$\mathbf{D}$ is the $|\mathcal{S}| \times |\mathcal{S}|$ diagonal matrix with the $\mu(s)$ on its diagonal, and $\mathbf{X}$ is the $|\mathcal{S}| \times d$ matrix with $\mathbf{x}(s)$ as its rows. From here it is clear that the inner matrix $\mathbf{D}(\mathbf{I} - \gamma\mathbf{P})$ is key to determining the positive definiteness of $\mathbf{A}$.

For a key matrix of this form, positive definiteness is assured if all of its columns sum to a nonnegative number. This was shown by Sutton (1988, p. 27) based on two previously established theorems. One theorem says that any matrix $\mathbf{M}$ is positive definite if and only if the symmetric matrix $\mathbf{S} = \mathbf{M} + \mathbf{M}^\top$ is positive definite (Sutton 1988, appendix). The second theorem says that any symmetric real matrix $\mathbf{S}$ is positive definite if all of its diagonal entries are positive and greater than the sum of the absolute values of the corresponding off-diagonal entries (Varga 1962, p. 23). For our key matrix, $\mathbf{D}(\mathbf{I} - \gamma\mathbf{P})$, the diagonal entries are positive and the off-diagonal entries are negative, so all we have to show is that each row sum plus the corresponding column sum is positive. The row sums are all positive because $\mathbf{P}$ is a stochastic matrix and $\gamma < 1$. Thus it only remains to show that the column sums are nonnegative. Note that the row vector of the column sums of any matrix $\mathbf{M}$ can be written as $\mathbf{1}^\top\mathbf{M}$, where $\mathbf{1}$ is the column vector with all components equal to 1. Let $\boldsymbol{\mu}$ denote the $|\mathcal{S}|$-vector of the $\mu(s)$, where $\boldsymbol{\mu} = \mathbf{P}^\top\boldsymbol{\mu}$ by virtue of μ being the stationary distribution. The column sums of our key matrix, then, are:

$$\begin{aligned}
\mathbf{1}^\top\mathbf{D}(\mathbf{I} - \gamma\mathbf{P}) &= \boldsymbol{\mu}^\top(\mathbf{I} - \gamma\mathbf{P}) \\
&= \boldsymbol{\mu}^\top - \gamma\boldsymbol{\mu}^\top\mathbf{P} \\
&= \boldsymbol{\mu}^\top - \gamma\boldsymbol{\mu}^\top && \text{(because } \boldsymbol{\mu} \text{ is the stationary distribution)} \\
&= (1 - \gamma)\boldsymbol{\mu}^\top,
\end{aligned}$$

all components of which are positive. Thus, the key matrix and its $\mathbf{A}$ matrix are positive definite, and on-policy TD(0) is stable. (Additional conditions and a schedule for reducing α over time are needed to prove convergence with probability one.)

At the TD fixed point, it has also been proven (in the continuing case) that the $\overline{\text{VE}}$ is within a bounded expansion of the lowest possible error:

$$\overline{\text{VE}}(\mathbf{w}_{\text{TD}}) \;\leq\; \frac{1}{1-\gamma} \min_{\mathbf{w}} \overline{\text{VE}}(\mathbf{w}). \tag{9.14}$$

That is, the asymptotic error of the TD method is no more than $\frac{1}{1-\gamma}$ times the smallest possible error, that attained in the limit by the Monte Carlo method. Because γ is often near one, this expansion factor can be quite large, so there is substantial potential loss in asymptotic performance with the TD method. On the other hand, recall that the TD methods are often of vastly reduced variance compared to Monte Carlo methods, and thus faster, as we saw in Chapters 6 and 7. Which method will be best depends on the nature of the approximation and problem, and on how long learning continues.

A bound analogous to (9.14) applies to other on-policy bootstrapping methods as well. For example, linear semi-gradient DP (Eq. 9.7 with $U_t \doteq \sum_a \pi(a|S_t) \sum_{s',r} p(s',r\,|\,S_t,a)[r+\gamma\hat{v}(s',\mathbf{w}_t)]$) with updates according to the on-policy distribution will also converge to the TD fixed point. One-step semi-gradient *action-value* methods, such as semi-gradient Sarsa(0) covered in the next chapter converge to an analogous fixed point and an analogous bound. For episodic tasks, there is a slightly different but related bound (see Bertsekas and Tsitsiklis, 1996). There are also a few technical conditions on the rewards, features, and decrease in the step-size parameter, which we have omitted here. The full details can be found in the original paper (Tsitsiklis and Van Roy, 1997).

Critical to these convergence results is that states are updated according to the on-policy distribution. For other update distributions, bootstrapping methods using function approximation may actually diverge to infinity. Examples of this and a discussion of possible solution methods are given in Chapter 11.

Example 9.2: Bootstrapping on the 1000-state Random Walk State aggregation is a special case of linear function approximation, so let's return to the 1000-state random walk to illustrate some of the observations made in this chapter. The left panel of Figure 9.2 shows the final value function learned by the semi-gradient TD(0) algorithm (page 203) using the same state aggregation as in Example 9.1. We see that the near-asymptotic TD approximation is indeed farther from the true values than the Monte Carlo approximation shown in Figure 9.1.

Nevertheless, TD methods retain large potential advantages in learning rate, and generalize Monte Carlo methods, as we investigated fully with n-step TD methods in Chapter 7. The right panel of Figure 9.2 shows results with an n-step semi-gradient TD method using state aggregation on the 1000-state random walk that are strikingly similar to those we obtained earlier with tabular methods and the 19-state random walk (Figure 7.2). To obtain such quantitatively similar results we switched the state aggregation to 20 groups of 50 states each. The 20 groups were then quantitatively close

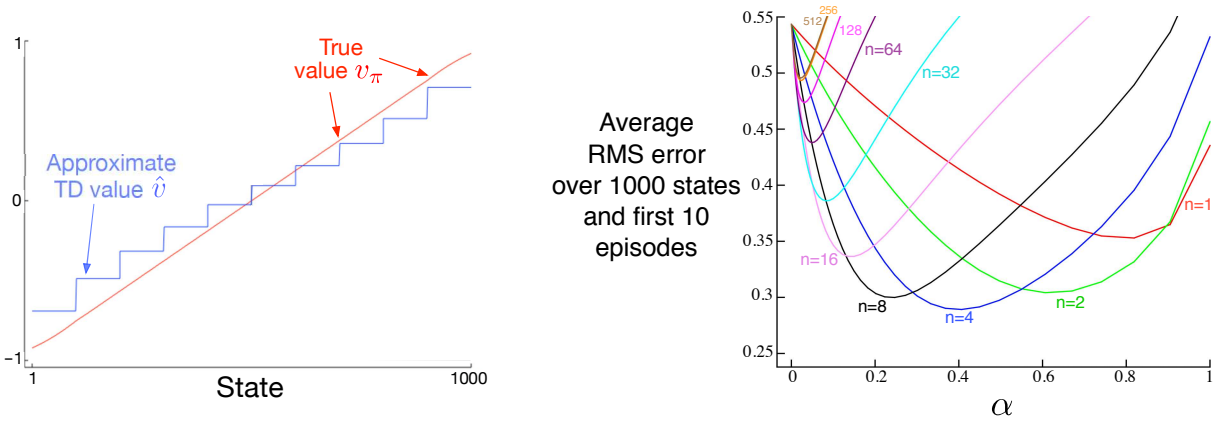


Figure 9.2: Bootstrapping with state aggregation on the 1000-state random walk task. *Left*: Asymptotic values of semi-gradient TD are worse than the asymptotic Monte Carlo values in Figure 9.1. *Right*: Performance of n-step methods with state-aggregation are strikingly similar to those with tabular representations (cf. Figure 7.2). These data are averages over 100 runs.

to the 19 states of the tabular problem. In particular, recall that state transitions were up to 100 states to the left or right. A typical transition would then be of 50 states to the right or left, which is quantitatively analogous to the single-state state transitions of the 19-state tabular system. To complete the match, we use here the same performance measure—an unweighted average of the RMS error over all states and over the first 10 episodes—rather than a $\overline{\text{VE}}$ objective as is otherwise more appropriate when using function approximation. ■

The semi-gradient n-step TD algorithm used in the example above is the natural extension of the tabular n-step TD algorithm presented in Chapter 7 to semi-gradient function approximation. Pseudocode is given in the box below.

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, a positive integer n
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations (S_t and R_t) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | | Take an action according to $\pi(\cdot|S_t)$
  | | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | | If S_{t+1} is terminal, then $T \leftarrow t+1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n},\mathbf{w})$ $(G_{\tau:\tau+n})$
  | | $\mathbf{w} \leftarrow \mathbf{w} + \alpha\,[G - \hat{v}(S_\tau,\mathbf{w})]\,\nabla\hat{v}(S_\tau,\mathbf{w})$
  Until $\tau = T - 1$

The key equation of this algorithm, analogous to (7.2), is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha\,[G_{t:t+n} - \hat{v}(S_t,\mathbf{w}_{t+n-1})]\,\nabla\hat{v}(S_t,\mathbf{w}_{t+n-1}), \qquad 0 \leq t < T, \tag{9.15}$$

where the n-step return is generalized from (7.1) to

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n},\mathbf{w}_{t+n-1}), \quad 0 \leq t \leq T-n. \tag{9.16}$$

Exercise 9.1 Show that tabular methods such as presented in Part I of this book are a special case of linear function approximation. What would the feature vectors be? □

9.5 Feature Construction for Linear Methods

Linear methods are interesting because of their convergence guarantees, but also because in practice they can be very efficient in terms of both data and computation. Whether or not this is so depends critically on how the states are represented in terms of features, which we investigate in this large section. Choosing features appropriate to the task is an important way of adding prior domain knowledge to reinforcement learning systems. Intuitively, the features should correspond to the aspects of the state space along which generalization may be appropriate. If we are valuing geometric objects, for example, we might want to have features for each possible shape, color, size, or function. If we are valuing states of a mobile robot, then we might want to have features for locations, degrees of remaining battery power, recent sonar readings, and so on.

A limitation of the linear form is that it cannot take into account any interactions between features, such as the presence of feature i being good only in the absence of feature j. For example, in the pole-balancing task (Example 3.4), high angular velocity can be either good or bad depending on the angle. If the angle is high, then high angular velocity means an imminent danger of falling—a bad state—whereas if the angle is low, then high angular velocity means the pole is righting itself—a good state. A linear value function could not represent this if its features coded separately for the angle and the angular velocity. It needs instead, or in addition, features for combinations of these two underlying state dimensions. In the following subsections we consider a variety of general ways of doing this.

9.5.1 Polynomials

The states of many problems are initially expressed as numbers, such as positions and velocities in the pole-balancing task (Example 3.4), the number of cars in each lot in the Jack's car rental problem (Example 4.2), or the gambler's capital in the gambler problem (Example 4.3). In these types of problems, function approximation for reinforcement learning has much in common with the familiar tasks of interpolation and regression. Various families of features commonly used for interpolation and regression can also be used in reinforcement learning. Polynomials make up one of the simplest families of features used for interpolation and regression. While the basic polynomial features we discuss here do not work as well as other types of features in reinforcement learning, they serve as a good introduction because they are simple and familiar.

As an example, suppose a reinforcement learning problem has states with two numerical dimensions. For a single representative state s, let its two numbers be $s_1 \in \mathbb{R}$ and $s_2 \in \mathbb{R}$. You might choose to represent s simply by its two state dimensions, so that $\mathbf{x}(s) = (s_1, s_2)^\top$, but then you would not be able to take into account any interactions between these dimensions. In addition, if both s_1 and s_2 were zero, then the approximate value would have to also be zero. Both limitations can be overcome by instead representing s by the four-dimensional feature vector $\mathbf{x}(s) = (1, s_1, s_2, s_1 s_2)^\top$. The initial 1 feature allows the representation of affine functions in the original state numbers, and the final product feature, $s_1 s_2$, enables interactions to be taken into account. Or you might choose to use higher-dimensional feature vectors like $\mathbf{x}(s) = (1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1 s_2^2, s_1^2 s_2, s_1^2 s_2^2)^\top$ to

take more complex interactions into account. Such feature vectors enable approximations as arbitrary quadratic functions of the state numbers—even though the approximation is still linear in the weights that have to be learned. Generalizing this example from two to k numbers, we can represent highly-complex interactions among a problem's state dimensions:

> Suppose each state s corresponds to k numbers, s_1, s_2, ..., s_k, with each $s_i \in \mathbb{R}$. For this k-dimensional state space, each order-n polynomial-basis feature x_i can be written as
>
> $$x_i(s) = \Pi_{j=1}^{k} s_j^{c_{i,j}}, \tag{9.17}$$
>
> where each $c_{i,j}$ is an integer in the set $\{0, 1, \ldots, n\}$ for an integer $n \geq 0$. These features make up the order-n polynomial basis for dimension k, which contains $(n+1)^k$ different features.

Higher-order polynomial bases allow for more accurate approximations of more complicated functions. But because the number of features in an order-n polynomial basis grows exponentially with the dimension k of the natural state space (if $n > 0$), it is generally necessary to select a subset of them for function approximation. This can be done using prior beliefs about the nature of the function to be approximated, and some automated selection methods developed for polynomial regression can be adapted to deal with the incremental and nonstationary nature of reinforcement learning.

Exercise 9.2 Why does (9.17) define $(n+1)^k$ distinct features for dimension k? □

Exercise 9.3 What n and $c_{i,j}$ produce the feature vectors $\mathbf{x}(s) = (1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1 s_2^2, s_1^2 s_2, s_1^2 s_2^2)^\top$? □

9.5.2 Fourier Basis

Another linear function approximation method is based on the time-honored Fourier series, which expresses periodic functions as weighted sums of sine and cosine basis functions (features) of different frequencies. (A function f is periodic if $f(x) = f(x+\tau)$ for all x and some period τ.) The Fourier series and the more general Fourier transform are widely used in applied sciences in part because if a function to be approximated is known, then the basis function weights are given by simple formulae and, further, with enough basis functions essentially any function can be approximated as accurately as desired. In reinforcement learning, where the functions to be approximated are unknown, Fourier basis functions are of interest because they are easy to use and can perform well in a range of reinforcement learning problems.

First consider the one-dimensional case. The usual Fourier series representation of a function of one dimension having period τ represents the function as a linear combination of sine and cosine functions that are each periodic with periods that evenly divide τ (in other words, whose frequencies are integer multiples of a fundamental frequency $1/\tau$). But if you are interested in approximating an aperiodic function defined over a bounded

interval, then you can use these Fourier basis features with τ set to the length of the interval. The function of interest is then just one period of the periodic linear combination of the sine and cosine features.

Furthermore, if you set τ to twice the length of the interval of interest and restrict attention to the approximation over the half interval $[0, \tau/2]$, then you can use just the cosine features. This is possible because you can represent any *even* function, that is, any function that is symmetric about the origin, with just the cosine basis. So any function over the half-period $[0, \tau/2]$ can be approximated as closely as desired with enough cosine features. (Saying "any function" is not exactly correct because the function has to be mathematically well-behaved, but we skip this technicality here.) Alternatively, it is possible to use just sine features, linear combinations of which are always *odd* functions, that is functions that are anti-symmetric about the origin. But it is generally better to keep just the cosine features because "half-even" functions tend to be easier to approximate than "half-odd" functions because the latter are often discontinuous at the origin. Of course, this does not rule out using both sine and cosine features to approximate over the interval $[0, \tau/2]$, which might be advantageous in some circumstances.

Following this logic and letting $\tau = 2$ so that the features are defined over the half-τ interval $[0, 1]$, the one-dimensional order-n Fourier cosine basis consists of the $n + 1$ features

$$x_i(s) = \cos(i\pi s), \quad s \in [0, 1],$$

for $i = 0, \ldots, n$. Figure 9.3 shows one-dimensional Fourier cosine features x_i, for $i = 1, 2, 3, 4$; x_0 is a constant function.

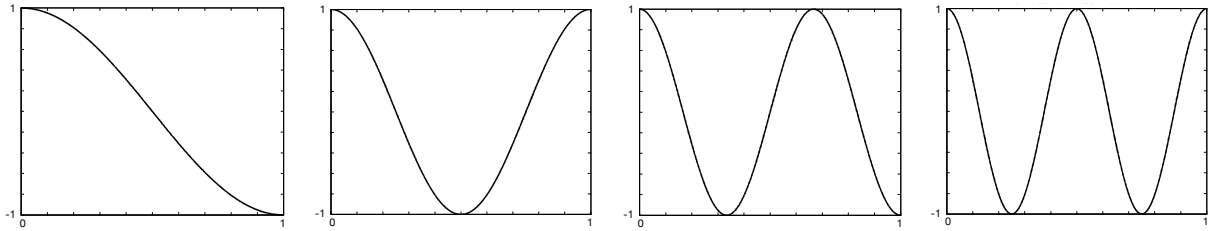


Figure 9.3: One-dimensional Fourier cosine-basis features x_i, $i = 1, 2, 3, 4$, for approximating functions over the interval $[0, 1]$. After Konidaris et al. (2011).

This same reasoning applies to the Fourier cosine series approximation in the multi-dimensional case as described in the box below.

Suppose each state s corresponds to a vector of k numbers, $\mathbf{s} = (s_1, s_2, ..., s_k)^\top$, with each $s_i \in [0, 1]$. The ith feature in the order-n Fourier cosine basis can then be written

$$x_i(s) = \cos\left(\pi \mathbf{s}^\top \mathbf{c}^i\right), \tag{9.18}$$

where $\mathbf{c}^i = (c_1^i, \ldots, c_k^i)^\top$, with $c_j^i \in \{0, \ldots, n\}$ for $j = 1, \ldots, k$ and $i = 1, \ldots, (n+1)^k$. This defines a feature for each of the $(n + 1)^k$ possible integer vectors $\mathbf{c}^i$. The inner

product $\mathbf{s}^\top \mathbf{c}^i$ has the effect of assigning an integer in $\{0, \ldots, n\}$ to each dimension of $\mathbf{s}$. As in the one-dimensional case, this integer determines the feature's frequency along that dimension. The features can of course be shifted and scaled to suit the bounded state space of a particular application.

As an example, consider the $k = 2$ case in which $\mathbf{s} = (s_1, s_2)^\top$, where each $\mathbf{c}^i = (c_1^i, c_2^i)^\top$. Figure 9.4 shows a selection of six Fourier cosine features, each labeled by the vector $\mathbf{c}^i$ that defines it (s_1 is the horizontal axis and $\mathbf{c}^i$ is shown as a row vector with the index i omitted). Any zero in $\mathbf{c}$ means the feature is constant along that state dimension. So if $\mathbf{c} = (0, 0)^\top$, the feature is constant over both dimensions; if $\mathbf{c} = (c_1, 0)^\top$ the feature is constant over the second dimension and varies over the first with frequency depending on c_1; and similarly, for $\mathbf{c} = (0, c_2)^\top$. When $\mathbf{c} = (c_1, c_2)^\top$ with neither $c_j = 0$, the feature varies along both dimensions and represents an interaction between the two state variables. The values of c_1 and c_2 determine the frequency along each dimension, and their ratio gives the direction of the interaction.

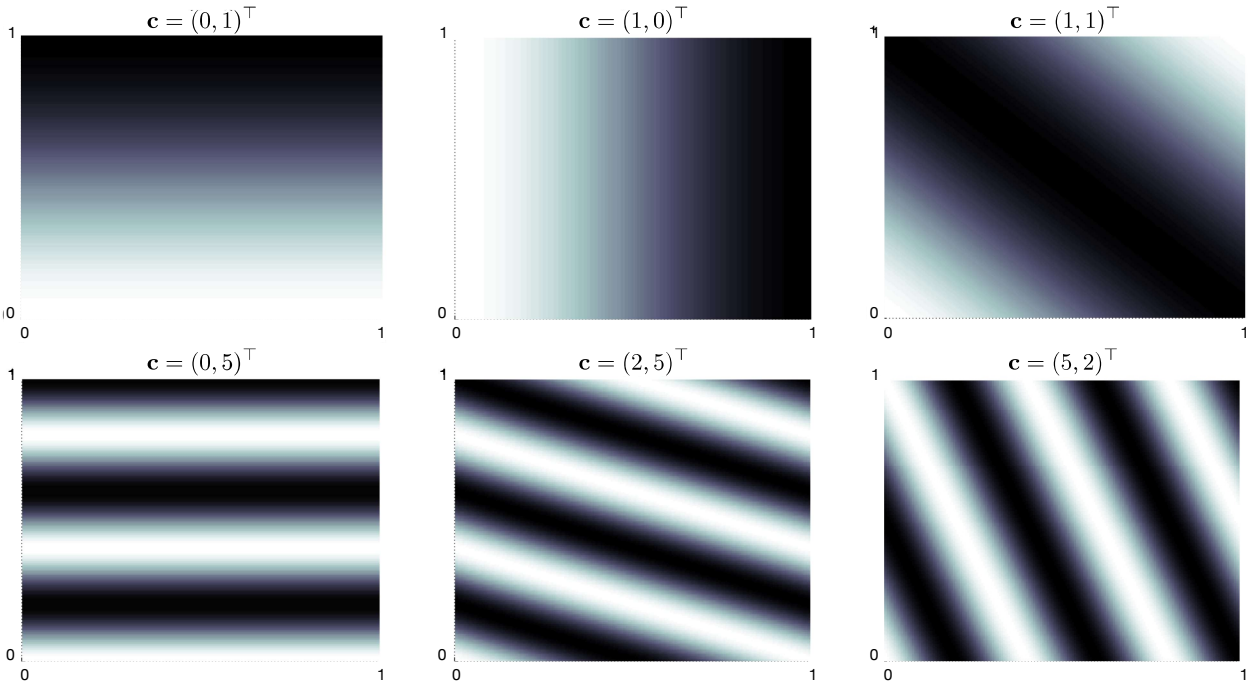


Figure 9.4: A selection of six two-dimensional Fourier cosine features, each labeled by the vector $\mathbf{c}^i$ that defines it (s_1 is the horizontal axis, and $\mathbf{c}^i$ is shown with the index i omitted). After Konidaris et al. (2011).

When using Fourier cosine features with a learning algorithm such as (9.7), semi-gradient TD(0), or semi-gradient Sarsa, it may be helpful to use a different step-size parameter for each feature. If α is the basic step-size parameter, then Konidaris, Osentoski, and Thomas (2011) suggest setting the step-size parameter for feature x_i to $\alpha_i = \alpha / \sqrt{(c_1^i)^2 + \cdots + (c_k^i)^2}$ (except when each $c_j^i = 0$, in which case $\alpha_i = \alpha$).

Fourier cosine features with Sarsa can produce good performance compared to several other collections of basis functions, including polynomial and radial basis functions. Not surprisingly, however, Fourier features have trouble with discontinuities because it is difficult to avoid "ringing" around points of discontinuity unless very high frequency basis functions are included.

The number of features in the order-n Fourier basis grows exponentially with the dimension of the state space, but if that dimension is small enough (e.g., $k \leq 5$), then one can select n so that all of the order-n Fourier features can be used. This makes the selection of features more-or-less automatic. For high dimension state spaces, however, it is necessary to select a subset of these features. This can be done using prior beliefs about the nature of the function to be approximated, and some automated selection methods can be adapted to deal with the incremental and nonstationary nature of reinforcement learning. An advantage of Fourier basis features in this regard is that it is easy to select features by setting the $\mathbf{c}^i$ vectors to account for suspected interactions among the state variables and by limiting the values in the $\mathbf{c}^j$ vectors so that the approximation can filter out high frequency components considered to be noise. On the other hand, because Fourier features are non-zero over the entire state space (with the few zeros excepted), they represent global properties of states, which can make it difficult to find good ways to represent local properties.

Figure 9.5 shows learning curves comparing the Fourier and polynomial bases on the 1000-state random walk example. In general, we do not recommend using polynomials for online learning.[2]

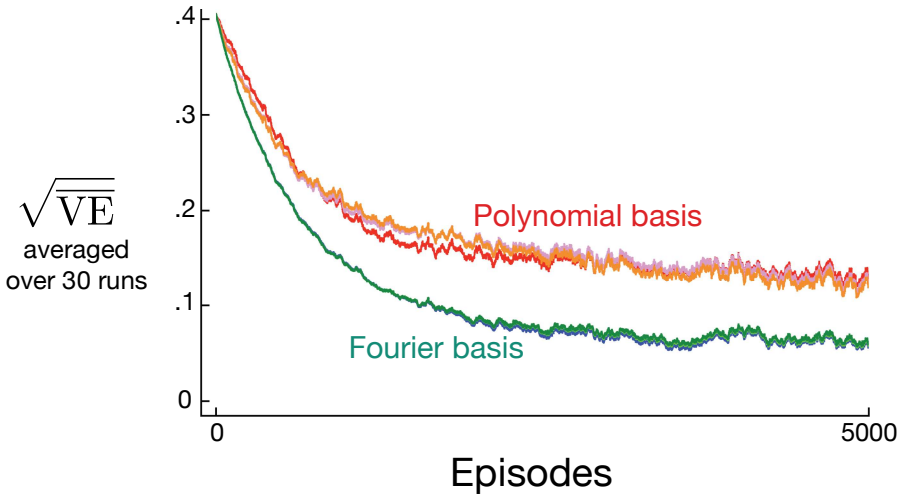


Figure 9.5: Fourier basis vs polynomials on the 1000-state random walk. Shown are learning curves for the gradient Monte Carlo method with Fourier and polynomial bases of order 5, 10, and 20. The step-size parameters were roughly optimized for each case: $\alpha = 0.0001$ for the polynomial basis and $\alpha = 0.00005$ for the Fourier basis. The performance measure (y-axis) is the root mean square value error (9.1).

[2]There are families of polynomials more complicated than those we have discussed, for example, different families of orthogonal polynomials, and these might work better, but at present there is little experience with them in reinforcement learning.

9.5.3 Coarse Coding

Consider a task in which the natural representation of the state set is a continuous two-dimensional space. One kind of representation for this case is made up of features corresponding to *circles* in state space, as shown to the right. If the state is inside a circle, then the corresponding feature has the value 1 and is said to be *present*; otherwise the feature is 0 and is said to be *absent*. This kind of 1–0-valued feature is called a *binary feature*. Given a state, which binary features are present indicate within which circles the state lies, and thus coarsely code for its location. Representing a state with features that overlap in this way (although they need not be circles or binary) is known as *coarse coding*.

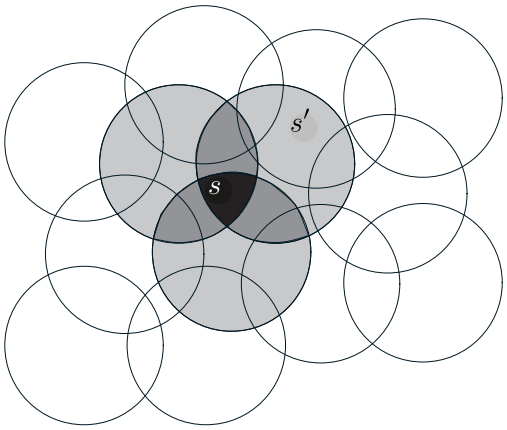


Figure 9.6: Coarse coding. Generalization from state s to state s' depends on the number of their features whose receptive fields (in this case, circles) overlap. These states have one feature in common, so there will be slight generalization between them.

Assuming linear gradient-descent function approximation, consider the effect of the size and density of the circles. Corresponding to each circle is a single weight (a component of $\mathbf{w}$) that is affected by learning. If we train at one state, a point in the space, then the weights of all circles intersecting that state will be affected. Thus, by (9.8), the approximate value function will be affected at all states within the union of the circles, with a greater effect the more circles a point has "in common" with the state, as shown in Figure 9.6. If the circles are small, then the generalization will be over a short distance, as in Figure 9.7 (left), whereas if they are large, it will be over a large distance, as in Figure 9.7 (middle). Moreover,

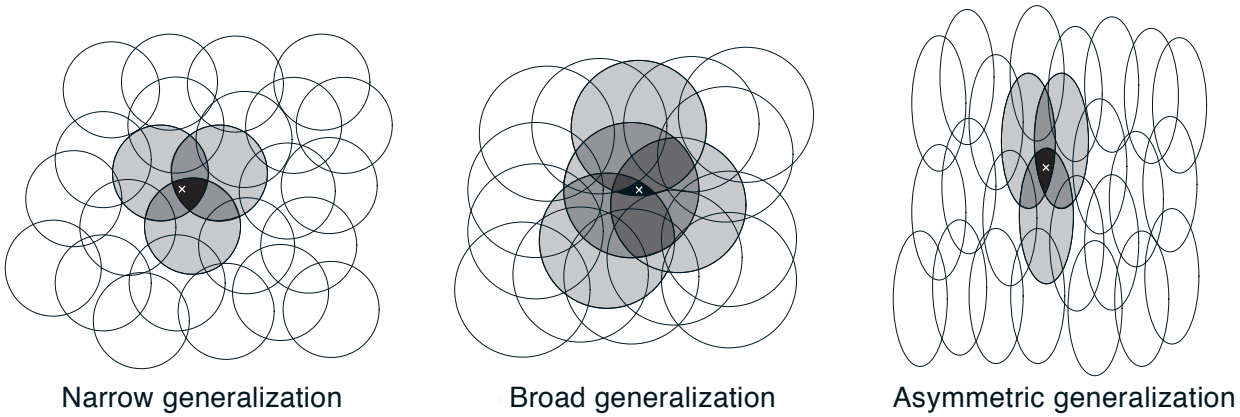


Figure 9.7: Generalization in linear function approximation methods is determined by the sizes and shapes of the features' receptive fields. All three of these cases have roughly the same number and density of features.

the shape of the features will determine the nature of the generalization. For example, if they are not strictly circular, but are elongated in one direction, then generalization will be similarly affected, as in Figure 9.7 (right).

Features with large receptive fields give broad generalization, but might also seem to limit the learned function to a coarse approximation, unable to make discriminations much finer than the width of the receptive fields. Happily, this is not the case. Initial generalization from one point to another is indeed controlled by the size and shape of the receptive fields, but acuity, the finest discrimination ultimately possible, is controlled more by the total number of features.

Example 9.3: Coarseness of Coarse Coding This example illustrates the effect on learning of the size of the receptive fields in coarse coding. Linear function approximation based on coarse coding and (9.7) was used to learn a one-dimensional square-wave function (shown at the top of Figure 9.8). The values of this function were used as the targets, U_t. With just one dimension, the receptive fields were intervals rather than circles. Learning was repeated with three different sizes of the intervals: narrow, medium, and broad, as shown at the bottom of the figure. All three cases had the same density of features, about 50 over the extent of the function being learned. Training examples were generated uniformly at random over this extent. The step-size parameter was $\alpha = \frac{0.2}{n}$, where n is the number of features that were present at one time. Figure 9.8 shows the functions learned in all three cases over the course of learning. Note that the width of the features had a strong effect early in learning. With broad features, the generalization tended to be broad; with narrow features, only the close neighbors of each trained point were changed, causing the function learned to be more bumpy. However, the final function learned was affected only slightly by the width of the features. Receptive field shape tends to have a strong effect on generalization but little effect on asymptotic solution quality.

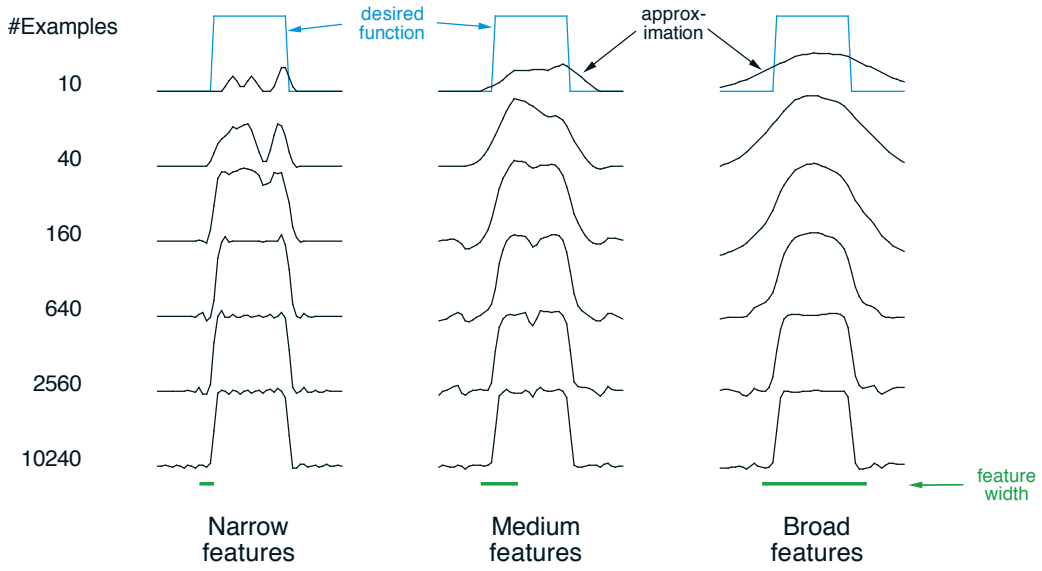


Figure 9.8: Example of feature width's strong effect on initial generalization (first row) and weak effect on asymptotic accuracy (last row). ■

9.5.4 Tile Coding

Tile coding is a form of coarse coding for multi-dimensional continuous spaces that is flexible and computationally efficient. It may be the most practical feature representation for modern sequential digital computers.

In tile coding the receptive fields of the features are grouped into partitions of the state space. Each such partition is called a *tiling*, and each element of the partition is called a *tile*. For example, the simplest tiling of a two-dimensional state space is a uniform grid such as that shown on the left side of Figure 9.9. The tiles or receptive field here are squares rather than the circles in Figure 9.6. If just this single tiling were used, then the state indicated by the white spot would be represented by the single feature whose tile it falls within; generalization would be complete to all states within the same tile and nonexistent to states outside it. With just one tiling, we would not have coarse coding but just a case of state aggregation.

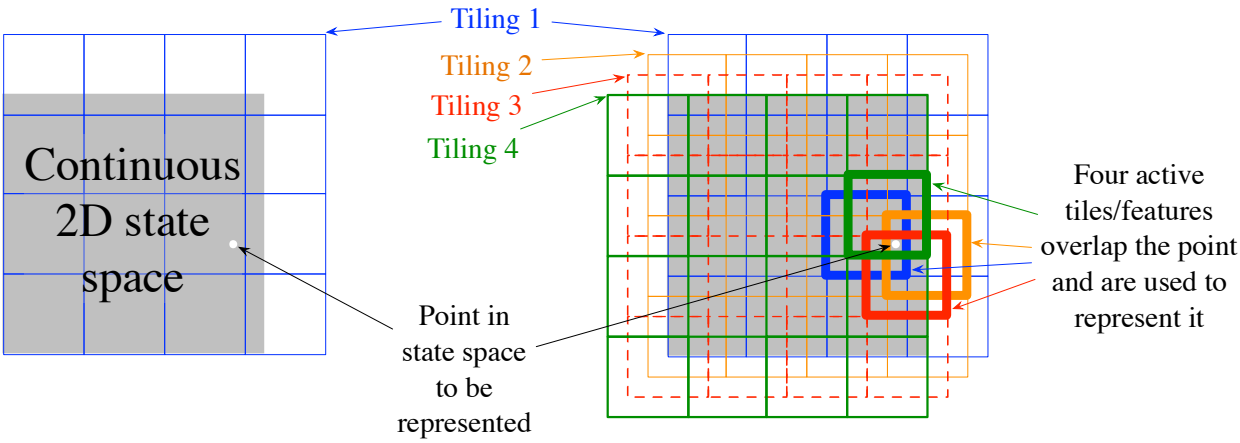


Figure 9.9: Multiple, overlapping grid-tilings on a limited two-dimensional space. These tilings are offset from one another by a uniform amount in each dimension.

To get the strengths of coarse coding requires overlapping receptive fields, and by definition the tiles of a partition do not overlap. To get true coarse coding with tile coding, multiple tilings are used, each offset by a fraction of a tile width. A simple case with four tilings is shown on the right side of Figure 9.9. Every state, such as that indicated by the white spot, falls in exactly one tile in each of the four tilings. These four tiles correspond to four features that become active when the state occurs. Specifically, the feature vector $\mathbf{x}(s)$ has one component for each tile in each tiling. In this example there are $4 \times 4 \times 4 = 64$ components, all of which will be 0 except for the four corresponding to the tiles that s falls within. Figure 9.10 shows the advantage of multiple offset tilings (coarse coding) over a single tiling on the 1000-state random walk example.

An immediate practical advantage of tile coding is that, because it works with partitions, the overall number of features that are active at one time is the same for any state. Exactly one feature is present in each tiling, so the total number of features present is always the same as the number of tilings. This allows the step-size parameter, α, to

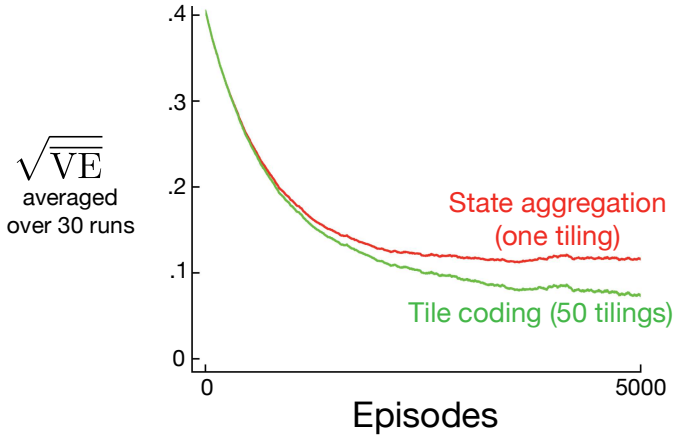


Figure 9.10: Why we use coarse coding. Shown are learning curves on the 1000-state random walk example for the gradient Monte Carlo algorithm with a single tiling and with multiple tilings. The space of 1000 states was treated as a single continuous dimension, covered with tiles each 200 states wide. The multiple tilings were offset from each other by 4 states. The step-size parameter was set so that the initial learning rate in the two cases was the same, $\alpha = 0.0001$ for the single tiling and $\alpha = 0.0001/50$ for the 50 tilings.

be set in an easy, intuitive way. For example, choosing $\alpha = \frac{1}{n}$, where n is the number of tilings, results in exact one-trial learning. If the example $s \mapsto v$ is trained on, then whatever the prior estimate, $\hat{v}(s,\mathbf{w}_t)$, the new estimate will be $\hat{v}(s,\mathbf{w}_{t+1}) = v$. Usually one wishes to change more slowly than this, to allow for generalization and stochastic variation in target outputs. For example, one might choose $\alpha = \frac{1}{10n}$, in which case the estimate for the trained state would move one-tenth of the way to the target in one update, and neighboring states will be moved less, proportional to the number of tiles they have in common.

Tile coding also gains computational advantages from its use of binary feature vectors. Because each component is either 0 or 1, the weighted sum making up the approximate value function (9.8) is almost trivial to compute. Rather than performing d multiplications and additions, one simply computes the indices of the $n \ll d$ active features and then adds up the n corresponding components of the weight vector.

Generalization occurs to states other than the one trained if those states fall within any of the same tiles, proportional to the number of tiles in common. Even the choice of how to offset the tilings from each other affects generalization. If they are offset uniformly in each dimension, as they were in Figure 9.9, then different states can generalize in qualitatively different ways, as shown in the upper half of Figure 9.11. Each of the eight subfigures show the pattern of generalization from a trained state to nearby points. In this example there are eight tilings, thus 64 subregions within a tile that generalize distinctly, but all according to one of these eight patterns. Note how uniform offsets result in a strong effect along the diagonal in many patterns. These artifacts can be avoided if the tilings are offset asymmetrically, as shown in the lower half of the figure. These lower generalization patterns are better because they are all well centered on the trained state with no obvious asymmetries.

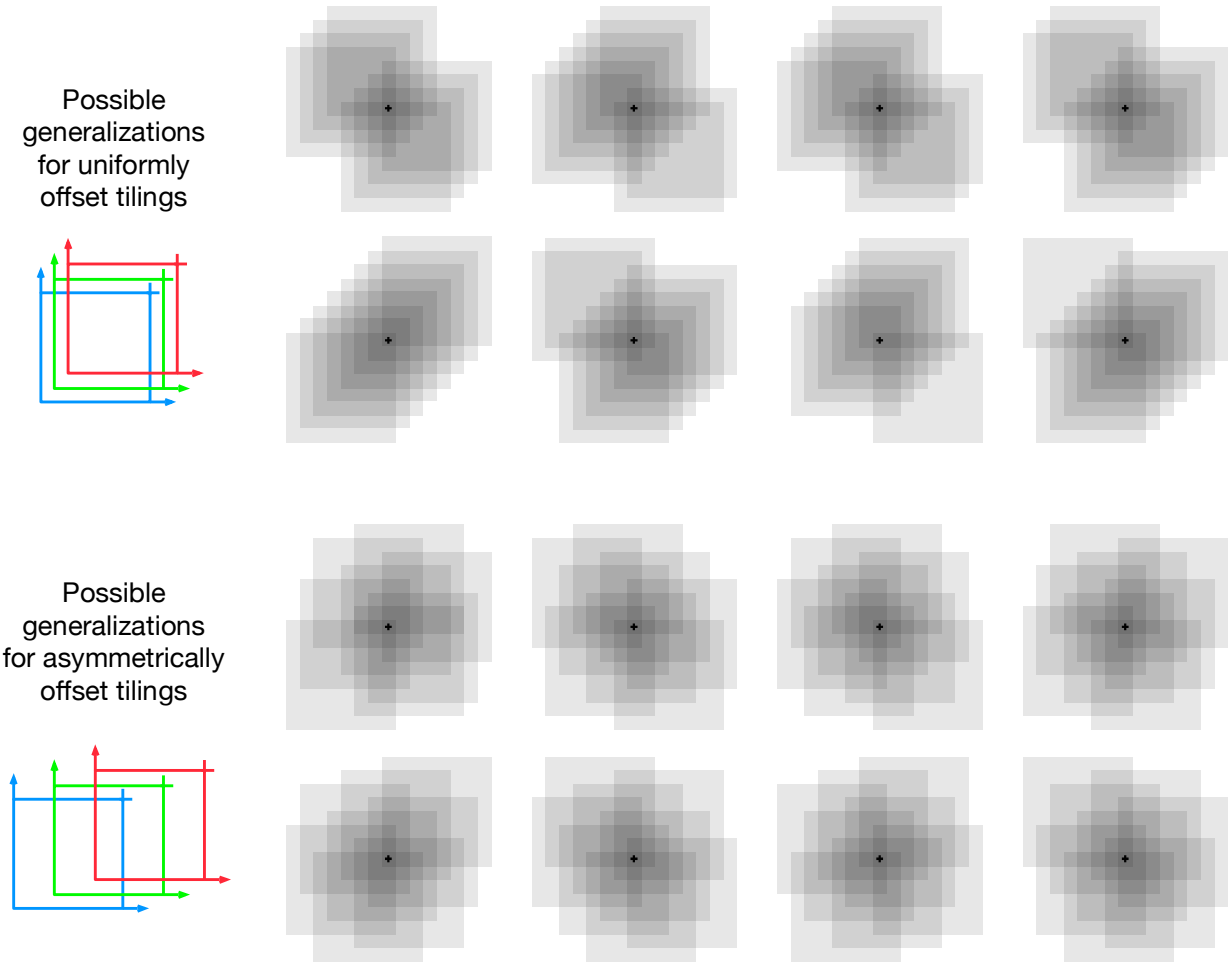


Figure 9.11: Why tile asymmetrical offsets are preferred in tile coding. Shown is the strength of generalization from a trained state, indicated by the small black plus, to nearby states, for the case of eight tilings. If the tilings are uniformly offset (above), then there are diagonal artifacts and substantial variations in the generalization, whereas with asymmetrically offset tilings the generalization is more spherical and homogeneous.

Tilings in all cases are offset from each other by a fraction of a tile width in each dimension. If w denotes the tile width and n the number of tilings, then $\frac{w}{n}$ is a fundamental unit. Within small squares $\frac{w}{n}$ on a side, all states activate the same tiles, have the same feature representation, and the same approximated value. If a state is moved by $\frac{w}{n}$ in any cartesian direction, the feature representation changes by one component/tile. Uniformly offset tilings are offset from each other by exactly this unit distance. For a two-dimensional space, we say that each tiling is offset by the displacement vector $(1, 1)$, meaning that it is offset from the previous tiling by $\frac{w}{n}$ times this vector. In these terms, the asymmetrically offset tilings shown in the lower part of Figure 9.11 are offset by a displacement vector of $(1, 3)$.

Extensive studies have been made of the effect of different displacement vectors on the generalization of tile coding (Parks and Militzer, 1991; An, 1991; An, Miller and Parks,

1991; Miller, An, Glanz and Carter, 1990), assessing their homegeneity and tendency toward diagonal artifacts like those seen for the $(1, 1)$ displacement vectors. Based on this work, Miller and Glanz (1996) recommend using displacement vectors consisting of the first odd integers. In particular, for a continuous space of dimension k, a good choice is to use the first odd integers $(1, 3, 5, 7, \ldots, 2k-1)$, with n (the number of tilings) set to an integer power of 2 greater than or equal to $4k$. This is what we have done to produce the tilings in the lower half of Figure 9.11, in which $k = 2$, $n = 2^3 \geq 4k$, and the displacement vector is $(1, 3)$. In a three-dimensional case, the first four tilings would be offset in total from a base position by $(0, 0, 0)$, $(1, 3, 5)$, $(2, 6, 10)$, and $(3, 9, 15)$. Open-source software that can efficiently make tilings like this for any k is readily available.

In choosing a tiling strategy, one has to pick the number of the tilings and the shape of the tiles. The number of tilings, along with the size of the tiles, determines the resolution or fineness of the asymptotic approximation, as in general coarse coding and illustrated in Figure 9.8. The shape of the tiles will determine the nature of generalization as in Figure 9.7. Square tiles will generalize roughly equally in each dimension as indicated in Figure 9.11 (lower). Tiles that are elongated along one dimension, such as the stripe tilings in Figure 9.12 (middle), will promote generalization along that dimension. The tilings in Figure 9.12 (middle) are also denser and thinner on the left, promoting discrimination along the horizontal dimension at lower values along that dimension. The diagonal stripe tiling in Figure 9.12 (right) will promote generalization along one diagonal. In higher dimensions, axis-aligned stripes correspond to ignoring some of the dimensions in some of the tilings, that is, to hyperplanar slices. Irregular tilings such as shown in Figure 9.12 (left) are also possible, though rare in practice and beyond the standard software.

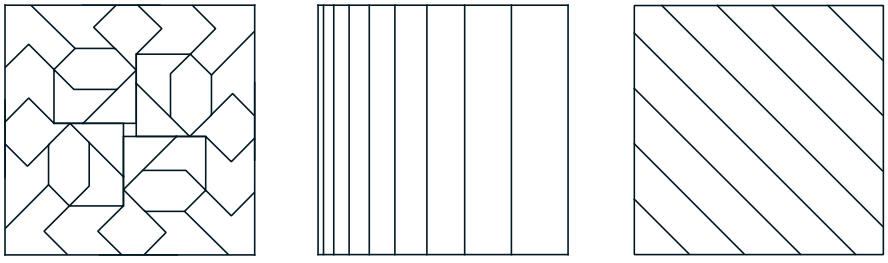
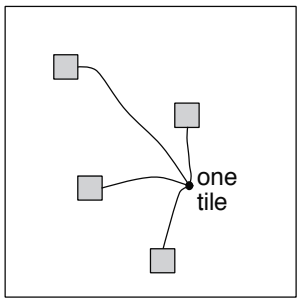


Figure 9.12: Tilings need not be grids. They can be arbitrarily shaped and non-uniform, while still in many cases being computationally efficient to compute.

In practice, it is often desirable to use different shaped tiles in different tilings. For example, one might use some vertical stripe tilings and some horizontal stripe tilings. This would encourage generalization along either dimension. However, with stripe tilings alone it is not possible to learn that a particular conjunction of horizontal and vertical coordinates has a distinctive value (whatever is learned for it will bleed into states with the same horizontal and vertical coordinates). For this one needs the conjunctive rectangular tiles such as originally shown in Figure 9.9. With multiple tilings—some horizontal, some vertical, and some conjunctive—one can get everything: a preference for generalizing along each dimension, yet the ability to learn specific values for conjunctions (see Sutton,

1996 for examples). The choice of tilings determines generalization, and until this choice can be effectively automated, it is important that tile coding enables the choice to be made flexibly and in a way that makes sense to people.

Another useful trick for reducing memory requirements is *hashing*—a consistent pseudo-random collapsing of a large tiling into a much smaller set of tiles. Hashing produces tiles consisting of noncontiguous, disjoint regions randomly spread throughout the state space, but that still form an exhaustive partition. For example, one tile might consist of the four subtiles shown to the right. Through hashing, memory requirements are often reduced by large factors with little loss of performance. This is possible because high resolution is needed in only a small fraction of the state space. Hashing frees us from the curse of dimensionality in the sense that memory requirements need not be exponential in the number of dimensions, but need merely match the real demands of the task. Open-source implementations of tile coding commonly include efficient hashing.



Exercise 9.4 Suppose we believe that one of two state dimensions is more likely to have an effect on the value function than is the other, that generalization should be primarily across this dimension rather than along it. What kind of tilings could be used to take advantage of this prior knowledge? □

9.5.5 Radial Basis Functions

Radial basis functions (RBFs) are the natural generalization of coarse coding to continuous-valued features. Rather than each feature being either 0 or 1, it can be anything in the interval $[0, 1]$, reflecting various *degrees* to which the feature is present. A typical RBF feature, x_i, has a Gaussian (bell-shaped) response $x_i(s)$ dependent only on the distance between the state, s, and the feature's prototypical or center state, c_i, and relative to the feature's width, σ_i:

$$x_i(s) \doteq \exp\left(-\frac{||s - c_i||^2}{2\sigma_i^2}\right).$$

The norm or distance metric of course can be chosen in whatever way seems most appropriate to the states and task at hand. The figure below shows a one-dimensional example with a Euclidean distance metric.

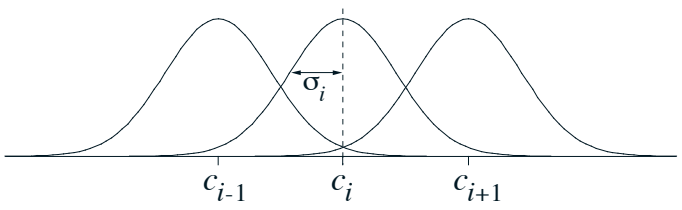


Figure 9.13: One-dimensional radial basis functions.

The primary advantage of RBFs over binary features is that they produce approximate functions that vary smoothly and are differentiable. Although this is appealing, in most cases it has no practical significance. Nevertheless, extensive studies have been made of graded response functions such as RBFs in the context of tile coding (An, 1991; Miller et al., 1991; An et al., 1991; Lane, Handelman and Gelfand, 1992). All of these methods require substantial additional computational complexity (over tile coding) and often reduce performance when there are more than two state dimensions. In high dimensions the edges of tiles are much more important, and it has proven difficult to obtain well controlled graded tile activations near the edges.

An *RBF network* is a linear function approximator using RBFs for its features. Learning is defined by equations (9.7) and (9.8), exactly as in other linear function approximators. In addition, some learning methods for RBF networks change the centers and widths of the features as well, bringing them into the realm of nonlinear function approximators. Nonlinear methods may be able to fit target functions much more precisely. The downside to RBF networks, and to nonlinear RBF networks especially, is greater computational complexity and, often, more manual tuning before learning is robust and efficient.

9.6 Selecting Step-Size Parameters Manually

Most SGD methods require the designer to select an appropriate step-size parameter α. Ideally this selection would be automated, and in some cases it has been, but for most cases it is still common practice to set it manually. To do this, and to better understand the algorithms, it is useful to develop some intuitive sense of the role of the step-size parameter. Can we say in general how it should be set?

Theoretical considerations are unfortunately of little help. The theory of stochastic approximation gives us conditions (2.7) on a slowly decreasing step-size sequence that are sufficient to guarantee convergence, but these tend to result in learning that is too slow. The classical choice $\alpha_t = 1/t$, which produces sample averages in tabular MC methods, is not appropriate for TD methods, for nonstationary problems, or for any method using function approximation. For linear methods, there are recursive least-squares methods that set an optimal *matrix* step size, and these methods can be extended to temporal-difference learning as in the LSTD method described in Section 9.8, but these require $O(d^2)$ step-size parameters, or d times more parameters than we are learning. For this reason we rule them out for use on large problems where function approximation is most needed.

To get some intuitive feel for how to set the step-size parameter manually, it is best to go back momentarily to the tabular case. There we can understand that a step size of $\alpha = 1$ will result in a complete elimination of the sample error after one target (see (2.4) with a step size of one). As discussed on page 201, we usually want to learn slower than this. In the tabular case, a step size of $\alpha = \frac{1}{10}$ would take about 10 experiences to converge approximately to their mean target, and if we wanted to learn in 100 experiences we would use $\alpha = \frac{1}{100}$. In general, if $\alpha = \frac{1}{\tau}$, then the tabular estimate for a state will approach the mean of its targets, with the most recent targets having the greatest effect, after about τ experiences with the state.

With general function approximation there is not such a clear notion of *number* of experiences with a state, as each state may be similar to and dissimilar from all the others to various degrees. However, there is a similar rule that gives similar behavior in the case of linear function approximation. Suppose you wanted to learn in about τ experiences with substantially the same feature vector. A good rule of thumb for setting the step-size parameter of linear SGD methods is then

$$\alpha \doteq \left(\tau \mathbb{E}\left[\mathbf{x}^\top \mathbf{x}\right]\right)^{-1}, \tag{9.19}$$

where $\mathbf{x}$ is a random feature vector chosen from the same distribution as input vectors will be in the SGD. This method works best if the feature vectors do not vary greatly in length; ideally $\mathbf{x}^\top \mathbf{x}$ is a constant.

Exercise 9.5 Suppose you are using tile coding to transform a seven-dimensional continuous state space into binary feature vectors to estimate a state value function $\hat{v}(s,\mathbf{w}) \approx v_\pi(s)$. You believe that the dimensions do not interact strongly, so you decide to use eight tilings of each dimension separately (stripe tilings), for $7 \times 8 = 56$ tilings. In addition, in case there are some pairwise interactions between the dimensions, you also take all $\binom{7}{2} = 21$ pairs of dimensions and tile each pair conjunctively with rectangular tiles. You make two tilings for each pair of dimensions, making a grand total of $21 \times 2 + 56 = 98$ tilings. Given these feature vectors, you suspect that you still have to average out some noise, so you decide that you want learning to be gradual, taking about 10 presentations with the same feature vector before learning nears its asymptote. What step-size parameter α should you use? Why? □

Exercise 9.6 If $\tau = 1$ and $\mathbf{x}(S_t)^\top \mathbf{x}(S_t) = \mathbb{E}\left[\mathbf{x}^\top \mathbf{x}\right]$, prove that (9.19) together with (9.7) and linear function approximation results in the error being reduced to zero in one update.

9.7 Nonlinear Function Approximation: Artificial Neural Networks

Artificial neural networks (ANNs) are widely used for nonlinear function approximation. An ANN is a network of interconnected units that have some of the properties of neurons, the main components of nervous systems. ANNs have a long history, with the latest advances in training deeply-layered ANNs (deep learning) being responsible for some of the most impressive abilities of machine learning systems, including reinforcement learning systems. In Chapter 16 we describe several impressive examples of reinforcement learning systems that use ANN function approximation.

Figure 9.14 shows a generic feedforward ANN, meaning that there are no loops in the network, that is, there are no paths within the network by which a unit's output can influence its input. The network in the figure has an output layer consisting of two output units, an input layer with four input units, and two "hidden layers": layers that are neither input nor output layers. A real-valued weight is associated with each link. A weight roughly corresponds to the efficacy of a synaptic connection in a real neural network (see Section 15.1). If an ANN has at least one loop in its connections, it is a recurrent rather

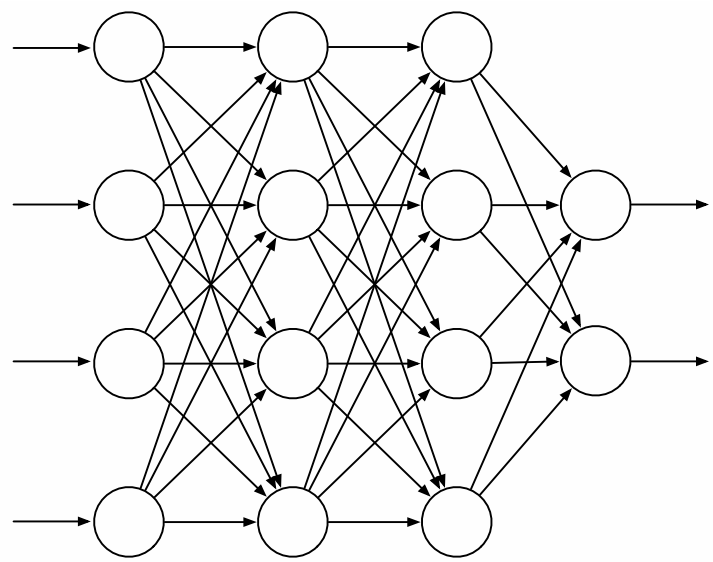


Figure 9.14: A generic feedforward ANN with four input units, two output units, and two hidden layers.

than a feedforward ANN. Although both feedforward and recurrent ANNs have been used in reinforcement learning, here we look only at the simpler feedforward case.

The units (the circles in Figure 9.14) are typically semi-linear units, meaning that they compute a weighted sum of their input signals and then apply to the result a nonlinear function, called the *activation function*, to produce the unit's output, or activation. Different activation functions are used, but they are typically S-shaped, or sigmoid, functions such as the logistic function $f(x) = 1/(1 + e^{-x})$, though sometimes the rectifier nonlinearity $f(x) = \max(0, x)$ is used. A step function like $f(x) = 1$ if $x \geq \theta$, and 0 otherwise, results in a binary unit with threshold θ. The units in a network's input layer are somewhat different in having their activations set to externally-supplied values that are the inputs to the function the network is approximating.

The activation of each output unit of a feedforward ANN is a nonlinear function of the activation patterns over the network's input units. The functions are parameterized by the network's connection weights. An ANN with no hidden layers can represent only a very small fraction of the possible input-output functions. However an ANN with a single hidden layer containing a large enough finite number of sigmoid units can approximate any continuous function on a compact region of the network's input space to any degree of accuracy (Cybenko, 1989). This is also true for other nonlinear activation functions that satisfy mild conditions, but nonlinearity is essential: if all the units in a multi-layer feedforward ANN have linear activation functions, the entire network is equivalent to a network with no hidden layers (because linear functions of linear functions are themselves linear).

Despite this "universal approximation" property of one-hidden-layer ANNs, both experience and theory show that approximating the complex functions needed for many artificial intelligence tasks is made easier—indeed may require—abstractions that are hierarchical compositions of many layers of lower-level abstractions, that is, abstractions

produced by deep architectures such as ANNs with many hidden layers. (See Bengio, 2009, for a thorough review.) The successive layers of a deep ANN compute increasingly abstract representations of the network's "raw" input, with each unit providing a feature contributing to a hierarchical representation of the overall input-output function of the network.

Training the hidden layers of an ANN is therefore a way to automatically create features appropriate for a given problem so that hierarchical representations can be produced without relying exclusively on hand-crafted features. This has been an enduring challenge for artificial intelligence and explains why learning algorithms for ANNs with hidden layers have received so much attention over the years. ANNs typically learn by a stochastic gradient method (Section 9.3). Each weight is adjusted in a direction aimed at improving the network's overall performance as measured by an objective function to be either minimized or maximized. In the most common supervised learning case, the objective function is the expected error, or loss, over a set of labeled training examples. In reinforcement learning, ANNs can use TD errors to learn value functions, or they can aim to maximize expected reward as in a gradient bandit (Section 2.8) or a policy-gradient algorithm (Chapter 13). In all of these cases it is necessary to estimate how a change in each connection weight would influence the network's overall performance, in other words, to estimate the partial derivative of an objective function with respect to each weight, given the current values of all the network's weights. The gradient is the vector of these partial derivatives.

The most successful way to do this for ANNs with hidden layers (provided the units have differentiable activation functions) is the backpropagation algorithm, which consists of alternating forward and backward passes through the network. Each forward pass computes the activation of each unit given the current activations of the network's input units. After each forward pass, a backward pass efficiently computes a partial derivative for each weight. (As in other stochastic gradient learning algorithms, the vector of these partial derivatives is an estimate of the true gradient.) In Section 15.10 we discuss methods for training ANNs with hidden layers that use reinforcement learning principles instead of backpropagation. These methods are less efficient than the backpropagation algorithm, but they may be closer to how real neural networks learn.

The backpropagation algorithm can produce good results for shallow networks having 1 or 2 hidden layers, but it may not work well for deeper ANNs. In fact, training a network with $k + 1$ hidden layers can actually result in poorer performance than training a network with k hidden layers, even though the deeper network can represent all the functions that the shallower network can (Bengio, 2009). Explaining results like these is not easy, but several factors are important. First, the large number of weights in a typical deep ANN makes it difficult to avoid the problem of overfitting, that is, the problem of failing to generalize correctly to cases on which the network has not been trained. Second, backpropagation does not work well for deep ANNs because the partial derivatives computed by its backward passes either decay rapidly toward the input side of the network, making learning by deep layers extremely slow, or the partial derivatives grow rapidly toward the input side of the network, making learning unstable. Methods for dealing with these problems are largely responsible for many impressive recent results

achieved by systems that use deep ANNs.

Overfitting is a problem for any function approximation method that adjusts functions with many degrees of freedom on the basis of limited training data. It is less of a problem for online reinforcement learning that does not rely on limited training sets, but generalizing effectively is still an important issue. Overfitting is a problem for ANNs in general, but especially so for deep ANNs because they tend to have very large numbers of weights. Many methods have been developed for reducing overfitting. These include stopping training when performance begins to decrease on validation data different from the training data (cross validation), modifying the objective function to discourage complexity of the approximation (regularization), and introducing dependencies among the weights to reduce the number of degrees of freedom (e.g., weight sharing).

A particularly effective method for reducing overfitting by deep ANNs is the dropout method introduced by Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov (2014). During training, units are randomly removed from the network (dropped out) along with their connections. This can be thought of as training a large number of "thinned" networks. Combining the results of these thinned networks at test time is a way to improve generalization performance. The dropout method efficiently approximates this combination by multiplying each outgoing weight of a unit by the probability that that unit was retained during training. Srivastava et al. found that this method significantly improves generalization performance. It encourages individual hidden units to learn features that work well with random collections of other features. This increases the versatility of the features formed by the hidden units so that the network does not overly specialize to rarely-occurring cases.

Hinton, Osindero, and Teh (2006) took a major step toward solving the problem of training the deep layers of a deep ANN in their work with deep belief networks, layered networks closely related to the deep ANNs discussed here. In their method, the deepest layers are trained one at a time using an unsupervised learning algorithm. Without relying on the overall objective function, unsupervised learning can extract features that capture statistical regularities of the input stream. The deepest layer is trained first, then with input provided by this trained layer, the next deepest layer is trained, and so on, until the weights in all, or many, of the network's layers are set to values that now act as initial values for supervised learning. The network is then fine-tuned by backpropagation with respect to the overall objective function. Studies show that this approach generally works much better than backpropagation with weights initialized with random values. The better performance of networks trained with weights initialized this way could be due to many factors, but one idea is that this method places the network in a region of weight space from which a gradient-based algorithm can make good progress.

Batch normalization (Ioffe and Szegedy, 2015) is another technique that makes it easier to train deep ANNs. It has long been known that ANN learning is easier if the network input is normalized, for example, by adjusting each input variable to have zero mean and unit variance. Batch normalization for training deep ANNs normalizes the output of deep layers before they feed into the following layer. Ioffe and Szegedy (2015) used statistics from subsets, or "mini-batches," of training examples to normalize these between-layer signals to improve the learning rate of deep ANNs.

Another technique useful for training deep ANNs is *deep residual learning* (He, Zhang, Ren, and Sun, 2016). Sometimes it is easier to learn how a function differs from the identity function than to learn the function itself. Then adding this difference, or residual function, to the input produces the desired function. In deep ANNs, a block of layers can be made to learn a residual function simply by adding shortcut, or skip, connections around the block. These connections add the input to the block to its output, and no additional weights are needed. He et al. (2016) evaluated this method using deep convolutional networks with skip connections around every pair of adjacent layers, finding substantial improvement over networks without the skip connections on benchmark image classification tasks. Both batch normalization and deep residual learning were used in the reinforcement learning application to the game of Go that we describe in Chapter 16.

A type of deep ANN that has proven to be very successful in applications, including impressive reinforcement learning applications (Chapter 16), is the *deep convolutional network*. This type of network is specialized for processing high-dimensional data arranged in spatial arrays, such as images. It was inspired by how early visual processing works in the brain (LeCun, Bottou, Bengio and Haffner, 1998). Because of its special architecture, a deep convolutional network can be trained by backpropagation without resorting to methods like those described above to train the deep layers.

Figure 9.15 illustrates the architecture of a deep convolutional network. This instance, from LeCun et al. (1998), was designed to recognize hand-written characters. It consists of alternating convolutional and subsampling layers, followed by several fully connected final layers. Each convolutional layer produces a number of feature maps. A feature map is a pattern of activity over an array of units, where each unit performs the same operation on data in its receptive field, which is the part of the data it "sees" from the preceding layer (or from the external input in the case of the first convolutional layer). The units of a feature map are identical to one another except that their receptive fields, which are all the same size and shape, are shifted to different locations on the arrays of incoming data. Units in the same feature map share the same weights. This means that a feature map detects the same feature no matter where it is located in the input

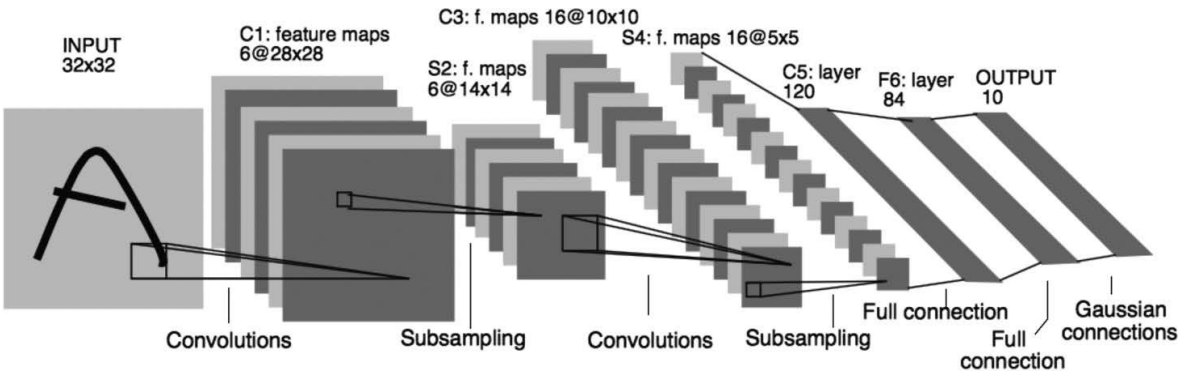


Figure 9.15: Deep Convolutional Network. Republished with permission of Proceedings of the IEEE, from Gradient-based learning applied to document recognition, LeCun, Bottou, Bengio, and Haffner, volume 86, 1998; permission conveyed through Copyright Clearance Center, Inc.

array. In the network in Figure 9.15, for example, the first convolutional layer produces 6 feature maps, each consisting of 28×28 units. Each unit in each feature map has a 5×5 receptive field, and these receptive fields overlap (in this case by four columns and four rows). Consequently, each of the 6 feature maps is specified by just 25 adjustable weights.

The subsampling layers of a deep convolutional network reduce the spatial resolution of the feature maps. Each feature map in a subsampling layer consists of units that average over a receptive field of units in the feature maps of the preceding convolutional layer. For example, each unit in each of the 6 feature maps in the first subsampling layer of the network of Figure 9.15 averages over a 2×2 non-overlapping receptive field over one of the feature maps produced by the first convolutional layer, resulting in six 14×14 feature maps. Subsampling layers reduce the network's sensitivity to the spatial locations of the features detected, that is, they help make the network's responses spatially invariant. This is useful because a feature detected at one place in an image is likely to be useful at other places as well.

Advances in the design and training of ANNs—of which we have only mentioned a few—all contribute to reinforcement learning. Although current reinforcement learning theory is mostly limited to methods using tabular or linear function approximation methods, the impressive performances of notable reinforcement learning applications owe much of their success to nonlinear function approximation by multi-layer ANNs. We discuss several of these applications in Chapter 16.

9.8 Least-Squares TD

All the methods we have discussed so far in this chapter have required computation per time step proportional to the number of parameters. With more computation, however, one can do better. In this section we present a method for linear function approximation that is arguably the best that can be done for this case.

As we established in Section 9.4 TD(0) with linear function approximation converges asymptotically (for appropriately decreasing step sizes) to the TD fixed point:

$$\mathbf{w}_{\text{TD}} = \mathbf{A}^{-1}\mathbf{b},$$

where

$$\mathbf{A} \doteq \mathbb{E}\big[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top\big] \quad \text{and} \quad \mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t]\,.$$

Why, one might ask, must we compute this solution iteratively? This is wasteful of data! Could one not do better by computing estimates of $\mathbf{A}$ and $\mathbf{b}$, and then directly computing the TD fixed point? The *Least-Squares TD* algorithm, commonly known as *LSTD*, does exactly this. It forms the natural estimates

$$\widehat{\mathbf{A}}_t \doteq \sum_{k=0}^{t-1} \mathbf{x}_k(\mathbf{x}_k - \gamma\mathbf{x}_{k+1})^\top + \varepsilon\mathbf{I} \quad \text{and} \quad \widehat{\mathbf{b}}_t \doteq \sum_{k=0}^{t-1} R_{k+1}\mathbf{x}_k, \tag{9.20}$$

where $\mathbf{I}$ is the identity matrix, and $\varepsilon\mathbf{I}$, for some small $\varepsilon > 0$, ensures that $\widehat{\mathbf{A}}_t$ is always invertible. It might seem that these estimates should both be divided by t, and indeed they should; as defined here, these are really estimates of t *times* $\mathbf{A}$ and t *times* $\mathbf{b}$. However, the extra t factors cancel out when LSTD uses these estimates to estimate the TD fixed point as

$$\mathbf{w}_t \doteq \widehat{\mathbf{A}}_t^{-1}\widehat{\mathbf{b}}_t. \tag{9.21}$$

This algorithm is the most data efficient form of linear TD(0), but it is also more expensive computationally. Recall that semi-gradient TD(0) requires memory and per-step computation that is only $O(d)$.

How complex is LSTD? As it is written above the complexity seems to increase with t, but the two approximations in (9.20) could be implemented incrementally using the techniques we have covered earlier (e.g., in Chapter 2) so that they can be done in constant time per step. Even so, the update for $\widehat{\mathbf{A}}_t$ would involve an outer product (a column vector times a row vector) and thus would be a matrix update; its computational complexity would be $O(d^2)$, and of course the memory required to hold the $\widehat{\mathbf{A}}_t$ matrix would be $O(d^2)$.

A potentially greater problem is that our final computation (9.21) uses the inverse of $\widehat{\mathbf{A}}_t$, and the computational complexity of a general inverse computation is $O(d^3)$. Fortunately, an inverse of a matrix of our special form—a sum of outer products—can also be updated incrementally with only $O(d^2)$ computations, as

$$\begin{aligned}
\widehat{\mathbf{A}}_t^{-1} &= \left(\widehat{\mathbf{A}}_{t-1} + \mathbf{x}_{t-1}(\mathbf{x}_{t-1} - \gamma\mathbf{x}_t)^\top\right)^{-1} && \text{(from (9.20))} \\
&= \widehat{\mathbf{A}}_{t-1}^{-1} - \frac{\widehat{\mathbf{A}}_{t-1}^{-1}\mathbf{x}_{t-1}(\mathbf{x}_{t-1} - \gamma\mathbf{x}_t)^\top\widehat{\mathbf{A}}_{t-1}^{-1}}{1 + (\mathbf{x}_{t-1} - \gamma\mathbf{x}_t)^\top\widehat{\mathbf{A}}_{t-1}^{-1}\mathbf{x}_{t-1}}, && (9.22)
\end{aligned}$$

for $t > 0$, with $\widehat{\mathbf{A}}_0 \doteq \varepsilon\mathbf{I}$. Although the identity (9.22), known as *the Sherman-Morrison formula*, is superficially complicated, it involves only vector-matrix and vector-vector multiplications and thus is only $O(d^2)$. Thus we can store the inverse matrix $\widehat{\mathbf{A}}_t^{-1}$, maintain it with (9.22), and then use it in (9.21), all with only $O(d^2)$ memory and per-step computation. The complete algorithm is given in the box on the next page.

Of course, $O(d^2)$ is still significantly more expensive than the $O(d)$ of semi-gradient TD. Whether the greater data efficiency of LSTD is worth this computational expense depends on how large d is, how important it is to learn quickly, and the expense of other parts of the system. The fact that LSTD requires no step-size parameter is sometimes also touted, but the advantage of this is probably overstated. LSTD does not require a step size, but it does requires ε; if ε is chosen too small the sequence of inverses can vary wildly, and if ε is chosen too large then learning is slowed. In addition, LSTD's lack of a step-size parameter means that it never forgets. This is sometimes desirable, but it is problematic if the target policy π changes as it does in reinforcement learning and GPI. In control applications, LSTD typically has to be combined with some other mechanism to induce forgetting, mooting any initial advantage of not requiring a step-size parameter.

LSTD for estimating $\hat{v} = \mathbf{w}^\top \mathbf{x}(\cdot) \approx v_\pi$ ($O(d^2)$ version)

Input: feature representation $\mathbf{x} : \mathcal{S}^+ \to \mathbb{R}^d$ such that $\mathbf{x}(terminal) = \mathbf{0}$
Algorithm parameter: small $\varepsilon > 0$

$\widehat{\mathbf{A}^{-1}} \leftarrow \varepsilon^{-1}\mathbf{I}$ — A $d \times d$ matrix
$\widehat{\mathbf{b}} \leftarrow \mathbf{0}$ — A d-dimensional vector
Loop for each episode:
  Initialize S; $\mathbf{x} \leftarrow \mathbf{x}(S)$
  Loop for each step of episode:
    Choose and take action $A \sim \pi(\cdot|S)$, observe R, S'; $\mathbf{x}' \leftarrow \mathbf{x}(S')$
    $\mathbf{v} \leftarrow \widehat{\mathbf{A}^{-1}}^\top (\mathbf{x} - \gamma\mathbf{x}')$
    $\widehat{\mathbf{A}^{-1}} \leftarrow \widehat{\mathbf{A}^{-1}} - \big(\widehat{\mathbf{A}^{-1}}\mathbf{x}\big)\mathbf{v}^\top / \big(1 + \mathbf{v}^\top \mathbf{x}\big)$
    $\widehat{\mathbf{b}} \leftarrow \widehat{\mathbf{b}} + R\mathbf{x}$
    $\mathbf{w} \leftarrow \widehat{\mathbf{A}^{-1}}\widehat{\mathbf{b}}$
    $S \leftarrow S'$; $\mathbf{x} \leftarrow \mathbf{x}'$
  until S' is terminal

9.9 Memory-based Function Approximation

So far we have discussed the *parametric* approach to approximating value functions. In this approach, a learning algorithm adjusts the parameters of a functional form intended to approximate the value function over a problem's entire state space. Each update, $s \mapsto g$, is a training example used by the learning algorithm to change the parameters with the aim of reducing the approximation error. After the update, the training example can be discarded (although it might be saved to be used again). When an approximate value of a state (which we will call the *query state*) is needed, the function is simply evaluated at that state using the latest parameters produced by the learning algorithm.

Memory-based function approximation methods are very different. They simply save training examples in memory as they arrive (or at least save a subset of the examples) without updating any parameters. Then, whenever a query state's value estimate is needed, a set of examples is retrieved from memory and used to compute a value estimate for the query state. This approach is sometimes called *lazy learning* because processing training examples is postponed until the system is queried to provide an output.

Memory-based function approximation methods are prime examples of *nonparametric* methods. Unlike parametric methods, the approximating function's form is not limited to a fixed parameterized class of functions, such as linear functions or polynomials, but is instead determined by the training examples themselves, together with some means for combining them to output estimated values for query states. As more training examples accumulate in memory, one expects nonparametric methods to produce increasingly accurate approximations of any target function.

There are many different memory-based methods depending on how the stored training examples are selected and how they are used to respond to a query. Here, we focus on *local-learning* methods that approximate a value function only locally in the neighborhood of the current query state. These methods retrieve a set of training examples from memory whose states are judged to be the most relevant to the query state, where relevance usually depends on the distance between states: the closer a training example's state is to the query state, the more relevant it is considered to be, where distance can be defined in many different ways. After the query state is given a value, the local approximation is discarded.

The simplest example of the memory-based approach is the *nearest neighbor* method, which simply finds the example in memory whose state is closest to the query state and returns that example's value as the approximate value of the query state. In other words, if the query state is s, and $s' \mapsto g$ is the example in memory in which s' is the closest state to s, then g is returned as the approximate value of s. Slightly more complicated are *weighted average* methods that retrieve a set of nearest neighbor examples and return a weighted average of their target values, where the weights generally decrease with increasing distance between their states and the query state. *Locally weighted regression* is similar, but it fits a surface to the values of a set of nearest states by means of a parametric approximation method that minimizes a weighted error measure like (9.1), where the weights depend on distances from the query state. The value returned is the evaluation of the locally-fitted surface at the query state, after which the local approximation surface is discarded.

Being nonparametric, memory-based methods have the advantage over parametric methods of not limiting approximations to pre-specified functional forms. This allows accuracy to improve as more data accumulates. Memory-based *local* approximation methods have other properties that make them well suited for reinforcement learning. Because trajectory sampling is of such importance in reinforcement learning, as discussed in Section 8.6, memory-based local methods can focus function approximation on local neighborhoods of states (or state–action pairs) visited in real or simulated trajectories. There may be no need for global approximation because many areas of the state space will never (or almost never) be reached. In addition, memory-based methods allow an agent's experience to have a relatively immediate affect on value estimates in the neighborhood of the current state, in contrast with a parametric method's need to incrementally adjust parameters of a global approximation.

Avoiding global approximation is also a way to address the curse of dimensionality. For example, for a state space with k dimensions, a tabular method storing a global approximation requires memory exponential in k. On the other hand, in storing examples for a memory-based method, each example requires memory proportional to k, and the memory required to store, say, n examples is linear in n. Nothing is exponential in k or n. Of course, the critical remaining issue is whether a memory-based method can answer queries quickly enough to be useful to an agent. A related concern is how speed degrades as the size of the memory grows. Finding nearest neighbors in a large database can take too long to be practical in many applications.

Proponents of memory-based methods have developed ways to accelerate the nearest neighbor search. Using parallel computers or special purpose hardware is one approach; another is the use of special multi-dimensional data structures to store the training data. One data structure studied for this application is the k-d tree (short for k-dimensional tree), which recursively splits a k-dimensional space into regions arranged as nodes of a binary tree. Depending on the amount of data and how it is distributed over the state space, nearest-neighbor search using k-d trees can quickly eliminate large regions of the space in the search for neighbors, making the searches feasible in some problems where naive searches would take too long.

Locally weighted regression additionally requires fast ways to do the local regression computations which have to be repeated to answer each query. Researchers have developed many ways to address these problems, including methods for forgetting entries in order to keep the size of the database within bounds. The Bibliographic and Historical Comments section at the end of this chapter points to some of the relevant literature, including a selection of papers describing applications of memory-based learning to reinforcement learning.

9.10 Kernel-based Function Approximation

Memory-based methods such as the weighted average and locally weighted regression methods described above depend on assigning weights to examples $s' \mapsto g$ in the database depending on the distance between s' and a query states s. The function that assigns these weights is called a *kernel function*, or simply a *kernel*. In the weighted average and locally weighted regressions methods, for example, a kernel function $k : \mathbb{R} \to \mathbb{R}$ assigns weights to distances between states. More generally, weights do not have to depend on distances; they can depend on some other measure of similarity between states. In this case, $k : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$, so that $k(s, s')$ is the weight given to data about s' in its influence on answering queries about s.

Viewed slightly differently, $k(s, s')$ is a measure of the strength of generalization from s' to s. Kernel functions numerically express how *relevant* knowledge about any state is to any other state. As an example, the strengths of generalization for tile coding shown in Figure 9.11 correspond to different kernel functions resulting from uniform and asymmetrical tile offsets. Although tile coding does not explicitly use a kernel function in its operation, it generalizes according to one. In fact, as we discuss more below, the strength of generalization resulting from linear parametric function approximation can always be described by a kernel function.

Kernel regression is the memory-based method that computes a kernel weighted average of the targets of *all* examples stored in memory, assigning the result to the query state. If $\mathcal{D}$ is the set of stored examples, and $g(s')$ denotes the target for state s' in a stored example, then kernel regression approximates the target function, in this case a value function depending on $\mathcal{D}$, as

$$\hat{v}(s,\mathcal{D}) = \sum_{s' \in \mathcal{D}} k(s, s')g(s'). \tag{9.23}$$

The weighted average method described above is a special case in which $k(s, s')$ is non-zero only when s and s' are close to one another so that the sum need not be computed over all of $\mathcal{D}$.

A common kernel is the Gaussian radial basis function (RBF) used in RBF function approximation as described in Section 9.5.5. In the method described there, RBFs are features whose centers and widths are either fixed from the start, with centers presumably concentrated in areas where many examples are expected to fall, or are adjusted in some way during learning. Barring methods that adjust centers and widths, this is a linear parametric method whose parameters are the weights of each RBF, which are typically learned by stochastic gradient, or semi-gradient, descent. The form of the approximation is a linear combination of the pre-determined RBFs. Kernel regression with an RBF kernel differs from this in two ways. First, it is memory-based: the RBFs are centered on the states of the stored examples. Second, it is nonparametric: there are no parameters to learn; the response to a query is given by (9.23).

Of course, many issues have to be addressed for practical implementation of kernel regression, issues that are beyond the scope of our brief discussion. However, it turns out that any linear parametric regression method like those we described in Section 9.4, with states represented by feature vectors $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_d(s))^\top$, can be recast as kernel regression where $k(s, s')$ is the inner product of the feature vector representations of s and s'; that is

$$k(s, s') = \mathbf{x}(s)^\top \mathbf{x}(s'). \tag{9.24}$$

Kernel regression with this kernel function produces the same approximation that a linear parametric method would if it used these feature vectors and learned with the same training data.

We skip the mathematical justification for this, which can be found in any modern machine learning text, such as Bishop (2006), and simply point out an important implication. Instead of constructing features for linear parametric function approximators, one can instead construct kernel functions directly without referring at all to feature vectors. Not all kernel functions can be expressed as inner products of feature vectors as in (9.24), but a kernel function that can be expressed like this can offer significant advantages over the equivalent parametric method. For many sets of feature vectors, (9.24) has a compact functional form that can be evaluated without any computation taking place in the d-dimensional feature space. In these cases, kernel regression is much less complex than directly using a linear parametric method with states represented by these feature vectors. This is the so-called "kernel trick" that allows effectively working in the high-dimension of an expansive feature space while actually working only with the set of stored training examples. The kernel trick is the basis of many machine learning methods, and researchers have shown how it can sometimes benefit reinforcement learning.

9.11 Looking Deeper at On-policy Learning: Interest and Emphasis

The algorithms we have considered so far in this chapter have treated all the states encountered equally, as if they were all equally important. In some cases, however, we are more interested in some states than others. In discounted episodic problems, for example, we may be more interested in accurately valuing early states in the episode than in later states where discounting may have made the rewards much less important to the value of the start state. Or, if an action-value function is being learned, it may be less important to accurately value poor actions whose value is much less than the greedy action. Function approximation resources are always limited, and if they were used in a more targeted way, then performance could be improved.

One reason we have treated all states encountered equally is that then we are updating according to the on-policy distribution, for which stronger theoretical results are available for semi-gradient methods. Recall that the on-policy distribution was defined as the distribution of states encountered in an MDP while following the target policy. Now we will generalize this concept significantly. Rather than having one on-policy distribution for the MDP, we will have many. All of them will have in common that they are a distribution of states encountered in trajectories while following the target policy, but they will vary in how the trajectories are, in a sense, initiated.

We now introduce some new concepts. First we introduce a non-negative scalar measure, a random variable I_t called *interest*, indicating the degree to which we are interested in accurately valuing the state (or state–action pair) at time t. If we don't care at all about the state, then the interest should be zero; if we fully care, it might be one, though it is formally allowed to take any non-negative value. The interest can be set in any causal way; for example, it may depend on the trajectory up to time t or the learned parameters at time t. The distribution μ in the $\overline{\text{VE}}$ (9.1) is then defined as the distribution of states encountered while following the target policy, weighted by the interest. Second, we introduce another non-negative scalar random variable, the *emphasis* M_t. This scalar multiplies the learning update and thus emphasizes or de-emphasizes the learning done at time t. The general n-step learning rule, replacing (9.15), is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha M_t \left[G_{t:t+n} - \hat{v}(S_t, \mathbf{w}_{t+n-1})\right] \nabla \hat{v}(S_t, \mathbf{w}_{t+n-1}), \qquad 0 \leq t < T, \tag{9.25}$$

with the n-step return given by (9.16) and the emphasis determined recursively from the interest by:

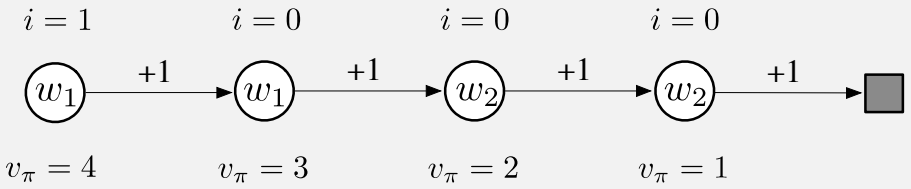
$$M_t = I_t + \gamma^n M_{t-n}, \qquad 0 \leq t < T, \tag{9.26}$$

with $M_t \doteq 0$, for all $t < 0$. These equations are taken to include the Monte Carlo case, for which $G_{t:t+n} = G_t$, all the updates are made at end of the episode, $n = T - t$, and $M_t = I_t$.

Example 9.4 illustrates how interest and emphasis can result in more accurate value estimates.

Example 9.4: Interest and Emphasis

To see the potential benefits of using interest and emphasis, consider the four-state Markov reward process shown below:



Episodes start in the leftmost state, then transition one state to the right, with a reward of +1, on each step until the terminal state is reached. The true value of the first state is thus 4, of the second state 3, and so on as shown below each state. These are the true values; the estimated values can only approximate these because they are constrained by the parameterization. There are two components to the parameter vector $\mathbf{w} = (w_1, w_2)^\top$, and the parameterization is as written inside each state. The estimated values of the first two states are given by w_1 alone and thus must be the same even though their true values are different. Similarly, the estimated values of the third and fourth states are given by w_2 alone and must be the same even though their true values are different. Suppose that we are interested in accurately valuing only the leftmost state; we assign it an interest of 1 while all the other states are assigned an interest of 0, as indicated above the states.

First consider applying gradient Monte Carlo algorithms to this problem. The algorithms presented earlier in this chapter that do not take into account interest and emphasis (in (9.7) and the box on page 202) will converge (for decreasing step sizes) to the parameter vector $\mathbf{w}_\infty = (3.5, 1.5)$, which gives the first state—the only one we are interested in—a value of 3.5 (i.e., intermediate between the true values of the first and second states). The methods presented in this section that do use interest and emphasis, on the other hand, will learn the value of the first state exactly correctly; w_1 will converge to 4 while w_2 will never be updated because the emphasis is zero in all states save the leftmost.

Now consider applying two-step semi-gradient TD methods. The methods from earlier in this chapter without interest and emphasis (in (9.15) and (9.16) and the box on page 209) will again converge to $\mathbf{w}_\infty = (3.5, 1.5)$, while the methods with interest and emphasis converge to $\mathbf{w}_\infty = (4, 2)$. The latter produces the exactly correct values for the first state and for the third state (which the first state bootstraps from) while never making any updates corresponding to the second or fourth states.

9.12 Summary

Reinforcement learning systems must be capable of *generalization* if they are to be applicable to artificial intelligence or to large engineering applications. To achieve this, any of a broad range of existing methods for *supervised-learning function approximation* can be used simply by treating each update as a training example.

Perhaps the most suitable supervised learning methods are those using *parameterized function approximation*, in which the policy is parameterized by a weight vector $\mathbf{w}$. Although the weight vector has many components, the state space is much larger still, and we must settle for an approximate solution. We defined the *mean square value error*, $\overline{\text{VE}}(\mathbf{w})$, as a measure of the error in the values $v_{\pi_\mathbf{w}}(s)$ for a weight vector $\mathbf{w}$ under the *on-policy distribution*, μ. The $\overline{\text{VE}}$ gives us a clear way to rank different value-function approximations in the on-policy case.

To find a good weight vector, the most popular methods are variations of *stochastic gradient descent* (SGD). In this chapter we have focused on the *on-policy* case with a *fixed policy*, also known as policy evaluation or prediction; a natural learning algorithm for this case is *n-step semi-gradient TD*, which includes gradient Monte Carlo and semi-gradient TD(0) algorithms as the special cases when $n=\infty$ and $n=1$ respectively. Semi-gradient TD methods are not true gradient methods. In such bootstrapping methods (including DP), the weight vector appears in the update target, yet this is not taken into account in computing the gradient—thus they are *semi*-gradient methods. As such, they cannot rely on classical SGD results.

Nevertheless, good results can be obtained for semi-gradient methods in the special case of *linear* function approximation, in which the value estimates are sums of features times corresponding weights. The linear case is the most well understood theoretically and works well in practice when provided with appropriate features. Choosing the features is one of the most important ways of adding prior domain knowledge to reinforcement learning systems. They can be chosen as polynomials, but this case generalizes poorly in the online learning setting typically considered in reinforcement learning. Better is to choose features according the Fourier basis, or according to some form of coarse coding with sparse overlapping receptive fields. Tile coding is a form of coarse coding that is particularly computationally efficient and flexible. Radial basis functions are useful for one- or two-dimensional tasks in which a smoothly varying response is important. LSTD is the most data-efficient linear TD prediction method, but requires computation proportional to the square of the number of weights, whereas all the other methods are of complexity linear in the number of weights. Nonlinear methods include artificial neural networks trained by backpropagation and variations of SGD; these methods have become very popular in recent years under the name *deep reinforcement learning*.

Linear semi-gradient n-step TD is guaranteed to converge under standard conditions, for all n, to a $\overline{\text{VE}}$ that is within a bound of the optimal error (achieved asymptotically by Monte Carlo methods). This bound is always tighter for higher n and approaches zero as $n \to \infty$. However, in practice very high n results in very slow learning, and some degree of bootstrapping ($n < \infty$) is usually preferable, just as we saw in comparisons of tabular n-step methods in Chapter 7 and in comparisons of tabular TD and Monte Carlo methods in Chapter 6.

Exercise 9.7 One of the simplest artificial neural networks consists of a single semi-linear unit with a logistic nonlinearity. The need to handle approximate value functions of this form is common in games that end with either a win or a loss, in which case the value of a state can be interpreted as the probability of winning. Derive the learning algorithm for this case, from (9.7), such that no gradient notation appears.

**Exercise 9.8* Arguably, the squared error used to derive (9.7) is inappropriate for the case treated in the preceding exercise, and the right error measure is the *cross-entropy loss* (which you can find on Wikipedia). Repeat the derivation in Section 9.3, using the cross-entropy loss instead of the squared error in (9.4), all the way to an explicit form with no gradient or logarithm notation in it. Is your final form more complex, or simpler, than that you obtained in the preceding exercise?

Bibliographical and Historical Remarks

Generalization and function approximation have always been an integral part of reinforcement learning. Bertsekas and Tsitsiklis (1996), Bertsekas (2012), and Sugiyama et al. (2013) present the state of the art in function approximation in reinforcement learning. Some of the early work with function approximation in reinforcement learning is discussed at the end of this section.

9.3 Gradient-descent methods for minimizing mean square error in supervised learning are well known. Widrow and Hoff (1960) introduced the least-mean-square (LMS) algorithm, which is the prototypical incremental gradient-descent algorithm. Details of this and related algorithms are provided in many texts (e.g., Widrow and Stearns, 1985; Bishop, 1995; Duda and Hart, 1973).

Semi-gradient TD(0) was first explored by Sutton (1984, 1988), as part of the linear TD(λ) algorithm that we will treat in Chapter 12. The term "semi-gradient" to describe these bootstrapping methods is new to the second edition of this book.

The earliest use of state aggregation in reinforcement learning may have been Michie and Chambers's BOXES system (1968). The theory of state aggregation in reinforcement learning has been developed by Singh, Jaakkola, and Jordan (1995) and Tsitsiklis and Van Roy (1996). State aggregation has been used in dynamic programming from its earliest days (e.g., Bellman, 1957a).

9.4 Sutton (1988) proved convergence of linear TD(0) in the mean to the minimal $\overline{\text{VE}}$ solution for the case in which the feature vectors, $\{\mathbf{x}(s) : s \in \mathcal{S}\}$, are linearly independent. Convergence with probability 1 was proved by several researchers at about the same time (Peng, 1993; Dayan and Sejnowski, 1994; Tsitsiklis, 1994; Gurvits, Lin, and Hanson, 1994). In addition, Jaakkola, Jordan, and Singh (1994) proved convergence under online updating. All of these results assumed linearly independent feature vectors, which implies at least as many components to $\mathbf{w}_t$ as there are states. Convergence for the more important case of general (dependent) feature vectors was first shown by Dayan (1992). A significant

generalization and strengthening of Dayan's result was proved by Tsitsiklis and Van Roy (1997). They proved the main result presented in this section, the bound on the asymptotic error of linear bootstrapping methods.

9.5 Our presentation of the range of possibilities for linear function approximation is based on that by Barto (1990).

9.5.2 Konidaris, Osentoski, and Thomas (2011) introduced the Fourier basis in a simple form suitable for reinforcement learning problems with multi-dimensional continuous state spaces and functions that do not have to be periodic.

9.5.3 The term *coarse coding* is due to Hinton (1984), and our Figure 9.6 is based on one of his figures. Waltz and Fu (1965) provide an early example of this type of function approximation in a reinforcement learning system.

9.5.4 Tile coding, including hashing, was introduced by Albus (1971, 1981). He described it in terms of his "cerebellar model articulator controller," or CMAC, as tile coding is sometimes known in the literature. The term "tile coding" was new to the first edition of this book, though the idea of describing CMAC in these terms is taken from Watkins (1989). Tile coding has been used in many reinforcement learning systems (e.g., Shewchuk and Dean, 1990; Lin and Kim, 1991; Miller, Scalera, and Kim, 1994; Sofge and White, 1992; Tham, 1994; Sutton, 1996; Watkins, 1989) as well as in other types of learning control systems (e.g., Kraft and Campagna, 1990; Kraft, Miller, and Dietz, 1992). This section draws heavily on the work of Miller and Glanz (1996). General software for tile coding is available in several languages (e.g., see `http://incompleteideas.net/tiles/tiles3.html`).

9.5.5 Function approximation using radial basis functions has received wide attention ever since being related to ANNs by Broomhead and Lowe (1988). Powell (1987) reviewed earlier uses of RBFs, and Poggio and Girosi (1989, 1990) extensively developed and applied this approach.

9.6 Automatic methods for adapting the step-size parameter include RMSprop (Tieleman and Hinton, 2012), Adam (Kingma and Ba, 2015), stochastic meta-descent methods such as Delta-Bar-Delta (Jacobs, 1988), its incremental generalization (Sutton, 1992b, c; Mahmood et al., 2012), and nonlinear generalizations (Schraudolph, 1999, 2002). Methods explicitly designed for reinforcement learning include AlphaBound (Dabney and Barto, 2012), SID and NOSID (Dabney, 2014), TIDBD (Kearney et al., in preparation) and the application of stochastic meta-descent to policy gradient learning (Schraudolph, Yu, and Aberdeen, 2006).

9.7 The introduction of the threshold logic unit as an abstract model neuron by McCulloch and Pitts (1943) was the beginning of ANNs. The history of ANNs as learning methods for classification or regression has passed through several stages: roughly, the Perceptron (Rosenblatt, 1962) and ADALINE (ADAptive LINear Element) (Widrow and Hoff, 1960) stage of learning by single-layer ANNs, the

error-backpropagation stage (LeCun, 1985; Rumelhart, Hinton, and Williams, 1986) of learning by multi-layer ANNs, and the current deep-learning stage with its emphasis on representation learning (e.g., Bengio, Courville, and Vincent, 2012; Goodfellow, Bengio, and Courville, 2016). Examples of the many books on ANNs are Haykin (1994), Bishop (1995), and Ripley (2007).

ANNs as function approximation for reinforcement learning goes back to the early work of Farley and Clark (1954), who used reinforcement-like learning to modify the weights of linear threshold functions representing policies. Widrow, Gupta, and Maitra (1973) presented a neuron-like linear threshold unit implementing a learning process they called *learning with a critic* or *selective bootstrap adaptation*, a reinforcement-learning variant of the ADALINE algorithm. Werbos (1987, 1994) developed an approach to prediction and control that uses ANNs trained by error backpropation to learn policies and value functions using TD-like algorithms. Barto, Sutton, and Brouwer (1981) and Barto and Sutton (1981b) extended the idea of an associative memory network (e.g., Kohonen, 1977; Anderson, Silverstein, Ritz, and Jones, 1977) to reinforcement learning. Barto, Anderson, and Sutton (1982) used a two-layer ANN to learn a nonlinear control policy, and emphasized the first layer's role of learning a suitable representation. Hampson (1983, 1989) was an early proponent of multilayer ANNs for learning value functions. Barto, Sutton, and Anderson (1983) presented an actor–critic algorithm in the form of an ANN learning to balance a simulated pole (see Sections 15.7 and 15.8). Barto and Anandan (1985) introduced a stochastic version of Widrow et al.'s (1973) selective bootstrap algorithm called the *associative reward-penalty* (A_{R-P}) *algorithm*. Barto (1985, 1986) and Barto and Jordan (1987) described multi-layer ANNs consisting of $\mathrm{A_{R-P}}$ units trained with a globally-broadcast reinforcement signal to learn classification rules that are not linearly separable. Barto (1985) discussed this approach to ANNs and how this type of learning rule is related to others in the literature at that time. (See Section 15.10 for additional discussion of this approach to training multi-layer ANNs.) Anderson (1986, 1987, 1989) evaluated numerous methods for training multilayer ANNs and showed that an actor–critic algorithm in which both the actor and critic were implemented by two-layer ANNs trained by error backpropagation outperformed single-layer ANNs in the pole-balancing and tower of Hanoi tasks. Williams (1988) described several ways that backpropagation and reinforcement learning can be combined for training ANNs. Gullapalli (1990) and Williams (1992) devised reinforcement learning algorithms for neuron-like units having continuous, rather than binary, outputs. Barto, Sutton, and Watkins (1990) argued that ANNs can play significant roles for approximating functions required for solving sequential decision problems. Williams (1992) related REINFORCE learning rules (Section 13.3) to the error backpropagation method for training multi-layer ANNs. Tesauro's TD-Gammon (Tesauro 1992, 1994; Section 16.1) influentially demonstrated the learning abilities of TD(λ) algorithm with function approximation by multi-layer ANNs in learning to play backgammon. The *AlphaGo*, *AlphaGo Zero*, and *AlphaZero* programs of Silver et al. (2016, 2017a, b; Section 16.6) used reinforcement learning with

deep convolutional ANNs in achieving impressive results with the game of Go. Schmidhuber (2015) reviews applications of ANNs in reinforcement learning, including applications of recurrent ANNs.

9.8 LSTD is due to Bradtke and Barto (see Bradtke, 1993, 1994; Bradtke and Barto, 1996; Bradtke, Ydstie, and Barto, 1994), and was further developed by Boyan (1999, 2002), Nedić and Bertsekas (2003), and Yu (2010). The incremental update of the inverse matrix has been known at least since 1949 (Sherman and Morrison, 1949). An extension of least-squares methods to control was introduced by Lagoudakis and Parr (2003; Buşoniu, Lazaric, Ghavamzadeh, Munos, Babuška, and De Schutter, 2012).

9.9 Our discussion of memory-based function approximation is largely based on the review of locally weighted learning by Atkeson, Moore, and Schaal (1997). Atkeson (1992) discussed the use of locally weighted regression in memory-based robot learning and supplied an extensive bibliography covering the history of the idea. Stanfill and Waltz (1986) influentially argued for the importance of memory based methods in artificial intelligence, especially in light of parallel architectures then becoming available, such as the Connection Machine. Baird and Klopf (1993) introduced a novel memory-based approach and used it as the function approximation method for Q-learning applied to the pole-balancing task. Schaal and Atkeson (1994) applied locally weighted regression to a robot juggling control problem, where it was used to learn a system model. Peng (1995) used the pole-balancing task to experiment with several nearest-neighbor methods for approximating value functions, policies, and environment models. Tadepalli and Ok (1996) obtained promising results with locally-weighted linear regression to learn a value function for a simulated automatic guided vehicle task. Bottou and Vapnik (1992) demonstrated surprising efficiency of several local learning algorithms compared to non-local algorithms in some pattern recognition tasks, discussing the impact of local learning on generalization.

Bentley (1975) introduced k-d trees and reported observing average running time of $O(\log n)$ for nearest neighbor search over n records. Friedman, Bentley, and Finkel (1977) clarified the algorithm for nearest neighbor search with k-d trees. Omohundro (1987) discussed efficiency gains possible with hierarchical data structures such as k-d-trees. Moore, Schneider, and Deng (1997) introduced the use of k-d trees for efficient locally weighted regression.

9.10 The origin of kernel regression is the *method of potential functions* of Aizerman, Braverman, and Rozonoer (1964). They likened the data to point electric charges of various signs and magnitudes distributed over space. The resulting electric potential over space produced by summing the potentials of the point charges corresponded to the interpolated surface. In this analogy, the kernel function is the potential of a point charge, which falls off as the reciprocal of the distance from the charge. Connell and Utgoff (1987) applied an actor–critic method to the pole-balancing task in which the critic approximated the value function

using kernel regression with an inverse-distance weighting. Predating widespread interest in kernel regression in machine learning, these authors did not use the term kernel, but referred to "Shepard's method" (Shepard, 1968). Other kernel-based approaches to reinforcement learning include those of Ormoneit and Sen (2002), Dietterich and Wang (2002), Xu, Xie, Hu, and Lu (2005), Taylor and Parr (2009), Barreto, Precup, and Pineau (2011), and Bhat, Farias, and Moallemi (2012).

9.11 For Emphatic-TD methods, see the bibliographical notes to Section 11.8.

The earliest example we know of in which function approximation methods were used for learning value functions was Samuel's checkers player (1959, 1967). Samuel followed Shannon's (1950) suggestion that a value function did not have to be exact to be a useful guide to selecting moves in a game and that it might be approximated by a linear combination of features. In addition to linear function approximation, Samuel experimented with lookup tables and hierarchical lookup tables called signature tables (Griffith, 1966, 1974; Page, 1977; Biermann, Fairfield, and Beres, 1982).

At about the same time as Samuel's work, Bellman and Dreyfus (1959) proposed using function approximation methods with DP. (It is tempting to think that Bellman and Samuel had some influence on one another, but we know of no reference to the other in the work of either.) There is now a fairly extensive literature on function approximation methods and DP, such as multigrid methods and methods using splines and orthogonal polynomials (e.g., Bellman and Dreyfus, 1959; Bellman, Kalaba, and Kotkin, 1963; Daniel, 1976; Whitt, 1978; Reetz, 1977; Schweitzer and Seidmann, 1985; Chow and Tsitsiklis, 1991; Kushner and Dupuis, 1992; Rust, 1996).

Holland's (1986) classifier system used a selective feature-match technique to generalize evaluation information across state–action pairs. Each classifier matched a subset of states having specified values for a subset of features, with the remaining features having arbitrary values ("wild cards"). These subsets were then used in a conventional state-aggregation approach to function approximation. Holland's idea was to use a genetic algorithm to evolve a set of classifiers that collectively would implement a useful action-value function. Holland's ideas influenced the early research of the authors on reinforcement learning, but we focused on different approaches to function approximation. As function approximators, classifiers are limited in several ways. First, they are state-aggregation methods, with concomitant limitations in scaling and in representing smooth functions efficiently. In addition, the matching rules of classifiers can implement only aggregation boundaries that are parallel to the feature axes. Perhaps the most important limitation of conventional classifier systems is that the classifiers are learned via the genetic algorithm, an evolutionary method. As we discussed in Chapter 1, there is available during learning much more detailed information about how to learn than can be used by evolutionary methods. This perspective led us to instead adapt supervised learning methods for use in reinforcement learning, specifically gradient-descent and ANN methods. These differences between Holland's approach and ours are not surprising because Holland's ideas were developed during a period when ANNs were generally regarded as being too weak in computational power to be useful, whereas our work was at the beginning of

the period that saw widespread questioning of that conventional wisdom. There remain many opportunities for combining aspects of these different approaches.

Christensen and Korf (1986) experimented with regression methods for modifying coefficients of linear value function approximations in the game of chess. Chapman and Kaelbling (1991) and Tan (1991) adapted decision-tree methods for learning value functions. Explanation-based learning methods have also been adapted for learning value functions, yielding compact representations (Yee, Saxena, Utgoff, and Barto, 1990; Dietterich and Flann, 1995).

Chapter 10

On-policy Control with Approximation

In this chapter we return to the control problem, now with parametric approximation of the action-value function $\hat{q}(s, a, \mathbf{w}) \approx q_*(s, a)$, where $\mathbf{w} \in \mathbb{R}^d$ is a finite-dimensional weight vector. We continue to restrict attention to the on-policy case, leaving off-policy methods to Chapter 11. The present chapter features the semi-gradient Sarsa algorithm, the natural extension of semi-gradient TD(0) (last chapter) to action values and to on-policy control. In the episodic case, the extension is straightforward, but in the continuing case we have to take a few steps backward and re-examine how we have used discounting to define an optimal policy. Surprisingly, once we have genuine function approximation we have to give up discounting and switch to a new "average-reward" formulation of the control problem, with new "differential" value functions.

Starting first in the episodic case, we extend the function approximation ideas presented in the last chapter from state values to action values. Then we extend them to control following the general pattern of on-policy GPI, using ε-greedy for action selection. We show results for n-step linear Sarsa on the Mountain Car problem. Then we turn to the continuing case and repeat the development of these ideas for the average-reward case with differential values.

10.1 Episodic Semi-gradient Control

The extension of the semi-gradient prediction methods of Chapter 9 to action values is straightforward. In this case it is the approximate action-value function, $\hat{q} \approx q_\pi$, that is represented as a parameterized functional form with weight vector $\mathbf{w}$. Whereas before we considered random training examples of the form $S_t \mapsto U_t$, now we consider examples of the form $S_t, A_t \mapsto U_t$. The update target U_t can be any approximation of $q_\pi(S_t, A_t)$, including the usual backed-up values such as the full Monte Carlo return (G_t) or any of the n-step Sarsa returns (7.4). The general gradient-descent update for action-value

prediction is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[U_t - \hat{q}(S_t, A_t, \mathbf{w}_t)\Big]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t). \tag{10.1}$$

For example, the update for the one-step Sarsa method is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)\Big]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t). \tag{10.2}$$

We call this method *episodic semi-gradient one-step Sarsa*. For a constant policy, this method converges in the same way that TD(0) does, with the same kind of error bound (9.14).

To form control methods, we need to couple such action-value prediction methods with techniques for policy improvement and action selection. Suitable techniques applicable to continuous actions, or to actions from large discrete sets, are a topic of ongoing research with as yet no clear resolution. On the other hand, if the action set is discrete and not too large, then we can use the techniques already developed in previous chapters. That is, for each possible action a available in the next state S_{t+1}, we can compute $\hat{q}(S_{t+1}, a, \mathbf{w}_t)$ and then find the greedy action $A^*_{t+1} = \arg\max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t)$. Policy improvement is then done (in the on-policy case treated in this chapter) by changing the estimation policy to a soft approximation of the greedy policy such as the ε-greedy policy. Actions are selected according to this same policy. Pseudocode for the complete algorithm is given in the box.

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

Example 10.1: Mountain Car Task Consider the task of driving an underpowered car up a steep mountain road, as suggested by the diagram in the upper left of Figure 10.1. The difficulty is that gravity is stronger than the car's engine, and even at full throttle the car cannot accelerate up the steep slope. The only solution is to first move away from the goal and up the opposite slope on the left. Then, by applying full throttle the car

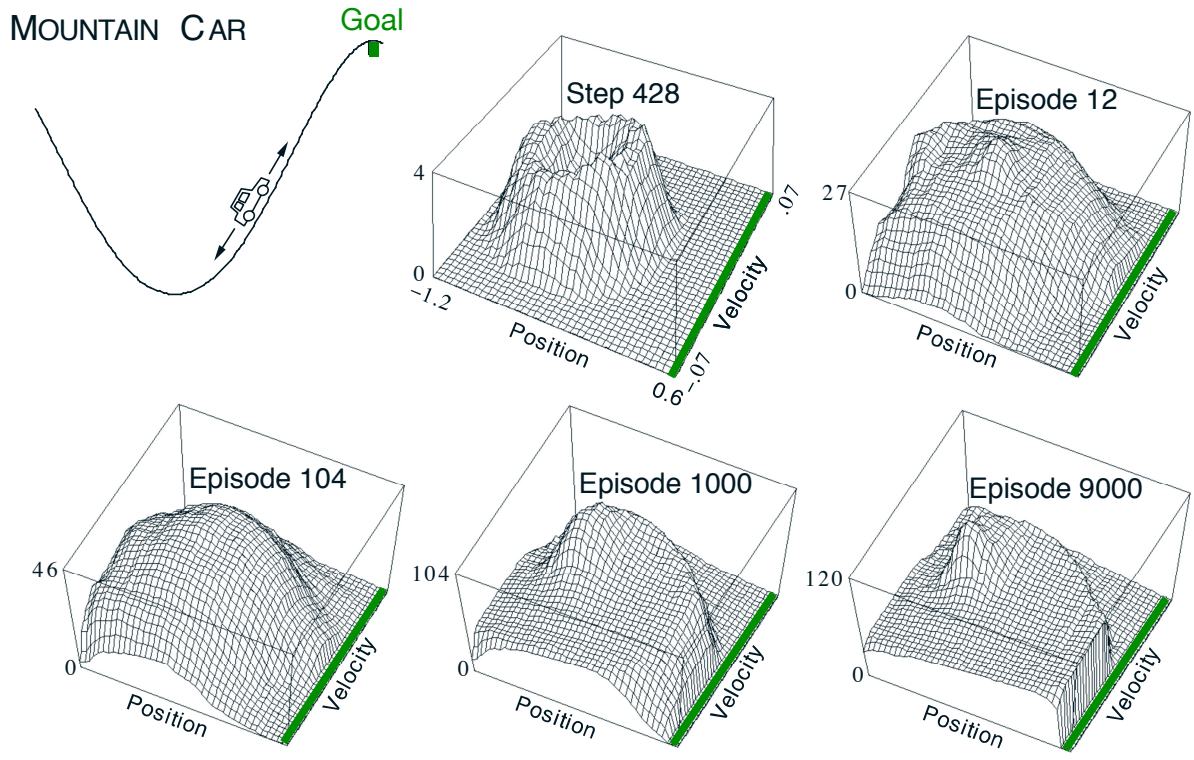


Figure 10.1: The Mountain Car task (upper left panel) and the cost-to-go function ($-\max_a \hat{q}(s, a, \mathbf{w})$) learned during one run.

can build up enough inertia to carry it up the steep slope even though it is slowing down the whole way. This is a simple example of a continuous control task where things have to get worse in a sense (farther from the goal) before they can get better. Many control methodologies have great difficulties with tasks of this kind unless explicitly aided by a human designer.

The reward in this problem is -1 on all time steps until the car moves past its goal position at the top of the mountain, which ends the episode. There are three possible actions: full throttle forward ($+1$), full throttle reverse (-1), and zero throttle (0). The car moves according to a simplified physics. Its position, x_t, and velocity, $\dot{x}_t$, are updated by

$$x_{t+1} \doteq \mathit{bound}\big[x_t + \dot{x}_{t+1}\big]$$

$$\dot{x}_{t+1} \doteq \mathit{bound}\big[\dot{x}_t + 0.001A_t - 0.0025\cos(3x_t)\big],$$

where the *bound* operation enforces $-1.2 \leq x_{t+1} \leq 0.5$ and $-0.07 \leq \dot{x}_{t+1} \leq 0.07$. In addition, when x_{t+1} reached the left bound, $\dot{x}_{t+1}$ was reset to zero. When it reached the right bound, the goal was reached and the episode was terminated. Each episode started from a random position $x_t \in [-0.6, -0.4)$ and zero velocity. To convert the two continuous state variables to binary features, we used grid-tilings as in Figure 9.9. We used 8 tilings, with each tile covering 1/8th of the bounded distance in each dimension,

and asymmetrical offsets as described in Section 9.5.4.[1] The feature vectors $\mathbf{x}(s,a)$ created by tile coding were then combined linearly with the parameter vector to approximate the action-value function:

$$\hat{q}(s,a,\mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s,a) = \sum_{i=1}^{d} w_i \cdot x_i(s,a), \tag{10.3}$$

for each pair of state, s, and action, a.

Figure 10.1 shows what typically happens while learning to solve this task with this form of function approximation.[2] Shown is the negative of the value function (the *cost-to-go* function) learned on a single run. The initial action values were all zero, which was optimistic (all true values are negative in this task), causing extensive exploration to occur even though the exploration parameter, ε, was 0. This can be seen in the middle-top panel of the figure, labeled "Step 428". At this time not even one episode had been completed, but the car has oscillated back and forth in the valley, following circular trajectories in state space. All the states visited frequently are valued worse than unexplored states, because the actual rewards have been worse than what was (unrealistically) expected. This continually drives the agent away from wherever it has been, to explore new states, until a solution is found.

Figure 10.2 shows several learning curves for semi-gradient Sarsa on this problem, with various step sizes.

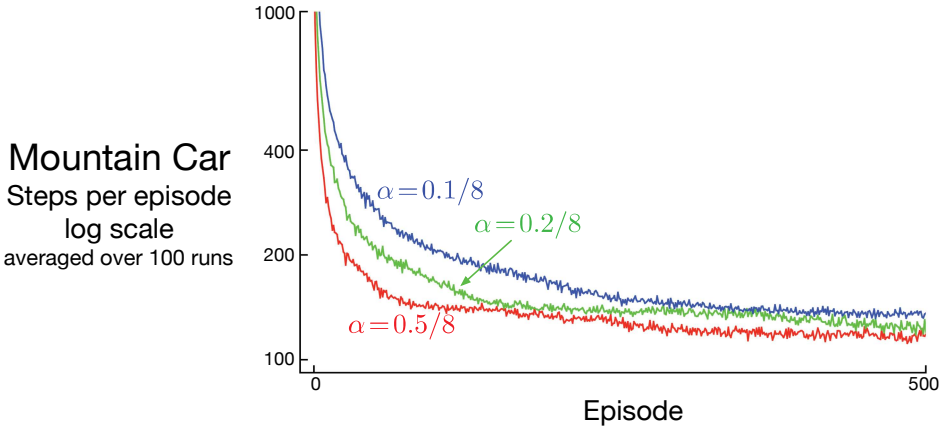


Figure 10.2: Mountain Car learning curves for the semi-gradient Sarsa method with tile-coding function approximation and ε-greedy action selection. ■

[1] In particular, we used the tile-coding software, available at `http://incompleteideas.net/tiles/tiles3.html`, with `iht=IHT(4096)` and `tiles(iht,8,[8*x/(0.5+1.2),8*xdot/(0.07+0.07)],[A])` to get the indices of the ones in the feature vector for state (`x`, `xdot`) and action `A`.

[2] This data is actually from the "semi-gradient Sarsa(λ)" algorithm that we will not meet until Chapter 12, but semi-gradient Sarsa would behave similarly.

10.2 Semi-gradient n-step Sarsa

We can obtain an n-step version of episodic semi-gradient Sarsa by using an n-step return as the update target in the semi-gradient Sarsa update equation (10.1). The n-step return immediately generalizes from its tabular form (7.4) to a function approximation form:

$$G_{t:t+n} \doteq R_{t+1}+\gamma R_{t+2}+\cdots+\gamma^{n-1}R_{t+n}+\gamma^{n}\hat{q}(S_{t+n},A_{t+n},\mathbf{w}_{t+n-1}), \quad t+n<T, \tag{10.4}$$

with $G_{t:t+n} \doteq G_t$ if $t+n \geq T$, as usual. The n-step update equation is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha\left[G_{t:t+n} - \hat{q}(S_t,A_t,\mathbf{w}_{t+n-1})\right]\nabla\hat{q}(S_t,A_t,\mathbf{w}_{t+n-1}), \qquad 0 \leq t < T. \tag{10.5}$$

Complete pseudocode is given in the box below.

Episodic semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_*$ or q_π

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Input: a policy π (if estimating q_π)
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$, a positive integer n
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations (S_t, A_t, and R_t) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$ or ε-greedy wrt $\hat{q}(S_0,\cdot,\mathbf{w})$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
    If $t < T$, then:
      Take action A_t
      Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
      If S_{t+1} is terminal, then:
        $T \leftarrow t+1$
      else:
        Select and store $A_{t+1} \sim \pi(\cdot|S_{t+1})$ or ε-greedy wrt $\hat{q}(S_{t+1},\cdot,\mathbf{w})$
    $\tau \leftarrow t-n+1$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1}R_i$
      If $\tau+n < T$, then $G \leftarrow G + \gamma^n\hat{q}(S_{\tau+n},A_{\tau+n},\mathbf{w})$ ($G_{\tau:\tau+n}$)
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\left[G - \hat{q}(S_\tau,A_\tau,\mathbf{w})\right]\nabla\hat{q}(S_\tau,A_\tau,\mathbf{w})$
  Until $\tau = T-1$

As we have seen before, performance is best if an intermediate level of bootstrapping is used, corresponding to an n larger than 1. Figure 10.3 shows how this algorithm tends to learn faster and obtain a better asymptotic performance at $n=8$ than at $n=1$ on the Mountain Car task. Figure 10.4 shows the results of a more detailed study of the effect of the parameters α and n on the rate of learning on this task.

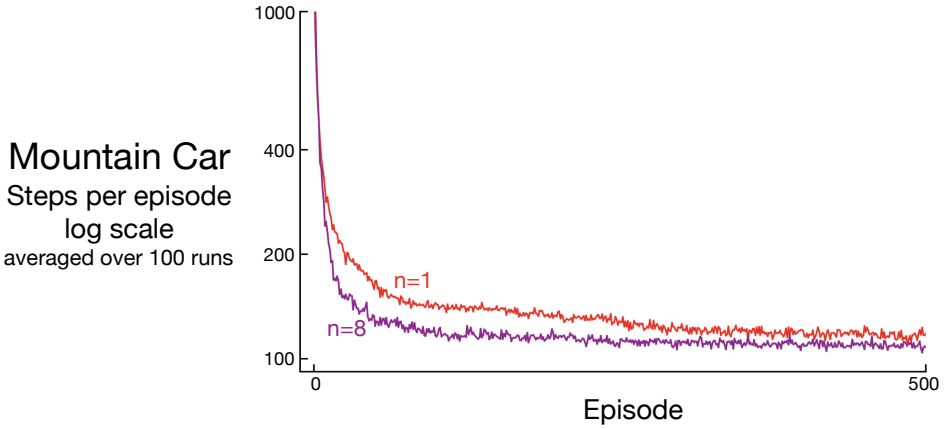


Figure 10.3: Performance of one-step vs 8-step semi-gradient Sarsa on the Mountain Car task. Good step sizes were used: $\alpha = 0.5/8$ for $n = 1$ and $\alpha = 0.3/8$ for $n = 8$.

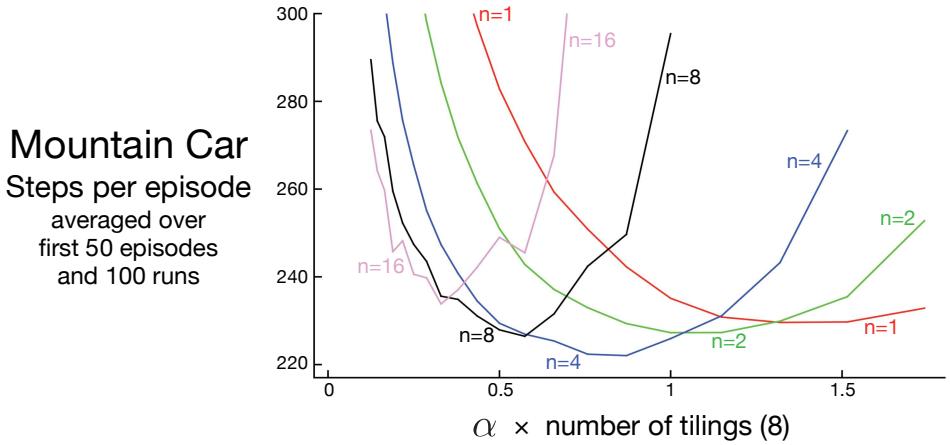


Figure 10.4: Effect of the α and n on early performance of n-step semi-gradient Sarsa and tile-coding function approximation on the Mountain Car task. As usual, an intermediate level of bootstrapping ($n = 4$) performed best. These results are for selected α values, on a log scale, and then connected by straight lines. The standard errors ranged from 0.5 (less than the line width) for $n = 1$ to about 4 for $n = 16$, so the main effects are all statistically significant.

Exercise 10.1 We have not explicitly considered or given pseudocode for any Monte Carlo methods in this chapter. What would they be like? Why is it reasonable not to give pseudocode for them? How would they perform on the Mountain Car task? □

Exercise 10.2 Give pseudocode for semi-gradient one-step *Expected* Sarsa for control. □

Exercise 10.3 Why do the results shown in Figure 10.4 have higher standard errors at large n than at small n? □

10.3 Average Reward: A New Problem Setting for Continuing Tasks

We now introduce a third classical setting—alongside the episodic and discounted settings—for formulating the goal in Markov decision problems (MDPs). Like the discounted setting, the *average reward* setting applies to continuing problems, problems for which the interaction between agent and environment goes on and on forever without termination or start states. Unlike that setting, however, there is no discounting—the agent cares just as much about delayed rewards as it does about immediate reward. The average-reward setting is one of the major settings commonly considered in the classical theory of dynamic programming and less-commonly in reinforcement learning. As we discuss in the next section, the discounted setting is problematic with function approximation, and thus the average-reward setting is needed to replace it.

In the average-reward setting, the quality of a policy π is defined as the average rate of reward, or simply *average reward*, while following that policy, which we denote as $r(\pi)$:

$$\begin{aligned} r(\pi) &\doteq \lim_{h\to\infty} \frac{1}{h} \sum_{t=1}^{h} \mathbb{E}[R_t \mid S_0, A_{0:t-1} \sim \pi] && (10.6)\\ &= \lim_{t\to\infty} \mathbb{E}[R_t \mid S_0, A_{0:t-1} \sim \pi]\,, && (10.7)\\ &= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a) r, \end{aligned}$$

where the expectations are conditioned on the initial state, S_0, and on the subsequent actions, $A_0, A_1, \ldots, A_{t-1}$, being taken according to π. The second and third equations hold if the steady-state distribution, $\mu_\pi(s) \doteq \lim_{t\to\infty} \Pr\{S_t = s \mid A_{0:t-1} \sim \pi\}$, exists and is independent of S_0, in other words, if the MDP is *ergodic*. In an ergodic MDP, the starting state and any early decision made by the agent can have only a temporary effect; in the long run the expectation of being in a state depends only on the policy and the MDP transition probabilities. Ergodicity is sufficient but not necessary to guarantee the existence of the limit in (10.6).

There are subtle distinctions that can be drawn between different kinds of optimality in the undiscounted continuing case. Nevertheless, for most practical purposes it may be adequate simply to order policies according to their average reward per time step, in other words, according to their $r(\pi)$. This quantity is essentially the average reward under π, as suggested by (10.7), or the *reward rate*. In particular, we consider all policies that attain the maximal value of $r(\pi)$ to be optimal.

Note that the steady state distribution μ_π is the special distribution under which, if you select actions according to π, you remain in the same distribution. That is, for which

$$\sum_s \mu_\pi(s) \sum_a \pi(a|s) p(s' \,|\, s, a) = \mu_\pi(s'). \tag{10.8}$$

In the average-reward setting, returns are defined in terms of differences between rewards and the average reward:

$$G_t \doteq R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \cdots. \tag{10.9}$$

This is known as the *differential* return, and the corresponding value functions are known as *differential* value functions. Differential value functions are defined in terms of the new return just as conventional value functions were defined in terms of the discounted return; thus we will use the same notation, $v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s]$ and $q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$ (similarly for v_* and q_*), for differential value functions. Differential value functions also have Bellman equations, just slightly different from those we have seen earlier. We simply remove all γs and replace all rewards by the difference between the reward and the true average reward:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{r,s'} p(s', r \,|\, s, a)\Big[r - r(\pi) + v_\pi(s')\Big],$$

$$q_\pi(s,a) = \sum_{r,s'} p(s', r \,|\, s, a)\Big[r - r(\pi) + \sum_{a'} \pi(a'|s') q_\pi(s', a')\Big],$$

$$v_*(s) = \max_a \sum_{r,s'} p(s', r \,|\, s, a)\Big[r - \max_\pi r(\pi) + v_*(s')\Big], \text{ and}$$

$$q_*(s,a) = \sum_{r,s'} p(s', r \,|\, s, a)\Big[r - \max_\pi r(\pi) + \max_{a'} q_*(s', a')\Big]$$

(cf. (3.14), Exercise 3.17, (3.19), and (3.20)).

There is also a differential form of the two TD errors:

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t), \tag{10.10}$$

and

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t), \tag{10.11}$$

where $\bar{R}_t$ is an estimate at time t of the average reward $r(\pi)$. With these alternate definitions, most of our algorithms and many theoretical results carry through to the average-reward setting without change.

For example, an average reward version of semi-gradient Sarsa could be defined just as in (10.2) except with the differential version of the TD error. That is, by

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \delta_t \nabla \hat{q}(S_t, A_t, \mathbf{w}_t), \tag{10.12}$$

with δ_t given by (10.11). Pseudocode for a complete algorithm is given in the box on the next page. One limitation of this algorithm is that it does not converge to the differential values but to the differential values plus an arbitrary offset. Notice that the Bellman equations and TD errors given above are unaffected if all the values are shifted by the same amount. Thus, the offset may not matter in practice. How this algorithm could be changed to eliminate the offset is an interesting question for future research.

Exercise 10.4 Give pseudocode for a differential version of semi-gradient Q-learning. □

Exercise 10.5 What equations are needed (beyond 10.10) to specify the differential version of TD(0)? □

Differential semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step sizes $\alpha, \beta > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)

Initialize state S, and action A
Loop for each step:
  Take action A, observe R, S'
  Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
  $\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$
  $\bar{R} \leftarrow \bar{R} + \beta\delta$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S, A, \mathbf{w})$
  $S \leftarrow S'$
  $A \leftarrow A'$

Exercise 10.6 Suppose there is an MDP that under any policy produces the deterministic sequence of rewards $+1, 0, +1, 0, +1, 0, \ldots$ going on forever. Technically, this violates ergodicity; there is no stationary limiting distribution μ_π and the limit (10.7) does not exist. Nevertheless, the average reward (10.6) is well defined. What is it? Now consider two states in this MDP. From A, the reward sequence is exactly as described above, starting with a $+1$, whereas, from B, the reward sequence starts with a 0 and then continues with $+1, 0, +1, 0, \ldots$. We would like to compute the differential values of A and B. Unfortunately, the differential return (10.9) is not well defined when starting from these states as the implicit limit does not exist. To repair this, one could alternatively define the differential value of a state as

$$v_\pi(s) \doteq \lim_{\gamma \to 1} \lim_{h \to \infty} \sum_{t=0}^{h} \gamma^t \Big(\mathbb{E}_\pi[R_{t+1} | S_0 = s] - r(\pi) \Big). \tag{10.13}$$

Under this definition, what are the differential values of states A and B? □

Exercise 10.7 Consider a Markov reward process consisting of a ring of three states A, B, and C, with state transitions going deterministically around the ring. A reward of $+1$ is received upon arrival in A and otherwise the reward is 0. What are the differential values of the three states, using (10.13)? □

Exercise 10.8 The pseudocode in the box on page 251 updates $\bar{R}_t$ using δ_t as an error rather than simply $R_{t+1} - \bar{R}_t$. Both errors work, but using δ_t is better. To see why, consider the ring MRP of three states from Exercise 10.7. The estimate of the average reward should tend towards its true value of $\frac{1}{3}$. Suppose it was already there and was held stuck there. What would the sequence of $R_{t+1} - \bar{R}_t$ errors be? What would the sequence of δ_t errors be (using Equation 10.10)? Which error sequence would produce a more stable estimate of the average reward if the estimate were allowed to change in response to the errors? Why? □

Example 10.2: An Access-Control Queuing Task This is a decision task involving access control to a set of 10 servers. Customers of four different priorities arrive at a single queue. If given access to a server, the customers pay a reward of 1, 2, 4, or 8 to the server, depending on their priority, with higher priority customers paying more. In each time step, the customer at the head of the queue is either accepted (assigned to one of the servers) or rejected (removed from the queue, with a reward of zero). In either case, on the next time step the next customer in the queue is considered. The queue never empties, and the priorities of the customers in the queue are uniformly randomly distributed. Of course a customer cannot be served if there is no free server; the customer is always rejected in this case. Each busy server becomes free with probability $p = 0.06$ on each time step. Although we have just described them for definiteness, let us assume the statistics of arrivals and departures are unknown. The task is to decide on each step whether to accept or reject the next customer, on the basis of his priority and the number of free servers, so as to maximize long-term reward without discounting.

In this example we consider a tabular solution to this problem. Although there is no generalization between states, we can still consider it in the general function approximation setting as this setting generalizes the tabular setting. Thus we have a differential action-value estimate for each pair of state (number of free servers and priority of the customer at the head of the queue) and action (accept or reject). Figure 10.5 shows the solution found by differential semi-gradient Sarsa with parameters $\alpha = 0.01$, $\beta = 0.01$, and $\varepsilon = 0.1$. The initial action values and $\bar{R}$ were zero.

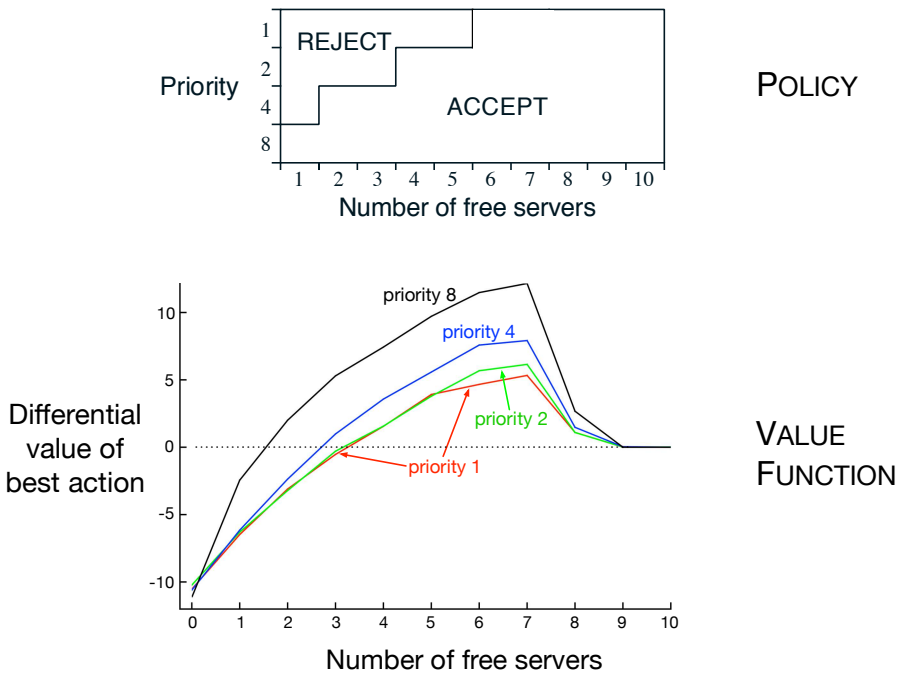


Figure 10.5: The policy and value function found by differential semi-gradient one-step Sarsa on the access-control queuing task after 2 million steps. The drop on the right of the graph is probably due to insufficient data; many of these states were never experienced. The value learned for $\bar{R}$ was about 2.31. (Note that priority 1 here is the lowest priority.) ■

10.4 Deprecating the Discounted Setting

The continuing, discounted problem formulation has been very useful in the tabular case, in which the returns from each state can be separately identified and averaged. But in the approximate case it is questionable whether one should ever use this problem formulation.

To see why, consider an infinite sequence of returns with no beginning or end, and no clearly identified states. The states might be represented only by feature vectors, which may do little to distinguish the states from each other. As a special case, all of the feature vectors may be the same. Thus one really has only the reward sequence (and the actions), and performance has to be assessed purely from these. How could it be done? One way is by averaging the rewards over a long interval—this is the idea of the average-reward setting. How could discounting be used? Well, for each time step we could measure the discounted return. Some returns would be small and some big, so again we would have to average them over a sufficiently large time interval. In the continuing setting there are no starts and ends, and no special time steps, so there is nothing else that could be done. However, if you do this, it turns out that the average of the discounted returns is proportional to the average reward. In fact, for policy π, the average of the discounted returns is always $r(\pi)/(1-\gamma)$, that is, it is essentially the average reward, $r(\pi)$. In particular, the *ordering* of all policies in the average discounted return setting would be exactly the same as in the average-reward setting. The discount rate γ thus has no effect on the problem formulation. It could in fact be *zero* and the ranking would be unchanged.

This surprising fact is proven in the box on the next page, but the basic idea can be seen via a symmetry argument. Each time step is exactly the same as every other. With discounting, every reward will appear exactly once in each position in some return. The tth reward will appear undiscounted in the $t-1$st return, discounted once in the $t-2$nd return, and discounted 999 times in the $t-1000$th return. The weight on the tth reward is thus $1+\gamma+\gamma^2+\gamma^3+\cdots = 1/(1-\gamma)$. Because all states are the same, they are all weighted by this, and thus the average of the returns will be this times the average reward, or $r(\pi)/(1-\gamma)$.

This example and the more general argument in the box show that if we optimized discounted value over the on-policy distribution, then the effect would be identical to optimizing *undiscounted* average reward; the actual value of γ would have no effect. This strongly suggests that discounting has no role to play in the definition of the control problem with function approximation. One can nevertheless go ahead and use discounting in solution methods. The discounting parameter γ changes from a problem parameter to a solution method parameter! Unfortunately, discounting algorithms with function approximation do not optimize discounted value over the on-policy distribution, and thus are not guaranteed to optimize average reward.

The root cause of the difficulties with the discounted control setting is that with function approximation we have lost the policy improvement theorem (Section 4.2). It is no longer true that if we change the policy to improve the discounted value of one state then we are guaranteed to have improved the overall policy in any useful sense. That guarantee was key to the theory of our reinforcement learning control methods. With

The Futility of Discounting in Continuing Problems

Perhaps discounting can be saved by choosing an objective that sums discounted values over the distribution with which states occur under the policy:

$$
\begin{aligned}
J(\pi) &= \sum_s \mu_\pi(s) v_\pi^\gamma(s) && \text{(where } v_\pi^\gamma \text{ is the discounted value function)} \\
&= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \,|\, s, a) \left[r + \gamma v_\pi^\gamma(s') \right] && \text{(Bellman Eq.)} \\
&= r(\pi) + \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \,|\, s, a) \gamma v_\pi^\gamma(s') && \text{(from (10.7))} \\
&= r(\pi) + \gamma \sum_{s'} v_\pi^\gamma(s') \sum_s \mu_\pi(s) \sum_a \pi(a|s) p(s' \,|\, s, a) && \text{(from (3.4))} \\
&= r(\pi) + \gamma \sum_{s'} v_\pi^\gamma(s') \mu_\pi(s') && \text{(from (10.8))} \\
&= r(\pi) + \gamma J(\pi) \\
&= r(\pi) + \gamma r(\pi) + \gamma^2 J(\pi) \\
&= r(\pi) + \gamma r(\pi) + \gamma^2 r(\pi) + \gamma^3 r(\pi) + \cdots \\
&= \frac{1}{1-\gamma} r(\pi).
\end{aligned}
$$

The proposed discounted objective orders policies identically to the undiscounted (average reward) objective. The discount rate γ does not influence the ordering!

function approximation we have lost it!

In fact, the lack of a policy improvement theorem is also a theoretical lacuna for the total-episodic and average-reward settings. Once we introduce function approximation we can no longer guarantee improvement for any setting. In Chapter 13 we introduce an alternative class of reinforcement learning algorithms based on parameterized policies, and there we have a theoretical guarantee called the "policy-gradient theorem" which plays a similar role as the policy improvement theorem. But for methods that learn action values we seem to be currently without a local improvement guarantee (possibly the approach taken by Perkins and Precup (2003) may provide a part of the answer). We do know that ε-greedification may sometimes result in an inferior policy, as policies may chatter among good policies rather than converge (Gordon, 1996a). This is an area with multiple open theoretical questions.

10.5 Differential Semi-gradient n-step Sarsa

In order to generalize to *n-step* bootstrapping, we need an n-step version of the TD error. We begin by generalizing the n-step return (7.4) to its differential form, with function approximation:

$$G_{t:t+n} \doteq R_{t+1} - \bar{R}_{t+n-1} + \cdots + R_{t+n} - \bar{R}_{t+n-1} + \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \quad (10.14)$$

where $\bar{R}$ is an estimate of $r(\pi)$, $n \geq 1$, and $t + n < T$. If $t + n \geq T$, then we define $G_{t:t+n} \doteq G_t$ as usual. The n-step TD error is then

$$\delta_t \doteq G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}), \quad (10.15)$$

after which we can apply our usual semi-gradient Sarsa update (10.12). Pseudocode for the complete algorithm is given in the box.

Differential semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_\pi$ or q_*

Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$, a policy π
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average-reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)
Algorithm parameters: step sizes $\alpha, \beta > 0$, small $\varepsilon > 0$, a positive integer n
All store and access operations (S_t, A_t, and R_t) can take their index mod $n + 1$

Initialize and store S_0 and A_0
Loop for each step, $t = 0, 1, 2, \ldots$:
  Take action A_t
  Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$, or ε-greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
  $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
  If $\tau \geq 0$:
    $\delta \leftarrow \sum_{i=\tau+1}^{\tau+n} (R_i - \bar{R}) + \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w}) - \hat{q}(S_\tau, A_\tau, \mathbf{w})$
    $\bar{R} \leftarrow \bar{R} + \beta\delta$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S_\tau, A_\tau, \mathbf{w})$

Exercise 10.9 In the differential semi-gradient n-step Sarsa algorithm, the step-size parameter on the average reward, β, needs to be quite small so that $\bar{R}$ becomes a good long-term estimate of the average reward. Unfortunately, $\bar{R}$ will then be biased by its initial value for many steps, which may make learning inefficient. Alternatively, one could use a sample average of the observed rewards for $\bar{R}$. That would initially adapt rapidly but in the long run would also adapt slowly. As the policy slowly changed, $\bar{R}$ would also change; the potential for such long-term nonstationarity makes sample-average methods ill-suited. In fact, the step-size parameter on the average reward is a perfect place to use the unbiased constant-step-size trick from Exercise 2.7. Describe the specific changes needed to the boxed algorithm for differential semi-gradient n-step Sarsa to use this trick. □

10.6 Summary

In this chapter we have extended the ideas of parameterized function approximation and semi-gradient descent, introduced in the previous chapter, to control. The extension is immediate for the episodic case, but for the continuing case we have to introduce a whole new problem formulation based on maximizing the *average reward setting* per time step. Surprisingly, the discounted formulation cannot be carried over to control in the presence of approximations. In the approximate case most policies cannot be represented by a value function. The arbitrary policies that remain need to be ranked, and the scalar average reward $r(\pi)$ provides an effective way to do this.

The average reward formulation involves new *differential* versions of value functions, Bellman equations, and TD errors, but all of these parallel the old ones, and the conceptual changes are small. There is also a new parallel set of differential algorithms for the average-reward case.

Bibliographical and Historical Remarks

10.1 Semi-gradient Sarsa with function approximation was first explored by Rummery and Niranjan (1994). Linear semi-gradient Sarsa with ε-greedy action selection does not converge in the usual sense, but does enter a bounded region near the best solution (Gordon, 1996a, 2001). Precup and Perkins (2003) showed convergence in a differentiable action selection setting. See also Perkins and Pendrith (2002) and Melo, Meyn, and Ribeiro (2008). The mountain–car example is based on a similar task studied by Moore (1990), but the exact form used here is from Sutton (1996).

10.2 Episodic n-step semi-gradient Sarsa is based on the forward Sarsa(λ) algorithm of van Seijen (2016). The empirical results shown here are new to the second edition of this text.

10.3 The average-reward formulation has been described for dynamic programming (e.g., Puterman, 1994) and from the point of view of reinforcement learning (Mahadevan, 1996; Tadepalli and Ok, 1994; Bertsekas and Tsitsiklis, 1996; Tsitsiklis and Van Roy, 1999). The algorithm described here is the on-policy analog of the "R-learning" algorithm introduced by Schwartz (1993). 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 differential or *relative* values. The access-control queuing example was suggested by the work of Carlström and Nordström (1997).

10.4 The recognition of the limitations of discounting as a formulation of the reinforcement learning problem with function approximation became apparent to the authors shortly after the publication of the first edition of this text. Singh, Jaakkola, and Jordan (1994) may have been the first to observe it in print.

Chapter 11

*Off-policy Methods with Approximation

This book has treated on-policy and off-policy learning methods since Chapter 5 primarily as two alternative ways of handling the conflict between exploitation and exploration inherent in learning forms of generalized policy iteration. The two chapters preceding this have treated the *on*-policy case with function approximation, and in this chapter we treat the *off*-policy case with function approximation. The extension to function approximation turns out to be significantly different and harder for off-policy learning than it is for on-policy learning. The tabular off-policy methods developed in Chapters 6 and 7 readily extend to semi-gradient algorithms, but these algorithms do not converge as robustly as they do under on-policy training. In this chapter we explore the convergence problems, take a closer look at the theory of linear function approximation, introduce a notion of learnability, and then discuss new algorithms with stronger convergence guarantees for the off-policy case. In the end we will have improved methods, but the theoretical results will not be as strong, nor the empirical results as satisfying, as they are for on-policy learning. Along the way, we will gain a deeper understanding of approximation in reinforcement learning for on-policy learning as well as off-policy learning.

Recall that in off-policy learning we seek to learn a value function for a *target policy* π, given data due to a different *behavior policy* b. In the prediction case, both policies are static and given, and we seek to learn either state values $\hat{v} \approx v_\pi$ or action values $\hat{q} \approx q_\pi$. In the control case, action values are learned, and both policies typically change during learning—π being the greedy policy with respect to $\hat{q}$, and b being something more exploratory such as the ε-greedy policy with respect to $\hat{q}$.

The challenge of off-policy learning can be divided into two parts, one that arises in the tabular case and one that arises only with function approximation. The first part of the challenge has to do with the target of the update (not to be confused with the target policy), and the second part has to do with the distribution of the updates. The techniques related to importance sampling developed in Chapters 5 and 7 deal with the first part; these may increase variance but are needed in all successful algorithms,

tabular and approximate. The extension of these techniques to function approximation are quickly dealt with in the first section of this chapter.

Something more is needed for the second part of the challenge of off-policy learning with function approximation because the distribution of updates in the off-policy case is not according to the on-policy distribution. The on-policy distribution is important to the stability of semi-gradient methods. Two general approaches have been explored to deal with this. One is to use importance sampling methods again, this time to warp the update distribution back to the on-policy distribution, so that semi-gradient methods are guaranteed to converge (in the linear case). The other is to develop true gradient methods that do not rely on any special distribution for stability. We present methods based on both approaches. This is a cutting-edge research area, and it is not clear which of these approaches is most effective in practice.

11.1 Semi-gradient Methods

We begin by describing how the methods developed in earlier chapters for the off-policy case extend readily to function approximation as semi-gradient methods. These methods address the first part of the challenge of off-policy learning (changing the update targets) but not the second part (changing the update distribution). Accordingly, these methods may diverge in some cases, and in that sense are not sound, but still they are often successfully used. Remember that these methods *are* guaranteed stable and asymptotically unbiased for the tabular case, which corresponds to a special case of function approximation. So it may still be possible to combine them with feature selection methods in such a way that the combined system could be assured stable. In any event, these methods are simple and thus a good place to start.

In Chapter 7 we described a variety of tabular off-policy algorithms. To convert them to semi-gradient form, we simply replace the update to an array (V or Q) to an update to a weight vector ($\mathbf{w}$), using the approximate value function ($\hat{v}$ or $\hat{q}$) and its gradient. Many of these algorithms use the per-step importance sampling ratio:

$$\rho_t \doteq \rho_{t:t} = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}. \tag{11.1}$$

For example, the one-step, state-value algorithm is semi-gradient off-policy TD(0), which is just like the corresponding on-policy algorithm (page 203) except for the addition of ρ_t:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\rho_t\delta_t\nabla\hat{v}(S_t,\mathbf{w}_t), \tag{11.2}$$

where δ_t is defined appropriately depending on whether the problem is episodic and discounted, or continuing and undiscounted using average reward:

$$\delta_t \doteq R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t), \text{ or} \tag{11.3}$$

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t). \tag{11.4}$$

For action values, the one-step algorithm is semi-gradient Expected Sarsa:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t \nabla \hat{q}(S_t, A_t, \mathbf{w}_t), \text{ with} \tag{11.5}$$

$$\delta_t \doteq R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})\hat{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t), \text{ or} \qquad \text{(episodic)}$$

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \sum_a \pi(a|S_{t+1})\hat{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t). \qquad \text{(continuing)}$$

Note that this algorithm does not use importance sampling. In the tabular case it is clear that this is appropriate because the only sample action is A_t, and in learning its value we do not have to consider any other actions. With function approximation it is less clear because we might want to weight different state–action pairs differently once they all contribute to the same overall approximation. Proper resolution of this issue awaits a more thorough understanding of the theory of function approximation in reinforcement learning.

In the multi-step generalizations of these algorithms, both the state-value and action-value algorithms involve importance sampling. The n-step version of semi-gradient Sarsa is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha\rho_{t+1}\cdots\rho_{t+n}\left[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})\right]\nabla\hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}) \tag{11.6}$$

with

$$G_{t:t+n} \doteq R_{t+1} + \cdots + \gamma^{n-1}R_{t+n} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \text{ or} \qquad \text{(episodic)}$$

$$G_{t:t+n} \doteq R_{t+1} - \bar{R}_t + \cdots + R_{t+n} - \bar{R}_{t+n-1} + \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \qquad \text{(continuing)}$$

where here we are being slightly informal in our treatment of the ends of episodes. In the first equation, the ρ_ks for $k \geq T$ (where T is the last time step of the episode) should be taken to be 1, and $G_{t:t+n}$ should be taken to be G_t if $t + n \geq T$.

Recall that we also presented in Chapter 7 an off-policy algorithm that does not involve importance sampling at all: the n-step tree-backup algorithm. Here is its semi-gradient version:

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha\left[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})\right]\nabla\hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}), \tag{11.7}$$

$$G_{t:t+n} \doteq \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}) + \sum_{k=t}^{t+n-1} \delta_k \prod_{i=t+1}^{k} \gamma\pi(A_i|S_i), \tag{11.8}$$

with δ_t as defined at the top of this page for Expected Sarsa. We also defined in Chapter 7 an algorithm that unifies all action-value algorithms: n-step $Q(\sigma)$. We leave the semi-gradient form of that algorithm, and also of the n-step state-value algorithm, as exercises for the reader.

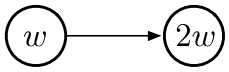
Exercise 11.1 Convert the equation of n-step off-policy TD (7.9) to semi-gradient form. Give accompanying definitions of the return for both the episodic and continuing cases. □

**Exercise 11.2* Convert the equations of n-step $Q(\sigma)$ (7.11 and 7.17) to semi-gradient form. Give definitions that cover both the episodic and continuing cases. □

11.2 Examples of Off-policy Divergence

In this section we begin to discuss the second part of the challenge of off-policy learning with function approximation—that the distribution of updates does not match the on-policy distribution. We describe some instructive counterexamples to off-policy learning—cases where semi-gradient and other simple algorithms are unstable and diverge.

To establish intuitions, it is best to consider first a very simple example. Suppose, perhaps as part of a larger MDP, there are two states whose estimated values are of the functional form w and $2w$, where the parameter vector $\mathbf{w}$ consists of only a single component w. This occurs under linear function approximation if the feature vectors for the two states are each simple numbers (single-component vectors), in this case 1 and 2. In the first state, only one action is available, and it results deterministically in a transition to the second state with a reward of 0:



where the expressions inside the two circles indicate the two state's values.

Suppose initially $w = 10$. The transition will then be from a state of estimated value 10 to a state of estimated value 20. It will look like a good transition, and w will be increased to raise the first state's estimated value. If γ is nearly 1, then the TD error will be nearly 10, and, if $\alpha = 0.1$, then w will be increased to nearly 11 in trying to reduce the TD error. However, the second state's estimated value will also be increased, to nearly 22. If the transition occurs again, then it will be from a state of estimated value $\approx$11 to a state of estimated value $\approx$22, for a TD error of $\approx$11—larger, not smaller, than before. It will look even more like the first state is undervalued, and its value will be increased again, this time to $\approx$12.1. This looks bad, and in fact with further updates w will diverge to infinity.

To see this definitively we have to look more carefully at the sequence of updates. The TD error on a transition between the two states is

$$\delta_t = R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t) = 0 + \gamma 2w_t - w_t = (2\gamma - 1)w_t,$$

and the off-policy semi-gradient TD(0) update (from (11.2)) is

$$w_{t+1} = w_t + \alpha\rho_t\delta_t\nabla\hat{v}(S_t,w_t) = w_t + \alpha \cdot 1 \cdot (2\gamma - 1)w_t \cdot 1 = \big(1 + \alpha(2\gamma - 1)\big)w_t.$$

Note that the importance sampling ratio, ρ_t, is 1 on this transition because there is only one action available from the first state, so its probabilities of being taken under the target and behavior policies must both be 1. In the final update above, the new parameter is the old parameter times a scalar constant, $1 + \alpha(2\gamma - 1)$. If this constant is greater than 1, then the system is unstable and w will go to positive or negative infinity depending on its initial value. Here this constant is greater than 1 whenever $\gamma > 0.5$. Note that stability does not depend on the specific step size, as long as $\alpha > 0$. Smaller or larger step sizes would affect the rate at which w goes to infinity, but not whether it goes there or not.

Key to this example is that the one transition occurs repeatedly without w being updated on other transitions. This is possible under off-policy training because the

behavior policy might select actions on those other transitions which the target policy never would. For these transitions, ρ_t would be zero and no update would be made. Under on-policy training, however, ρ_t is always one. Each time there is a transition from the w state to the $2w$ state, increasing w, there would also have to be a transition out of the $2w$ state. That transition would reduce w, unless it were to a state whose value was higher (because $\gamma < 1$) than $2w$, and then that state would have to be followed by a state of even higher value, or else again w would be reduced. Each state can support the one before only by creating a higher expectation. Eventually the piper must be paid. In the on-policy case the promise of future reward must be kept and the system is kept in check. But in the off-policy case, a promise can be made and then, after taking an action that the target policy never would, forgotten and forgiven.

This simple example communicates much of the reason why off-policy training can lead to divergence, but it is not completely convincing because it is not complete—it is just a fragment of a complete MDP. Can there really be a complete system with instability? A simple complete example of divergence is *Baird's counterexample*. Consider the episodic seven-state, two-action MDP shown in Figure 11.1. The dashed action takes the system to one of the six upper states with equal probability, whereas the solid action takes the system to the seventh state. The behavior policy b selects the dashed and solid actions with probabilities $\frac{6}{7}$ and $\frac{1}{7}$, so that the next-state distribution under it is uniform (the same for all nonterminal states), which is also the starting distribution for each episode. The target policy π always takes the solid action, and so the on-policy distribution (for π) is concentrated in the seventh state. The reward is zero on all transitions. The discount rate is $\gamma = 0.99$.

Consider estimating the state-value under the linear parameterization indicated by the expression shown in each state circle. For example, the estimated value of the leftmost state is $2w_1 + w_8$, where the subscript corresponds to the component of the

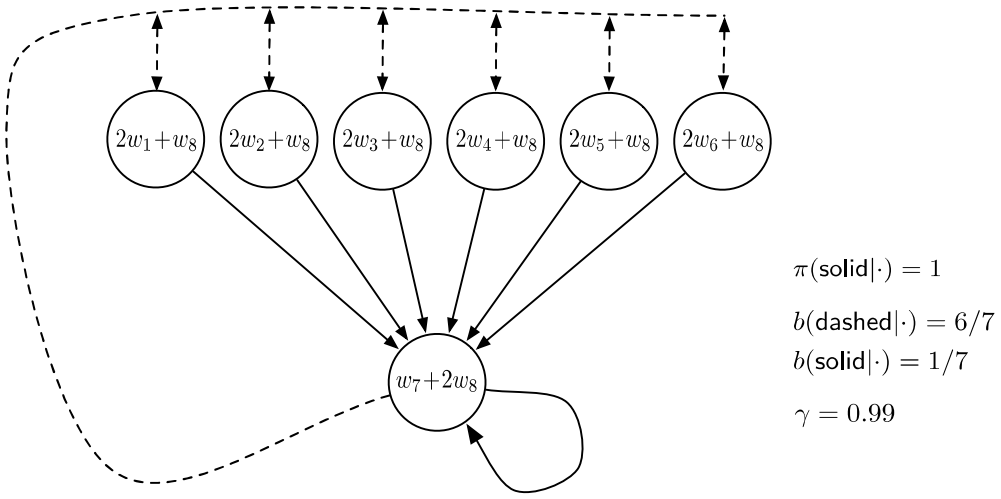


Figure 11.1: Baird's counterexample. The approximate state-value function for this Markov process is of the form shown by the linear expressions inside each state. The solid action usually results in the seventh state, and the dashed action usually results in one of the other six states, each with equal probability. The reward is always zero.

overall weight vector $\mathbf{w} \in \mathbb{R}^8$; this corresponds to a feature vector for the first state being $\mathbf{x}(1) = (2,0,0,0,0,0,0,1)^\top$. The reward is zero on all transitions, so the true value function is $v_\pi(s) = 0$, for all s, which can be exactly approximated if $\mathbf{w} = \mathbf{0}$. In fact, there are many solutions, as there are more components to the weight vector (8) than there are nonterminal states (7). Moreover, the set of feature vectors, $\{\mathbf{x}(s) : s \in \mathcal{S}\}$, is a linearly independent set. In all these ways this task seems a favorable case for linear function approximation.

If we apply semi-gradient TD(0) to this problem (11.2), then the weights diverge to infinity, as shown in Figure 11.2 (left). The instability occurs for any positive step size, no matter how small. In fact, it even occurs if an expected update is done as in dynamic programming (DP), as shown in Figure 11.2 (right). That is, if the weight vector, $\mathbf{w}_k$, is updated for all states at the same time in a semi-gradient way, using the DP (expectation-based) target:

$$\mathbf{w}_{k+1} \doteq \mathbf{w}_k + \frac{\alpha}{|\mathcal{S}|} \sum_s \Big(\mathbb{E}_\pi[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_k) \mid S_t = s] - \hat{v}(s, \mathbf{w}_k) \Big) \nabla \hat{v}(s, \mathbf{w}_k). \quad (11.9)$$

In this case, there is no randomness and no asynchrony, just as in a classical DP update. The method is conventional except in its use of semi-gradient function approximation. Yet still the system is unstable.

If we alter just the distribution of DP updates in Baird's counterexample, from the uniform distribution to the on-policy distribution (which generally requires asynchronous updating), then convergence is guaranteed to a solution with error bounded by (9.14). This example is striking because the TD and DP methods used are arguably the simplest

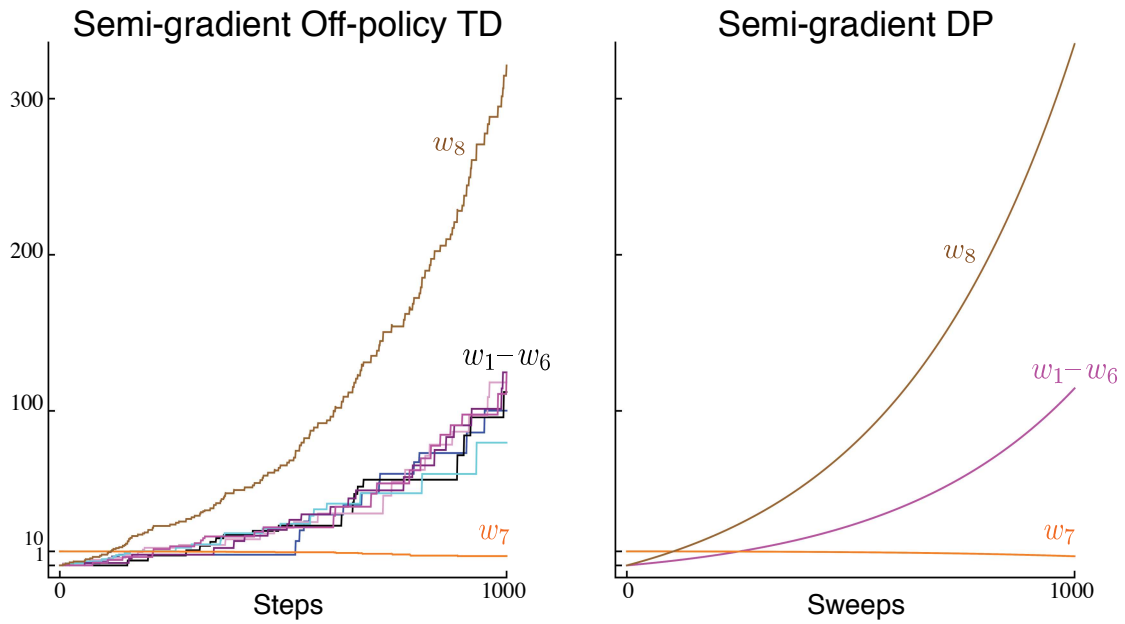


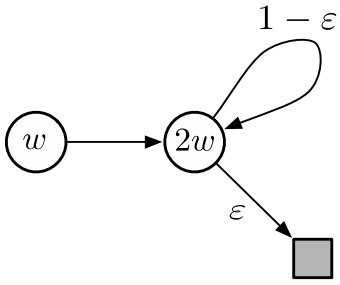
Figure 11.2: Demonstration of instability on Baird's counterexample. Shown are the evolution of the components of the parameter vector $\mathbf{w}$ of the two semi-gradient algorithms. The step size was $\alpha = 0.01$, and the initial weights were $\mathbf{w} = (1,1,1,1,1,1,10,1)^\top$.

and best-understood bootstrapping methods, and the linear, semi-descent method used is arguably the simplest and best-understood kind of function approximation. The example shows that even the simplest combination of bootstrapping and function approximation can be unstable if the updates are not done according to the on-policy distribution.

There are also counterexamples similar to Baird's showing divergence for Q-learning. This is cause for concern because otherwise Q-learning has the best convergence guarantees of all control methods. Considerable effort has gone into trying to find a remedy to this problem or to obtain some weaker, but still workable, guarantee. For example, it may be possible to guarantee convergence of Q-learning as long as the behavior policy is sufficiently close to the target policy, for example, when it is the ε-greedy policy. To the best of our knowledge, Q-learning has never been found to diverge in this case, but there has been no theoretical analysis. In the rest of this section we present several other ideas that have been explored.

Suppose that instead of taking just a step toward the expected one-step return on each iteration, as in Baird's counterexample, we actually change the value function all the way to the best, least-squares approximation. Would this solve the instability problem? Of course it would if the feature vectors, $\{\mathbf{x}(s) : s \in \mathcal{S}\}$, formed a linearly independent set, as they do in Baird's counterexample, because then exact approximation is possible on each iteration and the method reduces to standard tabular DP. But of course the point here is to consider the case when an exact solution is *not* possible. In this case stability is not guaranteed even when forming the best approximation at each iteration, as shown in the example.

Example 11.1: Tsitsiklis and Van Roy's Counterexample This example shows that linear function approximation would not work with DP even if the least-squares solution was found at each step. The counterexample is formed by extending the w-to-$2w$ example (from earlier in this section) with a terminal state, as shown to the right. As before, the estimated value of the first state is w, and the estimated value of the second state is $2w$. The reward is zero on all transitions, so the true values are zero at both states, which is exactly representable with $w = 0$. If we set w_{k+1} at each step so as to minimize the $\overline{\text{VE}}$ between the estimated value and the expected one-step return, then we have



$$\begin{aligned} w_{k+1} &= \operatorname*{arg\,min}_{w\in\mathbb{R}} \sum_{s\in\mathcal{S}} \Big(\hat{v}(s,w) - \mathbb{E}_\pi\big[R_{t+1} + \gamma\hat{v}(S_{t+1},w_k) \;\big|\; S_t = s\big]\Big)^2 \\ &= \operatorname*{arg\,min}_{w\in\mathbb{R}} \big(w - \gamma 2w_k\big)^2 + \big(2w - (1-\varepsilon)\gamma 2w_k\big)^2 \\ &= \frac{6-4\varepsilon}{5}\gamma w_k. \end{aligned} \tag{11.10}$$

The sequence $\{w_k\}$ diverges when $\gamma > \frac{5}{6-4\varepsilon}$ and $w_0 \neq 0$. ■

Another way to try to prevent instability is to use special methods for function approximation. In particular, stability is guaranteed for function approximation methods that do not extrapolate from the observed targets. These methods, called *averagers*, include nearest neighbor methods and locally weighted regression, but not popular methods such as tile coding and artificial neural networks (ANNs).

Exercise 11.3 (programming) Apply one-step semi-gradient Q-learning to Baird's counterexample and show empirically that its weights diverge. □

11.3 The Deadly Triad

Our discussion so far can be summarized by saying that the danger of instability and divergence arises whenever we combine all of the following three elements, making up what we call *the deadly triad*:

Function approximation A powerful, scalable way of generalizing from a state space much larger than the memory and computational resources (e.g., linear function approximation or ANNs).

Bootstrapping Update targets that include existing estimates (as in dynamic programming or TD methods) rather than relying exclusively on actual rewards and complete returns (as in MC methods).

Off-policy training Training on a distribution of transitions other than that produced by the target policy. Sweeping through the state space and updating all states uniformly, as in dynamic programming, does not respect the target policy and is an example of off-policy training.

In particular, note that the danger is *not* due to control or to generalized policy iteration. Those cases are more complex to analyze, but the instability arises in the simpler prediction case whenever it includes all three elements of the deadly triad. The danger is also *not* due to learning or to uncertainties about the environment, because it occurs just as strongly in planning methods, such as dynamic programming, in which the environment is completely known.

If any two elements of the deadly triad are present, but not all three, then instability can be avoided. It is natural, then, to go through the three and see if there is any one that can be given up.

Of the three, *function approximation* most clearly cannot be given up. We need methods that scale to large problems and to great expressive power. We need at least linear function approximation with many features and parameters. State aggregation or nonparametric methods whose complexity grows with data are too weak or too expensive. Least-squares methods such as LSTD are of quadratic complexity and are therefore too expensive for large problems.

Doing without *bootstrapping* is possible, at the cost of computational and data efficiency. Perhaps most important are the losses in computational efficiency. Monte Carlo (non-bootstrapping) methods require memory to save everything that happens between making

each prediction and obtaining the final return, and all their computation is done once the final return is obtained. The cost of these computational issues is not apparent on serial von Neumann computers, but would be on specialized hardware. With bootstrapping and eligibility traces (Chapter 12), data can be dealt with when and where it is generated, then need never be used again. The savings in communication and memory made possible by bootstrapping are great.

The losses in data efficiency by giving up *bootstrapping* are also significant. We have seen this repeatedly, such as in Chapters 7 (Figure 7.2) and 9 (Figure 9.2), where some degree of bootstrapping performed much better than Monte Carlo methods on the random-walk prediction task, and in Chapter 10 where the same was seen on the Mountain-Car control task (Figure 10.4). Many other problems show much faster learning with bootstrapping (e.g., see Figure 12.14). Bootstrapping often results in faster learning because it allows learning to take advantage of the state property, the ability to recognize a state upon returning to it. On the other hand, bootstrapping can impair learning on problems where the state representation is poor and causes poor generalization (e.g., this seems to be the case on Tetris, see Şimşek, Algórta, and Kothiyal, 2016). A poor state representation can also result in bias; this is the reason for the poorer bound on the asymptotic approximation quality of bootstrapping methods (Equation 9.14). On balance, the ability to bootstrap has to be considered extremely valuable. One may sometimes choose not to use it by selecting long n-step updates (or a large bootstrapping parameter, $\lambda \approx 1$; see Chapter 12) but often bootstrapping greatly increases efficiency. It is an ability that we would very much like to keep in our toolkit.

Finally, there is *off-policy learning*; can we give that up? On-policy methods are often adequate. For model-free reinforcement learning, one can simply use Sarsa rather than Q-learning. Off-policy methods free behavior from the target policy. This could be considered an appealing convenience but not a necessity. However, off-policy learning *is essential* to other anticipated use cases, cases that we have not yet mentioned in this book but may be important to the larger goal of creating a powerful intelligent agent.

In these use cases, the agent learns not just a single value function and single policy, but large numbers of them in parallel. There is extensive psychological evidence that people and animals learn to predict many different sensory events, not just rewards. We can be surprised by unusual events, and correct our predictions about them, even if they are of neutral valence (neither good nor bad). This kind of prediction presumably underlies predictive models of the world such as are used in planning. We predict what we will see after eye movements, how long it will take to walk home, the probability of making a jump shot in basketball, and the satisfaction we will get from taking on a new project. In all these cases, the events we would like to predict depend on our acting in a certain way. To learn them all, in parallel, requires learning from the one stream of experience. There are many target policies, and thus the one behavior policy cannot equal all of them. Yet parallel learning is conceptually possible because the behavior policy may overlap in part with many of the target policies. To take full advantage of this requires off-policy learning.

11.4 Linear Value-function Geometry

To better understand the stability challenge of off-policy learning, it is helpful to think about value function approximation more abstractly and independently of how learning is done. We can imagine the space of all possible state-value functions—all functions from states to real numbers $v : \mathcal{S} \to \mathbb{R}$. Most of these value functions do not correspond to any policy. More important for our purposes is that most are not representable by the function approximator, which by design has far fewer parameters than there are states.

Given an enumeration of the state space $\mathcal{S} = \{s_1, s_2, \ldots, s_{|\mathcal{S}|}\}$, any value function v corresponds to a vector listing the value of each state in order $[v(s_1), v(s_2), \ldots, v(s_{|\mathcal{S}|})]^\top$. This vector representation of a value function has as many components as there are states. In most cases where we want to use function approximation, this would be far too many components to represent the vector explicitly. Nevertheless, the idea of this vector is conceptually useful. In the following, we treat a value function and its vector representation interchangeably.

To develop intuitions, consider the case with three states $\mathcal{S} = \{s_1, s_2, s_3\}$ and two parameters $\mathbf{w} = (w_1, w_2)^\top$. We can then view all value functions/vectors as points in a three-dimensional space. The parameters provide an alternative coordinate system over a two-dimensional subspace. Any weight vector $\mathbf{w} = (w_1, w_2)^\top$ is a point in the two-dimensional subspace and thus also a complete value function $v_\mathbf{w}$ that assigns values to all three states. With general function approximation the relationship between the full space and the subspace of representable functions could be complex, but in the case of *linear* value-function approximation the subspace is a simple plane, as suggested by Figure 11.3.

Now consider a single fixed policy π. We assume that its true value function, v_π, is too complex to be represented exactly as an approximation. Thus v_π is not in the subspace; in the figure it is depicted as being above the planar subspace of representable functions.

If v_π cannot be represented exactly, what representable value function is closest to it? This turns out to be a subtle question with multiple answers. To begin, we need a measure of the distance between two value functions. Given two value functions v_1 and v_2, we can talk about the vector difference between them, $v = v_1 - v_2$. If v is small, then the two value functions are close to each other. But how are we to measure the size of this difference vector? The conventional Euclidean norm is not appropriate because, as discussed in Section 9.2, some states are more important than others because they occur more frequently or because we are more interested in them (Section 9.11). As in Section 9.2, let us use the distribution $\mu : \mathcal{S} \to [0, 1]$ to specify the degree to which we care about different states being accurately valued (often taken to be the on-policy distribution). We can then define the distance between value functions using the norm

$$\|v\|^2_\mu \doteq \sum_{s \in \mathcal{S}} \mu(s) v(s)^2. \tag{11.11}$$

Note that the $\overline{\text{VE}}$ from Section 9.2 can be written simply using this norm as $\overline{\text{VE}}(\mathbf{w}) = \|v_\mathbf{w} - v_\pi\|^2_\mu$. For any value function v, the operation of finding its closest value function in the subspace of representable value functions is a projection operation. We define a

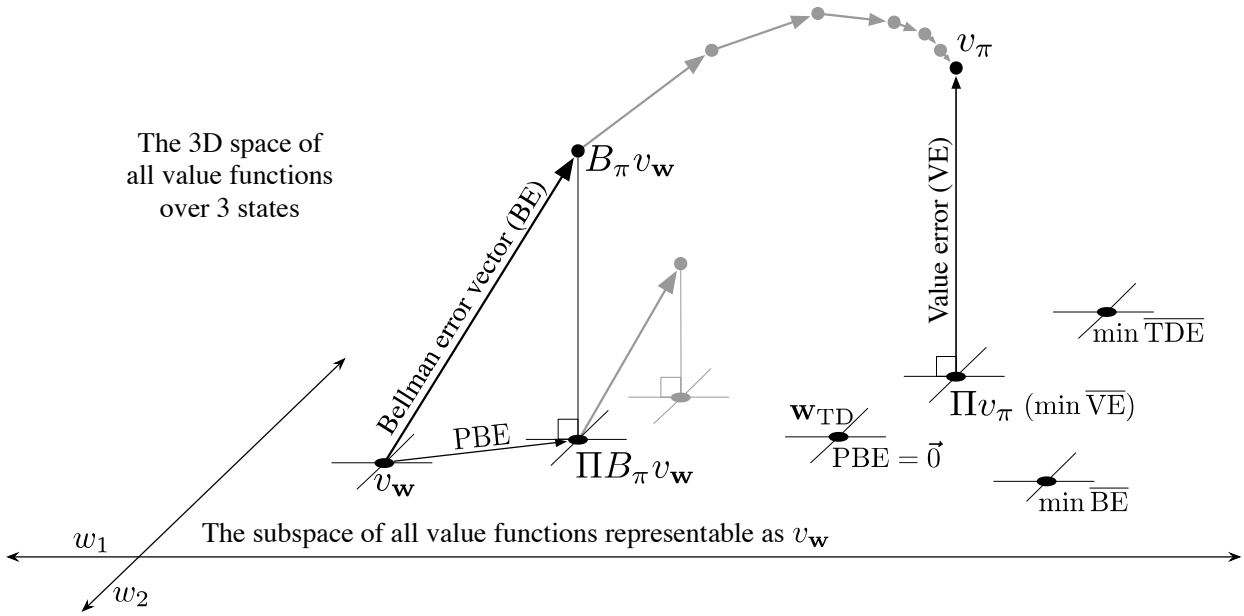


Figure 11.3: The geometry of linear value-function approximation. Shown is the three-dimensional space of all value functions over three states, while shown as a plane is the subspace of all value functions representable by a linear function approximator with parameter $\mathbf{w} = (w_1, w_2)^\top$. The true value function v_π is in the larger space and can be projected down (into the subspace, using a projection operator Π) to its best approximation in the value error (VE) sense. The best approximators in the Bellman error (BE), projected Bellman error (PBE), and temporal difference error (TDE) senses are all potentially different and are shown in the lower right. (VE, BE, and PBE are all treated as the corresponding vectors in this figure.) The Bellman operator takes a value function in the plane to one outside, which can then be projected back. If you iteratively applied the Bellman operator outside the space (shown in gray above) you would reach the true value function, as in conventional dynamic programming. If instead you kept projecting back into the subspace at each step, as in the lower step shown in gray, then the fixed point would be the point of vector-zero PBE.

projection operator Π that takes an arbitrary value function to the representable function that is closest in our norm:

$$\Pi v \doteq v_{\mathbf{w}} \quad \text{where} \quad \mathbf{w} = \operatorname*{arg\,min}_{\mathbf{w} \in \mathbb{R}^d} \|v - v_{\mathbf{w}}\|_\mu^2 . \tag{11.12}$$

The representable value function that is closest to the true value function v_π is thus its projection, Πv_π, as suggested in Figure 11.3. This is the solution asymptotically found by Monte Carlo methods, albeit often very slowly. The projection operation is discussed more fully in the box on the next page.

TD methods find different solutions. To understand their rationale, recall that the Bellman equation for value function v_π is

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a) \left[r + \gamma v_\pi(s') \right], \qquad \text{for all } s \in \mathcal{S}. \tag{11.16}$$

The projection matrix

For a linear function approximator, the projection operation is linear, which implies that it can be represented as an $|\mathcal{S}| \times |\mathcal{S}|$ matrix:

$$\Pi \doteq \mathbf{X}\left(\mathbf{X}^\top \mathbf{D}\mathbf{X}\right)^{-1}\mathbf{X}^\top \mathbf{D}, \tag{11.13}$$

where, as in Section 9.4, $\mathbf{D}$ denotes the $|\mathcal{S}| \times |\mathcal{S}|$ diagonal matrix with the $\mu(s)$ on the diagonal, and $\mathbf{X}$ denotes the $|\mathcal{S}| \times d$ matrix whose rows are the feature vectors $\mathbf{x}(s)^\top$, one for each state s. If the inverse in (11.13) does not exist, then the pseudoinverse is substituted. Using these matrices, the squared norm of a vector can be written

$$\|v\|_\mu^2 = v^\top \mathbf{D} v, \tag{11.14}$$

and the approximate linear value function can be written

$$v_{\mathbf{w}} = \mathbf{X}\mathbf{w}. \tag{11.15}$$

The true value function v_π is the only value function that solves (11.16) exactly. If an approximate value function $v_{\mathbf{w}}$ were substituted for v_π, the difference between the right and left sides of the modified equation could be used as a measure of how far off $v_{\mathbf{w}}$ is from v_π. We call this the *Bellman error* at state s:

$$\bar{\delta}_{\mathbf{w}}(s) \doteq \left(\sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_{\mathbf{w}}(s')\right]\right) - v_{\mathbf{w}}(s) \tag{11.17}$$

$$= \mathbb{E}_\pi\left[R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t) \;\middle|\; S_t = s, A_t \sim \pi\right], \tag{11.18}$$

which shows clearly the relationship of the Bellman error to the TD error (11.3). The Bellman error is the expectation of the TD error.

The vector of all the Bellman errors, at all states, $\bar{\delta}_{\mathbf{w}} \in \mathbb{R}^{|\mathcal{S}|}$, is called the *Bellman error vector* (shown as BE in Figure 11.3). The overall size of this vector, in the norm, is an overall measure of the error in the value function, called the *mean square Bellman error*:

$$\overline{\text{BE}}(\mathbf{w}) = \left\|\bar{\delta}_{\mathbf{w}}\right\|_\mu^2. \tag{11.19}$$

It is not possible in general to reduce the $\overline{\text{BE}}$ to zero (at which point $v_{\mathbf{w}} = v_\pi$), but for linear function approximation there is a unique value of $\mathbf{w}$ for which the $\overline{\text{BE}}$ is minimized. This point in the representable-function subspace (labeled $\min \overline{\text{BE}}$ in Figure 11.3) is different in general from that which minimizes the $\overline{\text{VE}}$ (shown as Πv_π). Methods that seek to minimize the $\overline{\text{BE}}$ are discussed in the next two sections.

The Bellman error vector is shown in Figure 11.3 as the result of applying the *Bellman operator* $B_\pi : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$ to the approximate value function. The Bellman operator is

defined by

$$(B_\pi v)(s) \doteq \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a) \left[r + \gamma v(s') \right], \tag{11.20}$$

for all $s \in \mathcal{S}$, $v : \mathcal{S} \to \mathbb{R}$. The Bellman error vector for $v_{\mathbf{w}}$ can be written $\bar{\delta}_{\mathbf{w}} = B_\pi v_{\mathbf{w}} - v_{\mathbf{w}}$.

If the Bellman operator is applied to a value function in the representable subspace, then, in general, it will produce a new value function that is outside the subspace, as suggested in the figure. In dynamic programming (without function approximation), this operator is applied repeatedly to the points outside the representable space, as suggested by the gray arrows in the top of Figure 11.3. Eventually that process converges to the true value function v_π, the only fixed point for the Bellman operator, the only value function for which

$$v_\pi = B_\pi v_\pi, \tag{11.21}$$

which is just another way of writing the Bellman equation for π (11.16).

With function approximation, however, the intermediate value functions lying outside the subspace cannot be represented. The gray arrows in the upper part of Figure 11.3 cannot be followed because after the first update (dark line) the value function must be projected back into something representable. The next iteration then begins within the subspace; the value function is again taken outside of the subspace by the Bellman operator and then mapped back by the projection operator, as suggested by the lower gray arrow and line. Following these arrows is a DP-like process with approximation.

In this case we are interested in the projection of the Bellman error vector back into the representable space. This is the projected Bellman error vector $\Pi\bar{\delta}_{\mathbf{w}}$, shown in Figure 11.3 as PBE. The size of this vector, in the norm, is another measure of error in the approximate value function. For any approximate value function $v_{\mathbf{w}}$, we define the *mean square Projected Bellman error*, denoted $\overline{\text{PBE}}$, as

$$\overline{\text{PBE}}(\mathbf{w}) = \left\| \Pi\bar{\delta}_{\mathbf{w}} \right\|_\mu^2. \tag{11.22}$$

With linear function approximation there always exists an approximate value function (within the subspace) with zero $\overline{\text{PBE}}$; this is the TD fixed point, $\mathbf{w}_{\text{TD}}$, introduced in Section 9.4. As we have seen, this point is not always stable under semi-gradient TD methods and off-policy training. As shown in the figure, this value function is generally different from those minimizing $\overline{\text{VE}}$ or $\overline{\text{BE}}$. Methods that are guaranteed to converge to it are discussed in Sections 11.7 and 11.8.

11.5 Gradient Descent in the Bellman Error

Armed with a better understanding of value function approximation and its various objectives, we return now to the challenge of stability in off-policy learning. We would like to apply the approach of stochastic gradient descent (SGD, Section 9.3), in which updates are made that in expectation are equal to the negative gradient of an objective

function. These methods always go downhill (in expectation) in the objective and because of this are typically stable with excellent convergence properties. Among the algorithms investigated so far in this book, only the Monte Carlo methods are true SGD methods. These methods converge robustly under both on-policy and off-policy training as well as for general nonlinear (differentiable) function approximators, though they are often slower than semi-gradient methods with bootstrapping, which are not SGD methods. Semi-gradient methods may diverge under off-policy training, as we have seen earlier in this chapter, and under contrived cases of nonlinear function approximation (Tsitsiklis and Van Roy, 1997). With a true SGD method such divergence would not be possible.

The appeal of SGD is so strong that great effort has gone into finding a practical way of harnessing it for reinforcement learning. The starting place of all such efforts is the choice of an error or objective function to optimize. In this and the next section we explore the origins and limits of the most popular proposed objective function, that based on the *Bellman error* introduced in the previous section. Although this has been a popular and influential approach, the conclusion that we reach here is that it is a misstep and yields no good learning algorithms. On the other hand, this approach fails in an interesting way that offers insight into what might constitute a good approach.

To begin, let us consider not the Bellman error, but something more immediate and naive. Temporal difference learning is driven by the TD error. Why not take the minimization of the expected square of the TD error as the objective? In the general function-approximation case, the one-step TD error with discounting is

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t).$$

A possible objective function then is what one might call the *mean square TD error*:

$$\begin{aligned}
\overline{\text{TDE}}(\mathbf{w}) &= \sum_{s\in\mathcal{S}} \mu(s)\mathbb{E}\big[\delta_t^2 \;\big|\; S_t{=}s, A_t{\sim}\pi\big] \\
&= \sum_{s\in\mathcal{S}} \mu(s)\mathbb{E}\big[\rho_t\delta_t^2 \;\big|\; S_t{=}s, A_t{\sim}b\big] \\
&= \mathbb{E}_b\big[\rho_t\delta_t^2\big]\,. \qquad\qquad \text{(if } \mu \text{ is the distribution encountered under } b\text{)}
\end{aligned}$$

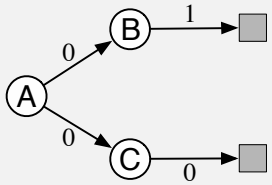
The last equation is of the form needed for SGD; it gives the objective as an expectation that can be sampled from experience (remember the experience is due to the behavior policy b). Thus, following the standard SGD approach, one can derive the per-step update based on a sample of this expected value:

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha\nabla(\rho_t\delta_t^2) \\
&= \mathbf{w}_t - \alpha\rho_t\delta_t\nabla\delta_t \\
&= \mathbf{w}_t + \alpha\rho_t\delta_t\big(\nabla\hat{v}(S_t,\mathbf{w}_t) - \gamma\nabla\hat{v}(S_{t+1},\mathbf{w}_t)\big), \qquad (11.23)
\end{aligned}$$

which you will recognize as the same as the semi-gradient TD algorithm (11.2) except for the additional final term. This term completes the gradient and makes this a true SGD algorithm with excellent convergence guarantees. Let us call this algorithm the *naive*

residual-gradient algorithm (after Baird, 1995). Although the naive residual-gradient algorithm converges robustly, it does not necessarily converge to a desirable place.

Example 11.2: A-split example, showing the naiveté of the naive residual-gradient algorithm



Consider the three-state episodic MRP shown to the right. Episodes begin in state A and then 'split' stochastically, half the time going to B (and then invariably going on to terminate with a reward of 1) and half the time going to state C (and then invariably terminating with a reward of zero). Reward for the first transition, out of A, is always zero whichever way the episode goes. As this is an episodic problem, we can take γ to be 1. We also assume on-policy training, so that ρ_t is always 1, and tabular function approximation, so that the learning algorithms are free to give arbitrary, independent values to all three states. Thus, this should be an easy problem.

What should the values be? From A, half the time the return is 1, and half the time the return is 0; A should have value $\frac{1}{2}$. From B the return is always 1, so its value should be 1, and similarly from C the return is always 0, so its value should be 0. These are the true values and, as this is a tabular problem, all the methods presented previously converge to them exactly.

However, the naive residual-gradient algorithm finds different values for B and C. It converges with B having a value of $\frac{3}{4}$ and C having a value of $\frac{1}{4}$ (A converges correctly to $\frac{1}{2}$). These are in fact the values that minimize the $\overline{\text{TDE}}$.

Let us compute the $\overline{\text{TDE}}$ for these values. The first transition of each episode is either up from A's $\frac{1}{2}$ to B's $\frac{3}{4}$, a change of $\frac{1}{4}$, or down from A's $\frac{1}{2}$ to C's $\frac{1}{4}$, a change of $-\frac{1}{4}$. Because the reward is zero on these transitions, and $\gamma = 1$, these changes *are* the TD errors, and thus the squared TD error is always $\frac{1}{16}$ on the first transition. The second transition is similar; it is either up from B's $\frac{3}{4}$ to a reward of 1 (and a terminal state of value 0), or down from C's $\frac{1}{4}$ to a reward of 0 (again with a terminal state of value 0). Thus, the TD error is always $\pm\frac{1}{4}$, for a squared error of $\frac{1}{16}$ on the second step. Thus, for this set of values, the $\overline{\text{TDE}}$ on both steps is $\frac{1}{16}$.

Now let's compute the $\overline{\text{TDE}}$ for the true values (B at 1, C at 0, and A at $\frac{1}{2}$). In this case the first transition is either from $\frac{1}{2}$ up to 1, at B, or from $\frac{1}{2}$ down to 0, at C; in either case the absolute error is $\frac{1}{2}$ and the squared error is $\frac{1}{4}$. The second transition has zero error because the starting value, either 1 or 0 depending on whether the transition is from B or C, always exactly matches the immediate reward and return. Thus the squared TD error is $\frac{1}{4}$ on the first transition and 0 on the second, for a mean reward over the two transitions of $\frac{1}{8}$. As $\frac{1}{8}$ is bigger that $\frac{1}{16}$, this solution is worse according to the $\overline{\text{TDE}}$. On this simple problem, the true values do not have the smallest $\overline{\text{TDE}}$.

A tabular representation is used in the A-split example, so the true state values can be exactly represented, yet the naive residual-gradient algorithm finds different values, and these values have lower $\overline{\text{TDE}}$ than do the true values. Minimizing the $\overline{\text{TDE}}$ is naive; by penalizing all TD errors it achieves something more like temporal smoothing than accurate prediction.

A better idea would seem to be minimizing the mean square Bellman error ($\overline{\text{BE}}$). If the exact values are learned, the Bellman error is zero everywhere. Thus, a Bellman-error-minimizing algorithm should have no trouble with the A-split example. We cannot expect to achieve zero Bellman error in general, as it would involve finding the true value function, which we presume is outside the space of representable value functions. But getting close to this ideal is a natural-seeming goal. As we have seen, the Bellman error is also closely related to the TD error. The Bellman error for a state is the expected TD error in that state. So let's repeat the derivation above with the expected TD error (all expectations here are implicitly conditional on S_t):

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha\nabla(\mathbb{E}_\pi[\delta_t]^2) \\
&= \mathbf{w}_t - \frac{1}{2}\alpha\nabla(\mathbb{E}_b[\rho_t\delta_t]^2) \\
&= \mathbf{w}_t - \alpha\mathbb{E}_b[\rho_t\delta_t]\,\nabla\mathbb{E}_b[\rho_t\delta_t] \\
&= \mathbf{w}_t - \alpha\mathbb{E}_b\big[\rho_t(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w}))\big]\,\mathbb{E}_b[\rho_t\nabla\delta_t] \\
&= \mathbf{w}_t + \alpha\Big[\mathbb{E}_b\big[\rho_t(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}))\big] - \hat{v}(S_t,\mathbf{w})\Big]\Big[\nabla\hat{v}(S_t,\mathbf{w}) - \gamma\mathbb{E}_b\big[\rho_t\nabla\hat{v}(S_{t+1},\mathbf{w})\big]\Big].
\end{aligned}$$

This update and various ways of sampling it are referred to as the *residual-gradient algorithm*. If you simply used the sample values in all the expectations, then the equation above reduces almost exactly to (11.23), the naive residual-gradient algorithm.[1] But this is naive, because the equation above involves the next state, S_{t+1}, appearing in two expectations that are multiplied together. To get an unbiased sample of the product, two independent samples of the next state are required, but during normal interaction with an external environment only one is obtained. One expectation or the other can be sampled, but not both.

There are two ways to make the residual-gradient algorithm work. One is in the case of deterministic environments. If the transition to the next state is deterministic, then the two samples will necessarily be the same, and the naive algorithm is valid. The other way is to obtain *two* independent samples of the next state, S_{t+1}, from S_t, one for the first expectation and another for the second expectation. In real interaction with an environment, this would not seem possible, but when interacting with a simulated environment, it is. One simply rolls back to the previous state and obtains an alternate next state before proceeding forward from the first next state. In either of these cases the residual-gradient algorithm is guaranteed to converge to a minimum of the $\overline{\text{BE}}$ under the usual conditions on the step-size parameter. As a true SGD method, this convergence is

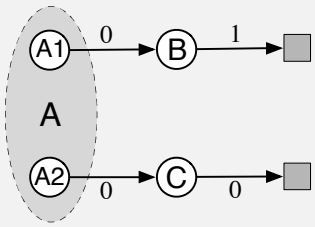
[1] For state values there remains a small difference in the treatment of the importance sampling ratio ρ_t. In the analagous action-value case (which is the most important case for control algorithms), the residual-gradient algorithm would reduce exactly to the naive version.

robust, applying to both linear and nonlinear function approximators. In the linear case, convergence is always to the *unique* $\mathbf{w}$ that minimizes the $\overline{\text{BE}}$.

However, there remain at least three ways in which the convergence of the residual-gradient method is unsatisfactory. The first of these is that empirically it is slow, much slower that semi-gradient methods. Indeed, proponents of this method have proposed increasing its speed by combining it with faster semi-gradient methods initially, then gradually switching over to residual gradient for the convergence guarantee (Baird and Moore, 1999). The second way in which the residual-gradient algorithm is unsatisfactory is that it still seems to converge to the wrong values. It does get the right values in all tabular cases, such as the A-split example, as for those an exact solution to the Bellman

Example 11.3: A-presplit example, a counterexample for the $\overline{\text{BE}}$

Consider the three-state episodic MRP shown to the right: Episodes start in either A1 or A2, with equal probability. These two states look exactly the same to the function approximator, like a single state A whose feature representation is distinct from and unrelated to the feature representation of the other two states, B and C, which are also distinct from each other. Specifically, the parameter of the function approximator has three components, one giving the value of state B, one giving the value of state C, and one giving the value of both states A1 and A2. Other than the selection of the initial state, the system is deterministic. If it starts in A1, then it transitions to B with a reward of 0 and then on to termination with a reward of 1. If it starts in A2, then it transitions to C, and then to termination, with both rewards zero.



To a learning algorithm, seeing only the features, the system looks identical to the A-split example. The system seems to always start in A, followed by either B or C with equal probability, and then terminating with a 1 or a 0 depending deterministically on the previous state. As in the A-split example, the true values of B and C are 1 and 0, and the best shared value of A1 and A2 is $\frac{1}{2}$, by symmetry.

Because this problem appears externally identical to the A-split example, we already know what values will be found by the algorithms. Semi-gradient TD converges to the ideal values just mentioned, while the naive residual-gradient algorithm converges to values of $\frac{3}{4}$ and $\frac{1}{4}$ for B and C respectively. All state transitions are deterministic, so the non-naive residual-gradient algorithm will also converge to these values (it is the same algorithm in this case). It follows then that this 'naive' solution must also be the one that minimizes the $\overline{\text{BE}}$, and so it is. On a deterministic problem, the Bellman errors and TD errors are all the same, so the $\overline{\text{BE}}$ is always the same as the $\overline{\text{TDE}}$. Optimizing the $\overline{\text{BE}}$ on this example gives rise to the same failure mode as with the naive residual-gradient algorithm on the A-split example.

equation is possible. But if we examine examples with genuine function approximation, then the residual-gradient algorithm, and indeed the $\overline{\text{BE}}$ objective, seem to find the wrong value functions. One of the most telling such examples is the variation on the A-split example known as the A-*pre*split example, shown on the preceding page, in which the residual-gradient algorithm finds the same poor solution as its naive version. This example shows intuitively that minimizing the $\overline{\text{BE}}$ (which the residual-gradient algorithm surely does) may not be a desirable goal.

The third way in which the convergence of the residual-gradient algorithm is not satisfactory is explained in the next section. Like the second way, the third way is also a problem with the $\overline{\text{BE}}$ objective itself rather than with any particular algorithm for achieving it.

11.6 The Bellman Error is Not Learnable

The concept of learnability that we introduce in this section is different from that commonly used in machine learning. There, a hypothesis is said to be "learnable" if it is *efficiently* learnable, meaning that it can be learned within a polynomial rather than an exponential number of examples. Here we use the term in a more basic way, to mean learnable at all, with any amount of experience. It turns out many quantities of apparent interest in reinforcement learning cannot be learned even from an infinite amount of experiential data. These quantities are well defined and can be computed given knowledge of the internal structure of the environment, but cannot be computed or estimated from the observed sequence of feature vectors, actions, and rewards.[2] We say that they are not *learnable*. It will turn out that the Bellman error objective ($\overline{\text{BE}}$) introduced in the last two sections is not learnable in this sense. That the Bellman error objective cannot be learned from the observable data is probably the strongest reason not to seek it.

To make the concept of learnability clear, let's start with some simple examples. Consider the two Markov reward processes[3] (MRPs) diagrammed below:

Where two edges leave a state, both transitions are assumed to occur with equal probability, and the numbers indicate the reward received. All the states appear the same; they all produce the same single-component feature vector $x = 1$ and have approximated value w. Thus, the only varying part of the data trajectory is the reward sequence. The left MRP stays in the same state and emits an endless stream of 0s and 2s at random, each with 0.5 probability. The right MRP, on every step, either stays in its current state or

[2] They would of course be estimated if the *state* sequence were observed rather than only the corresponding feature vectors.

[3] All MRPs can be considered MDPs with a single action in all states; what we conclude about MRPs here applies as well to MDPs.

switches to the other, with equal probability. The reward is deterministic in this MRP, always a 0 from one state and always a 2 from the other, but because the each state is equally likely on each step, the observable data is again an endless stream of 0s and 2s at random, identical to that produced by the left MRP. (We can assume the right MRP starts in one of two states at random with equal probability.) Thus, even given an infinite amount of data, it would not be possible to tell which of these two MRPs was generating it. In particular, we could not tell if the MRP has one state or two, is stochastic or deterministic. These things are not learnable.

This pair of MRPs also illustrates that the $\overline{\text{VE}}$ objective (9.1) is not learnable. If $\gamma = 0$, then the true values of the three states (in both MRPs), left to right, are 1, 0, and 2. Suppose $w = 1$. Then the $\overline{\text{VE}}$ is 0 for the left MRP and 1 for the right MRP. Because the $\overline{\text{VE}}$ is different in the two problems, yet the data generated has the same distribution, the $\overline{\text{VE}}$ cannot be learned. The $\overline{\text{VE}}$ is not a unique function of the data distribution. And if it cannot be learned, then how could the $\overline{\text{VE}}$ possibly be useful as an objective for learning?

If an objective cannot be learned, it does indeed draw its utility into question. In the case of the $\overline{\text{VE}}$, however, there is a way out. Note that the same solution, $w = 1$, is optimal for both MRPs above (assuming μ is the same for the two indistinguishable states in the right MRP). Is this a coincidence, or could it be generally true that all MDPs with the same data distribution also have the same optimal parameter vector? If this is true—and we will show next that it is—then the $\overline{\text{VE}}$ remains a usable objective. The $\overline{\text{VE}}$ is not learnable, but the parameter that optimizes it is!

To understand this, it is useful to bring in another natural objective function, this time one that is clearly learnable. One error that is always observable is that between the value estimate at each time and the return from that time. The *mean square return error*, denoted $\overline{\text{RE}}$, is the expectation, under μ, of the square of this error. In the on-policy case the $\overline{\text{RE}}$ can be written

$$\begin{aligned}\overline{\text{RE}}(\mathbf{w}) &= \mathbb{E}\Big[\big(G_t - \hat{v}(S_t,\mathbf{w})\big)^2\Big] \\ &= \overline{\text{VE}}(\mathbf{w}) + \mathbb{E}\Big[\big(G_t - v_\pi(S_t)\big)^2\Big]. \end{aligned} \tag{11.24}$$

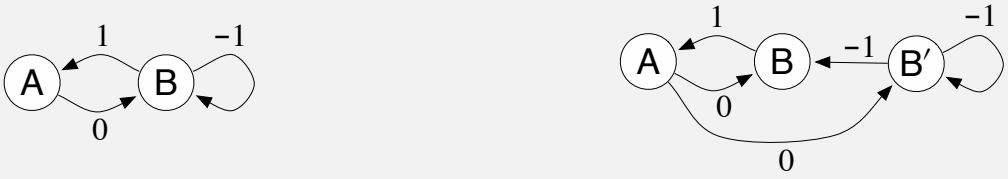
Thus, the two objectives are the same except for a variance term that does not depend on the parameter vector. The two objectives must therefore have the same optimal parameter value $\mathbf{w}^*$. The overall relationships are summarized in the left side of Figure 11.4.

**Exercise 11.4* Prove (11.24). Hint: Write the $\overline{\text{RE}}$ as an expectation over possible states s of the expectation of the squared error given that $S_t = s$. Then add and subtract the true value of state s from the error (before squaring), grouping the subtracted true value with the return and the added true value with the estimated value. Then, if you expand the square, the most complex term will end up being zero, leaving you with (11.24). □

Now let us return to the $\overline{\text{BE}}$. The $\overline{\text{BE}}$ is like the $\overline{\text{VE}}$ in that it can be computed from knowledge of the MDP but is not learnable from data. But it is not like the $\overline{\text{VE}}$ in that its minimum solution is not learnable. The box on the next page presents a counterexample—two MRPs that generate the same data distribution but whose minimizing parameter vector is different, proving that the optimal parameter vector is not a function of the

Example 11.4: Counterexample to the learnability of the Bellman error

To show the full range of possibilities we need a slightly more complex pair of Markov reward processes (MRPs) than those considered earlier. Consider the following two MRPs:



Where two edges leave a state, both transitions are assumed to occur with equal probability, and the numbers indicate the reward received. The MRP on the left has two states that are represented distinctly. The MRP on the right has three states, two of which, B and B′, appear the same and must be given the same approximate value. Specifically, $\mathbf{w}$ has two components and the value of state A is given by the first component and the value of B and B′ is given by the second. The second MRP has been designed so that equal time is spent in all three states, so we can take $\mu(s) = \frac{1}{3}$, for all s.

Note that the observable data distribution is identical for the two MRPs. In both cases the agent will see single occurrences of A followed by a 0, then some number of apparent Bs, each followed by a -1 except the last, which is followed by a 1, then we start all over again with a single A and a 0, etc. All the statistical details are the same as well; in both MRPs, the probability of a string of k Bs is 2^{-k}.

Now suppose $\mathbf{w} = \mathbf{0}$. In the first MRP, this is an exact solution, and the $\overline{\text{BE}}$ is zero. In the second MRP, this solution produces a squared error in both B and B′ of 1, such that $\overline{\text{BE}} = \mu(\text{B})1 + \mu(\text{B}')1 = \frac{2}{3}$. These two MRPs, which generate the same data distribution, have different $\overline{\text{BE}}$s; the $\overline{\text{BE}}$ is not learnable.

Moreover (and unlike the earlier example for the $\overline{\text{VE}}$) the minimizing value of $\mathbf{w}$ is different for the two MRPs. For the first MRP, $\mathbf{w} = \mathbf{0}$ minimizes the $\overline{\text{BE}}$ for any γ. For the second MRP, the minimizing $\mathbf{w}$ is a complicated function of γ, but in the limit, as $\gamma \to 1$, it is $(-\frac{1}{2}, 0)^\top$. Thus the solution that minimizes $\overline{\text{BE}}$ cannot be estimated from data alone; knowledge of the MRP beyond what is revealed in the data is required. In this sense, it is impossible in principle to pursue the $\overline{\text{BE}}$ as an objective for learning.

It may be surprising that in the second MRP the $\overline{\text{BE}}$-minimizing value of A is so far from zero. Recall that A has a dedicated weight and thus its value is unconstrained by function approximation. A is followed by a reward of 0 and transition to a state with a value of nearly 0, which suggests $v_{\mathbf{w}}(\text{A})$ should be 0; why is its optimal value substantially negative rather than 0? The answer is that making $v_{\mathbf{w}}(\text{A})$ negative reduces the error upon arriving in A from B. The reward on this deterministic transition is 1, which implies that B should have a value 1 more than A. Because B's value is approximately zero, A's value is driven toward -1. The $\overline{\text{BE}}$-minimizing value of $\approx -\frac{1}{2}$ for A is a compromise between reducing the errors on leaving and on entering A.

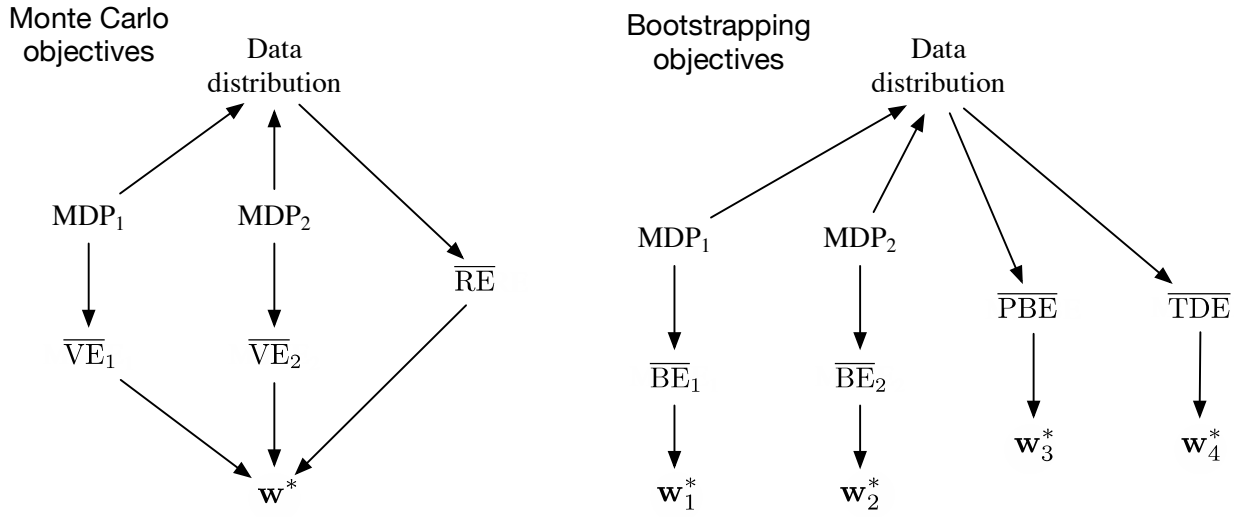


Figure 11.4: Causal relationships among the data distribution, MDPs, and various objectives. **Left, Monte Carlo objectives:** Two different MDPs can produce the same data distribution yet also produce different $\overline{\text{VE}}$s, proving that the $\overline{\text{VE}}$ objective cannot be determined from data and is not learnable. However, all such $\overline{\text{VE}}$s must have the same optimal parameter vector, $\mathbf{w}^*$! Moreover, this same $\mathbf{w}^*$ can be determined from another objective, the $\overline{\text{RE}}$, which *is* uniquely determined from the data distribution. Thus $\mathbf{w}^*$ and the $\overline{\text{RE}}$ are learnable even though the $\overline{\text{VE}}$s are not. **Right, Bootstrapping objectives:** Two different MDPs can produce the same data distribution yet also produce different $\overline{\text{BE}}$s *and* have different minimizing parameter vectors; these are not learnable from the data distribution. The $\overline{\text{PBE}}$ and $\overline{\text{TDE}}$ objectives and their (different) minima can be directly determined from data and thus are learnable.

data and thus cannot be learned from it. The other bootstrapping objectives that we have considered, the $\overline{\text{PBE}}$ and $\overline{\text{TDE}}$, can be determined from data (are learnable) and determine optimal solutions that are in general different from each other and the $\overline{\text{BE}}$ minimums. The general case is summarized in the right side of Figure 11.4.

Thus, the $\overline{\text{BE}}$ is not learnable; it cannot be estimated from feature vectors and other observable data. This limits the $\overline{\text{BE}}$ to model-based settings. There can be no algorithm that minimizes the $\overline{\text{BE}}$ without access to the underlying MDP states beyond the feature vectors. The residual-gradient algorithm is only able to minimize $\overline{\text{BE}}$ because it is allowed to double sample from the same state—not a state that has the same feature vector, but one that is guaranteed to be the same underlying state. We can see now that there is no way around this. Minimizing the $\overline{\text{BE}}$ requires some such access to the nominal, underlying MDP. This is an important limitation of the $\overline{\text{BE}}$ beyond that identified in the A-presplit example on page 273. All this directs more attention toward the $\overline{\text{PBE}}$.

11.7 Gradient-TD Methods

We now consider SGD methods for minimizing the $\overline{\text{PBE}}$. As true SGD methods, these *Gradient-TD methods* have robust convergence properties even under off-policy training and nonlinear function approximation. Remember that in the linear case there is always an exact solution, the TD fixed point $\mathbf{w}_{\text{TD}}$, at which the $\overline{\text{PBE}}$ is zero. This solution could be found by least-squares methods (Section 9.8), but only by methods of quadratic $O(d^2)$ complexity in the number of parameters. We seek instead an SGD method, which should be $O(d)$ and have robust convergence properties. Gradient-TD methods come close to achieving these goals, at the cost of a rough doubling of computational complexity.

To derive an SGD method for the $\overline{\text{PBE}}$ (assuming linear function approximation) we begin by expanding and rewriting the objective (11.22) in matrix terms:

$$\begin{aligned}
\overline{\text{PBE}}(\mathbf{w}) &= \left\|\Pi\bar{\delta}_{\mathbf{w}}\right\|_{\mu}^2 \\
&= (\Pi\bar{\delta}_{\mathbf{w}})^\top \mathbf{D}\Pi\bar{\delta}_{\mathbf{w}} && \text{(from (11.14))} \\
&= \bar{\delta}_{\mathbf{w}}^\top \Pi^\top \mathbf{D}\Pi\bar{\delta}_{\mathbf{w}} \\
&= \bar{\delta}_{\mathbf{w}}^\top \mathbf{D}\mathbf{X}\big(\mathbf{X}^\top\mathbf{D}\mathbf{X}\big)^{-1}\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}} && (11.25) \\
&\quad \text{(using (11.13) and the identity } \Pi^\top\mathbf{D}\Pi = \mathbf{D}\mathbf{X}\left(\mathbf{X}^\top\mathbf{D}\mathbf{X}\right)^{-1}\mathbf{X}^\top\mathbf{D}\text{)} \\
&= \big(\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}}\big)^\top\big(\mathbf{X}^\top\mathbf{D}\mathbf{X}\big)^{-1}\big(\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}}\big). && (11.26)
\end{aligned}$$

The gradient with respect to $\mathbf{w}$ is

$$\nabla\overline{\text{PBE}}(\mathbf{w}) = 2\nabla\big[\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}}\big]^\top\big(\mathbf{X}^\top\mathbf{D}\mathbf{X}\big)^{-1}\big(\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}}\big).$$

To turn this into an SGD method, we have to sample something on every time step that has this quantity as its expected value. Let us take μ to be the distribution of states visited under the behavior policy. All three of the factors above can then be written in terms of expectations under this distribution. For example, the last factor can be written

$$\mathbf{X}^\top\mathbf{D}\bar{\delta}_{\mathbf{w}} = \sum_s \mu(s)\mathbf{x}(s)\bar{\delta}_{\mathbf{w}}(s) = \mathbb{E}[\rho_t\delta_t\mathbf{x}_t]\,,$$

which is just the expectation of the semi-gradient TD(0) update (11.2). The first factor is the transpose of the gradient of this update:

$$\begin{aligned}
\nabla\mathbb{E}[\rho_t\delta_t\mathbf{x}_t]^\top &= \mathbb{E}\big[\rho_t\nabla\delta_t^\top\mathbf{x}_t^\top\big] \\
&= \mathbb{E}\big[\rho_t\nabla(R_{t+1} + \gamma\mathbf{w}^\top\mathbf{x}_{t+1} - \mathbf{w}^\top\mathbf{x}_t)^\top\mathbf{x}_t^\top\big] && \text{(using episodic } \delta_t\text{)} \\
&= \mathbb{E}\big[\rho_t(\gamma\mathbf{x}_{t+1} - \mathbf{x}_t)\mathbf{x}_t^\top\big]\,.
\end{aligned}$$

Finally, the middle factor is the inverse of the expected outer-product matrix of the feature vectors:

$$\mathbf{X}^\top\mathbf{D}\mathbf{X} = \sum_s \mu(s)\mathbf{x}(s)\mathbf{x}(s)^\top = \mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]\,.$$

Substituting these expectations for the three factors in our expression for the gradient of the $\overline{\text{PBE}}$, we get

$$\nabla\overline{\text{PBE}}(\mathbf{w}) = 2\mathbb{E}\big[\rho_t(\gamma\mathbf{x}_{t+1} - \mathbf{x}_t)\mathbf{x}_t^\top\big]\,\mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]^{-1}\,\mathbb{E}[\rho_t\delta_t\mathbf{x}_t]\,. \tag{11.27}$$

It might not be obvious that we have made any progress by writing the gradient in this form. It is a product of three expressions and the first and last are not independent. They both depend on the next feature vector $\mathbf{x}_{t+1}$; we cannot simply sample both of these expectations and then multiply the samples. This would give us a biased estimate of the gradient just as in the residual-gradient algorithm.

Another idea would be to estimate the three expectations separately and then combine them to produce an unbiased estimate of the gradient. This would work, but would require a lot of computational resources, particularly to store the first two expectations, which are $d \times d$ matrices, and to compute the inverse of the second. This idea can be improved. If two of the three expectations are estimated and stored, then the third could be sampled and used in conjunction with the two stored quantities. For example, you could store estimates of the second two quantities (using the increment inverse-updating techniques in Section 9.8) and then sample the first expression. Unfortunately, the overall algorithm would still be of quadratic complexity (of order $O(d^2)$).

The idea of storing some estimates separately and then combining them with samples is a good one and is also used in Gradient-TD methods. Gradient-TD methods estimate and store *the product* of the second two factors in (11.27). These factors are a $d \times d$ matrix and a d-vector, so their product is just a d-vector, like $\mathbf{w}$ itself. We denote this second learned vector as $\mathbf{v}$:

$$\mathbf{v} \approx \mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]^{-1}\,\mathbb{E}[\rho_t\delta_t\mathbf{x}_t]\,. \tag{11.28}$$

This form is familiar to students of linear supervised learning. It is the solution to a linear least-squares problem that tries to approximate $\rho_t\delta_t$ from the features. The standard SGD method for incrementally finding the vector $\mathbf{v}$ that minimizes the expected squared error $\big(\mathbf{v}^\top\mathbf{x}_t - \rho_t\delta_t\big)^2$ is known as the Least Mean Square (LMS) rule (here augmented with an importance sampling ratio):

$$\mathbf{v}_{t+1} \doteq \mathbf{v}_t + \beta\rho_t\left(\delta_t - \mathbf{v}_t^\top\mathbf{x}_t\right)\mathbf{x}_t,$$

where $\beta > 0$ is another step-size parameter. We can use this method to effectively achieve (11.28) with $O(d)$ storage and per-step computation.

Given a stored estimate $\mathbf{v}_t$ approximating (11.28), we can update our main parameter vector $\mathbf{w}_t$ using SGD methods based on (11.27). The simplest such rule is

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha\nabla\overline{\text{PBE}}(\mathbf{w}_t) && \text{(the general SGD rule)}\\
&= \mathbf{w}_t - \frac{1}{2}\alpha 2\mathbb{E}\big[\rho_t(\gamma\mathbf{x}_{t+1} - \mathbf{x}_t)\mathbf{x}_t^\top\big]\,\mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]^{-1}\,\mathbb{E}[\rho_t\delta_t\mathbf{x}_t] && \text{(from (11.27))}\\
&= \mathbf{w}_t + \alpha\mathbb{E}\big[\rho_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})\mathbf{x}_t^\top\big]\,\mathbb{E}\big[\mathbf{x}_t\mathbf{x}_t^\top\big]^{-1}\,\mathbb{E}[\rho_t\delta_t\mathbf{x}_t] && (11.29)\\
&\approx \mathbf{w}_t + \alpha\mathbb{E}\big[\rho_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})\mathbf{x}_t^\top\big]\,\mathbf{v}_t && \text{(based on (11.28))}\\
&\approx \mathbf{w}_t + \alpha\rho_t\left(\mathbf{x}_t - \gamma\mathbf{x}_{t+1}\right)\mathbf{x}_t^\top\mathbf{v}_t. && \text{(sampling)}
\end{aligned}$$

This algorithm is called *GTD2*. Note that if the final inner product $(\mathbf{x}_t^\top \mathbf{v}_t)$ is done first, then the entire algorithm is of $O(d)$ complexity.

A slightly better algorithm can be derived by doing a few more analytic steps before substituting in $\mathbf{v}_t$. Continuing from (11.29):

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha \mathbb{E}\big[\rho_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})\mathbf{x}_t^\top\big]\, \mathbb{E}\big[\mathbf{x}_t \mathbf{x}_t^\top\big]^{-1}\, \mathbb{E}[\rho_t \delta_t \mathbf{x}_t] \\
&= \mathbf{w}_t + \alpha \left(\mathbb{E}\big[\rho_t \mathbf{x}_t \mathbf{x}_t^\top\big] - \gamma \mathbb{E}\big[\rho_t \mathbf{x}_{t+1} \mathbf{x}_t^\top\big]\right) \mathbb{E}\big[\mathbf{x}_t \mathbf{x}_t^\top\big]^{-1}\, \mathbb{E}[\rho_t \delta_t \mathbf{x}_t] \\
&= \mathbf{w}_t + \alpha \left(\mathbb{E}\big[\mathbf{x}_t \mathbf{x}_t^\top\big] - \gamma \mathbb{E}\big[\rho_t \mathbf{x}_{t+1} \mathbf{x}_t^\top\big]\right) \mathbb{E}\big[\mathbf{x}_t \mathbf{x}_t^\top\big]^{-1}\, \mathbb{E}[\rho_t \delta_t \mathbf{x}_t] \\
&= \mathbf{w}_t + \alpha \Big(\mathbb{E}[\rho_t \delta_t \mathbf{x}_t] - \gamma \mathbb{E}\big[\rho_t \mathbf{x}_{t+1} \mathbf{x}_t^\top\big]\, \mathbb{E}\big[\mathbf{x}_t \mathbf{x}_t^\top\big]^{-1}\, \mathbb{E}[\rho_t \delta_t \mathbf{x}_t]\Big) \\
&\approx \mathbf{w}_t + \alpha \left(\mathbb{E}[\rho_t \delta_t \mathbf{x}_t] - \gamma \mathbb{E}\big[\rho_t \mathbf{x}_{t+1} \mathbf{x}_t^\top\big]\, \mathbf{v}_t\right) && \text{(based on (11.28))} \\
&\approx \mathbf{w}_t + \alpha \rho_t \left(\delta_t \mathbf{x}_t - \gamma \mathbf{x}_{t+1} \mathbf{x}_t^\top \mathbf{v}_t\right), && \text{(sampling)}
\end{aligned}$$

which again is $O(d)$ if the final product $(\mathbf{x}_t^\top \mathbf{v}_t)$ is done first. This algorithm is known as either *TD(0) with gradient correction (TDC)* or, alternatively, as *GTD(0)*.

Figure 11.5 shows a sample and the expected behavior of TDC on Baird's counterexample. As intended, the $\overline{\text{PBE}}$ falls to zero, but note that the individual components of the parameter vector do not approach zero. In fact, these values are still far from

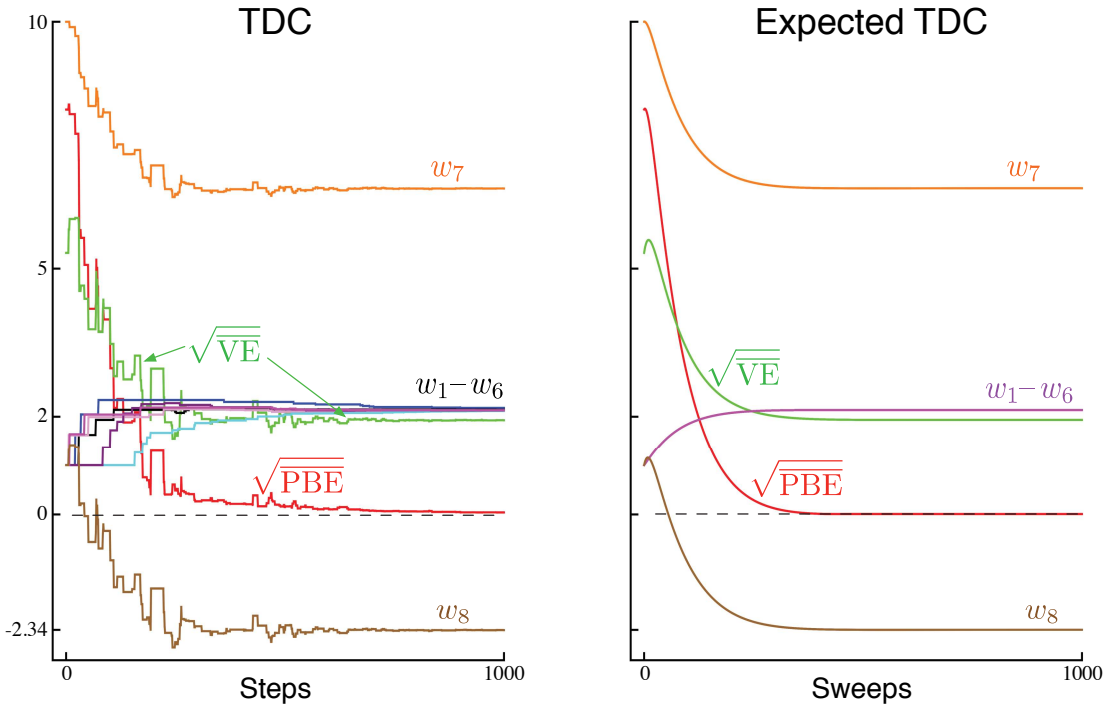


Figure 11.5: The behavior of the TDC algorithm on Baird's counterexample. On the left is shown a typical single run, and on the right is shown the expected behavior of this algorithm if the updates are done synchronously (analogous to (11.9), except for the two TDC parameter vectors). The step sizes were $\alpha = 0.005$ and $\beta = 0.05$.

an optimal solution, $\hat{v}(s) = 0$, for all s, for which $\mathbf{w}$ would have to be proportional to $(1, 1, 1, 1, 1, 1, 4, -2)^\top$. After 1000 iterations we are still far from an optimal solution, as we can see from the $\overline{\text{VE}}$, which remains almost 2. The system is actually converging to an optimal solution, but progress is extremely slow because the $\overline{\text{PBE}}$ is already so close to zero.

GTD2 and TDC both involve two learning processes, a primary one for $\mathbf{w}$ and a secondary one for $\mathbf{v}$. The logic of the primary learning process relies on the secondary learning process having finished, at least approximately, whereas the secondary learning process proceeds without being influenced by the first. We call this sort of asymmetrical dependence a *cascade*. In cascades we often assume that the secondary learning process is proceeding faster and thus is always at its asymptotic value, ready and accurate to assist the primary learning process. The convergence proofs for these methods often make this assumption explicitly. These are called *two-time-scale* proofs. The fast time scale is that of the secondary learning process, and the slower time scale is that of the primary learning process. If α is the step size of the primary learning process, and β is the step size of the secondary learning process, then these convergence proofs will typically require that in the limit $\beta \to 0$ and $\frac{\alpha}{\beta} \to 0$.

Gradient-TD methods are currently the most well understood and widely used stable off-policy methods. There are extensions to action values and control (GQ, Maei et al., 2010), to eligibility traces (GTD(λ) and GQ(λ), Maei, 2011; Maei and Sutton, 2010), and to nonlinear function approximation (Maei et al., 2009). There have also been proposed hybrid algorithms midway between semi-gradient TD and gradient TD (Hackman, 2012; White and White, 2016). Hybrid-TD algorithms behave like Gradient-TD algorithms in states where the target and behavior policies are very different, and behave like semi-gradient algorithms in states where the target and behavior policies are the same. Finally, the Gradient-TD idea has been combined with the ideas of proximal methods and control variates to produce more efficient methods (Mahadevan et al., 2014; Du et al., 2017).

11.8 Emphatic-TD Methods

We turn now to the second major strategy that has been extensively explored for obtaining a cheap and efficient off-policy learning method with function approximation. Recall that linear semi-gradient TD methods are efficient and stable when trained under the on-policy distribution, and that we showed in Section 9.4 that this has to do with the positive definiteness of the matrix $\mathbf{A}$ (9.11)[4] and the match between the on-policy state distribution μ_π and the state-transition probabilities $p(s \,|\, s, a)$ under the target policy. In off-policy learning, we reweight the state transitions using importance sampling so that they become appropriate for learning about the target policy, but the state distribution is still that of the behavior policy. There is a mismatch. A natural idea is to somehow reweight the states, emphasizing some and de-emphasizing others, so as to return the distribution of updates to the on-policy distribution. There would then be a match, and stability and convergence would follow from existing results. This is the idea of

[4] In the off-policy case, the matrix $\mathbf{A}$ is generally defined as $\mathbb{E}_{s\sim b}\big[\mathbf{x}(s)\mathbb{E}\big[\mathbf{x}(S_{t+1})^\top \,\big|\, S_t = s, A_t \sim \pi\big]\big]$.

Emphatic-TD methods, first introduced for on-policy training in Section 9.11.

Actually, the notion of "the on-policy distribution" is not quite right, as there are many on-policy distributions, and any one of these is sufficient to guarantee stability. Consider an undiscounted episodic problem. The way episodes terminate is fully determined by the transition probabilities, but there may be several different ways the episodes might begin. However the episodes start, if all state transitions are due to the target policy, then the state distribution that results is an on-policy distribution. You might start close to the terminal state and visit only a few states with high probability before ending the episode. Or you might start far away and pass through many states before terminating. Both are on-policy distributions, and training on both with a linear semi-gradient method would be guaranteed to be stable. However the process starts, an on-policy distribution results as long as all states encountered are updated up until termination.

If there is discounting, it can be treated as partial or probabilistic termination for these purposes. If $\gamma = 0.9$, then we can consider that with probability 0.1 the process terminates on every time step and then immediately restarts in the state that is transitioned to. A discounted problem is one that is continually terminating and restarting with probability $1-\gamma$ on every step. This way of thinking about discounting is an example of a more general notion of *pseudo termination*—termination that does not affect the sequence of state transitions, but does affect the learning process and the quantities being learned. This kind of pseudo termination is important to off-policy learning because the restarting is optional—remember we can start any way we want to—and the termination relieves the need to keep including encountered states within the on-policy distribution. That is, if we don't consider the new states as restarts, then discounting quickly gives us a limited on-policy distribution.

The one-step Emphatic-TD algorithm for learning episodic state values is defined by:

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t),$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha M_t \rho_t \delta_t \nabla \hat{v}(S_t,\mathbf{w}_t),$$

$$M_t = \gamma \rho_{t-1} M_{t-1} + I_t,$$

with I_t, the *interest*, being arbitrary and M_t, the *emphasis*, being initialized to $M_{-1} = 0$. How does this algorithm perform on Baird's counterexample? Figure 11.6 shows the trajectory in expectation of the components of the parameter vector (for the case in which $I_t = 1$, for all t). There are some oscillations but eventually everything converges and the $\overline{\text{VE}}$ goes to zero. These trajectories are obtained by iteratively computing the expectation of the parameter vector trajectory without any of the variance due to sampling of transitions and rewards. We do not show the results of applying the Emphatic-TD algorithm directly because its variance on Baird's counterexample is so high that it is nigh impossible to get consistent results in computational experiments. The algorithm converges to the optimal solution in theory on this problem, but in practice it does not. We turn to the topic of reducing the variance of all these algorithms in the next section.

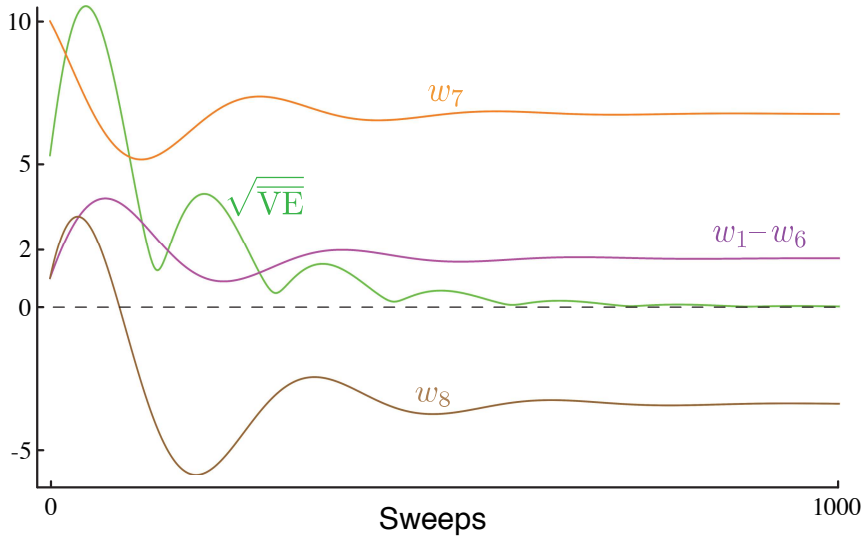


Figure 11.6: The behavior of the one-step Emphatic-TD algorithm in expectation on Baird's counterexample. The step size was $\alpha = 0.03$.

11.9 Reducing Variance

Off-policy learning is inherently of greater variance than on-policy learning. This is not surprising; if you receive data less closely related to a policy, you should expect to learn less about the policy's values. In the extreme, one may be able to learn nothing. You can't expect to learn how to drive by cooking dinner, for example. Only if the target and behavior policies are related, if they visit similar states and take similar actions, should one be able to make significant progress in off-policy training.

On the other hand, any policy has many neighbors, many similar policies with considerable overlap in states visited and actions chosen, and yet which are not identical. The raison d'être of off-policy learning is to enable generalization to this vast number of related-but-not-identical policies. The problem remains of how to make the best use of the experience. Now that we have some methods that are stable in expected value (if the step sizes are set right), attention naturally turns to reducing the variance of the estimates. There are many possible ideas, and we can just touch on a few of them in this introductory text.

Why is controlling variance especially critical in off-policy methods based on importance sampling? As we have seen, importance sampling often involves products of policy ratios. The ratios are always one in expectation (5.13), but their actual values may be very high or as low as zero. Successive ratios are uncorrelated, so their products are also always one in expected value, but they can be of very high variance. Recall that these ratios multiply the step size in SGD methods, so high variance means taking steps that vary greatly in their sizes. This is problematic for SGD because of the occasional very large steps. They must not be so large as to take the parameter to a part of the space with a very different gradient. SGD methods rely on averaging over multiple steps to get a good sense of the gradient, and if they make large moves from single samples they become unreliable. If the step-size parameter is set small enough to prevent this, then the expected step

can end up being very small, resulting in very slow learning. The notions of momentum (Derthick, 1984), of Polyak-Ruppert averaging (Polyak, 1990; Ruppert, 1988; Polyak and Juditsky, 1992), or further extensions of these ideas may significantly help. Methods for adaptively setting separate step sizes for different components of the parameter vector are also pertinent (e.g., Jacobs, 1988; Sutton, 1992b, c), as are the "importance weight aware" updates of Karampatziakis and Langford (2010).

In Chapter 5 we saw how weighted importance sampling is significantly better behaved, with lower variance updates, than ordinary importance sampling. However, adapting weighted importance sampling to function approximation is challenging and can probably only be done approximately with $O(d)$ complexity (Mahmood and Sutton, 2015).

The Tree Backup algorithm (Section 7.5) shows that it is possible to perform some off-policy learning without using importance sampling. This idea has been extended to the off-policy case to produce stable and more efficient methods by Munos, Stepleton, Harutyunyan, and Bellemare (2016) and by Mahmood, Yu and Sutton (2017).

Another, complementary strategy is to allow the target policy to be determined in part by the behavior policy, in such a way that it never can be so different from it to create large importance sampling ratios. For example, the target policy can be defined by reference to the behavior policy, as in the "recognizers" proposed by Precup et al. (2006).

11.10 Summary

Off-policy learning is a tempting challenge, testing our ingenuity in designing stable and efficient learning algorithms. Tabular Q-learning makes off-policy learning seem easy, and it has natural generalizations to Expected Sarsa and to the Tree Backup algorithm. But as we have seen in this chapter, the extension of these ideas to significant function approximation, even linear function approximation, involves new challenges and forces us to deepen our understanding of reinforcement learning algorithms.

Why go to such lengths? One reason to seek off-policy algorithms is to give flexibility in dealing with the tradeoff between exploration and exploitation. Another is to free behavior from learning, and avoid the tyranny of the target policy. TD learning appears to hold out the possibility of learning about multiple things in parallel, of using one stream of experience to solve many tasks simultaneously. We can certainly do this in special cases, just not in every case that we would like to or as efficiently as we would like to.

In this chapter we divided the challenge of off-policy learning into two parts. The first part, correcting the targets of learning for the behavior policy, is straightforwardly dealt with using the techniques devised earlier for the tabular case, albeit at the cost of increasing the variance of the updates and thereby slowing learning. High variance will probably always remains a challenge for off-policy learning.

The second part of the challenge of off-policy learning emerges as the instability of semi-gradient TD methods that involve bootstrapping. We seek powerful function approximation, off-policy learning, and the efficiency and flexibility of bootstrapping

TD methods, but it is challenging to combine all three aspects of this *deadly triad* in one algorithm without introducing the potential for instability. There have been several attempts. The most popular has been to seek to perform true stochastic gradient descent (SGD) in the Bellman error (a.k.a. the Bellman residual). However, our analysis concludes that this is not an appealing goal in many cases, and that anyway it is impossible to achieve with a learning algorithm—the gradient of the $\overline{\text{BE}}$ is not learnable from experience that reveals only feature vectors and not underlying states. Another approach, Gradient-TD methods, performs SGD in the *projected* Bellman error. The gradient of the $\overline{\text{PBE}}$ is learnable with $O(d)$ complexity, but at the cost of a second parameter vector with a second step size. The newest family of methods, Emphatic-TD methods, refine an old idea for reweighting updates, emphasizing some and de-emphasizing others. In this way they restore the special properties that make on-policy learning stable with computationally simple semi-gradient methods.

The whole area of off-policy learning is relatively new and unsettled. Which methods are best or even adequate is not yet clear. Are the complexities of the new methods introduced at the end of this chapter really necessary? Which of them can be combined effectively with variance reduction methods? The potential for off-policy learning remains tantalizing, the best way to achieve it still a mystery.

Bibliographical and Historical Remarks

11.1 The first semi-gradient method was linear TD(λ) (Sutton, 1988). The name "semi-gradient" is more recent (Sutton, 2015a). Semi-gradient off-policy TD(0) with general importance-sampling ratio may not have been explicitly stated until Sutton, Mahmood, and White (2016), but the action-value forms were introduced by Precup, Sutton, and Singh (2000), who also did eligibility trace forms of these algorithms (see Chapter 12). Their continuing, undiscounted forms have not been significantly explored. The n-step forms given here are new.

11.2 The earliest w-to-$2w$ example was given by Tsitsiklis and Van Roy (1996), who also introduced the specific counterexample in the box on page 263. Baird's counterexample is due to Baird (1995), though the version we present here is slightly modified. Averaging methods for function approximation were developed by Gordon (1995, 1996b). Other examples of instability with off-policy DP methods and more complex methods of function approximation are given by Boyan and Moore (1995). Bradtke (1993) gives an example in which Q-learning using linear function approximation in a linear quadratic regulation problem converges to a destabilizing policy.

11.3 The deadly triad was first identified by Sutton (1995b) and thoroughly analyzed by Tsitsiklis and Van Roy (1997). The name "deadly triad" is due to Sutton (2015a).

11.4 This kind of linear analysis was pioneered by Tsitsiklis and Van Roy (1996; 1997), including the dynamic programming operator. Diagrams like Figure 11.3 were

introduced by Lagoudakis and Parr (2003).

What we have called the Bellman operator, and denoted B_π, is more commonly denoted T^π and called a "dynamic programming operator," while a generalized form, denoted $T^{(\lambda)}$, is called the "TD(λ) operator" (Tsitsiklis and Van Roy, 1996, 1997).

11.5 The $\overline{\text{BE}}$ was first proposed as an objective function for dynamic programming by Schweitzer and Seidmann (1985). Baird (1995, 1999) extended it to TD learning based on stochastic gradient descent. In the literature, $\overline{\text{BE}}$ minimization is often referred to as Bellman residual minimization.

The earliest A-split example is due to Dayan (1992). The two forms given here were introduced by Sutton et al. (2009a).

11.6 The contents of this section are new to this text.

11.7 Gradient-TD methods were introduced by Sutton, Szepesvári, and Maei (2009b). The methods highlighted in this section were introduced by Sutton et al. (2009a) and Mahmood et al. (2014). A major extension to proximal TD methods was developed by Mahadevan et al. (2014). The most sensitive empirical investigations to date of Gradient-TD and related methods are given by Geist and Scherrer (2014), Dann, Neumann, and Peters (2014), White (2015), and Ghiassian, Patterson, White, Sutton, and White (2018). Recent developments in the theory of Gradient-TD methods are presented by Yu (2017).

11.8 Emphatic-TD methods were introduced by Sutton, Mahmood, and White (2016). Full convergence proofs and other theory were later established by Yu (2015; 2016; Yu, Mahmood, and Sutton, 2017), Hallak, Tamar, and Mannor (2015), and Hallak, Tamar, Munos, and Mannor (2016).

Chapter 12

Eligibility Traces

Eligibility traces are one of the basic mechanisms of reinforcement learning. For example, in the popular TD(λ) algorithm, the λ refers to the use of an eligibility trace. Almost any temporal-difference (TD) method, such as Q-learning or Sarsa, can be combined with eligibility traces to obtain a more general method that may learn more efficiently.

Eligibility traces unify and generalize TD and Monte Carlo methods. When TD methods are augmented with eligibility traces, they produce a family of methods spanning a spectrum that has Monte Carlo methods at one end ($\lambda = 1$) and one-step TD methods at the other ($\lambda = 0$). In between are intermediate methods that are often better than either extreme method. Eligibility traces also provide a way of implementing Monte Carlo methods online and on continuing problems without episodes.

Of course, we have already seen one way of unifying TD and Monte Carlo methods: the n-step TD methods of Chapter 7. What eligibility traces offer beyond these is an elegant algorithmic mechanism with significant computational advantages. The mechanism is a short-term memory vector, the *eligibility trace* $\mathbf{z}_t \in \mathbb{R}^d$, that parallels the long-term weight vector $\mathbf{w}_t \in \mathbb{R}^d$. The rough idea is that when a component of $\mathbf{w}_t$ participates in producing an estimated value, then the corresponding component of $\mathbf{z}_t$ is bumped up and then begins to fade away. Learning will then occur in that component of $\mathbf{w}_t$ if a nonzero TD error occurs before the trace falls back to zero. The trace-decay parameter $\lambda \in [0, 1]$ determines the rate at which the trace falls.

The primary computational advantage of eligibility traces over n-step methods is that only a single trace vector is required rather than a store of the last n feature vectors. Learning also occurs continually and uniformly in time rather than being delayed and then catching up at the end of the episode. In addition learning can occur and affect behavior immediately after a state is encountered rather than being delayed n steps.

Eligibility traces illustrate that a learning algorithm can sometimes be implemented in a different way to obtain computational advantages. Many algorithms are most naturally formulated and understood as an update of a state's value based on events that follow that state over multiple future time steps. For example, Monte Carlo methods (Chapter 5) update a state based on all the future rewards, and n-step TD methods (Chapter 7)

update based on the next n rewards and state n steps in the future. Such formulations, based on looking forward from the updated state, are called *forward views*. Forward views are always somewhat complex to implement because the update depends on later things that are not available at the time. However, as we show in this chapter it is often possible to achieve nearly the same updates—and sometimes *exactly* the same updates—with an algorithm that uses the current TD error, looking backward to recently visited states using an eligibility trace. These alternate ways of looking at and implementing learning algorithms are called *backward views*. Backward views, transformations between forward views and backward views, and equivalences between them, date back to the introduction of temporal difference learning but have become much more powerful and sophisticated since 2014. Here we present the basics of the modern view.

As usual, first we fully develop the ideas for state values and prediction, then extend them to action values and control. We develop them first for the on-policy case then extend them to off-policy learning. Our treatment pays special attention to the case of linear function approximation, for which the results with eligibility traces are stronger. All these results apply also to the tabular and state aggregation cases because these are special cases of linear function approximation.

12.1 The λ-return

In Chapter 7 we defined an n-step return as the sum of the first n rewards plus the estimated value of the state reached in n steps, each appropriately discounted (7.1). The general form of that equation, for any parameterized function approximator, is

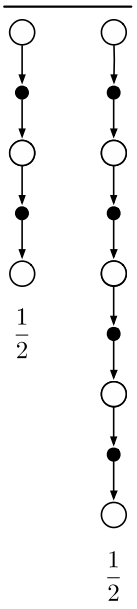
$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n}, \mathbf{w}_{t+n-1}), \quad 0 \leq t \leq T-n, \tag{12.1}$$

where $\hat{v}(s,\mathbf{w})$ is the approximate value of state s given weight vector $\mathbf{w}$ (Chapter 9), and T is the time of episode termination, if any. We noted in Chapter 7 that each n-step return, for $n \geq 1$, is a valid update target for a tabular learning update, just as it is for an approximate SGD learning update such as (9.7).

Now we note that a valid update can be done not just toward any n-step return, but toward any *average* of n-step returns for different ns. For example, an update can be done toward a target that is half of a two-step return and half of a four-step return: $\frac{1}{2}G_{t:t+2} + \frac{1}{2}G_{t:t+4}$. Any set of n-step returns can be averaged in this way, even an infinite set, as long as the weights on the component returns are positive and sum to 1. The composite return possesses an error reduction property similar to that of individual n-step returns (7.3) and thus can be used to construct updates with guaranteed convergence properties. Averaging produces a substantial new range of algorithms. For example, one could average one-step and infinite-step returns to obtain another way of interrelating TD and Monte Carlo methods. In principle, one could even average experience-based updates with DP updates to get a simple combination of experience-based and model-based methods (cf. Chapter 8).

An update that averages simpler component updates is called a *compound update*. The backup diagram for a compound update consists of the backup diagrams for each of the component updates with a horizontal line above them and the weighting fractions below.

For example, the compound update for the case mentioned at the start of this section, mixing half of a two-step return and half of a four-step return, has the diagram shown to the right. A compound update can only be done when the longest of its component updates is complete. The update at the right, for example, could only be done at time $t+4$ for the estimate formed at time t. In general one would like to limit the length of the longest component update because of the corresponding delay in the updates.



The TD(λ) algorithm can be understood as one particular way of averaging n-step updates. This average contains all the n-step updates, each weighted proportionally to λ^{n-1} (where $\lambda \in [0,1)$), and is normalized by a factor of $1-\lambda$ to ensure that the weights sum to 1 (Figure 12.1). The resulting update is toward a return, called the *λ-return*, defined in its state-based form by

$$G_t^\lambda \doteq (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n}. \tag{12.2}$$

Figure 12.2 further illustrates the weighting on the sequence of n-step returns in the λ-return. The one-step return is given the largest weight, $1-\lambda$; the two-step return is given the next largest weight, $(1-\lambda)\lambda$; the three-step return is given the weight $(1-\lambda)\lambda^2$; and so on. The weight fades by λ with each additional step. After a terminal state has been reached, all subsequent n-step returns are equal to the conventional return, G_t. If

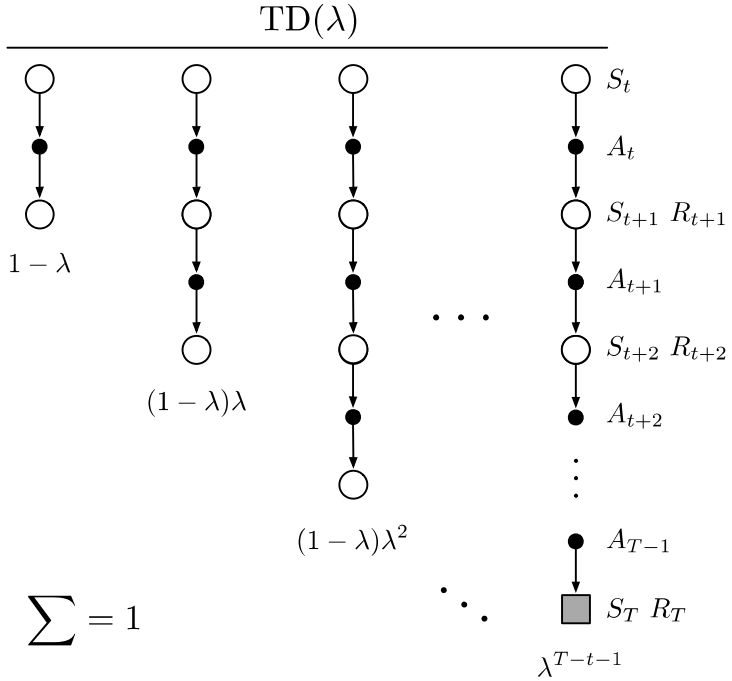


Figure 12.1: The backup diagram for TD(λ). If $\lambda = 0$, then the overall update reduces to its first component, the one-step TD update, whereas if $\lambda = 1$, then the overall update reduces to its last component, the Monte Carlo update.

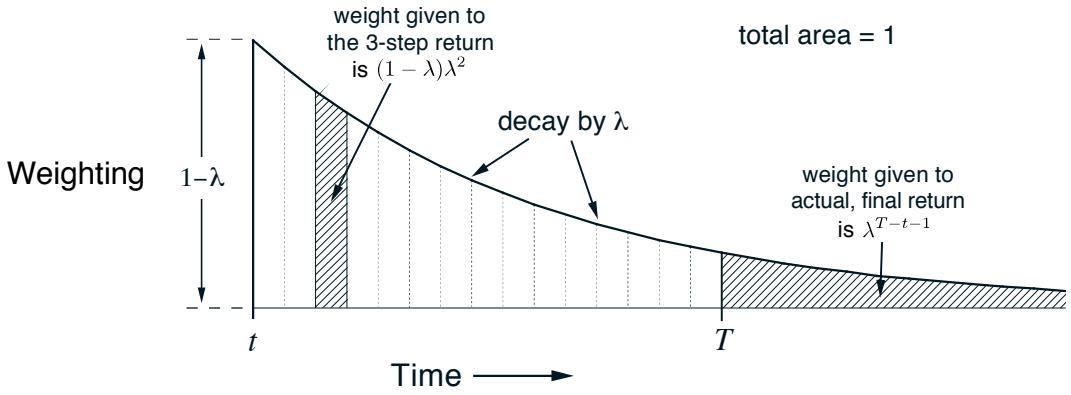


Figure 12.2: Weighting given in the λ-return to each of the n-step returns.

we want, we can separate these post-termination terms from the main sum, yielding

$$G_t^\lambda \;=\; (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_{t:t+n} \;+\; \lambda^{T-t-1}G_t, \tag{12.3}$$

as indicated in the figures. This equation makes it clearer what happens when $\lambda = 1$. In this case the main sum goes to zero, and the remaining term reduces to the conventional return. Thus, for $\lambda = 1$, updating according to the λ-return is a Monte Carlo algorithm. On the other hand, if $\lambda = 0$, then the λ-return reduces to $G_{t:t+1}$, the one-step return. Thus, for $\lambda = 0$, updating according to the λ-return is a one-step TD method.

Exercise 12.1 Just as the return can be written recursively in terms of the first reward and itself one-step later (3.9), so can the λ-return. Derive the analogous recursive relationship from (12.2) and (12.1). □

Exercise 12.2 The parameter λ characterizes how fast the exponential weighting in Figure 12.2 falls off, and thus how far into the future the λ-return algorithm looks in determining its update. But a rate factor such as λ is sometimes an awkward way of characterizing the speed of the decay. For some purposes it is better to specify a time constant, or half-life. What is the equation relating λ and the half-life, τ_λ, the time by which the weighting sequence will have fallen to half of its initial value? □

We are now ready to define our first learning algorithm based on the λ-return: the *off-line λ-return algorithm*. As an off-line algorithm, it makes no changes to the weight vector during the episode. Then, at the end of the episode, a whole sequence of off-line updates are made according to our usual semi-gradient rule, using the λ-return as the target:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[G_t^\lambda - \hat{v}(S_t,\mathbf{w}_t)\Big]\nabla\hat{v}(S_t,\mathbf{w}_t), \quad t = 0,\ldots,T-1. \tag{12.4}$$

The λ-return gives us an alternative way of moving smoothly between Monte Carlo and one-step TD methods that can be compared with the n-step bootstrapping way developed in Chapter 7. There we assessed effectiveness on a 19-state random walk task (Example 7.1, page 144). Figure 12.3 shows the performance of the off-line λ-return algorithm on this task alongside that of the n-step methods (repeated from Figure 7.2). The experiment was just as described earlier except that for the λ-return algorithm we varied λ instead of n. The performance measure used is the estimated root-mean-square error between the correct and estimated values of each state measured at the end of the episode, averaged over the first 10 episodes and the 19 states. Note that overall performance of the off-line λ-return algorithms is comparable to that of the n-step algorithms. In both cases we get best performance with an intermediate value of the bootstrapping parameter, n for n-step methods and λ for the off-line λ-return algorithm.

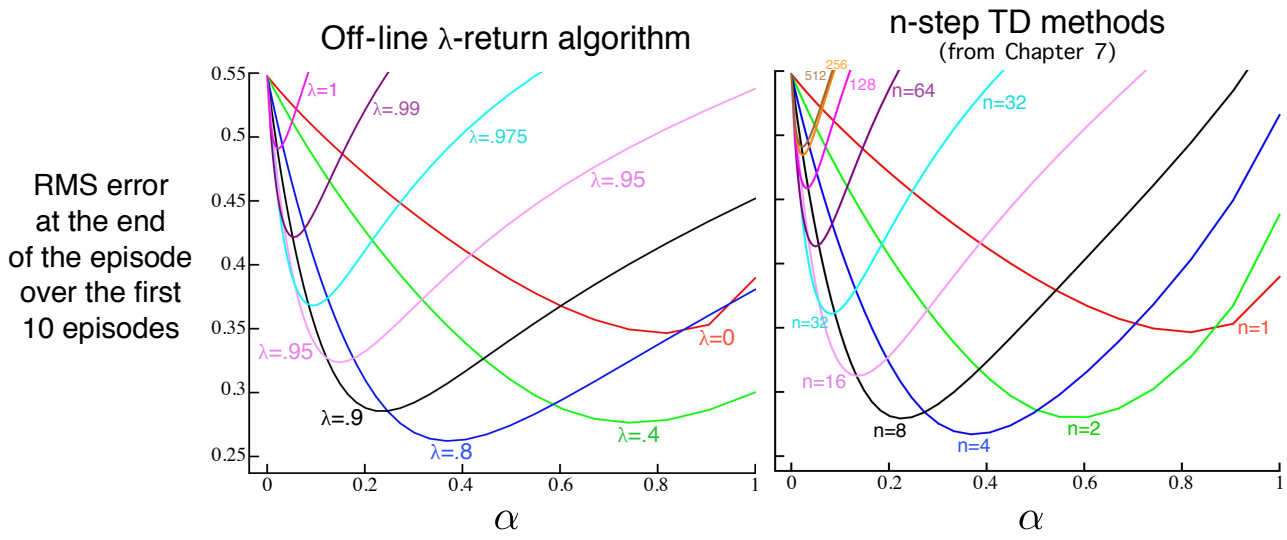


Figure 12.3: 19-state Random walk results (Example 7.1): Performance of the off-line λ-return algorithm alongside that of the n-step TD methods. In both case, intermediate values of the bootstrapping parameter (λ or n) performed best. The results with the off-line λ-return algorithm are slightly better at the best values of α and λ, and at high α.

The approach that we have been taking so far is what we call the theoretical, or *forward*, view of a learning algorithm. For each state visited, we look forward in time to all the future rewards and decide how best to combine them. We might imagine ourselves riding the stream of states, looking forward from each state to determine its update, as suggested by Figure 12.4. After looking forward from and updating one state, we move on to the next and never have to work with the preceding state again. Future states, on the other hand, are viewed and processed repeatedly, once from each vantage point preceding them.

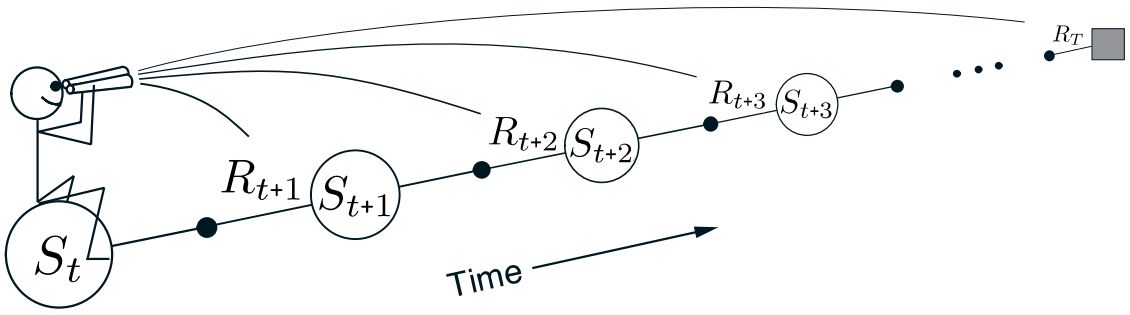


Figure 12.4: The forward view. We decide how to update each state by looking forward to future rewards and states.

12.2 TD(λ)

TD(λ) is one of the oldest and most widely used algorithms in reinforcement learning. It was the first algorithm for which a formal relationship was shown between a more theoretical forward view and a more computationally-congenial backward view using eligibility traces. Here we will show empirically that it approximates the off-line λ-return algorithm presented in the previous section.

TD(λ) improves over the off-line λ-return algorithm in three ways. First it updates the weight vector on every step of an episode rather than only at the end, and thus its estimates may be better sooner. Second, its computations are equally distributed in time rather than all at the end of the episode. And third, it can be applied to continuing problems rather than just to episodic problems. In this section we present the semi-gradient version of TD(λ) with function approximation.

With function approximation, the eligibility trace is a vector $\mathbf{z}_t \in \mathbb{R}^d$ with the same number of components as the weight vector $\mathbf{w}_t$. Whereas the weight vector is a long-term memory, accumulating over the lifetime of the system, the eligibility trace is a short-term memory, typically lasting less time than the length of an episode. Eligibility traces assist in the learning process; their only consequence is that they affect the weight vector, and then the weight vector determines the estimated value.

In TD(λ), the eligibility trace vector is initialized to zero at the beginning of the episode, is incremented on each time step by the value gradient, and then fades away by $\gamma\lambda$:

$$\begin{aligned} &\mathbf{z}_{-1} \doteq \mathbf{0}, \\ &\mathbf{z}_t \doteq \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{v}(S_t,\mathbf{w}_t), \quad 0 \leq t \leq T, \end{aligned} \tag{12.5}$$

where γ is the discount rate and λ is the parameter introduced in the previous section, which we henceforth call the trace-decay parameter. The eligibility trace keeps track of which components of the weight vector have contributed, positively or negatively, to recent state valuations, where "recent" is defined in terms of $\gamma\lambda$. (Recall that in linear function approximation, $\nabla\hat{v}(S_t,\mathbf{w}_t)$ is the feature vector, $\mathbf{x}_t$, in which case the eligibility trace vector is just a sum of past, fading, input vectors.) The trace is said to indicate the eligibility of each component of the weight vector for undergoing learning changes

should a reinforcing event occur. The reinforcing events we are concerned with are the moment-by-moment one-step TD errors. The TD error for state-value prediction is

$$\delta_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t). \tag{12.6}$$

In TD(λ), the weight vector is updated on each step proportional to the scalar TD error and the vector eligibility trace:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t. \tag{12.7}$$

Semi-gradient TD(λ) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0,1]$
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  $\mathbf{z} \leftarrow \mathbf{0}$ (a d-dimensional vector)
  Loop for each step of episode:
  | Choose $A \sim \pi(\cdot|S)$
  | Take action A, observe R, S'
  | $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \nabla\hat{v}(S,\mathbf{w})$
  | $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
  | $S \leftarrow S'$
  until S' is terminal

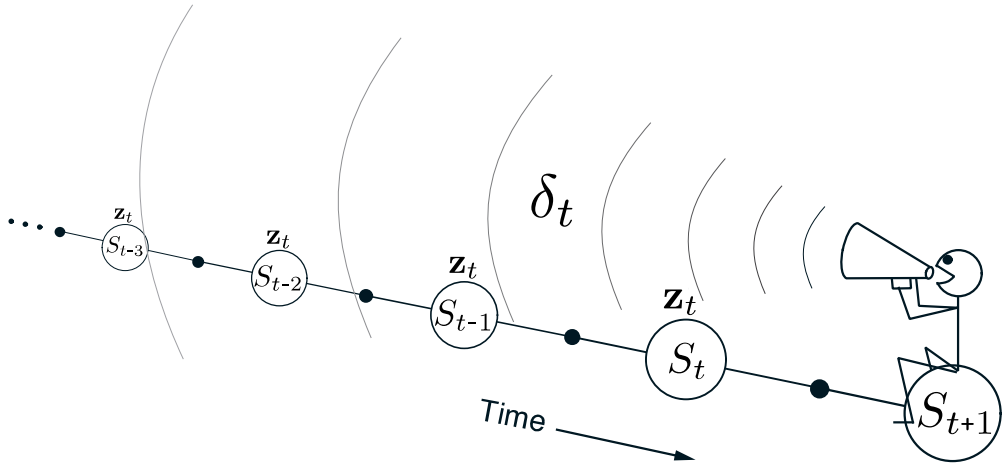


Figure 12.5: The backward or mechanistic view of TD(λ). Each update depends on the current TD error combined with the current eligibility traces of past events.

TD(λ) is oriented backward in time. At each moment we look at the current TD error and assign it backward to each prior state according to how much that state contributed to the current eligibility trace at that time. We might imagine ourselves riding along the stream of states, computing TD errors, and shouting them back to the previously visited states, as suggested by Figure 12.5. Where the TD error and traces come together, we get the update given by (12.7), changing the values of those past states for when they occur again in the future.

To better understand the backward view of TD(λ), consider what happens at various values of λ. If $\lambda = 0$, then by (12.5) the trace at t is exactly the value gradient corresponding to S_t. Thus the TD(λ) update (12.7) reduces to the one-step semi-gradient TD update treated in Chapter 9 (and, in the tabular case, to the simple TD rule (6.2)). This is why that algorithm was called TD(0). In terms of Figure 12.5, TD(0) is the case in which only the one state preceding the current one is updated by the TD error (other states may have their value estimates changed by generalization due to function approximation). For larger values of λ, but still $\lambda < 1$, more of the preceding states are updated, but each more temporally distant state is updated less because the corresponding eligibility trace is smaller, as suggested by the figure. We say that the earlier states are given less *credit* for the TD error.

If $\lambda = 1$, then the credit given to earlier states falls only by γ per step. This turns out to be just the right thing to do to achieve Monte Carlo behavior. For example, remember that the TD error, δ_t, includes an undiscounted term of R_{t+1}. In passing this back k steps it needs to be discounted, like any reward in a return, by γ^k, which is just what the falling eligibility trace achieves. If $\lambda = 1$ and $\gamma = 1$, then the eligibility traces do not decay at all with time. In this case the method behaves like a Monte Carlo method for an undiscounted, episodic task. If $\lambda = 1$, the algorithm is also known as TD(1).

TD(1) is a way of implementing Monte Carlo algorithms that is more general than those presented earlier and that significantly increases their range of applicability. Whereas the earlier Monte Carlo methods were limited to episodic tasks, TD(1) can be applied to discounted continuing tasks as well. Moreover, TD(1) can be performed incrementally and online. One disadvantage of Monte Carlo methods is that they learn nothing from an episode until it is over. For example, if a Monte Carlo control method takes an action that produces a very poor reward but does not end the episode, then the agent's tendency to repeat the action will be undiminished during the episode. Online TD(1), on the other hand, learns in an n-step TD way from the incomplete ongoing episode, where the n steps are all the way up to the current step. If something unusually good or bad happens during an episode, control methods based on TD(1) can learn immediately and alter their behavior on that same episode.

It is revealing to revisit the 19-state random walk example (Example 7.1) to see how well TD(λ) does in approximating the off-line λ-return algorithm. The results for both algorithms are shown in Figure 12.6. For each λ value, if α is selected optimally for it (or smaller), then the two algorithms perform virtually identically. If α is chosen larger than is optimal, however, then the λ-return algorithm is only a little worse whereas TD(λ) is much worse and may even be unstable. This is not catastrophic for TD(λ) on this problem, as these higher parameter values are not what one would want to use anyway, but for other problems it can be a significant weakness.

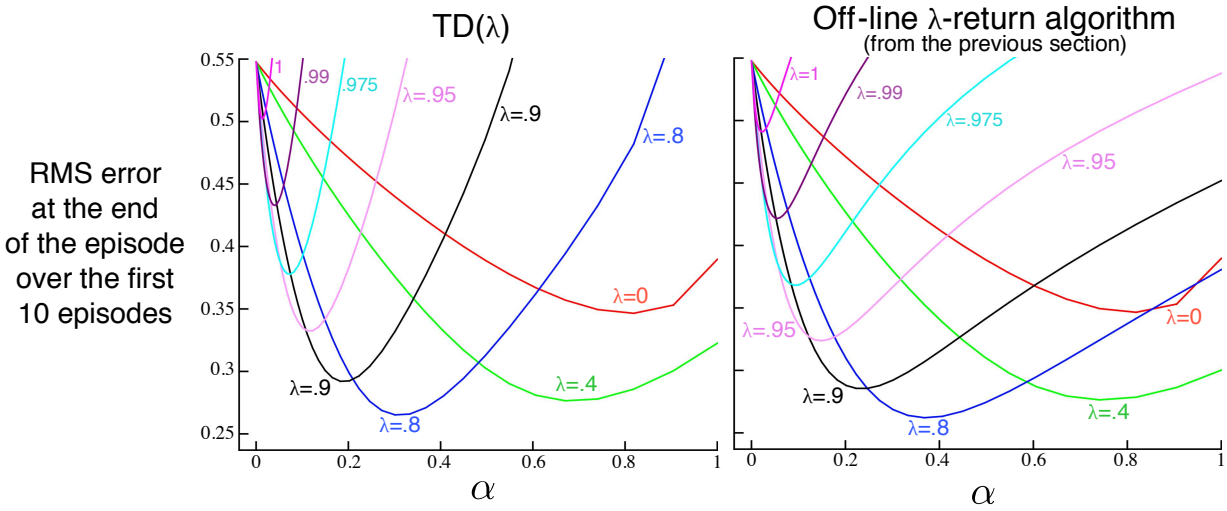


Figure 12.6: 19-state Random walk results (Example 7.1): Performance of TD(λ) alongside that of the off-line λ-return algorithm. The two algorithms performed virtually identically at low (less than optimal) α values, but TD(λ) was worse at high α values.

Linear TD(λ) has been proved to converge in the on-policy case if the step-size parameter is reduced over time according to the usual conditions (2.7). Just as discussed in Section 9.4, convergence is not to the minimum-error weight vector, but to a nearby weight vector that depends on λ. The bound on solution quality presented in that section (9.14) can now be generalized to apply for any λ. For the continuing discounted case,

$$\overline{\text{VE}}(\mathbf{w}_\infty) \quad \leq \quad \frac{1-\gamma\lambda}{1-\gamma} \min_{\mathbf{w}} \overline{\text{VE}}(\mathbf{w}). \tag{12.8}$$

That is, the asymptotic error is no more than $\frac{1-\gamma\lambda}{1-\gamma}$ times the smallest possible error. As λ approaches 1, the bound approaches the minimum error (and it is loosest at $\lambda = 0$). In practice, however, $\lambda = 1$ is often the poorest choice, as will be illustrated later in Figure 12.14.

Exercise 12.3 Some insight into how TD(λ) can closely approximate the off-line λ-return algorithm can be gained by seeing that the latter's error term (in brackets in (12.4)) can be written as the sum of TD errors (12.6) for a single fixed $\mathbf{w}$. Show this, following the pattern of (6.6), and using the recursive relationship for the λ-return you obtained in Exercise 12.1. □

Exercise 12.4 Use your result from the preceding exercise to show that, if the weight updates over an episode were computed on each step but not actually used to change the weights ($\mathbf{w}$ remained fixed), then the sum of TD(λ)'s weight updates would be the same as the sum of the off-line λ-return algorithm's updates. □

12.3 *n*-step Truncated λ-return Methods

The off-line λ-return algorithm is an important ideal, but it is of limited utility because it uses the λ-return (12.2), which is not known until the end of the episode. In the

continuing case, the λ-return is technically never known, as it depends on n-step returns for arbitrarily large n, and thus on rewards arbitrarily far in the future. However, the dependence becomes weaker for longer-delayed rewards, falling by $\gamma\lambda$ for each step of delay. A natural approximation, then, would be to truncate the sequence after some number of steps. Our existing notion of n-step returns provides a natural way to do this in which the missing rewards are replaced with estimated values.

In general, we define the *truncated λ-return* for time t, given data only up to some later horizon, h, as

$$G^{\lambda}_{t:h} \doteq (1-\lambda)\sum_{n=1}^{h-t-1}\lambda^{n-1}G_{t:t+n} + \lambda^{h-t-1}G_{t:h}, \qquad 0 \le t < h \le T. \tag{12.9}$$

If you compare this equation with the λ-return (12.3), it is clear that the horizon h is playing the same role as was previously played by T, the time of termination. Whereas in the λ-return there is a residual weight given to the conventional return G_t, here it is given to the longest available n-step return, $G_{t:h}$ (Figure 12.2).

The truncated λ-return immediately gives rise to a family of n-step λ-return algorithms similar to the n-step methods of Chapter 7. In all of these algorithms, updates are delayed by n steps and only take into account the first n rewards, but now all the k-step returns are included for $1 \le k \le n$ (whereas the earlier n-step algorithms used only the n-step return), weighted geometrically as in Figure 12.2. In the state-value case, this family of algorithms is known as Truncated TD(λ), or TTD(λ). The compound backup diagram, shown in Figure 12.7, is similar to that for TD(λ) (Figure 12.1) except that the longest component update is at most n steps rather than always going all the way to the

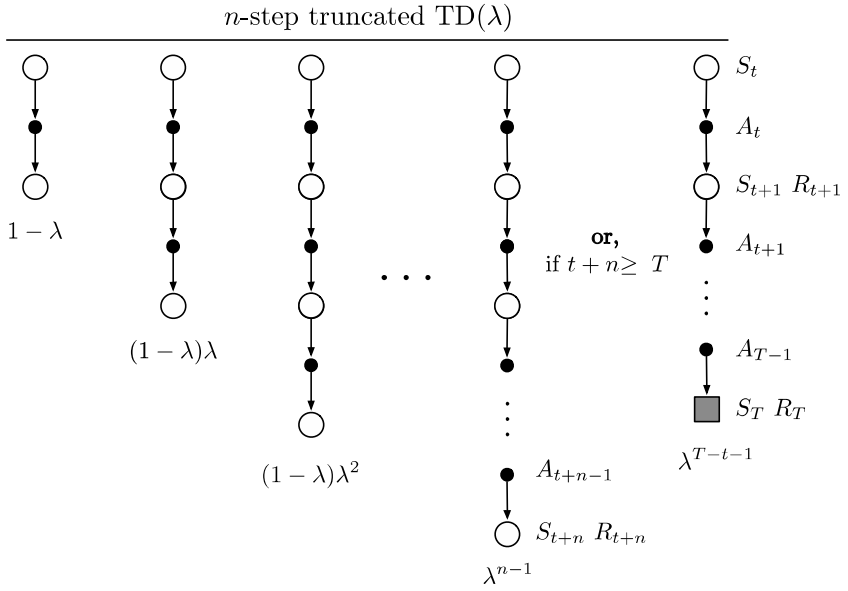


Figure 12.7: The backup diagram for Truncated TD(λ).

end of the episode. TTD(λ) is defined by (cf. (9.15)):

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha \left[G^{\lambda}_{t:t+n} - \hat{v}(S_t,\mathbf{w}_{t+n-1})\right] \nabla \hat{v}(S_t,\mathbf{w}_{t+n-1}), \qquad 0 \leq t < T.$$

This algorithm can be implemented efficiently so that per-step computation does not scale with n (though of course memory must). Much as in n-step TD methods, no updates are made on the first $n-1$ time steps of each episode, and $n-1$ additional updates are made upon termination. Efficient implementation relies on the fact that the k-step λ-return can be written exactly as

$$G^{\lambda}_{t:t+k} = \hat{v}(S_t,\mathbf{w}_{t-1}) + \sum_{i=t}^{t+k-1} (\gamma\lambda)^{i-t}\delta'_i, \tag{12.10}$$

where

$$\delta'_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_{t-1}).$$

Exercise 12.5 Several times in this book (often in exercises) we have established that returns can be written as sums of TD errors if the value function is held constant. Why is (12.10) another instance of this? Prove (12.10). □

12.4 Redoing Updates: Online λ-return Algorithm

Choosing the truncation parameter n in Truncated TD(λ) involves a tradeoff. n should be large so that the method closely approximates the off-line λ-return algorithm, but it should also be small so that the updates can be made sooner and can influence behavior sooner. Can we get the best of both? Well, yes, in principle we can, albeit at the cost of computational complexity.

The idea is that, on each time step as you gather a new increment of data, you go back and redo all the updates since the beginning of the current episode. The new updates will be better than the ones you previously made because now they can take into account the time step's new data. That is, the updates are always towards an n-step truncated λ-return target, but they always use the latest horizon. In each pass over that episode you can use a slightly longer horizon and obtain slightly better results. Recall that the truncated λ-return is defined in (12.9) as

$$G^{\lambda}_{t:h} \doteq (1-\lambda) \sum_{n=1}^{h-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{h-t-1} G_{t:h}.$$

Let us step through how this target could ideally be used if computational complexity was not an issue. The episode begins with an estimate at time 0 using the weights $\mathbf{w}_0$ from the end of the previous episode. Learning begins when the data horizon is extended to time step 1. The target for the estimate at step 0, given the data up to horizon 1, could only be the one-step return $G_{0:1}$, which includes R_1 and bootstraps from the estimate $\hat{v}(S_1,\mathbf{w}_0)$. Note that this is exactly what $G^{\lambda}_{0:1}$ is, with the sum in the first term of the

equation degenerating to zero. Using this update target, we construct $\mathbf{w}_1$. Then, after advancing the data horizon to step 2, what do we do? We have new data in the form of R_2 and S_2, as well as the new $\mathbf{w}_1$, so now we can construct a better update target $G^{\lambda}_{0:2}$ for the first update from S_0 as well as a better update target $G^{\lambda}_{1:2}$ for the second update from S_1. Using these improved targets, we redo the updates at S_1 and S_2, starting again from $\mathbf{w}_0$, to produce $\mathbf{w}_2$. Now we advance the horizon to step 3 and repeat, going all the way back to produce three new targets, redoing all updates starting from the original $\mathbf{w}_0$ to produce $\mathbf{w}_3$, and so on. Each time the horizon is advanced, all the updates are redone starting from $\mathbf{w}_0$ using the weight vector from the preceding horizon.

This conceptual algorithm involves multiple passes over the episode, one at each horizon, each generating a different sequence of weight vectors. To describe it clearly we have to distinguish between the weight vectors computed at the different horizons. Let us use $\mathbf{w}^h_t$ to denote the weights used to generate the value at time t in the sequence up to horizon h. The first weight vector $\mathbf{w}^h_0$ in each sequence is that inherited from the previous episode (so they are the same for all h), and the last weight vector $\mathbf{w}^h_h$ in each sequence defines the ultimate weight-vector sequence of the algorithm. At the final horizon $h = T$ we obtain the final weights $\mathbf{w}^T_T$ which will be passed on to form the initial weights of the next episode. With these conventions, the three first sequences described in the previous paragraph can be given explicitly:

$$h = 1: \quad \mathbf{w}^1_1 \doteq \mathbf{w}^1_0 + \alpha \left[G^{\lambda}_{0:1} - \hat{v}(S_0,\mathbf{w}^1_0)\right] \nabla \hat{v}(S_0,\mathbf{w}^1_0),$$

$$\begin{aligned} h = 2: \quad \mathbf{w}^2_1 &\doteq \mathbf{w}^2_0 + \alpha \left[G^{\lambda}_{0:2} - \hat{v}(S_0,\mathbf{w}^2_0)\right] \nabla \hat{v}(S_0,\mathbf{w}^2_0), \\ \mathbf{w}^2_2 &\doteq \mathbf{w}^2_1 + \alpha \left[G^{\lambda}_{1:2} - \hat{v}(S_1,\mathbf{w}^2_1)\right] \nabla \hat{v}(S_1,\mathbf{w}^2_1), \end{aligned}$$

$$\begin{aligned} h = 3: \quad \mathbf{w}^3_1 &\doteq \mathbf{w}^3_0 + \alpha \left[G^{\lambda}_{0:3} - \hat{v}(S_0,\mathbf{w}^3_0)\right] \nabla \hat{v}(S_0,\mathbf{w}^3_0), \\ \mathbf{w}^3_2 &\doteq \mathbf{w}^3_1 + \alpha \left[G^{\lambda}_{1:3} - \hat{v}(S_1,\mathbf{w}^3_1)\right] \nabla \hat{v}(S_1,\mathbf{w}^3_1), \\ \mathbf{w}^3_3 &\doteq \mathbf{w}^3_2 + \alpha \left[G^{\lambda}_{2:3} - \hat{v}(S_2,\mathbf{w}^3_2)\right] \nabla \hat{v}(S_2,\mathbf{w}^3_2). \end{aligned}$$

The general form for the update is

$$\mathbf{w}^h_{t+1} \doteq \mathbf{w}^h_t + \alpha \left[G^{\lambda}_{t:h} - \hat{v}(S_t,\mathbf{w}^h_t)\right] \nabla \hat{v}(S_t,\mathbf{w}^h_t), \quad 0 \le t < h \le T.$$

This update, together with $\mathbf{w}_t \doteq \mathbf{w}^t_t$ defines the *online λ-return algorithm.*

The online λ-return algorithm is fully online, determining a new weight vector $\mathbf{w}_t$ at each step t during an episode, using only information available at time t. Its main drawback is that it is computationally complex, passing over the portion of the episode experienced so far on every step. Note that it is strictly more complex than the off-line λ-return algorithm, which passes through all the steps at the time of termination but does not make any updates during the episode. In return, the online algorithm can be expected to perform better than the off-line one, not only during the episode when it makes an update while the off-line algorithm makes none, but also at the end of the episode because the weight vector used in bootstrapping (in $G^{\lambda}_{t:h}$) has had a larger number of informative

updates. This effect can be seen if one looks carefully at Figure 12.8, which compares the two algorithms on the 19-state random walk task.

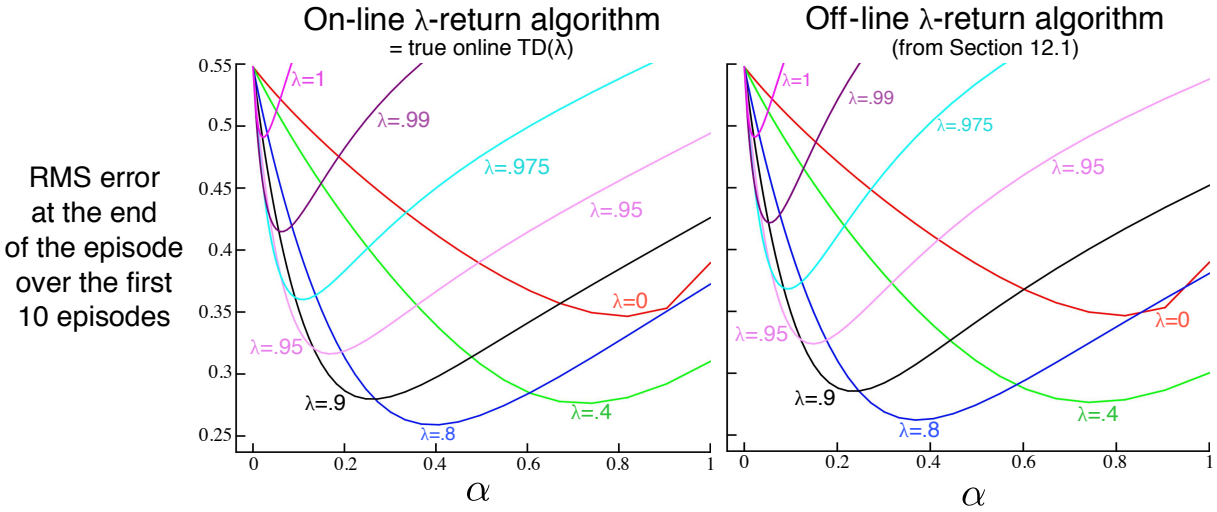


Figure 12.8: 19-state Random walk results (Example 7.1): Performance of online and off-line λ-return algorithms. The performance measure here is the $\overline{\text{VE}}$ at the end of the episode, which should be the best case for the off-line algorithm. Nevertheless, the online algorithm performs subtly better. For comparison, the $\lambda = 0$ line is the same for both methods.

12.5 True Online TD(λ)

The online λ-return algorithm just presented is currently the best performing temporal-difference algorithm. It is an ideal which online TD(λ) only approximates. As presented, however, the online λ-return algorithm is very complex. Is there a way to invert this forward-view algorithm to produce an efficient backward-view algorithm using eligibility traces? It turns out that there is indeed an exact computationally congenial implementation of the online λ-return algorithm for the case of linear function approximation. This implementation is known as the true online TD(λ) algorithm because it is "truer" to the ideal of the online λ-return algorithm than the TD(λ) algorithm is.

The derivation of true online TD(λ) is a little too complex to present here (see the next section and the appendix to the paper by van Seijen et al., 2016) but its strategy is simple. The sequence of weight vectors produced by the online λ-return algorithm can be arranged in a triangle:

$$\begin{array}{ccccccc}
\mathbf{w}_0^0 & & & & & & \\
\mathbf{w}_0^1 & \mathbf{w}_1^1 & & & & & \\
\mathbf{w}_0^2 & \mathbf{w}_1^2 & \mathbf{w}_2^2 & & & & \\
\mathbf{w}_0^3 & \mathbf{w}_1^3 & \mathbf{w}_2^3 & \mathbf{w}_3^3 & & & \\
\vdots & \vdots & \vdots & \vdots & \ddots & & \\
\mathbf{w}_0^T & \mathbf{w}_1^T & \mathbf{w}_2^T & \mathbf{w}_3^T & \cdots & \mathbf{w}_T^T
\end{array}$$

One row of this triangle is produced on each time step. It turns out that the weight vectors on the diagonal, the $\mathbf{w}_t^t$, are the only ones really needed. The first, $\mathbf{w}_0^0$, is the initial weight

vector of the episode, the last, $\mathbf{w}_T^T$, is the final weight vector, and each weight vector along the way, $\mathbf{w}_t^t$, plays a role in bootstrapping in the n-step returns of the updates. In the final algorithm the diagonal weight vectors are renamed without a superscript, $\mathbf{w}_t \doteq \mathbf{w}_t^t$. The strategy then is to find a compact, efficient way of computing each $\mathbf{w}_t^t$ from the one before. If this is done, for the linear case in which $\hat{v}(s,\mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s)$, then we arrive at the true online TD(λ) algorithm:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t \mathbf{z}_t + \alpha\left(\mathbf{w}_t^\top \mathbf{x}_t - \mathbf{w}_{t-1}^\top \mathbf{x}_t\right)(\mathbf{z}_t - \mathbf{x}_t),$$

where we have used the shorthand $\mathbf{x}_t \doteq \mathbf{x}(S_t)$, δ_t is defined as in TD(λ) (12.6), and $\mathbf{z}_t$ is defined by

$$\mathbf{z}_t \doteq \gamma\lambda \mathbf{z}_{t-1} + \left(1 - \alpha\gamma\lambda \mathbf{z}_{t-1}^\top \mathbf{x}_t\right)\mathbf{x}_t. \tag{12.11}$$

This algorithm has been proven to produce exactly the same sequence of weight vectors, $\mathbf{w}_t, 0 \le t \le T$, as the online λ-return algorithm (van Seijen et al. 2016). Thus the results on the random walk task on the left of Figure 12.8 are also its results on that task. Now, however, the algorithm is much less expensive. The memory requirements of true online TD(λ) are identical to those of conventional TD(λ), while the per-step computation is increased by about 50% (there is one more inner product in the eligibility-trace update). Overall, the per-step computational complexity remains of $O(d)$, the same as TD(λ). Pseudocode for the complete algorithm is given in the box.

True online TD(λ) for estimating $\mathbf{w}^\top \mathbf{x} \approx v_\pi$

Input: the policy π to be evaluated
Input: a feature function $\mathbf{x} : \mathcal{S}^+ \to \mathbb{R}^d$ such that $\mathbf{x}(terminal, \cdot) = \mathbf{0}$
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize state and obtain initial feature vector $\mathbf{x}$
  $\mathbf{z} \leftarrow \mathbf{0}$ (a d-dimensional vector)
  $V_{old} \leftarrow 0$ (a temporary scalar variable)
  Loop for each step of episode:
  | Choose $A \sim \pi$
  | Take action A, observe R, $\mathbf{x}'$ (feature vector of the next state)
  | $V \leftarrow \mathbf{w}^\top \mathbf{x}$
  | $V' \leftarrow \mathbf{w}^\top \mathbf{x}'$
  | $\delta \leftarrow R + \gamma V' - V$
  | $\mathbf{z} \leftarrow \gamma\lambda \mathbf{z} + \left(1 - \alpha\gamma\lambda \mathbf{z}^\top \mathbf{x}\right)\mathbf{x}$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha(\delta + V - V_{old})\mathbf{z} - \alpha(V - V_{old})\mathbf{x}$
  | $V_{old} \leftarrow V'$
  | $\mathbf{x} \leftarrow \mathbf{x}'$
  until $\mathbf{x}' = \mathbf{0}$ (signaling arrival at a terminal state)

The eligibility trace (12.11) used in true online TD(λ) is called a *dutch trace* to distinguish it from the trace (12.5) used in TD(λ), which is called an *accumulating trace*.

Earlier work often used a third kind of trace called the *replacing trace*, defined only for the tabular case or for binary feature vectors such as those produced by tile coding. The replacing trace is defined on a component-by-component basis depending on whether the component of the feature vector was 1 or 0:

$$z_{i,t} \doteq \begin{cases} 1 & \text{if } x_{i,t} = 1 \\ \gamma\lambda z_{i,t-1} & \text{otherwise.} \end{cases} \tag{12.12}$$

Nowadays, we see replacing traces as crude approximations to dutch traces, which largely supersede them. Dutch traces usually perform better than replacing traces and have a clearer theoretical basis. Accumulating traces remain of interest for nonlinear function approximations where dutch traces are not available.

12.6 *Dutch Traces in Monte Carlo Learning

Although eligibility traces are closely associated historically with TD learning, in fact they have nothing to do with it. In fact, eligibility traces arise even in Monte Carlo learning, as we show in this section. We show that the linear MC algorithm (Chapter 9), taken as a forward view, can be used to derive an equivalent yet computationally cheaper backward-view algorithm using dutch traces. This is the only equivalence of forward- and backward-views that we explicitly demonstrate in this book. It gives some of the flavor of the proof of equivalence of true online TD(λ) and the online λ-return algorithm, but is much simpler.

The linear version of the gradient Monte Carlo prediction algorithm (page 202) makes the following sequence of updates, one for each time step of the episode:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[G - \mathbf{w}_t^\top \mathbf{x}_t\right] \mathbf{x}_t, \qquad 0 \leq t < T. \tag{12.13}$$

To simplify the example, we assume here that the return G is a single reward received at the end of the episode (this is why G is not subscripted by time) and that there is no discounting. In this case the update is also known as the Least Mean Square (LMS) rule. As a Monte Carlo algorithm, all the updates depend on the final reward/return, so none can be made until the end of the episode. The MC algorithm is an off-line algorithm and we do not seek to improve this aspect of it. Rather we seek merely an implementation of this algorithm with computational advantages. We will still update the weight vector only at the end of the episode, but we will do some computation during each step of the episode and less at its end. This will give a more equal distribution of computation—$O(d)$ per step—and also remove the need to store the feature vectors at each step for use later at the end of each episode. Instead, we will introduce an additional vector memory, the eligibility trace, keeping in it a summary of all the feature vectors seen so far. This will be sufficient to efficiently recreate exactly the same overall update as the sequence of MC

updates (12.13), by the end of the episode:

$$\begin{aligned}
\mathbf{w}_T &= \mathbf{w}_{T-1} + \alpha\left(G - \mathbf{w}_{T-1}^\top \mathbf{x}_{T-1}\right)\mathbf{x}_{T-1} \\
&= \mathbf{w}_{T-1} + \alpha\mathbf{x}_{T-1}\left(-\mathbf{x}_{T-1}^\top \mathbf{w}_{T-1}\right) + \alpha G\mathbf{x}_{T-1} \\
&= \left(\mathbf{I} - \alpha\mathbf{x}_{T-1}\mathbf{x}_{T-1}^\top\right)\mathbf{w}_{T-1} + \alpha G\mathbf{x}_{T-1} \\
&= \mathbf{F}_{T-1}\mathbf{w}_{T-1} + \alpha G\mathbf{x}_{T-1}
\end{aligned}$$

where $\mathbf{F}_t \doteq \mathbf{I} - \alpha\mathbf{x}_t\mathbf{x}_t^\top$ is a *forgetting*, or *fading*, matrix. Now, recursing,

$$\begin{aligned}
&= \mathbf{F}_{T-1}\left(\mathbf{F}_{T-2}\mathbf{w}_{T-2} + \alpha G\mathbf{x}_{T-2}\right) + \alpha G\mathbf{x}_{T-1} \\
&= \mathbf{F}_{T-1}\mathbf{F}_{T-2}\mathbf{w}_{T-2} + \alpha G\left(\mathbf{F}_{T-1}\mathbf{x}_{T-2} + \mathbf{x}_{T-1}\right) \\
&= \mathbf{F}_{T-1}\mathbf{F}_{T-2}\left(\mathbf{F}_{T-3}\mathbf{w}_{T-3} + \alpha G\mathbf{x}_{T-3}\right) + \alpha G\left(\mathbf{F}_{T-1}\mathbf{x}_{T-2} + \mathbf{x}_{T-1}\right) \\
&= \mathbf{F}_{T-1}\mathbf{F}_{T-2}\mathbf{F}_{T-3}\mathbf{w}_{T-3} + \alpha G\left(\mathbf{F}_{T-1}\mathbf{F}_{T-2}\mathbf{x}_{T-3} + \mathbf{F}_{T-1}\mathbf{x}_{T-2} + \mathbf{x}_{T-1}\right) \\
&\;\;\vdots \\
&= \underbrace{\mathbf{F}_{T-1}\mathbf{F}_{T-2}\cdots\mathbf{F}_0\mathbf{w}_0}_{\mathbf{a}_{T-1}} \;+\; \alpha G\underbrace{\sum_{k=0}^{T-1}\mathbf{F}_{T-1}\mathbf{F}_{T-2}\cdots\mathbf{F}_{k+1}\mathbf{x}_k}_{\mathbf{z}_{T-1}} \\
&= \mathbf{a}_{T-1} + \alpha G\mathbf{z}_{T-1}\,, \qquad (12.14)
\end{aligned}$$

where $\mathbf{a}_{T-1}$ and $\mathbf{z}_{T-1}$ are the values at time $T-1$ of two auxiliary memory vectors that can be updated incrementally without knowledge of G and with $O(d)$ complexity per time step. The $\mathbf{z}_t$ vector is in fact a dutch-style eligibility trace. It is initialized to $\mathbf{z}_0 = \mathbf{x}_0$ and then updated according to

$$\begin{aligned}
\mathbf{z}_t &\doteq \sum_{k=0}^{t}\mathbf{F}_t\mathbf{F}_{t-1}\cdots\mathbf{F}_{k+1}\mathbf{x}_k, \qquad 1 \le t < T \\
&= \sum_{k=0}^{t-1}\mathbf{F}_t\mathbf{F}_{t-1}\cdots\mathbf{F}_{k+1}\mathbf{x}_k + \mathbf{x}_t \\
&= \mathbf{F}_t\sum_{k=0}^{t-1}\mathbf{F}_{t-1}\mathbf{F}_{t-2}\cdots\mathbf{F}_{k+1}\mathbf{x}_k + \mathbf{x}_t \\
&= \mathbf{F}_t\mathbf{z}_{t-1} + \mathbf{x}_t \\
&= \left(\mathbf{I} - \alpha\mathbf{x}_t\mathbf{x}_t^\top\right)\mathbf{z}_{t-1} + \mathbf{x}_t \\
&= \mathbf{z}_{t-1} - \alpha\mathbf{x}_t\mathbf{x}_t^\top\mathbf{z}_{t-1} + \mathbf{x}_t \\
&= \mathbf{z}_{t-1} - \alpha\left(\mathbf{z}_{t-1}^\top\mathbf{x}_t\right)\mathbf{x}_t + \mathbf{x}_t \\
&= \mathbf{z}_{t-1} + \left(1 - \alpha\mathbf{z}_{t-1}^\top\mathbf{x}_t\right)\mathbf{x}_t,
\end{aligned}$$

which is the dutch trace for the case of $\gamma\lambda{=}1$ (cf. Eq. 12.11). The $\mathbf{a}_t$ auxiliary vector is initialized to $\mathbf{a}_0 = \mathbf{w}_0$ and then updated according to

$$\mathbf{a}_t \;\doteq\; \mathbf{F}_t\mathbf{F}_{t-1}\cdots\mathbf{F}_0\mathbf{w}_0 \;=\; \mathbf{F}_t\mathbf{a}_{t-1} \;=\; \mathbf{a}_{t-1} - \alpha\mathbf{x}_t\mathbf{x}_t^\top\mathbf{a}_{t-1}, \quad 1 \le t < T.$$

The auxiliary vectors, $\mathbf{a}_t$ and $\mathbf{z}_t$, are updated on each time step $t < T$ and then, at time T when G is observed, they are used in (12.14) to compute $\mathbf{w}_T$. In this way we achieve exactly the same final result as the MC/LMS algorithm that has poor computational properties (12.13), but now with an incremental algorithm whose time and memory complexity per step is $O(d)$. This is surprising and intriguing because the notion of an eligibility trace (and the dutch trace in particular) has arisen in a setting without temporal-difference (TD) learning (in contrast to van Seijen and Sutton, 2014). It seems eligibility traces are not specific to TD learning at all; they are more fundamental than that. The need for eligibility traces seems to arise whenever one tries to learn long-term predictions in an efficient manner.

12.7 Sarsa(λ)

Very few changes in the ideas already presented in this chapter are required in order to extend eligibility-traces to action-value methods. To learn approximate action values, $\hat{q}(s, a, \mathbf{w})$, rather than approximate state values, $\hat{v}(s,\mathbf{w})$, we need to use the action-value form of the n-step return, from Chapter 10:

$$G_{t:t+n} \doteq R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \qquad t + n < T,$$

with $G_{t:t+n} \doteq G_t$ if $t + n \geq T$. Using this, we can form the action-value form of the λ-return, which is otherwise identical to the state-value form (12.3). The action-value form of the off-line λ-return algorithm (12.4) simply uses $\hat{q}$ rather than $\hat{v}$:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \Big[G_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w}_t) \Big] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t), \quad t = 0, \ldots, T-1, \tag{12.15}$$

where $G_t^\lambda \doteq G_{t:\infty}^\lambda$. The compound backup diagram for this forward view is shown in Figure 12.9. Notice the similarity to the diagram of the TD(λ) algorithm (Figure 12.1). The first update looks ahead one full step, to the next state–action pair, the second looks ahead two steps, to the second state–action pair, and so on. A final update is based on the complete return. The weighting of each n-step update in the λ-return is just as in TD(λ) and the λ-return algorithm (12.3).

The temporal-difference method for action values, known as *Sarsa(λ)*, approximates this forward view. It has the same update rule as given earlier for TD(λ):

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t,$$

except, naturally, using the action-value form of the TD error:

$$\delta_t \doteq R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t), \tag{12.16}$$

and the action-value form of the eligibility trace:

$$\begin{aligned} &\mathbf{z}_{-1} \doteq \mathbf{0}, \\ &\mathbf{z}_t \doteq \gamma \lambda \mathbf{z}_{t-1} + \nabla \hat{q}(S_t, A_t, \mathbf{w}_t), \quad 0 \leq t \leq T. \end{aligned}$$

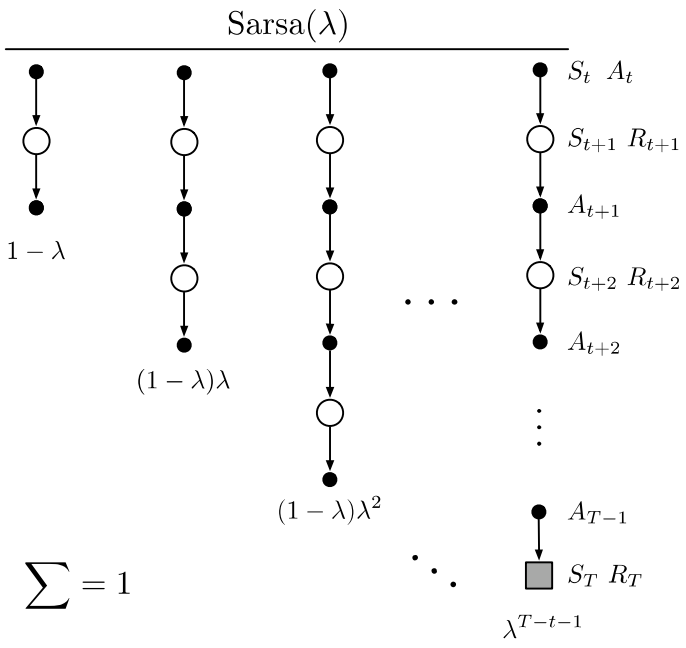
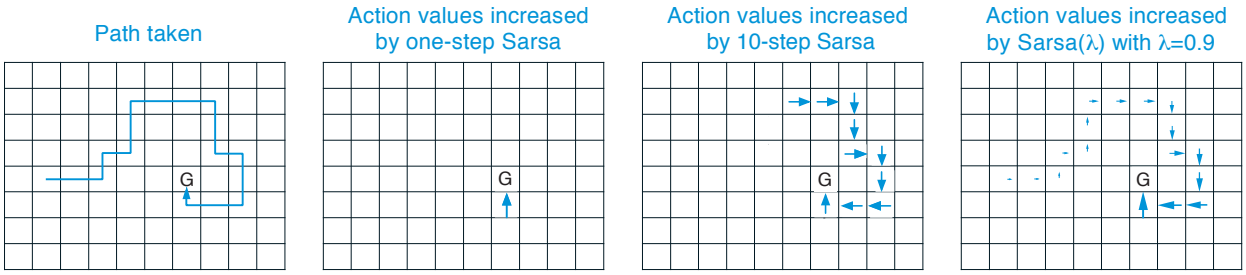


Figure 12.9: Sarsa(λ)'s backup diagram. Compare with Figure 12.1.

Complete pseudocode for Sarsa(λ) with linear function approximation, binary features, and either accumulating or replacing traces is given in the box on the next page. This pseudocode highlights a few optimizations possible in the special case of binary features (features are either active (=1) or inactive (=0)).

Example 12.1: Traces in Gridworld The use of eligibility traces can substantially increase the efficiency of control algorithms over one-step methods and even over n-step methods. The reason for this is illustrated by the gridworld example below.



The first panel shows the path taken by an agent in a single episode. The initial estimated values were zero, and all rewards were zero except for a positive reward at the goal location marked by G. The arrows in the other panels show, for various algorithms, which action-values would be increased, and by how much, upon reaching the goal. A one-step method would increment only the last action value, whereas an n-step method would equally increment the last n actions' values (assuming $\gamma = 1$), and an eligibility trace method would update all the action values up to the beginning of the episode, to different degrees, fading with recency. The fading strategy is often the best. ■

Sarsa(λ) with binary features and linear function approximation for estimating $\mathbf{w}^\top\mathbf{x} \approx q_\pi$ or q_*

Input: a function $\mathcal{F}(s,a)$ returning the set of (indices of) active features for s, a
Input: a policy π
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0,1]$, small $\varepsilon > 0$
Initialize: $\mathbf{w} = (w_1, \ldots, w_d)^\top \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$), $\mathbf{z} = (z_1, \ldots, z_d)^\top \in \mathbb{R}^d$

Loop for each episode:
  Initialize S
  Choose $A \sim \pi(\cdot|S)$ or ε-greedy according to $\hat{q}(S, \cdot, \mathbf{w})$
  $\mathbf{z} \leftarrow \mathbf{0}$
  Loop for each step of episode:
    Take action A, observe R, S'
    $\delta \leftarrow R$
    Loop for i in $\mathcal{F}(S, A)$:
      $\delta \leftarrow \delta - w_i$
      $z_i \leftarrow z_i + 1$ (accumulating traces)
      or $z_i \leftarrow 1$ (replacing traces)
    If S' is terminal then:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
      Go to next episode
    Choose $A' \sim \pi(\cdot|S')$ or ε-greedy according to $\hat{q}(S', \cdot, \mathbf{w})$
    Loop for i in $\mathcal{F}(S', A')$: $\delta \leftarrow \delta + \gamma w_i$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
    $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z}$
    $S \leftarrow S'; A \leftarrow A'$

Exercise 12.6 Modify the pseudocode for Sarsa(λ) to use dutch traces (12.11) without the other distinctive features of a true online algorithm. Assume linear function approximation and binary features. □

Example 12.2: Sarsa(λ) on Mountain Car Figure 12.10 (left) on the next page shows results with Sarsa(λ) on the Mountain Car task introduced in Example 10.1. The function approximation, action selection, and environmental details were exactly as in Chapter 10, and thus it is appropriate to numerically compare these results with the Chapter 10 results for n-step Sarsa (right side of the figure). The earlier results varied the update length n whereas here for Sarsa(λ) we vary the trace parameter λ, which plays a similar role. The fading-trace bootstrapping strategy of Sarsa(λ) appears to result in more efficient learning on this problem. ■

There is also an action-value version of our ideal TD method, the online λ-return algorithm (Section 12.4) and its efficient implementation as true online TD(λ) (Section 12.5). Everything in Section 12.4 goes through without change other than to use the action-value form of the n-step return given at the beginning of the current section. The analyses in Sections 12.5 and 12.6 also carry through for action values, the only change being the use

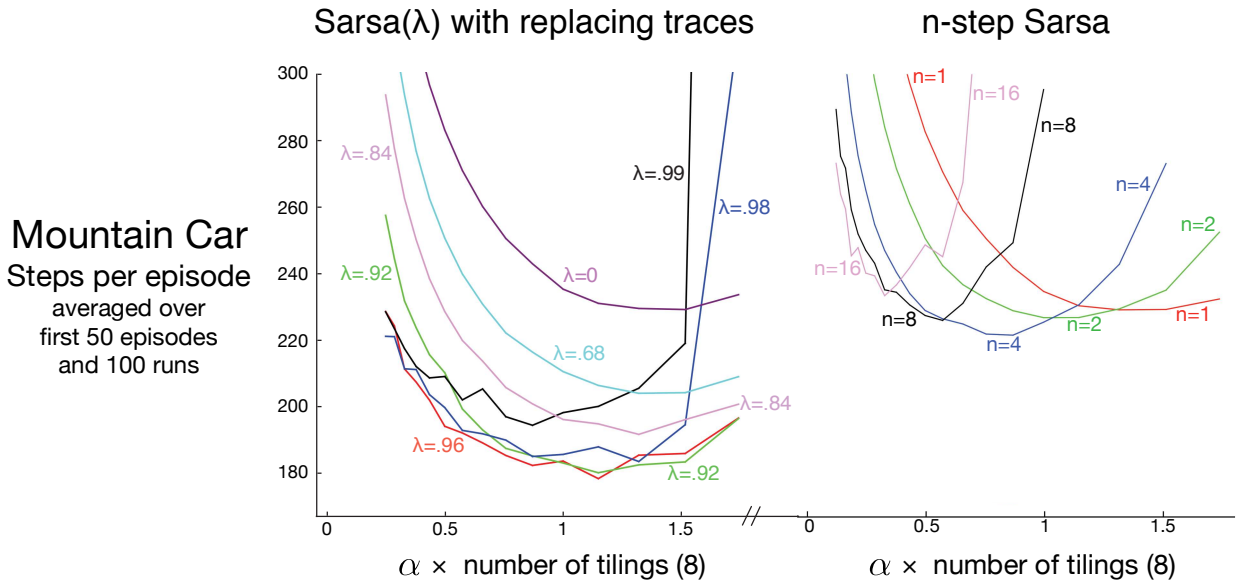


Figure 12.10: Early performance on the Mountain Car task of Sarsa(λ) with replacing traces and n-step Sarsa (copied from Figure 10.4) as a function of the step size, α.

of state–action feature vectors $\mathbf{x}_t = \mathbf{x}(S_t, A_t)$ instead of state feature vectors $\mathbf{x}_t = \mathbf{x}(S_t)$. Pseudocode for the resulting efficient algorithm, called *true online Sarsa(λ)* is given in the box on the next page. The figure below compares the performance of various versions of Sarsa(λ) on the Mountain Car example.

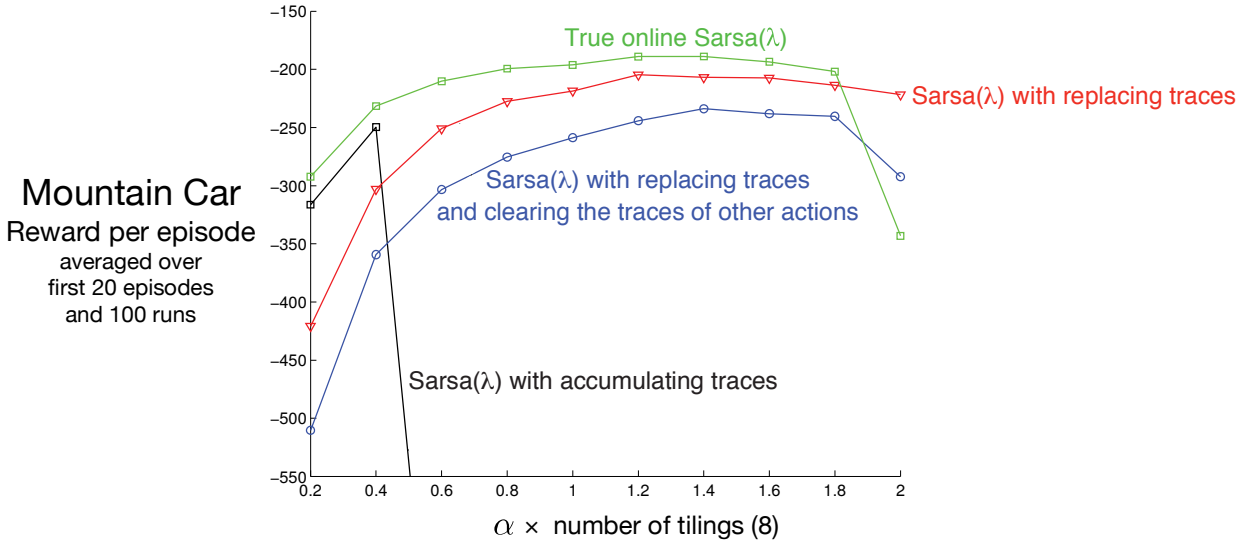


Figure 12.11: Summary comparison of Sarsa(λ) algorithms on the Mountain Car task. True online Sarsa(λ) performed better than regular Sarsa(λ) with both accumulating and replacing traces. Also included is a version of Sarsa(λ) with replacing traces in which, on each time step, the traces for the state and the actions not selected were set to zero.

True online Sarsa(λ) for estimating $\mathbf{w}^\top \mathbf{x} \approx q_\pi$ or q_*

Input: a feature function $\mathbf{x} : \mathcal{S}^+ \times \mathcal{A} \to \mathbb{R}^d$ such that $\mathbf{x}(terminal, \cdot) = \mathbf{0}$
Input: a policy π (if estimating q_π)
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$, small $\varepsilon > 0$
Initialize: $\mathbf{w} \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Choose $A \sim \pi(\cdot|S)$ or ε-greedy according to $\hat{q}(S, \cdot, \mathbf{w})$
  $\mathbf{x} \leftarrow \mathbf{x}(S, A)$
  $\mathbf{z} \leftarrow \mathbf{0}$
  $Q_{old} \leftarrow 0$
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose $A' \sim \pi(\cdot|S')$ or ε-greedy according to $\hat{q}(S', \cdot, \mathbf{w})$
    $\mathbf{x}' \leftarrow \mathbf{x}(S', A')$
    $Q \leftarrow \mathbf{w}^\top \mathbf{x}$
    $Q' \leftarrow \mathbf{w}^\top \mathbf{x}'$
    $\delta \leftarrow R + \gamma Q' - Q$
    $\mathbf{z} \leftarrow \gamma\lambda \mathbf{z} + \big(1 - \alpha\gamma\lambda \mathbf{z}^\top \mathbf{x}\big)\, \mathbf{x}$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha(\delta + Q - Q_{old})\mathbf{z} - \alpha(Q - Q_{old})\mathbf{x}$
    $Q_{old} \leftarrow Q'$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    $A \leftarrow A'$
  until S' is terminal

Finally, there is also a truncated version of Sarsa(λ), called *forward Sarsa(λ)* (van Seijen, 2016), which appears to be a particularly effective model-free control method for use in conjunction with multi-layer artificial neural networks.

12.8 Variable λ and γ

We are starting now to reach the end of our development of fundamental TD learning algorithms. To present the final algorithms in their most general forms, it is useful to generalize the degree of bootstrapping and discounting beyond constant parameters to functions potentially dependent on the state and action. That is, each time step will have a different λ and γ, denoted λ_t and γ_t. We change notation now so that $\lambda : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is now a function from states and actions to the unit interval such that $\lambda_t \doteq \lambda(S_t, A_t)$, and similarly, $\gamma : \mathcal{S} \to [0, 1]$ is a function from states to the unit interval such that $\gamma_t \doteq \gamma(S_t)$.

Introducing the function γ, the *termination function*, is particularly significant because it changes the return, the fundamental random variable whose expectation we seek to

estimate. Now the return is defined more generally as

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma_{t+1} G_{t+1} \\ &= R_{t+1} + \gamma_{t+1} R_{t+2} + \gamma_{t+1}\gamma_{t+2} R_{t+3} + \gamma_{t+1}\gamma_{t+2}\gamma_{t+3} R_{t+4} + \cdots \\ &= \sum_{k=t}^{\infty} \left(\prod_{i=t+1}^{k} \gamma_i \right) R_{k+1}, \end{aligned} \tag{12.17}$$

where, to assure the sums are finite, we require that $\prod_{k=t}^{\infty} \gamma_k = 0$ with probability one for all t. One convenient aspect of this definition is that it enables the episodic setting and its algorithms to be presented in terms of a single stream of experience, without special terminal states, start distributions, or termination times. An erstwhile terminal state becomes a state at which $\gamma(s) = 0$ and which transitions to the start distribution. In that way (and by choosing $\gamma(\cdot)$ as a constant in all other states) we can recover the classical episodic setting as a special case. State dependent termination includes other prediction cases such as *pseudo termination*, in which we seek to predict a quantity without altering the flow of the Markov process. Discounted returns can be thought of as such a quantity, in which case state-dependent termination unifies the episodic and discounted-continuing cases. (The undiscounted-continuing case still needs some special treatment.)

The generalization to variable bootstrapping is not a change in the problem, like discounting, but a change in the solution strategy. The generalization affects the λ-returns for states and actions. The new state-based λ-return can be written recursively as

$$G_t^{\lambda s} \doteq R_{t+1} + \gamma_{t+1} \left((1 - \lambda_{t+1}) \hat{v}(S_{t+1}, \mathbf{w}_t) + \lambda_{t+1} G_{t+1}^{\lambda s} \right), \tag{12.18}$$

where now we have added the "s" to the superscript λ to remind us that this is a return that bootstraps from state values, distinguishing it from returns that bootstrap from action values, which we present below with "a" in the superscript. This equation says that the λ-return is the first reward, undiscounted and unaffected by bootstrapping, plus possibly a second term to the extent that we are not discounting at the next state (that is, according to γ_{t+1}; recall that this is zero if the next state is terminal). To the extent that we aren't terminating at the next state, we have a second term which is itself divided into two cases depending on the degree of bootstrapping in the state. To the extent we are bootstrapping, this term is the estimated value at the state, whereas, to the extent that we are not bootstrapping, the term is the λ-return for the next time step. The action-based λ-return is either the Sarsa form

$$G_t^{\lambda a} \doteq R_{t+1} + \gamma_{t+1} \Big((1 - \lambda_{t+1}) \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) + \lambda_{t+1} G_{t+1}^{\lambda a} \Big), \tag{12.19}$$

or the Expected Sarsa form,

$$G_t^{\lambda a} \doteq R_{t+1} + \gamma_{t+1} \Big((1 - \lambda_{t+1}) \bar{V}_t(S_{t+1}) + \lambda_{t+1} G_{t+1}^{\lambda a} \Big), \tag{12.20}$$

where (7.8) is generalized to function approximation as

$$\bar{V}_t(s) \doteq \sum_a \pi(a|s) \hat{q}(s, a, \mathbf{w}_t). \tag{12.21}$$

Exercise 12.7 Generalize the three recursive equations above to their truncated versions, defining $G^{\lambda s}_{t:h}$ and $G^{\lambda a}_{t:h}$. □

12.9 Off-policy Traces with Control Variates

The final step is to incorporate importance sampling. For methods using non-truncated λ-returns, there is not a practical option in which the importance-sampling weighting is applied to the target return (as there is for n-step methods as explained in Section 7.3). Instead, we move directly to the bootstrapping generalization of per-decision importance sampling with control variates (Section 7.4).

In the state case, our final definition of the λ-return generalizes (12.18), after the model of (7.13), to

$$G^{\lambda s}_t \doteq \rho_t\Big(R_{t+1}+\gamma_{t+1}\big((1-\lambda_{t+1})\hat{v}(S_{t+1},\mathbf{w}_t)+\lambda_{t+1}G^{\lambda s}_{t+1}\big)\Big)+(1-\rho_t)\hat{v}(S_t,\mathbf{w}_t), \quad (12.22)$$

where $\rho_t = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$ is the usual single-step importance sampling ratio. Much like the other returns we have seen in this book, this final λ-return can be approximated simply in terms of sums of the state-based TD error,

$$\delta^s_t \doteq R_{t+1}+\gamma_{t+1}\hat{v}(S_{t+1},\mathbf{w}_t)-\hat{v}(S_t,\mathbf{w}_t), \quad (12.23)$$

as

$$G^{\lambda s}_t \approx \hat{v}(S_t,\mathbf{w}_t)+\rho_t\sum_{k=t}^{\infty}\delta^s_k\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i, \quad (12.24)$$

with the approximation becoming exact if the approximate value function does not change.

Exercise 12.8 Prove that (12.24) becomes exact if the value function does not change. To save writing, consider the case of $t=0$, and use the notation $V_k \doteq \hat{v}(S_k,\mathbf{w})$. □

Exercise 12.9 The truncated version of the general off-policy return is denoted $G^{\lambda s}_{t:h}$. Guess the correct equation, based on (12.24). □

The above form of the λ-return (12.24) is convenient to use in a forward-view update,

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t+\alpha\left(G^{\lambda s}_t-\hat{v}(S_t,\mathbf{w}_t)\right)\nabla\hat{v}(S_t,\mathbf{w}_t)\\ &\approx \mathbf{w}_t+\alpha\rho_t\left(\sum_{k=t}^{\infty}\delta^s_k\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i\right)\nabla\hat{v}(S_t,\mathbf{w}_t),\end{aligned}$$

which to the experienced eye looks like an eligibility-based TD update—the product is like an eligibility trace and it is multiplied by TD errors. But this is just one time step of a forward view. The relationship that we are looking for is that the forward-view update, summed over time, is approximately equal to a backward-view update, summed over time (this relationship is only approximate because again we ignore changes in the value

function). The sum of the forward-view update over time is

$$\begin{aligned}
\sum_{t=0}^{\infty}(\mathbf{w}_{t+1}-\mathbf{w}_t) &\approx \sum_{t=0}^{\infty}\sum_{k=t}^{\infty}\alpha\rho_t\delta_k^s\nabla\hat{v}(S_t,\mathbf{w}_t)\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i \\
&= \sum_{k=0}^{\infty}\sum_{t=0}^{k}\alpha\rho_t\nabla\hat{v}(S_t,\mathbf{w}_t)\delta_k^s\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i \\
&\qquad\qquad \text{(using the summation rule: } \textstyle\sum_{t=x}^{y}\sum_{k=t}^{y}=\sum_{k=x}^{y}\sum_{t=x}^{k}\text{)} \\
&= \sum_{k=0}^{\infty}\alpha\delta_k^s\sum_{t=0}^{k}\rho_t\nabla\hat{v}(S_t,\mathbf{w}_t)\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i,
\end{aligned}$$

which would be in the form of the sum of a backward-view TD update if the entire expression from the second sum on could be written and updated incrementally as an eligibility trace, which we now show can be done. That is, we show that if this expression was the trace at time k, then we could update it from its value at time $k-1$ by:

$$\begin{aligned}
\mathbf{z}_k &= \sum_{t=0}^{k}\rho_t\nabla\hat{v}(S_t,\mathbf{w}_t)\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i \\
&= \sum_{t=0}^{k-1}\rho_t\nabla\hat{v}(S_t,\mathbf{w}_t)\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i \quad + \quad \rho_k\nabla\hat{v}(S_k,\mathbf{w}_k) \\
&= \gamma_k\lambda_k\rho_k\underbrace{\sum_{t=0}^{k-1}\rho_t\nabla\hat{v}(S_t,\mathbf{w}_t)\prod_{i=t+1}^{k-1}\gamma_i\lambda_i\rho_i}_{\mathbf{z}_{k-1}} \quad + \quad \rho_k\nabla\hat{v}(S_k,\mathbf{w}_k) \\
&= \rho_k\big(\gamma_k\lambda_k\mathbf{z}_{k-1}+\nabla\hat{v}(S_k,\mathbf{w}_k)\big),
\end{aligned}$$

which, changing the index from k to t, is the general accumulating trace update for state values:

$$\mathbf{z}_t \doteq \rho_t\big(\gamma_t\lambda_t\mathbf{z}_{t-1}+\nabla\hat{v}(S_t,\mathbf{w}_t)\big), \tag{12.25}$$

This eligibility trace, together with the usual semi-gradient parameter-update rule for TD(λ) (12.7), forms a general TD(λ) algorithm that can be applied to either on-policy or off-policy data. In the on-policy case, the algorithm is exactly TD(λ) because ρ_t is alway 1 and (12.25) becomes the usual accumulating trace (12.5) (extended to variable λ and γ). In the off-policy case, the algorithm often works well but, as a semi-gradient method, is not guaranteed to be stable. In the next few sections we will consider extensions of it that do guarantee stability.

A very similar series of steps can be followed to derive the off-policy eligibility traces for *action*-value methods and corresponding general Sarsa(λ) algorithms. One could start with either recursive form for the general action-based λ-return, (12.19) or (12.20), but the latter (the Expected Sarsa form) works out to be simpler. We extend (12.20) to the

off-policy case after the model of (7.14) to produce

$$G_t^{\lambda a} \doteq R_{t+1}+\gamma_{t+1}\Big((1-\lambda_{t+1})\bar{V}_t(S_{t+1})+\lambda_{t+1}\big[\rho_{t+1}G_{t+1}^{\lambda a}+\bar{V}_t(S_{t+1})-\rho_{t+1}\hat{q}(S_{t+1},A_{t+1},\mathbf{w}_t)\big]\Big)$$
$$= R_{t+1} + \gamma_{t+1}\Big(\bar{V}_t(S_{t+1}) + \lambda_{t+1}\rho_{t+1}\big[G_{t+1}^{\lambda a} - \hat{q}(S_{t+1},A_{t+1},\mathbf{w}_t)\big]\Big) \tag{12.26}$$

where $\bar{V}_t(S_{t+1})$ is as given by (12.21). Again the λ-return can be written approximately as the sum of TD errors,

$$G_t^{\lambda a} \approx \hat{q}(S_t,A_t,\mathbf{w}_t) + \sum_{k=t}^{\infty}\delta_k^a \prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i, \tag{12.27}$$

using the expectation form of the action-based TD error:

$$\delta_t^a = R_{t+1} + \gamma_{t+1}\bar{V}_t(S_{t+1}) - \hat{q}(S_t,A_t,\mathbf{w}_t). \tag{12.28}$$

As before, the approximation becomes exact if the approximate value function does not change.

Exercise 12.10 Prove that (12.27) becomes exact if the value function does not change. To save writing, consider the case of $t = 0$, and use the notation $Q_k = \hat{q}(S_k,A_k,\mathbf{w})$. Hint: Start by writing out δ_0^a and $G_0^{\lambda a}$, then $G_0^{\lambda a} - Q_0$. □

Exercise 12.11 The truncated version of the general off-policy return is denoted $G_{t:h}^{\lambda a}$. Guess the correct equation for it, based on (12.27). □

Using steps entirely analogous to those for the state case, one can write a forward-view update based on (12.27), transform the sum of the updates using the summation rule, and finally derive the following form for the eligibility trace for action values:

$$\mathbf{z}_t \doteq \gamma_t\lambda_t\rho_t\mathbf{z}_{t-1} + \nabla\hat{q}(S_t,A_t,\mathbf{w}_t). \tag{12.29}$$

This eligibility trace, together with the expectation-based TD error (12.28) and the usual semi-gradient parameter-update rule (12.7), forms an elegant, efficient Expected Sarsa(λ) algorithm that can be applied to either on-policy or off-policy data. It is probably the best algorithm of this type at the current time (though of course it is not guaranteed to be stable until combined in some way with one of the methods presented in the following sections). In the on-policy case with constant λ and γ, and the usual state–action TD error (12.16), the algorithm would be identical to the Sarsa(λ) algorithm presented in Section 12.7.

Exercise 12.12 Show in detail the steps outlined above for deriving (12.29) from (12.27). Start with the update (12.15), substitute $G_t^{\lambda a}$ from (12.26) for G_t^{λ}, then follow similar steps as led to (12.25). □

At $\lambda = 1$, these algorithms become closely related to corresponding Monte Carlo algorithms. One might expect that an exact equivalence would hold for episodic problems and off-line updating, but in fact the relationship is subtler and slightly weaker than that. Under these most favorable conditions still there is not an episode by episode equivalence of updates, only of their expectations. This should not be surprising as these methods

make irrevocable updates as a trajectory unfolds, whereas true Monte Carlo methods would make no update for a trajectory if any action within it has zero probability under the target policy. In particular, all of these methods, even at $\lambda = 1$, still bootstrap in the sense that their targets depend on the current value estimates—it's just that the dependence cancels out in expected value. Whether this is a good or bad property in practice is another question. Recently, methods have been proposed that do achieve an exact equivalence (Sutton, Mahmood, Precup and van Hasselt, 2014). These methods require an additional vector of "provisional weights" that keep track of updates which have been made but may need to be retracted (or emphasized) depending on the actions taken later. The state and state–action versions of these methods are called PTD(λ) and PQ(λ) respectively, where the 'P' stands for Provisional.

The practical consequences of all these new off-policy methods have not yet been established. Undoubtedly, issues of high variance will arise as they do in all off-policy methods using importance sampling (Section 11.9).

If $\lambda < 1$, then all these off-policy algorithms involve bootstrapping and the deadly triad applies (Section 11.3), meaning that they can be guaranteed stable only for the tabular case, for state aggregation, and for other limited forms of function approximation. For linear and more-general forms of function approximation the parameter vector may diverge to infinity as in the examples in Chapter 11. As we discussed there, the challenge of off-policy learning has two parts. Off-policy eligibility traces deal effectively with the first part of the challenge, correcting for the expected value of the targets, but not at all with the second part of the challenge, having to do with the distribution of updates. Algorithmic strategies for meeting the second part of the challenge of off-policy learning with eligibility traces are summarized in Section 12.11.

Exercise 12.13 What are the dutch-trace and replacing-trace versions of off-policy eligibility traces for state-value and action-value methods? □

12.10 Watkins's Q(λ) to Tree-Backup(λ)

Several methods have been proposed over the years to extend Q-learning to eligibility traces. The original is *Watkins's Q(λ)*, which decays its eligibility traces in the usual way as long as a greedy action was taken, then cuts the traces to zero after the first non-greedy action. The backup diagram for Watkins's Q(λ) is shown in Figure 12.12. In Chapter 6, we unified Q-learning and Expected Sarsa in the off-policy version of the latter, which includes Q-learning as a special case, and generalizes it to arbitrary target policies, and in the previous section of this chapter we completed our treatment of Expected Sarsa by generalizing it to off-policy eligibility traces. In Chapter 7, however, we distinguished n-step Expected Sarsa from n-step Tree Backup, where the latter retained the property of not using importance sampling. It remains then to present the eligibility trace version of Tree Backup, which we will call *Tree-Backup(λ)*, or *TB(λ)* for short. This is arguably the true successor to Q-learning because it retains its appealing absence of importance sampling even though it can be applied to off-policy data.

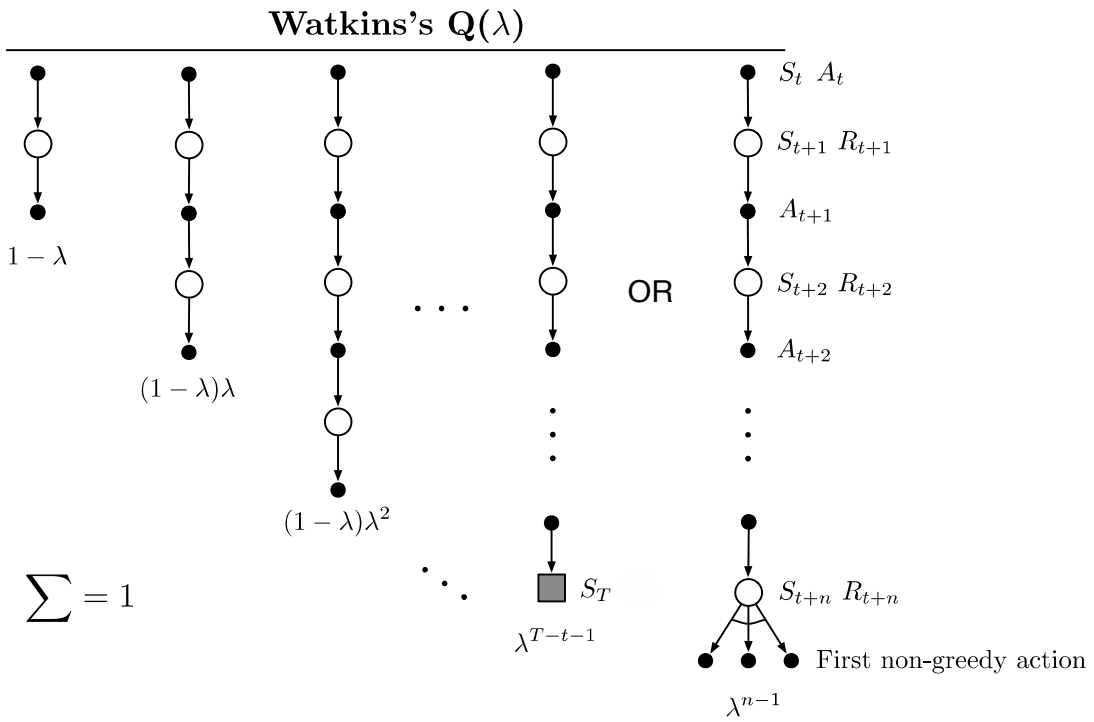


Figure 12.12: The backup diagram for Watkins's Q(λ). The series of component updates ends either with the end of the episode or with the first nongreedy action, whichever comes first.

The concept of TB(λ) is straightforward. As shown in its backup diagram in Figure 12.13, the tree-backup updates of each length (from Section 7.5) are weighted in the usual way dependent on the bootstrapping parameter λ. To get the detailed equations, with the right indices on the general bootstrapping and discounting parameters, it is best to start with a recursive form (12.20) for the λ-return using action values, and then expand the bootstrapping case of the target after the model of (7.16):

$$G_t^{\lambda a} \doteq R_{t+1}+\gamma_{t+1}\Big((1-\lambda_{t+1})\bar{V}_t(S_{t+1})+\lambda_{t+1}\Big[\sum_{a\neq A_{t+1}}\pi(a|S_{t+1})\hat{q}(S_{t+1},a,\mathbf{w}_t)+\pi(A_{t+1}|S_{t+1})G_{t+1}^{\lambda a}\Big]\Big)$$

$$= R_{t+1}+\gamma_{t+1}\Big(\bar{V}_t(S_{t+1})+\lambda_{t+1}\pi(A_{t+1}|S_{t+1})\Big(G_{t+1}^{\lambda a}-\hat{q}(S_{t+1},A_{t+1},\mathbf{w}_t)\Big)\Big)$$

As per the usual pattern, it can also be written approximately (ignoring changes in the approximate value function) as a sum of TD errors,

$$G_t^{\lambda a} \approx \hat{q}(S_t,A_t,\mathbf{w}_t)+\sum_{k=t}^{\infty}\delta_k^a\prod_{i=t+1}^{k}\gamma_i\lambda_i\pi(A_i|S_i),$$

using the expectation form of the action-based TD error (12.28).

Following the same steps as in the previous section, we arrive at a special eligibility trace update involving the target-policy probabilities of the selected actions,

$$\mathbf{z}_t \doteq \gamma_t\lambda_t\pi(A_t|S_t)\mathbf{z}_{t-1}+\nabla\hat{q}(S_t,A_t,\mathbf{w}_t).$$

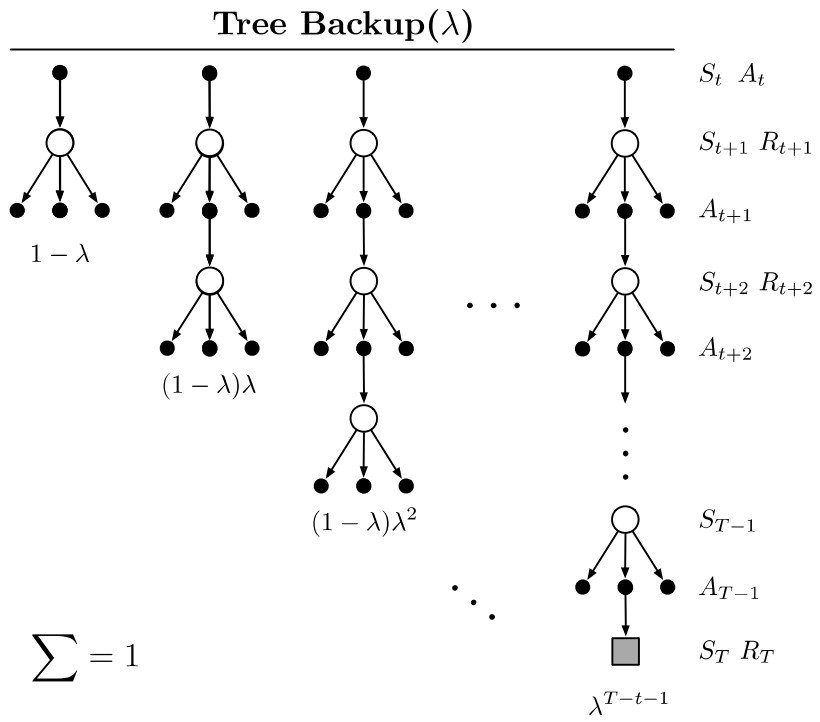


Figure 12.13: The backup diagram for the λ version of the Tree Backup algorithm.

This, together with the usual parameter-update rule (12.7), defines the TB(λ) algorithm. Like all semi-gradient algorithms, TB(λ) is not guaranteed to be stable when used with off-policy data and with a powerful function approximator. To obtain those assurances, TB(λ) would have to be combined with one of the methods presented in the next section.

**Exercise 12.14* How might Double Expected Sarsa be extended to eligibility traces? □

12.11 Stable Off-policy Methods with Traces

Several methods using eligibility traces have been proposed that achieve guarantees of stability under off-policy training, and here we present four of the most important using this book's standard notation, including general bootstrapping and discounting functions. All are based on either the Gradient-TD or the Emphatic-TD ideas presented in Sections 11.7 and 11.8. All the algorithms assume linear function approximation, though extensions to nonlinear function approximation can also be found in the literature.

GTD(λ) is the eligibility-trace algorithm analogous to TDC, the better of the two state-value Gradient-TD prediction algorithms discussed in Section 11.7. Its goal is to learn a parameter $\mathbf{w}_t$ such that $\hat{v}(s,\mathbf{w}) \doteq \mathbf{w}_t^\top \mathbf{x}(s) \approx v_\pi(s)$, even from data that is due to following another policy b. Its update is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t^s \mathbf{z}_t - \alpha\gamma_{t+1}(1-\lambda_{t+1})\left(\mathbf{z}_t^\top \mathbf{v}_t\right)\mathbf{x}_{t+1},$$

with δ_t^s, $\mathbf{z}_t$, and ρ_t defined in the usual ways for state values (12.23) (12.25) (11.1), and

$$\mathbf{v}_{t+1} \doteq \mathbf{v}_t + \beta\delta_t^s \mathbf{z}_t - \beta\left(\mathbf{v}_t^\top \mathbf{x}_t\right)\mathbf{x}_t, \tag{12.30}$$

where, as in Section 11.7, $\mathbf{v} \in \mathbb{R}^d$ is a vector of the same dimension as $\mathbf{w}$, initialized to $\mathbf{v}_0 = \mathbf{0}$, and $\beta > 0$ is a second step-size parameter.

GQ(λ) is the Gradient-TD algorithm for action values with eligibility traces. Its goal is to learn a parameter $\mathbf{w}_t$ such that $\hat{q}(s, a, \mathbf{w}_t) \doteq \mathbf{w}_t^\top \mathbf{x}(s, a) \approx q_\pi(s, a)$ from off-policy data. If the target policy is ε-greedy, or otherwise biased toward the greedy policy for $\hat{q}$, then GQ(λ) can be used as a control algorithm. Its update is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t^a \mathbf{z}_t - \alpha\gamma_{t+1}(1-\lambda_{t+1})\left(\mathbf{z}_t^\top \mathbf{v}_t\right)\bar{\mathbf{x}}_{t+1},$$

where $\bar{\mathbf{x}}_t$ is the average feature vector for S_t under the target policy,

$$\bar{\mathbf{x}}_t \doteq \sum_a \pi(a|S_t)\mathbf{x}(S_t, a),$$

δ_t^a is the expectation form of the TD error, which can be written

$$\delta_t^a \doteq R_{t+1} + \gamma_{t+1}\mathbf{w}_t^\top \bar{\mathbf{x}}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t,$$

$\mathbf{z}_t$ is defined in the usual way for action values (12.29), and the rest is as in GTD(λ), including the update for $\mathbf{v}_t$ (12.30).

HTD(λ) is a hybrid state-value algorithm combining aspects of GTD(λ) and TD(λ). Its most appealing feature is that it is a strict generalization of TD(λ) to off-policy learning, meaning that if the behavior policy happens to be the same as the target policy, then HTD(λ) becomes the same as TD(λ), which is not true for GTD(λ). This is appealing because TD(λ) is often faster than GTD(λ) when both algorithms converge, and TD(λ) requires setting only a single step size. HTD(λ) is defined by

$$\begin{aligned}
\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha\delta_t^s \mathbf{z}_t + \alpha\left((\mathbf{z}_t - \mathbf{z}_t^b)^\top \mathbf{v}_t\right)(\mathbf{x}_t - \gamma_{t+1}\mathbf{x}_{t+1}),\\
\mathbf{v}_{t+1} &\doteq \mathbf{v}_t + \beta\delta_t^s \mathbf{z}_t - \beta\left(\mathbf{z}_t^{b\top} \mathbf{v}_t\right)(\mathbf{x}_t - \gamma_{t+1}\mathbf{x}_{t+1}), \quad \text{with } \mathbf{v}_0 \doteq \mathbf{0},\\
\mathbf{z}_t &\doteq \rho_t\big(\gamma_t\lambda_t \mathbf{z}_{t-1} + \mathbf{x}_t\big), \quad \text{with } \mathbf{z}_{-1} \doteq \mathbf{0},\\
\mathbf{z}_t^b &\doteq \gamma_t\lambda_t \mathbf{z}_{t-1}^b + \mathbf{x}_t, \quad \text{with } \mathbf{z}_{-1}^b \doteq \mathbf{0},
\end{aligned}$$

where $\beta > 0$ again is a second step-size parameter. In addition to the second set of weights, $\mathbf{v}_t$, HTD(λ) also has a second set of eligibility traces, $\mathbf{z}_t^b$. These are conventional accumulating eligibility traces for the behavior policy and become equal to $\mathbf{z}_t$ if all the ρ_t are 1, which causes the last term in the $\mathbf{w}_t$ update to be zero and the overall update to reduce to TD(λ).

Emphatic TD(λ) is the extension of the one-step Emphatic-TD algorithm (Sections 9.11 and 11.8) to eligibility traces. The resultant algorithm retains strong off-policy convergence guarantees while enabling any degree of bootstrapping, albeit at the cost of

high variance and potentially slow convergence. Emphatic TD(λ) is defined by

$$\begin{aligned}
\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha\delta_t\mathbf{z}_t \\
\delta_t &\doteq R_{t+1} + \gamma_{t+1}\mathbf{w}_t^\top\mathbf{x}_{t+1} - \mathbf{w}_t^\top\mathbf{x}_t \\
\mathbf{z}_t &\doteq \rho_t\big(\gamma_t\lambda_t\mathbf{z}_{t-1} + M_t\mathbf{x}_t\big), \qquad \text{with } \mathbf{z}_{-1} \doteq \mathbf{0}, \\
M_t &\doteq \lambda_t I_t + (1-\lambda_t)F_t \\
F_t &\doteq \rho_{t-1}\gamma_t F_{t-1} + I_t, \qquad \text{with } F_0 \doteq i(S_0),
\end{aligned}$$

where $M_t \geq 0$ is the general form of *emphasis*, $F_t \geq 0$ is termed the *followon trace*, and $I_t \geq 0$ is the *interest*, as described in Section 11.8. Note that M_t, like δ_t, is not really an additional memory variable. It can be removed from the algorithm by substituting its definition into the eligibility-trace equation. Pseudocode and software for the true online version of Emphatic-TD(λ) are available on the web (Sutton, 2015b).

In the on-policy case ($\rho_t = 1$, for all t), Emphatic-TD(λ) is similar to conventional TD(λ), but still significantly different. In fact, whereas Emphatic-TD(λ) is guaranteed to converge for all state-dependent λ functions, TD(λ) is not. TD(λ) is guaranteed convergent only for all constant λ. See Yu's counterexample (Ghiassian, Rafiee, and Sutton, 2016).

12.12 Implementation Issues

It might at first appear that tabular methods using eligibility traces are much more complex than one-step methods. A naive implementation would require every state (or state–action pair) to update both its value estimate and its eligibility trace on every time step. This would not be a problem for implementations on single-instruction, multiple-data, parallel computers or in plausible artificial neural network (ANN) implementations, but it is a problem for implementations on conventional serial computers. Fortunately, for typical values of λ and γ the eligibility traces of almost all states are almost always nearly zero; only those states that have recently been visited will have traces significantly greater than zero and only these few states need to be updated to closely approximate these algorithms.

In practice, then, implementations on conventional computers may keep track of and update only the few traces that are significantly greater than zero. Using this trick, the computational expense of using traces in tabular methods is typically just a few times that of a one-step method. The exact multiple of course depends on λ and γ and on the expense of the other computations. Note that the tabular case is in some sense the worst case for the computational complexity of eligibility traces. When function approximation is used, the computational advantages of not using traces generally decrease. For example, if ANNs and backpropagation are used, then eligibility traces generally cause only a doubling of the required memory and computation per step. Truncated λ-return methods (Section 12.3) can be computationally efficient on conventional computers though they always require some additional memory.

12.13 Conclusions

Eligibility traces in conjunction with TD errors provide an efficient, incremental way of shifting and choosing between Monte Carlo and TD methods. The n-step methods of Chapter 7 also enabled this, but eligibility trace methods are more general, often faster to learn, and offer different computational complexity tradeoffs. This chapter has offered an introduction to the elegant, emerging theoretical understanding of eligibility traces for on- and off-policy learning and for variable bootstrapping and discounting. One aspect of this elegant theory is true online methods, which exactly reproduce the behavior of expensive ideal methods while retaining the computational congeniality of conventional TD methods. Another aspect is the possibility of derivations that automatically convert from intuitive forward-view methods to more efficient incremental backward-view algorithms. We illustrated this general idea in a derivation that started with a classical, expensive Monte Carlo algorithm and ended with a cheap incremental non-TD implementation using the same novel eligibility trace used in true online TD methods.

As we mentioned in Chapter 5, Monte Carlo methods may have advantages in non-Markov tasks because they do not bootstrap. Because eligibility traces make TD methods more like Monte Carlo methods, they also can have advantages in these cases. If one wants to use TD methods because of their other advantages, but the task is at least partially non-Markov, then the use of an eligibility trace method is indicated. Eligibility traces are the first line of defense against both long-delayed rewards and non-Markov tasks.

By adjusting λ, we can place eligibility trace methods anywhere along a continuum from Monte Carlo to one-step TD methods. Where shall we place them? We do not yet have a good theoretical answer to this question, but a clear empirical answer appears to be emerging. On tasks with many steps per episode, or many steps within the half-life of discounting, it appears significantly better to use eligibility traces than not to (e.g., see Figure 12.14). On the other hand, if the traces are so long as to produce a pure Monte Carlo method, or nearly so, then performance degrades sharply. An intermediate mixture appears to be the best choice. Eligibility traces should be used to bring us toward Monte Carlo methods, but not all the way there. In the future it may be possible to more finely vary the trade-off between TD and Monte Carlo methods by using variable λ, but at present it is not clear how this can be done reliably and usefully.

Methods using eligibility traces require more computation than one-step methods, but in return they offer significantly faster learning, particularly when rewards are delayed by many steps. Thus it often makes sense to use eligibility traces when data are scarce and cannot be repeatedly processed, as is often the case in online applications. On the other hand, in off-line applications in which data can be generated cheaply, perhaps from an inexpensive simulation, then it often does not pay to use eligibility traces. In these cases the objective is not to get more out of a limited amount of data, but simply to process as much data as possible as quickly as possible. In these cases the speedup per datum due to traces is typically not worth their computational cost, and one-step methods are favored.

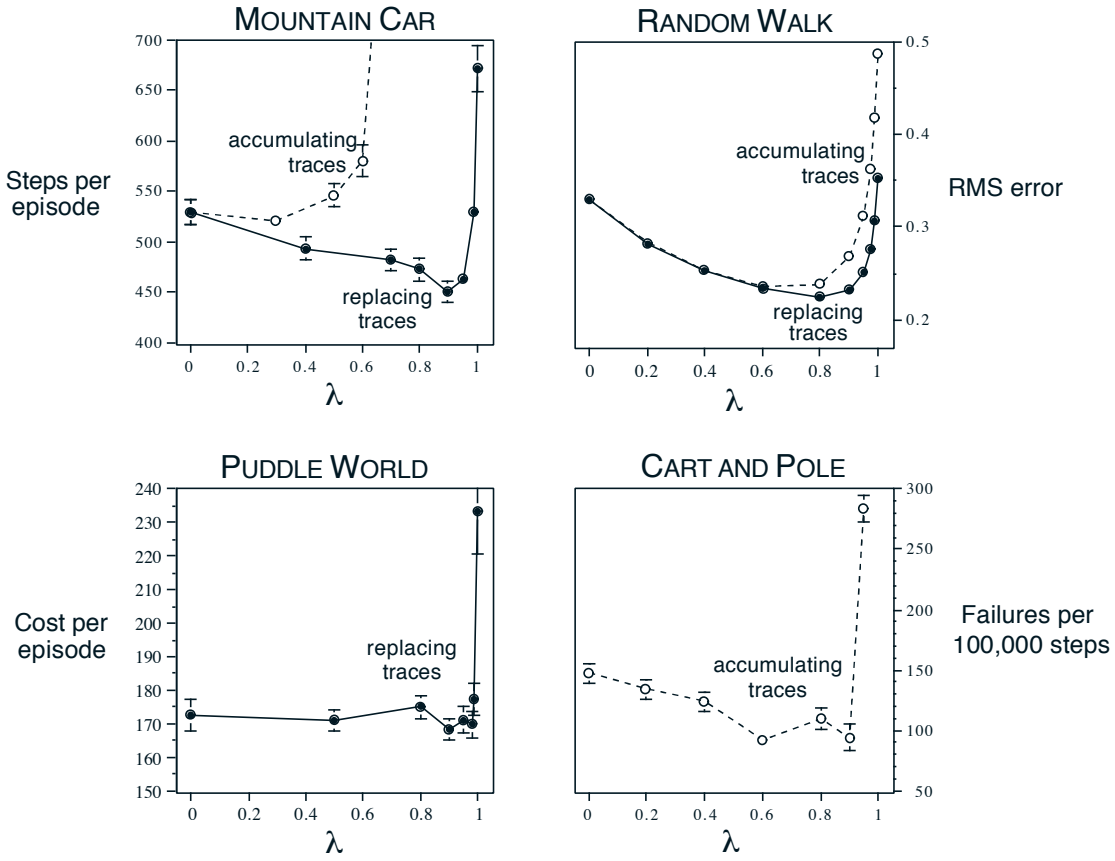


Figure 12.14: The effect of λ on reinforcement learning performance in four different test problems. In all cases, performance is generally best (a *lower* number in the graph) at an intermediate value of λ. The two left panels are applications to simple continuous-state control tasks using the Sarsa(λ) algorithm and tile coding, with either replacing or accumulating traces (Sutton, 1996). The upper-right panel is for policy evaluation on a random walk task using TD(λ) (Singh and Sutton, 1996). The lower right panel is unpublished data for the pole-balancing task (Example 3.4) from an earlier study (Sutton, 1984).

Bibliographical and Historical Remarks

Eligibility traces came into reinforcement learning via the fecund ideas of Klopf (1972). Our use of eligibility traces is based on Klopf's work (Sutton, 1978a, 1978b, 1978c; Barto and Sutton, 1981a, 1981b; Sutton and Barto, 1981a; Barto, Sutton, and Anderson, 1983; Sutton, 1984). We may have been the first to use the term "eligibility trace" (Sutton and Barto, 1981a). The idea that stimuli produce after effects in the nervous system that are important for learning is very old (see Chapter 14). Some of the earliest uses of eligibility traces were in the actor–critic methods discussed in Chapter 13 (Barto, Sutton, and Anderson, 1983; Sutton, 1984).

12.1 Compound updates were called "complex backups" in the first edition of this book.

The λ-return and its error-reduction properties were introduced by Watkins (1989) and further developed by Jaakkola, Jordan, and Singh (1994). The random walk results in this and subsequent sections are new to this text, as are the terms "forward view" and "backward view." The notion of a λ-return algorithm was introduced in the first edition of this text. The more refined treatment presented here was developed in conjunction with Harm van Seijen (e.g., van Seijen and Sutton, 2014).

12.2 TD(λ) with accumulating traces was introduced by Sutton (1988, 1984). Convergence in the mean was proved by Dayan (1992), and with probability 1 by many researchers, including Peng (1993), Dayan and Sejnowski (1994), Tsitsiklis (1994), and Gurvits, Lin, and Hanson (1994). The bound on the error of the asymptotic λ-dependent solution of linear TD(λ) is due to Tsitsiklis and Van Roy (1997).

12.3 Truncated TD methods were developed by Cichosz (1995) and van Seijen (2016).

12.4 The idea of redoing updates was extensively developed by van Seijen, originally under the name "best-match learning" (van Seijen, 2011; van Seijen, Whiteson, van Hasselt, and Weiring, 2011).

12.5 True online TD(λ) is primarily due to Harm van Seijen (van Seijen and Sutton, 2014; van Seijen et al., 2016) though some of its key ideas were discovered independently by Hado van Hasselt (personal communication). The name "dutch traces" is in recognition of the contributions of both scientists. Replacing traces are due to Singh and Sutton (1996).

12.6 The material in this section is from van Hasselt and Sutton (2015).

12.7 Sarsa(λ) with accumulating traces was first explored as a control method by Rummery and Niranjan (1994; Rummery, 1995). True Online Sarsa(λ) was

introduced by van Seijen and Sutton (2014). The algorithm on page 307 was adapted from van Seijen et al. (2016). The Mountain Car results were made for this text, except for Figure 12.11 which is adapted from van Seijen and Sutton (2014).

12.8 Perhaps the first published discussion of variable λ was by Watkins (1989), who pointed out that the cutting off of the update sequence (Figure 12.12) in his Q(λ) when a nongreedy action was selected could be implemented by temporarily setting λ to 0.

Variable λ was introduced in the first edition of this text. The roots of variable γ are in the work on options (Sutton, Precup, and Singh, 1999) and its precursors (Sutton, 1995a), becoming explicit in the GQ(λ) paper (Maei and Sutton, 2010), which also introduced some of these recursive forms for the λ-returns.

A different notion of variable λ has been developed by Yu (2012).

12.9 Off-policy eligibility traces were introduced by Precup et al. (2000, 2001), then further developed by Bertsekas and Yu (2009), Maei (2011; Maei and Sutton, 2010), Yu (2012), and by Sutton, Mahmood, Precup, and van Hasselt (2014). The last reference in particular gives a powerful forward view for off-policy TD methods with general state-dependent λ and γ. The presentation here seems to be new.

This section ends with an elegant Expected Sarsa(λ) algorithm. Although it is a natural algorithm, to our knowledge it has not previously been described or tested in the literature.

12.10 Watkins's Q(λ) is due to Watkins (1989). The tabular, episodic, off-line version has been proven convergent by Munos, Stepleton, Harutyunyan, and Bellemare (2016). Alternative Q(λ) algorithms were proposed by Peng and Williams (1994, 1996) and by Sutton, Mahmood, Precup, and van Hasselt (2014). Tree Backup(λ) is due to Precup, Sutton, and Singh (2000).

12.11 GTD(λ) is due to Maei (2011). GQ(λ) is due to Maei and Sutton (2010). HTD(λ) is due to White and White (2016) based on the one-step HTD algorithm introduced by Hackman (2012). The latest developments in the theory of Gradient-TD methods are by Yu (2017). Emphatic TD(λ) was introduced by Sutton, Mahmood, and White (2016), who proved its stability. Yu (2015, 2016) proved its convergence, and the algorithm was developed further by Hallak et al. (2015, 2016).

Chapter 13

Policy Gradient Methods

In this chapter we consider something new. So far in this book almost all the methods have been *action-value methods*; they learned the values of actions and then selected actions based on their estimated action values[1]; their policies would not even exist without the action-value estimates. In this chapter we consider methods that instead learn a *parameterized policy* that can select actions without consulting a value function. A value function may still be used to *learn* the policy parameter, but is not required for action selection. We use the notation $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ for the policy's parameter vector. Thus we write $\pi(a|s,\boldsymbol{\theta}) = \Pr\{A_t = a \mid S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\}$ for the probability that action a is taken at time t given that the environment is in state s at time t with parameter $\boldsymbol{\theta}$. If a method uses a learned value function as well, then the value function's weight vector is denoted $\mathbf{w} \in \mathbb{R}^d$ as usual, as in $\hat{v}(s,\mathbf{w})$.

In this chapter we consider methods for learning the policy parameter based on the gradient of some scalar performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameter. These methods seek to *maximize* performance, so their updates approximate gradient *ascent* in J:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)}, \tag{13.1}$$

where $\widehat{\nabla J(\boldsymbol{\theta}_t)} \in \mathbb{R}^{d'}$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument $\boldsymbol{\theta}_t$. All methods that follow this general schema we call *policy gradient methods*, whether or not they also learn an approximate value function. Methods that learn approximations to both policy and value functions are often called *actor–critic methods*, where 'actor' is a reference to the learned policy, and 'critic' refers to the learned value function, usually a state-value function. First we treat the episodic case, in which performance is defined as the value of the start state under the parameterized policy, before going on to consider the continuing case, in

[1] The lone exception is the gradient bandit algorithms of Section 2.8. In fact, that section goes through many of the same steps, in the single-state bandit case, as we go through here for full MDPs. Reviewing that section would be good preparation for fully understanding this chapter.

which performance is defined as the average reward rate, as in Section 10.3. In the end, we are able to express the algorithms for both cases in very similar terms.

13.1 Policy Approximation and its Advantages

In policy gradient methods, the policy can be parameterized in any way, as long as $\pi(a|s,\boldsymbol{\theta})$ is differentiable with respect to its parameters, that is, as long as $\nabla\pi(a|s,\boldsymbol{\theta})$ (the column vector of partial derivatives of $\pi(a|s,\boldsymbol{\theta})$ with respect to the components of $\boldsymbol{\theta}$) exists and is finite for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, and $\boldsymbol{\theta} \in \mathbb{R}^{d'}$. In practice, to ensure exploration we generally require that the policy never becomes deterministic (i.e., that $\pi(a|s,\boldsymbol{\theta}) \in (0,1)$, for all $s,a,\boldsymbol{\theta}$). In this section we introduce the most common parameterization for discrete action spaces and point out the advantages it offers over action-value methods. Policy-based methods also offer useful ways of dealing with continuous action spaces, as we describe later in Section 13.7.

If the action space is discrete and not too large, then a natural and common kind of parameterization is to form parameterized numerical preferences $h(s,a,\boldsymbol{\theta}) \in \mathbb{R}$ for each state–action pair. The actions with the highest preferences in each state are given the highest probabilities of being selected, for example, according to an exponential soft-max distribution:

$$\pi(a|s,\boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}}, \tag{13.2}$$

where $e \approx 2.71828$ is the base of the natural logarithm. Note that the denominator here is just what is required so that the action probabilities in each state sum to one. We call this kind of policy parameterization *soft-max in action preferences*.

The action preferences themselves can be parameterized arbitrarily. For example, they might be computed by a deep artificial neural network (ANN), where $\boldsymbol{\theta}$ is the vector of all the connection weights of the network (as in the AlphaGo system described in Section 16.6). Or the preferences could simply be linear in features,

$$h(s,a,\boldsymbol{\theta}) = \boldsymbol{\theta}^\top \mathbf{x}(s,a), \tag{13.3}$$

using feature vectors $\mathbf{x}(s,a) \in \mathbb{R}^{d'}$ constructed by any of the methods described in Section 9.5.

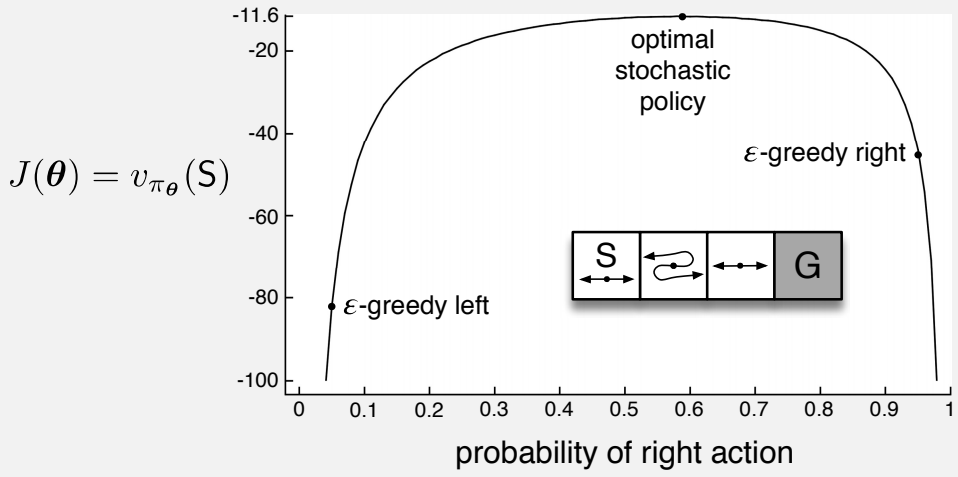
One advantage of parameterizing policies according to the soft-max in action preferences is that the approximate policy can approach a deterministic policy, whereas with ε-greedy action selection over action values there is always an ε probability of selecting a random action. Of course, one could select according to a soft-max distribution based on action values, but this alone would not allow the policy to approach a deterministic policy. Instead, the action-value estimates would converge to their corresponding true values, which would differ by a finite amount, translating to specific probabilities other than 0 and 1. If the soft-max distribution included a temperature parameter, then the temperature could be reduced over time to approach determinism, but in practice it would be difficult to choose the reduction schedule, or even the initial temperature, without more prior knowledge of the true action values than we would like to assume. Action preferences

are different because they do not approach specific values; instead they are driven to produce the optimal stochastic policy. If the optimal policy is deterministic, then the preferences of the optimal actions will be driven infinitely higher than all suboptimal actions (if permitted by the parameterization).

A second advantage of parameterizing policies according to the soft-max in action preferences is that it enables the selection of actions with arbitrary probabilities. In problems with significant function approximation, the best approximate policy may be stochastic. For example, in card games with imperfect information the optimal play is often to do two different things with specific probabilities, such as when bluffing in Poker. Action-value methods have no natural way of finding stochastic optimal policies, whereas policy approximating methods can, as shown in Example 13.1.

Example 13.1 Short corridor with switched actions

Consider the small corridor gridworld shown inset in the graph below. The reward is -1 per step, as usual. In each of the three nonterminal states there are only two actions, right and left. These actions have their usual consequences in the first and third states (left causes no movement in the first state), but in the second state they are reversed, so that right moves to the left and left moves to the right. The problem is difficult because all the states appear identical under the function approximation. In particular, we define $\mathbf{x}(s, \text{right}) = [1, 0]^\top$ and $\mathbf{x}(s, \text{left}) = [0, 1]^\top$, for all s. An action-value method with ε-greedy action selection is forced to choose between just two policies: choosing right with high probability $1 - \varepsilon/2$ on all steps or choosing left with the same high probability on all time steps. If $\varepsilon = 0.1$, then these two policies achieve a value (at the start state) of less than -44 and -82, respectively, as shown in the graph. A method can do significantly better if it can learn a specific probability with which to select right. The best probability is about 0.59, which achieves a value of about -11.6.



Perhaps the simplest advantage that policy parameterization may have over action-value parameterization is that the policy may be a simpler function to approximate. Problems vary in the complexity of their policies and action-value functions. For some, the action-value function is simpler and thus easier to approximate. For others, the policy is simpler. In the latter case a policy-based method will typically learn faster and yield a superior asymptotic policy (as in Tetris; see Şimşek, Algórta, and Kothiyal, 2016).

Finally, we note that the choice of policy parameterization is sometimes a good way of injecting prior knowledge about the desired form of the policy into the reinforcement learning system. This is often the most important reason for using a policy-based learning method.

Exercise 13.1 Use your knowledge of the gridworld and its dynamics to determine an *exact* symbolic expression for the optimal probability of selecting the right action in Example 13.1. □

13.2 The Policy Gradient Theorem

In addition to the practical advantages of policy parameterization over ε-greedy action selection, there is also an important theoretical advantage. With continuous policy parameterization the action probabilities change smoothly as a function of the learned parameter, whereas in ε-greedy selection the action probabilities may change dramatically for an arbitrarily small change in the estimated action values, if that change results in a different action having the maximal value. Largely because of this, stronger convergence guarantees are available for policy-gradient methods than for action-value methods. In particular, it is the continuity of the policy dependence on the parameters that enables policy-gradient methods to approximate gradient ascent (13.1).

The episodic and continuing cases define the performance measure, $J(\boldsymbol{\theta})$, differently and thus have to be treated separately to some extent. Nevertheless, we will try to present both cases uniformly, and we develop a notation so that the major theoretical results can be described with a single set of equations.

In this section we treat the episodic case, for which we define the performance measure as the value of the start state of the episode. We can simplify the notation without losing any meaningful generality by assuming that every episode starts in some particular (non-random) state s_0. Then, in the episodic case we define performance as

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0), \tag{13.4}$$

where $v_{\pi_{\boldsymbol{\theta}}}$ is the true value function for $\pi_{\boldsymbol{\theta}}$, the policy determined by $\boldsymbol{\theta}$. From here on in our discussion we will assume no discounting ($\gamma = 1$) for the episodic case, although for completeness we do include the possibility of discounting in the boxed algorithms.

With function approximation it may seem challenging to change the policy parameter in a way that ensures improvement. The problem is that performance depends on both the action selections and the distribution of states in which those selections are made, and that both of these are affected by the policy parameter. Given a state, the effect of the policy parameter on the actions, and thus on reward, can be computed in a relatively straightforward way from knowledge of the parameterization. But the effect of the policy

Proof of the Policy Gradient Theorem (episodic case)

With just elementary calculus and re-arranging of terms, we can prove the policy gradient theorem from first principles. To keep the notation simple, we leave it implicit in all cases that π is a function of $\boldsymbol{\theta}$, and all gradients are also implicitly with respect to $\boldsymbol{\theta}$. First note that the gradient of the state-value function can be written in terms of the action-value function as

$$
\begin{aligned}
\nabla v_\pi(s) &= \nabla \left[\sum_a \pi(a|s) q_\pi(s,a) \right], \quad \text{for all } s \in \mathcal{S} && \text{(Exercise 3.18)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla q_\pi(s,a) \Big] && \text{(product rule of calculus)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla \sum_{s',r} p(s',r\,|\,s,a) \big(r + v_\pi(s')\big) \Big] \\
& && \text{(Exercise 3.19 and Equation 3.2)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'\,|\,s,a) \nabla v_\pi(s') \Big] && \text{(Eq. 3.4)} \\
&= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'\,|\,s,a) && \text{(unrolling)} \\
&\qquad \sum_{a'} \big[\nabla \pi(a'|s') q_\pi(s',a') + \pi(a'|s') \sum_{s''} p(s''\,|\,s',a') \nabla v_\pi(s'') \big] \Big] \\
&= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \Pr(s \to x, k, \pi) \sum_a \nabla \pi(a|x) q_\pi(x,a),
\end{aligned}
$$

after repeated unrolling, where $\Pr(s \to x, k, \pi)$ is the probability of transitioning from state s to state x in k steps under policy π. It is then immediate that

$$
\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \nabla v_\pi(s_0) \\
&= \sum_s \left(\sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi) \right) \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&= \sum_s \eta(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && \text{(box page 199)} \\
&= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && \text{(Eq. 9.3)} \\
&\propto \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) && \text{(Q.E.D.)}
\end{aligned}
$$

on the state distribution is a function of the environment and is typically unknown. How can we estimate the performance gradient with respect to the policy parameter when the gradient depends on the unknown effect of policy changes on the state distribution?

Fortunately, there is an excellent theoretical answer to this challenge in the form of the *policy gradient theorem*, which provides an analytic expression for the gradient of performance with respect to the policy parameter (which is what we need to approximate for gradient ascent (13.1)) that does *not* involve the derivative of the state distribution. The policy gradient theorem for the episodic case establishes that

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}), \tag{13.5}$$

where the gradients are column vectors of partial derivatives with respect to the components of $\boldsymbol{\theta}$, and π denotes the policy corresponding to parameter vector $\boldsymbol{\theta}$. The symbol $\propto$ here means "proportional to". In the episodic case, the constant of proportionality is the average length of an episode, and in the continuing case it is 1, so that the relationship is actually an equality. The distribution μ here (as in Chapters 9 and 10) is the on-policy distribution under π (see page 199). The policy gradient theorem is proved for the episodic case in the box on the previous page.

13.3 REINFORCE: Monte Carlo Policy Gradient

We are now ready to derive our first policy-gradient learning algorithm. Recall our overall strategy of stochastic gradient ascent (13.1), which requires a way to obtain samples such that the expectation of the sample gradient is proportional to the actual gradient of the performance measure as a function of the parameter. The sample gradients need only be proportional to the gradient because any constant of proportionality can be absorbed into the step size α, which is otherwise arbitrary. The policy gradient theorem gives an exact expression proportional to the gradient; all that is needed is some way of sampling whose expectation equals or approximates this expression. Notice that the right-hand side of the policy gradient theorem is a sum over states weighted by how often the states occur under the target policy π; if π is followed, then states will be encountered in these proportions. Thus

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \\ &= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\boldsymbol{\theta}) \right].\end{aligned} \tag{13.6}$$

We could stop here and instantiate our stochastic gradient-ascent algorithm (13.1) as

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \sum_a \hat{q}(S_t,a,\mathbf{w}) \nabla \pi(a|S_t,\boldsymbol{\theta}), \tag{13.7}$$

where $\hat{q}$ is some learned approximation to q_π. This algorithm, which has been called an *all-actions* method because its update involves all of the actions, is promising and

deserving of further study, but our current interest is the classical REINFORCE algorithm (Willams, 1992) whose update at time t involves just A_t, the one action actually taken at time t.

We continue our derivation of REINFORCE by introducing A_t in the same way as we introduced S_t in (13.6)—by replacing a sum over the random variable's possible values by an expectation under π, and then sampling the expectation. Equation (13.6) involves an appropriate sum over actions, but each term is not weighted by $\pi(a|S_t,\boldsymbol{\theta})$ as is needed for an expectation under π. So we introduce such a weighting, without changing the equality, by multiplying and then dividing the summed terms by $\pi(a|S_t,\boldsymbol{\theta})$. Continuing from (13.6), we have

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \mathbb{E}_\pi\left[\sum_a \pi(a|S_t,\boldsymbol{\theta}) q_\pi(S_t,a)\frac{\nabla\pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})}\right] \\
&= \mathbb{E}_\pi\left[q_\pi(S_t,A_t)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}\right] && \text{(replacing } a \text{ by the sample } A_t\sim\pi) \\
&= \mathbb{E}_\pi\left[G_t\,\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}\right], && \text{(because } \mathbb{E}_\pi[G_t|S_t,A_t]=q_\pi(S_t,A_t))
\end{aligned}$$

where G_t is the return as usual. The final expression in brackets is exactly what is needed, a quantity that can be sampled on each time step whose expectation is proportional to the gradient. Using this sample to instantiate our generic stochastic gradient ascent algorithm (13.1) yields the REINFORCE update:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}. \tag{13.8}$$

This update has an intuitive appeal. Each increment is proportional to the product of a return G_t and a vector, the gradient of the probability of taking the action actually taken divided by the probability of taking that action. The vector is the direction in parameter space that most increases the probability of repeating the action A_t on future visits to state S_t. The update increases the parameter vector in this direction proportional to the return, and inversely proportional to the action probability. The former makes sense because it causes the parameter to move most in the directions that favor actions that yield the highest return. The latter makes sense because otherwise actions that are selected frequently are at an advantage (the updates will be more often in their direction) and might win out even if they do not yield the highest return.

Note that REINFORCE uses the complete return from time t, which includes all future rewards up until the end of the episode. In this sense REINFORCE is a Monte Carlo algorithm and is well defined only for the episodic case with all updates made in retrospect after the episode is completed (like the Monte Carlo algorithms in Chapter 5). This is shown explicitly in the boxed algorithm on the next page.

Notice that the update in the last line of pseudocode appears rather different from the REINFORCE update rule (13.8). One difference is that the pseudocode uses the compact expression $\nabla \ln \pi(A_t|S_t,\boldsymbol{\theta}_t)$ for the fractional vector $\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}$ in (13.8). That these two expressions for the vector are equivalent follows from the identity $\nabla \ln x = \frac{\nabla x}{x}$.

This vector has been given several names and notations in the literature; we will refer to it simply as the *eligibility vector*. Note that it is the only place that the policy parameterization appears in the algorithm.

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

The second difference between the pseudocode update and the REINFORCE update equation (13.8) is that the former includes a factor of γ^t. This is because, as mentioned earlier, in the text we are treating the non-discounted case ($\gamma = 1$) while in the boxed algorithms we are giving the algorithms for the general discounted case. All of the ideas go through in the discounted case with appropriate adjustments (including to the box on page 199) but involve additional complexity that distracts from the main ideas.

**Exercise 13.2* Generalize the box on page 199, the policy gradient theorem (13.5), the proof of the policy gradient theorem (page 325), and the steps leading to the REINFORCE update equation (13.8), so that (13.8) ends up with a factor of γ^t and thus aligns with the general algorithm given in the pseudocode. □

Figure 13.1 shows the performance of REINFORCE on the short-corridor gridworld from Example 13.1.

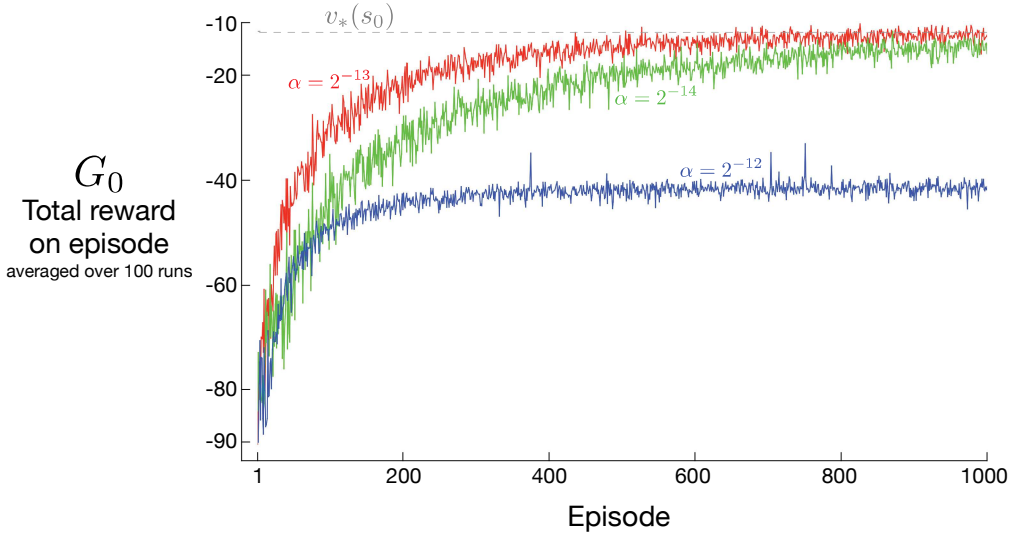


Figure 13.1: REINFORCE on the short-corridor gridworld (Example 13.1). With a good step size, the total reward per episode approaches the optimal value of the start state.

As a stochastic gradient method, REINFORCE has good theoretical convergence properties. By construction, the expected update over an episode is in the same direction as the performance gradient. This assures an improvement in expected performance for sufficiently small α, and convergence to a local optimum under standard stochastic approximation conditions for decreasing α. However, as a Monte Carlo method REINFORCE may be of high variance and thus produce slow learning.

Exercise 13.3 In Section 13.1 we considered policy parameterizations using the soft-max in action preferences (13.2) with linear action preferences (13.3). For this parameterization, prove that the eligibility vector is

$$\nabla \ln \pi(a|s,\boldsymbol{\theta}) = \mathbf{x}(s,a) - \sum_b \pi(b|s,\boldsymbol{\theta})\mathbf{x}(s,b), \tag{13.9}$$

using the definitions and elementary calculus. □

13.4 REINFORCE with Baseline

The policy gradient theorem (13.5) can be generalized to include a comparison of the action value to an arbitrary *baseline* $b(s)$:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a \Big(q_\pi(s,a) - b(s)\Big) \nabla \pi(a|s,\boldsymbol{\theta}). \tag{13.10}$$

The baseline can be any function, even a random variable, as long as it does not vary with a; the equation remains valid because the subtracted quantity is zero:

$$\sum_a b(s) \nabla \pi(a|s,\boldsymbol{\theta}) \;=\; b(s) \nabla \sum_a \pi(a|s,\boldsymbol{\theta}) \;=\; b(s) \nabla 1 \;=\; 0.$$

The policy gradient theorem with baseline (13.10) can be used to derive an update rule using similar steps as in the previous section. The update rule that we end up with is a new version of REINFORCE that includes a general baseline:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\Big(G_t - b(S_t)\Big) \frac{\nabla \pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}. \tag{13.11}$$

Because the baseline could be uniformly zero, this update is a strict generalization of REINFORCE. In general, the baseline leaves the expected value of the update unchanged, but it can have a large effect on its variance. For example, we saw in Section 2.8 that an analogous baseline can significantly reduce the variance (and thus speed the learning) of gradient bandit algorithms. In the bandit algorithms the baseline was just a number (the average of the rewards seen so far), but for MDPs the baseline should vary with state. In some states all actions have high values and we need a high baseline to differentiate the higher valued actions from the less highly valued ones; in other states all actions will have low values and a low baseline is appropriate.

One natural choice for the baseline is an estimate of the state value, $\hat{v}(S_t,\mathbf{w})$, where $\mathbf{w} \in \mathbb{R}^d$ is a weight vector learned by one of the methods presented in previous chapters.

Because REINFORCE is a Monte Carlo method for learning the policy parameter, $\boldsymbol{\theta}$, it seems natural to also use a Monte Carlo method to learn the state-value weights, $\mathbf{w}$. A complete pseudocode algorithm for REINFORCE with baseline using such a learned state-value function as the baseline is given in the box below.

REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

This algorithm has two step sizes, denoted $\alpha^{\boldsymbol{\theta}}$ and $\alpha^{\mathbf{w}}$ (where $\alpha^{\boldsymbol{\theta}}$ is the α in (13.11)). Choosing the step size for values (here $\alpha^{\mathbf{w}}$) is relatively easy; in the linear case we have rules of thumb for setting it, such as $\alpha^{\mathbf{w}} = 0.1/\mathbb{E}\big[\|\nabla \hat{v}(S_t,\mathbf{w})\|_{\mu}^2\big]$ (see Section 9.6). It is much less clear how to set the step size for the policy parameters, $\alpha^{\boldsymbol{\theta}}$, whose best value depends on the range of variation of the rewards and on the policy parameterization.

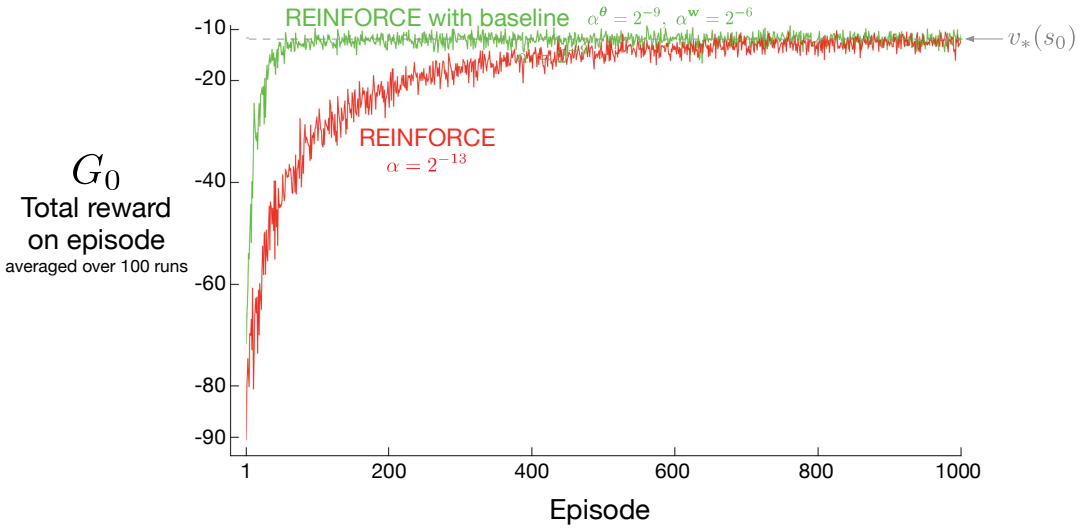


Figure 13.2: Adding a baseline to REINFORCE can make it learn much faster, as illustrated here on the short-corridor gridworld (Example 13.1). The step size used here for plain REINFORCE is that at which it performs best (to the nearest power of two; see Figure 13.1).

Figure 13.2 compares the behavior of REINFORCE with and without a baseline on the short-corridor gridword (Example 13.1). Here the approximate state-value function used in the baseline is $\hat{v}(s,\mathbf{w}) = w$. That is, $\mathbf{w}$ is a single component, w.

13.5 Actor–Critic Methods

In REINFORCE with baseline, the learned state-value function estimates the value of the *first* state of each state transition. This estimate sets a baseline for the subsequent return, but is made prior to the transition's action and thus cannot be used to assess that action. In actor–critic methods, on the other hand, the state-value function is applied also to the *second* state of the transition. The estimated value of the second state, when discounted and added to the reward, constitutes the one-step return, $G_{t:t+1}$, which is a useful estimate of the actual return and thus *is* a way of assessing the action. As we have seen in the TD learning of value functions throughout this book, the one-step return is often superior to the actual return in terms of its variance and computational congeniality, even though it introduces bias. We also know how we can flexibly modulate the extent of the bias with n-step returns and eligibility traces (Chapters 7 and 12). When the state-value function is used to assess actions in this way it is called a *critic*, and the overall policy-gradient method is termed an *actor–critic* method. Note that the bias in the gradient estimate is not due to bootstrapping as such; the actor would be biased even if the critic was learned by a Monte Carlo method.

First consider one-step actor–critic methods, the analog of the TD methods introduced in Chapter 6 such as TD(0), Sarsa(0), and Q-learning. The main appeal of one-step methods is that they are fully online and incremental, yet avoid the complexities of eligibility traces. They are a special case of the eligibility trace methods, but easier to understand. One-step actor–critic methods replace the full return of REINFORCE (13.11) with the one-step return (and use a learned state-value function as the baseline) as follows:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\Big(G_{t:t+1} - \hat{v}(S_t,\mathbf{w})\Big)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)} \tag{13.12}$$

$$= \boldsymbol{\theta}_t + \alpha\Big(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})\Big)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)} \tag{13.13}$$

$$= \boldsymbol{\theta}_t + \alpha\delta_t\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}. \tag{13.14}$$

The natural state-value-function learning method to pair with this is semi-gradient TD(0). Pseudocode for the complete algorithm is given in the box at the top of the next page. Note that it is now a fully online, incremental algorithm, with states, actions, and rewards processed as they occur and then never revisited.

One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\nabla\hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

The generalizations to the forward view of n-step methods and then to a λ-return algorithm are straightforward. The one-step return in (13.12) is merely replaced by $G_{t:t+n}$ or G_t^λ respectively. The backward view of the λ-return algorithm is also straightforward, using separate eligibility traces for the actor and critic, each after the patterns in Chapter 12. Pseudocode for the complete algorithm is given in the box below.

Actor–Critic with Eligibility Traces (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: trace-decay rates $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\lambda^{\mathbf{w}} \in [0,1]$; step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ (d'-component eligibility trace vector)
  $\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ (d-component eligibility trace vector)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{z}^{\mathbf{w}} \leftarrow \gamma\lambda^{\mathbf{w}}\mathbf{z}^{\mathbf{w}} + \nabla\hat{v}(S,\mathbf{w})$
    $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \gamma\lambda^{\boldsymbol{\theta}}\mathbf{z}^{\boldsymbol{\theta}} + I\nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\mathbf{z}^{\mathbf{w}}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\delta\mathbf{z}^{\boldsymbol{\theta}}$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

13.6 Policy Gradient for Continuing Problems

As discussed in Section 10.3, for continuing problems without episode boundaries we need to define performance in terms of the average rate of reward per time step:

$$\begin{aligned} J(\boldsymbol{\theta}) \doteq r(\pi) &\doteq \lim_{h\to\infty} \frac{1}{h} \sum_{t=1}^{h} \mathbb{E}[R_t \mid S_0, A_{0:t-1} \sim \pi] \\ &= \lim_{t\to\infty} \mathbb{E}[R_t \mid S_0, A_{0:t-1} \sim \pi] \\ &= \sum_s \mu(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a) r, \end{aligned} \tag{13.15}$$

where μ is the steady-state distribution under π, $\mu(s) \doteq \lim_{t\to\infty} \Pr\{S_t = s | A_{0:t} \sim \pi\}$, which is assumed to exist and to be independent of S_0 (an ergodicity assumption). Remember that this is the special distribution under which, if you select actions according to π, you remain in the same distribution:

$$\sum_s \mu(s) \sum_a \pi(a|s, \boldsymbol{\theta}) p(s' \,|\, s, a) = \mu(s'), \text{ for all } s' \in \mathcal{S}. \tag{13.16}$$

Complete pseudocode for the actor–critic algorithm in the continuing case (backward view) is given in the box below.

Actor–Critic with Eligibility Traces (continuing), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$
Algorithm parameters: $\lambda^{\mathbf{w}} \in [0, 1]$, $\lambda^{\boldsymbol{\theta}} \in [0, 1]$, $\alpha^{\mathbf{w}} > 0$, $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\bar{R}} > 0$
Initialize $\bar{R} \in \mathbb{R}$ (e.g., to 0)
Initialize state-value weights $\mathbf{w} \in \mathbb{R}^d$ and policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)
Initialize $S \in \mathcal{S}$ (e.g., to s_0)

$\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ (d-component eligibility trace vector)
$\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ (d'-component eligibility trace vector)
Loop forever (for each time step):
  $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
  Take action A, observe S', R
  $\delta \leftarrow R - \bar{R} + \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$
  $\bar{R} \leftarrow \bar{R} + \alpha^{\bar{R}} \delta$
  $\mathbf{z}^{\mathbf{w}} \leftarrow \lambda^{\mathbf{w}} \mathbf{z}^{\mathbf{w}} + \nabla \hat{v}(S, \mathbf{w})$
  $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \lambda^{\boldsymbol{\theta}} \mathbf{z}^{\boldsymbol{\theta}} + \nabla \ln \pi(A|S, \boldsymbol{\theta})$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \mathbf{z}^{\mathbf{w}}$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \mathbf{z}^{\boldsymbol{\theta}}$
  $S \leftarrow S'$

Naturally, in the continuing case, we define values, $v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t=s]$ and $q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t=s, A_t=a]$, with respect to the differential return:

$$G_t \doteq R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \cdots. \tag{13.17}$$

With these alternate definitions, the policy gradient theorem as given for the episodic case (13.5) remains true for the continuing case. A proof is given in the box on the next page. The forward and backward view equations also remain the same.

Proof of the Policy Gradient Theorem (continuing case)

The proof of the policy gradient theorem for the continuing case begins similarly to the episodic case. Again we leave it implicit in all cases that π is a function of $\boldsymbol{\theta}$ and that the gradients are with respect to $\boldsymbol{\theta}$. Recall that in the continuing case $J(\boldsymbol{\theta}) = r(\pi)$ (13.15) and that v_π and q_π denote values with respect to the differential return (13.17). The gradient of the state-value function can be written, for any $s \in \mathcal{S}$, as

$$\begin{aligned}
\nabla v_\pi(s) &= \nabla \left[\sum_a \pi(a|s) q_\pi(s,a) \right], \quad \text{for all } s \in \mathcal{S} && \text{(Exercise 3.18)} \\
&= \sum_a \Big[\nabla\pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla q_\pi(s,a) \Big] && \text{(product rule of calculus)} \\
&= \sum_a \Big[\nabla\pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla \sum_{s',r} p(s',r\,|\,s,a) \big(r - r(\boldsymbol{\theta}) + v_\pi(s')\big) \Big] \\
&= \sum_a \Big[\nabla\pi(a|s) q_\pi(s,a) + \pi(a|s) \big[-\nabla r(\boldsymbol{\theta}) + \sum_{s'} p(s'\,|\,s,a) \nabla v_\pi(s') \big] \Big].
\end{aligned}$$

After re-arranging terms, we obtain

$$\nabla r(\boldsymbol{\theta}) = \sum_a \Big[\nabla\pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'\,|\,s,a) \nabla v_\pi(s') \Big] - \nabla v_\pi(s).$$

Notice that the left-hand side can be written $\nabla J(\boldsymbol{\theta})$, and that it does not depend on s. Thus the right-hand side does not depend on s either, and we can safely sum it over all $s \in \mathcal{S}$, weighted by $\mu(s)$, without changing it (because $\sum_s \mu(s) = 1$):

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \sum_s \mu(s) \left(\sum_a \Big[\nabla\pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'\,|\,s,a) \nabla v_\pi(s') \Big] - \nabla v_\pi(s) \right) \\
&= \sum_s \mu(s) \sum_a \nabla\pi(a|s) q_\pi(s,a) \\
&\quad + \sum_s \mu(s) \sum_a \pi(a|s) \sum_{s'} p(s'\,|\,s,a) \nabla v_\pi(s') - \sum_s \mu(s) \nabla v_\pi(s)
\end{aligned}$$

$$
\begin{aligned}
&= \sum_s \mu(s) \sum_a \nabla\pi(a|s) q_\pi(s,a) \\
&\quad + \sum_{s'} \underbrace{\sum_s \mu(s) \sum_a \pi(a|s) p(s'|s,a)}_{\mu(s') \text{ (13.16)}} \nabla v_\pi(s') - \sum_s \mu(s) \nabla v_\pi(s) \\
&= \sum_s \mu(s) \sum_a \nabla\pi(a|s) q_\pi(s,a) + \sum_{s'} \mu(s') \nabla v_\pi(s') - \sum_s \mu(s) \nabla v_\pi(s) \\
&= \sum_s \mu(s) \sum_a \nabla\pi(a|s) q_\pi(s,a). \qquad \text{Q.E.D.}
\end{aligned}
$$

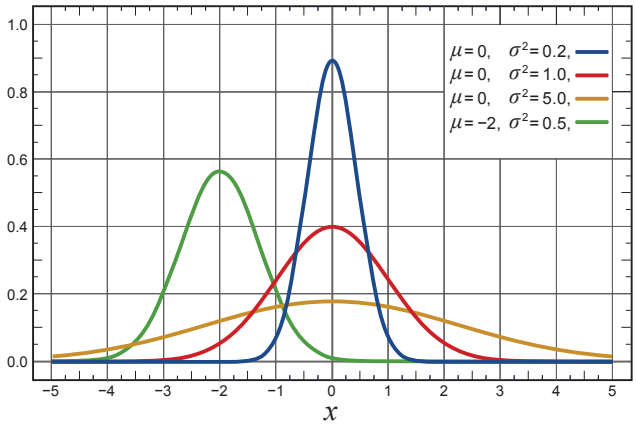
13.7 Policy Parameterization for Continuous Actions

Policy-based methods offer practical ways of dealing with large action spaces, even continuous spaces with an infinite number of actions. Instead of computing learned probabilities for each of the many actions, we instead learn statistics of the probability distribution. For example, the action set might be the real numbers, with actions chosen from a normal (Gaussian) distribution.

The probability density function for the normal distribution is conventionally written

$$
p(x) \doteq \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \tag{13.18}
$$

where μ and σ here are the mean and standard deviation of the normal distribution, and of course π here is just the number $\pi \approx 3.14159$. The probability density functions for several different means and standard deviations are shown to the right. The value $p(x)$ is the *density* of the probability at x, not the probability. It can be greater than 1; it is the total area under $p(x)$ that must sum to 1. In general, one can take the integral under $p(x)$ for any range of x values to get the probability of x falling within that range.



To produce a policy parameterization, the policy can be defined as the normal probability density over a real-valued scalar action, with mean and standard deviation given by parametric function approximators that depend on the state. That is,

$$
\pi(a|s,\boldsymbol{\theta}) \doteq \frac{1}{\sigma(s,\boldsymbol{\theta})\sqrt{2\pi}} \exp\left(-\frac{(a-\mu(s,\boldsymbol{\theta}))^2}{2\sigma(s,\boldsymbol{\theta})^2}\right), \tag{13.19}
$$

where $\mu : \mathcal{S}\times\mathbb{R}^{d'} \to \mathbb{R}$ and $\sigma : \mathcal{S}\times\mathbb{R}^{d'} \to \mathbb{R}^+$ are two parameterized function approximators.

To complete the example we need only give a form for these approximators. For this we divide the policy's parameter vector into two parts, $\boldsymbol{\theta} = [\boldsymbol{\theta}_\mu, \boldsymbol{\theta}_\sigma]^\top$, one part to be used for the approximation of the mean and one part for the approximation of the standard deviation. The mean can be approximated as a linear function. The standard deviation must always be positive and is better approximated as the exponential of a linear function. Thus

$$\mu(s, \boldsymbol{\theta}) \doteq {\boldsymbol{\theta}_\mu}^\top \mathbf{x}_\mu(s) \quad \text{and} \quad \sigma(s, \boldsymbol{\theta}) \doteq \exp\Big({\boldsymbol{\theta}_\sigma}^\top \mathbf{x}_\sigma(s)\Big), \tag{13.20}$$

where $\mathbf{x}_\mu(s)$ and $\mathbf{x}_\sigma(s)$ are state feature vectors perhaps constructed by one of the methods described in Section 9.5. With these definitions, all the algorithms described in the rest of this chapter can be applied to learn to select real-valued actions.

Exercise 13.4 Show that for the Gaussian policy parameterization (Equations 13.19 and 13.20) the eligibility vector has the following two parts:

$$\nabla \ln \pi(a|s, \boldsymbol{\theta}_\mu) = \frac{\nabla \pi(a|s, \boldsymbol{\theta}_\mu)}{\pi(a|s, \boldsymbol{\theta})} = \frac{1}{\sigma(s, \boldsymbol{\theta})^2}\big(a - \mu(s, \boldsymbol{\theta})\big)\mathbf{x}_\mu(s), \text{ and}$$

$$\nabla \ln \pi(a|s, \boldsymbol{\theta}_\sigma) = \frac{\nabla \pi(a|s, \boldsymbol{\theta}_\sigma)}{\pi(a|s, \boldsymbol{\theta})} = \left(\frac{\big(a - \mu(s, \boldsymbol{\theta})\big)^2}{\sigma(s, \boldsymbol{\theta})^2} - 1\right)\mathbf{x}_\sigma(s). \qquad \square$$

Exercise 13.5 A *Bernoulli-logistic unit* is a stochastic neuron-like unit used in some ANNs (Section 9.7). Its input at time t is a feature vector $\mathbf{x}(S_t)$; its output, A_t, is a random variable having two values, 0 and 1, with $\Pr\{A_t = 1\} = P_t$ and $\Pr\{A_t = 0\} = 1 - P_t$ (the Bernoulli distribution). Let $h(s, 0, \boldsymbol{\theta})$ and $h(s, 1, \boldsymbol{\theta})$ be the preferences in state s for the unit's two actions given policy parameter $\boldsymbol{\theta}$. Assume that the difference between the action preferences is given by a weighted sum of the unit's input vector, that is, assume that $h(s, 1, \boldsymbol{\theta}) - h(s, 0, \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \mathbf{x}(s)$, where $\boldsymbol{\theta}$ is the unit's weight vector.

(a) Show that if the exponential soft-max distribution (13.2) is used to convert action preferences to policies, then $P_t = \pi(1|S_t, \boldsymbol{\theta}_t) = 1/(1 + \exp(-\boldsymbol{\theta}_t^\top \mathbf{x}(S_t)))$ (the logistic function).

(b) What is the Monte-Carlo REINFORCE update of $\boldsymbol{\theta}_t$ to $\boldsymbol{\theta}_{t+1}$ upon receipt of return G_t?

(c) Express the eligibility $\nabla \ln \pi(a|s, \boldsymbol{\theta})$ for a Bernoulli-logistic unit, in terms of a, $\mathbf{x}(s)$, and $\pi(a|s, \boldsymbol{\theta})$ by calculating the gradient.

Hint for part (c): Define $P = \pi(1|s, \boldsymbol{\theta})$ and compute the derivative of the logarithm, for each action, using the chain rule on P. Combine the two results into one expression that depends on a and P, and then use the chain rule again, this time on $\boldsymbol{\theta}^\top \mathbf{x}(s)$, noting that the derivative of the logistic function $f(x) = 1/(1 + e^{-x})$ is $f(x)(1 - f(x))$. $\square$

13.8 Summary

Prior to this chapter, this book focused on *action-value methods*—meaning methods that learn action values and then use them to determine action selections. In this chapter, on the other hand, we considered methods that learn a parameterized policy that enables actions to be taken without consulting action-value estimates. In particular, we have considered *policy-gradient methods*—meaning methods that update the policy parameter on each step in the direction of an estimate of the gradient of performance with respect to the policy parameter.

Methods that learn and store a policy parameter have many advantages. They can learn specific probabilities for taking the actions. They can learn appropriate levels of exploration and approach deterministic policies asymptotically. They can naturally handle continuous action spaces. All these things are easy for policy-based methods but awkward or impossible for ε-greedy methods and for action-value methods in general. In addition, on some problems the policy is just simpler to represent parametrically than the value function; these problems are more suited to parameterized policy methods.

Parameterized policy methods also have an important theoretical advantage over action-value methods in the form of the *policy gradient theorem*, which gives an exact formula for how performance is affected by the policy parameter that does not involve derivatives of the state distribution. This theorem provides a theoretical foundation for all policy gradient methods.

The REINFORCE method follows directly from the policy gradient theorem. Adding a state-value function as a *baseline* reduces REINFORCE's variance without introducing bias. If the state-value function is also used to assess—or criticize—the policy's action selections, then the value function is called a *critic* and the policy is called an *actor*; the overall method is called an *actor–critic* method. The critic introduces bias into the actor's gradient estimates, but is often desirable for the same reason that bootstrapping TD methods are often superior to Monte Carlo methods (substantially reduced variance).

Overall, policy-gradient methods provide a significantly different set of strengths and weaknesses than action-value methods. Today they are less well understood in some respects, but a subject of excitement and ongoing research.

Bibliographical and Historical Remarks

Methods that we now see as related to policy gradients were actually some of the earliest to be studied in reinforcement learning (Witten, 1977; Barto, Sutton, and Anderson, 1983; Sutton, 1984; Williams, 1987, 1992) and in predecessor fields (see Phansalkar and Thathachar, 1995). They were largely supplanted in the 1990s by the action-value methods that are the focus of the other chapters of this book. In recent years, however, attention has returned to actor–critic methods and to policy-gradient methods in general. Among the further developments beyond what we cover here are natural-gradient methods (Amari, 1998; Kakade, 2002, Peters, Vijayakumar and Schaal, 2005; Peters and Schaal, 2008; Park, Kim and Kang, 2005; Bhatnagar, Sutton, Ghavamzadeh and Lee, 2009; see Grondman, Busoniu, Lopes and Babuska, 2012), deterministic policy-gradient methods

(Silver et al., 2014), off-policy policy-gradient methods (Degris, White, and Sutton, 2012; Maei, 2018), and entropy regularization (see Schulman, Chen, and Abbeel, 2017). Major applications include acrobatic helicopter autopilots and AlphaGo (Section 16.6).

Our presentation in this chapter is based primarily on that by Sutton, McAllester, Singh, and Mansour (2000), who introduced the term "policy gradient methods." A useful overview is provided by Bhatnagar et al. (2009). One of the earliest related works is by Aleksandrov, Sysoyev, and Shemeneva (1968). Thomas (2014) first realized that the factor of γ^t, as specified in the boxed algorithms of this chapter, was needed in the case of discounted episodic problems.

13.1 Example 13.1 and the results with it in this chapter were developed with Eric Graves.

13.2 The policy gradient theorem here and on page 334 was first obtained by Marbach and Tsitsiklis (1998, 2001) and then independently by Sutton et al. (2000). A similar expression was obtained by Cao and Chen (1997). Other early results are due to Konda and Tsitsiklis (2000, 2003), Baxter and Bartlett (2001), and Baxter, Bartlett, and Weaver (2001). Some additional results are developed by Sutton, Singh, and McAllester (2000).

13.3 REINFORCE is due to Williams (1987, 1992). Phansalkar and Thathachar (1995) proved both local and global convergence theorems for modified versions of REINFORCE algorithms.

The all-actions algorithm was first presented in an unpublished but widely circulated incomplete paper (Sutton, Singh, and McAllester, 2000) and then developed further by Ciosek and Whiteson (2017, 2018), who termed it "expected policy gradients," and by Asadi, Allen, Roderick, Mohamed, Konidaris, and Littman (2017), who called it "mean actor critic."

13.4 The baseline was introduced in Williams's (1987, 1992) original work. Greensmith, Bartlett, and Baxter (2004) analyzed an arguably better baseline (see Dick, 2015). Thomas and Brunskill (2017) argue that an action-dependent baseline can be used without incurring bias.

13.5–6 Actor–critic methods were among the earliest to be investigated in reinforcement learning (Witten, 1977; Barto, Sutton, and Anderson, 1983; Sutton, 1984). The algorithms presented here are based on the work of Degris, White, and Sutton (2012). Actor–critic methods are sometimes referred to as advantage actor–critic ("A2C") methods in the literature.

13.7 The first to show how continuous actions could be handled this way appears to have been Williams (1987, 1992). The figure on page 335 is adapted from Wikipedia.

Part III: Looking Deeper

In this last part of the book we look beyond the standard reinforcement learning ideas presented in the first two parts of the book to briefly survey their relationships with psychology and neuroscience, a sampling of reinforcement learning applications, and some of the active frontiers for future reinforcement learning research.

Chapter 14

Psychology

In previous chapters we developed ideas for algorithms based on computational considerations alone. In this chapter we look at some of these algorithms from another perspective: the perspective of psychology and its study of how animals learn. The goals of this chapter are, first, to discuss ways that reinforcement learning ideas and algorithms correspond to what psychologists have discovered about animal learning, and second, to explain the influence reinforcement learning is having on the study of animal learning. The clear formalism provided by reinforcement learning that systemizes tasks, returns, and algorithms is proving to be enormously useful in making sense of experimental data, in suggesting new kinds of experiments, and in pointing to factors that may be critical to manipulate and to measure. The idea of optimizing return over the long term that is at the core of reinforcement learning is contributing to our understanding of otherwise puzzling features of animal learning and behavior.

Some of the correspondences between reinforcement learning and psychological theories are not surprising because the development of reinforcement learning drew inspiration from psychological learning theories. However, as developed in this book, reinforcement learning explores idealized situations from the perspective of an artificial intelligence researcher or engineer, with the goal of solving computational problems with efficient algorithms, rather than to replicate or explain in detail how animals learn. As a result, some of the correspondences we describe connect ideas that arose independently in their respective fields. We believe these points of contact are specially meaningful because they expose computational principles important to learning, whether it is learning by artificial or by natural systems.

For the most part, we describe correspondences between reinforcement learning and learning theories developed to explain how animals like rats, pigeons, and rabbits learn in controlled laboratory experiments. Thousands of these experiments were conducted throughout the 20th century, and many are still being conducted today. Although sometimes dismissed as irrelevant to wider issues in psychology, these experiments probe subtle properties of animal learning, often motivated by precise theoretical questions. As psychology shifted its focus to more cognitive aspects of behavior, that is, to mental

processes such as thought and reasoning, animal learning experiments came to play less of a role in psychology than they once did. But this experimentation led to the discovery of learning principles that are elemental and widespread throughout the animal kingdom, principles that should not be neglected in designing artificial learning systems. In addition, as we shall see, some aspects of cognitive processing connect naturally to the computational perspective provided by reinforcement learning.

This chapter's final section includes references relevant to the connections we discuss as well as to connections we neglect. We hope this chapter encourages readers to probe all of these connections more deeply. Also included in this final section is a discussion of how the terminology used in reinforcement learning relates to that of psychology. Many of the terms and phrases used in reinforcement learning are borrowed from animal learning theories, but the computational/engineering meanings of these terms and phrases do not always coincide with their meanings in psychology.

14.1 Prediction and Control

The algorithms we describe in this book fall into two broad categories: algorithms for *prediction* and algorithms for *control*.[1] These categories arise naturally in solution methods for the reinforcement learning problem presented in Chapter 3. In many ways these categories respectively correspond to categories of learning extensively studied by psychologists: *classical*, or *Pavlovian*, *conditioning* and *instrumental*, or *operant*, *conditioning*. These correspondences are not completely accidental because of psychology's influence on reinforcement learning, but they are nevertheless striking because they connect ideas arising from different objectives.

The prediction algorithms presented in this book estimate quantities that depend on how features of an agent's environment are expected to unfold over the future. We specifically focus on estimating the amount of reward an agent can expect to receive over the future while it interacts with its environment. In this role, prediction algorithms are *policy evaluation algorithms*, which are integral components of algorithms for improving policies. But prediction algorithms are not limited to predicting future reward; they can predict any feature of the environment (see, for example, Modayil, White, and Sutton, 2014). The correspondence between prediction algorithms and classical conditioning rests on their common property of predicting upcoming stimuli, whether or not those stimuli are rewarding (or punishing).

The situation in an instrumental, or operant, conditioning experiment is different. Here, the experimental apparatus is set up so that an animal is given something it likes (a reward) or something it dislikes (a penalty) depending on what the animal did. The animal learns to increase its tendency to produce rewarded behavior and to decrease its tendency to produce penalized behavior. The reinforcing stimulus is said to be *contingent* on the animal's behavior, whereas in classical conditioning it is not (although it is difficult to remove all behavior contingencies in a classical conditioning experiment). Instrumental

[1] What control means for us is different from what it typically means in animal learning theories; there the environment controls the agent instead of the other way around. See our comments on terminology at the end of this chapter.

conditioning experiments are like those that inspired Thorndike's Law of Effect that we briefly discuss in Chapter 1. *Control* is at the core of this form of learning, which corresponds to the operation of reinforcement learning's policy-improvement algorithms.

Thinking of classical conditioning in terms of prediction, and instrumental conditioning in terms of control, is a starting point for connecting our computational view of reinforcement learning to animal learning, but in reality, the situation is more complicated than this. There is more to classical conditioning than prediction; it also involves action, and so is a mode of control, sometimes called *Pavlovian control*. Further, classical and instrumental conditioning interact in interesting ways, with both sorts of learning likely being engaged in most experimental situations. Despite these complications, aligning the classical/instrumental distinction with the prediction/control distinction is a convenient first approximation in connecting reinforcement learning to animal learning.

In psychology, the term reinforcement is used to describe learning in both classical and instrumental conditioning. Originally referring only to the strengthening of a pattern of behavior, it is frequently also used for the weakening of a pattern of behavior. A stimulus considered to be the cause of the change in behavior is called a reinforcer, whether or not it is contingent on the animal's previous behavior. At the end of this chapter we discuss this terminology in more detail and how it relates to terminology used in machine learning.

14.2 Classical Conditioning

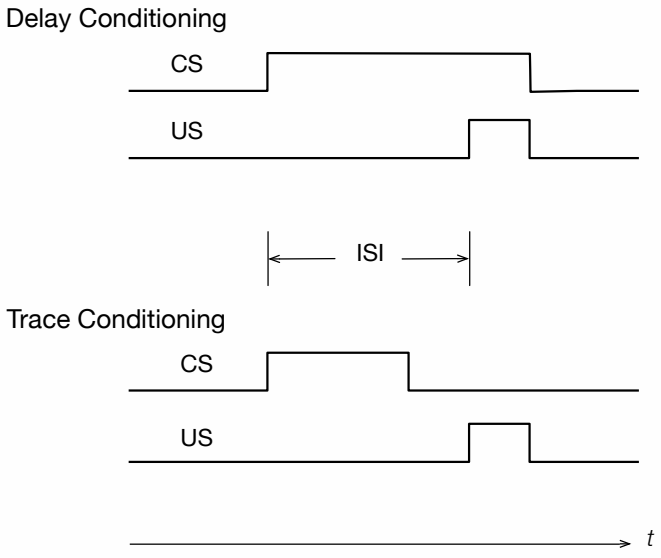
While studying the activity of the digestive system, the celebrated Russian physiologist Ivan Pavlov found that an animal's innate responses to certain triggering stimuli can come to be triggered by other stimuli that are quite unrelated to the inborn triggers. His experimental subjects were dogs that had undergone minor surgery to allow the intensity of their salivary reflex to be accurately measured. In one case he describes, the dog did not salivate under most circumstances, but about 5 seconds after being presented with food it produced about six drops of saliva over the next several seconds. After several repetitions of presenting another stimulus, one not related to food, in this case the sound of a metronome, shortly before the introduction of food, the dog salivated in response to the sound of the metronome in the same way it did to the food. "The activity of the salivary gland has thus been called into play by impulses of sound—a stimulus quite alien to food" (Pavlov, 1927, p. 22). Summarizing the significance of this finding, Pavlov wrote:

> It is pretty evident that under natural conditions the normal animal must respond not only to stimuli which themselves bring immediate benefit or harm, but also to other physical or chemical agencies—waves of sound, light, and the like—which in themselves only *signal* the approach of these stimuli; though it is not the sight and sound of the beast of prey which is in itself harmful to the smaller animal, but its teeth and claws. (Pavlov, 1927, p. 14)

Connecting new stimuli to innate reflexes in this way is now called classical, or Pavlovian, conditioning. Pavlov (or more exactly, his translators) called inborn responses (e.g.,

salivation in his demonstration described above) "unconditioned responses" (URs), their natural triggering stimuli (e.g., food) "unconditioned stimuli" (USs), and new responses triggered by predictive stimuli (e.g., here also salivation) "conditioned responses" (CRs). A stimulus that is initially neutral, meaning that it does not normally elicit strong responses (e.g., the metronome sound), becomes a "conditioned stimulus" (CS) as the animal learns that it predicts the US and so comes to produce a CR in response to the CS. These terms are still used in describing classical conditioning experiments (though better translations would have been "conditional" and "unconditional" instead of conditioned and unconditioned). The US is called a reinforcer because it reinforces producing a CR in response to the CS.

The arrangement of stimuli in two common types of classical conditioning experiments is shown to the right. In *delay conditioning*, the CS extends throughout the interstimulus interval, or ISI, which is the time interval between the CS onset and the US onset (with the CS ending when the US ends in a common version shown here). In *trace conditioning*, the US begins after the CS ends, and the time interval between CS offset and US onset is called the trace interval.



The salivation of Pavlov's dogs to the sound of a metronome is just one example of classical conditioning, which has been intensively studied across many response systems of many species of animals. URs are often preparatory in some way, like the salivation of Pavlov's dog, or protective in some way, like an eye blink in response to something irritating to the eye, or freezing in response to seeing a predator. Experiencing the CS-US predictive relationship over a series of trials causes the animal to learn that the CS predicts the US so that the animal can respond to the CS with a CR that prepares the animal for, or protects it from, the predicted US. Some CRs are similar to the UR but begin earlier and differ in ways that increase their effectiveness. In one intensively studied type of experiment, for example, a tone CS reliably predicts a puff of air (the US) to a rabbit's eye, triggering a UR consisting of the closure of a protective inner eyelid called the nictitating membrane. After one or more trials, the tone comes to trigger a CR consisting of membrane closure that begins before the air puff and eventually becomes timed so that peak closure occurs just when the air puff is likely to occur. This CR, being initiated in anticipation of the air puff and appropriately timed, offers better protection than simply initiating closure as a reaction to the irritating US. The ability to act in anticipation of important events by learning about predictive relationships among stimuli is so beneficial that it is widely present across the animal kingdom.

14.2.1 Blocking and Higher-order Conditioning

Many interesting properties of classical conditioning have been observed in experiments. Beyond the anticipatory nature of CRs, two widely observed properties figured prominently in the development of classical conditioning models: *blocking* and *higher-order conditioning*. Blocking occurs when an animal fails to learn a CR when a potential CS is presented along with another CS that had been used previously to condition the animal to produce that CR. For example, in the first stage of a blocking experiment involving rabbit nictitating membrane conditioning, a rabbit is first conditioned with a tone CS and an air puff US to produce the CR of closing its nictitating membrane in anticipation of the air puff. The experiment's second stage consists of additional trials in which a second stimulus, say a light, is added to the tone to form a compound tone/light CS followed by the same air puff US. In the experiment's third phase, the second stimulus alone—the light—is presented to the rabbit to see if the rabbit has learned to respond to it with a CR. It turns out that the rabbit produces very few, or no, CRs in response to the light: learning to the light had been *blocked* by the previous learning to the tone.[2] Blocking results like this challenged the idea that conditioning depends only on simple temporal contiguity, that is, that a necessary and sufficient condition for conditioning is that a US frequently follows a CS closely in time. In the next section we describe the *Rescorla–Wagner model* (Rescorla and Wagner, 1972) that offered an influential explanation for blocking.

Higher-order conditioning occurs when a previously-conditioned CS acts as a US in conditioning another initially neutral stimulus. Pavlov described an experiment in which his assistant first conditioned a dog to salivate to the sound of a metronome that predicted a food US, as described above. After this stage of conditioning, a number of trials were conducted in which a black square, to which the dog was initially indifferent, was placed in the dog's line of vision followed by the sound of the metronome—and this was *not* followed by food. In just ten trials, the dog began to salivate merely upon seeing the black square, despite the fact that the sight of it had never been followed by food. The sound of the metronome itself acted as a US in conditioning a salivation CR to the black square CS. This was second-order conditioning. If the black square had been used as a US to establish salivation CRs to another otherwise neutral CS, it would have been third-order conditioning, and so on. Higher-order conditioning is difficult to demonstrate, especially above the second order, in part because a higher-order reinforcer loses its reinforcing value due to not being repeatedly followed by the original US during higher-order conditioning trials. But under the right conditions, such as intermixing first-order trials with higher-order trials or by providing a general energizing stimulus, higher-order conditioning beyond the second order can be demonstrated. As we describe below, the *TD model of classical conditioning* uses the bootstrapping idea that is central to our approach to extend the Rescorla–Wagner model's account of blocking to include both the anticipatory nature of CRs and higher-order conditioning.

[2] Comparison with a control group is necessary to show that the previous conditioning to the tone is responsible for blocking learning to the light. This is done by trials with the tone/light CS but with no prior conditioning to the tone. Learning to the light in this case is unimpaired. Moore and Schmajuk (2008) give a full account of this procedure.

Higher-order instrumental conditioning occurs as well. In this case, a stimulus that consistently predicts primary reinforcement becomes a reinforcer itself, where reinforcement is primary if its rewarding or penalizing quality has been built into the animal by evolution. The predicting stimulus becomes a *secondary reinforcer*, or more generally, a *higher-order* or *conditioned reinforcer*—the latter being a better term when the predicted reinforcing stimulus is itself a secondary, or an even higher-order, reinforcer. A conditioned reinforcer delivers *conditioned reinforcement*: conditioned reward or conditioned penalty. Conditioned reinforcement acts like primary reinforcement in increasing an animal's tendency to produce behavior that leads to conditioned reward, and to decrease an animal's tendency to produce behavior that leads to conditioned penalty. (See our comments at the end of this chapter that explain how our terminology sometimes differs, as it does here, from terminology used in psychology.)

Conditioned reinforcement is a key phenomenon that explains, for instance, why we work for the conditioned reinforcer money, whose worth derives solely from what is predicted by having it. In actor–critic methods described in Section 13.5 (and discussed in the context of neuroscience in Sections 15.7 and 15.8), the critic uses a TD method to evaluate the actor's policy, and its value estimates provide conditioned reinforcement to the actor, allowing the actor to improve its policy. This analog of higher-order instrumental conditioning helps address the credit-assignment problem mentioned in Section 1.7 because the critic gives moment-by-moment reinforcement to the actor when the primary reward signal is delayed. We discuss this more below in Section 14.4.

14.2.2 The Rescorla–Wagner Model

Rescorla and Wagner created their model mainly to account for blocking. The core idea of the Rescorla–Wagner model is that an animal only learns when events violate its expectations, in other words, only when the animal is surprised (although without necessarily implying any *conscious* expectation or emotion). We first present Rescorla and Wagner's model using their terminology and notation before shifting to the terminology and notation we use to describe the TD model.

Here is how Rescorla and Wagner described their model. The model adjusts the "associative strength" of each component stimulus of a compound CS, which is a number representing how strongly or reliably that component is predictive of a US. When a compound CS consisting of several component stimuli is presented in a classical conditioning trial, the associative strength of each component stimulus changes in a way that depends on an associative strength associated with the entire stimulus compound, called the "aggregate associative strength," and not just on the associative strength of each component itself.

Rescorla and Wagner considered a compound CS AX, consisting of component stimuli A and X, where the animal may have already experienced stimulus A, and stimulus X might be new to the animal. Let V_{A}, V_{X}, and V_{AX} respectively denote the associative strengths of stimuli A, X, and the compound AX. Suppose that on a trial the compound CS AX is followed by a US, which we label stimulus Y. Then the associative strengths of

the stimulus components change according to these expressions:

$$\Delta V_{\mathrm{A}} = \alpha_{\mathrm{A}}\beta_{\mathrm{Y}}(R_{\mathrm{Y}} - V_{\mathrm{AX}})$$
$$\Delta V_{\mathrm{X}} = \alpha_{\mathrm{X}}\beta_{\mathrm{Y}}(R_{\mathrm{Y}} - V_{\mathrm{AX}}),$$

where $\alpha_{\mathrm{A}}\beta_{\mathrm{Y}}$ and $\alpha_{\mathrm{X}}\beta_{\mathrm{Y}}$ are the step-size parameters, which depend on the identities of the CS components and the US, and R_{Y} is the asymptotic level of associative strength that the US Y can support. (Rescorla and Wagner used λ here instead of R, but we use R to avoid confusion with our use of λ and because we usually think of this as the magnitude of a reward signal, with the caveat that the US in classical conditioning is not necessarily rewarding or penalizing.) A key assumption of the model is that the aggregate associative strength V_{AX} is equal to $V_{\mathrm{A}} + V_{\mathrm{X}}$. The associative strengths as changed by these Δs become the associative strengths at the beginning of the next trial.

To be complete, the model needs a response-generation mechanism, which is a way of mapping values of Vs to CRs. Because this mapping would depend on details of the experimental situation, Rescorla and Wagner did not specify a mapping but simply assumed that larger Vs would produce stronger or more likely CRs, and that negative Vs would mean that there would be no CRs.

The Rescorla–Wagner model accounts for the acquisition of CRs in a way that explains blocking. As long as the aggregate associative strength, V_{AX}, of the stimulus compound is below the asymptotic level of associative strength, R_{Y}, that the US Y can support, the prediction error $R_{\mathrm{Y}} - V_{\mathrm{AX}}$ is positive. This means that over successive trials the associative strengths V_{A} and V_{X} of the component stimuli increase until the aggregate associative strength V_{AX} equals R_{Y}, at which point the associative strengths stop changing (unless the US changes). When a new component is added to a compound CS to which the animal has already been conditioned, further conditioning with the augmented compound produces little or no increase in the associative strength of the added CS component because the error has already been reduced to zero, or to a low value. The occurrence of the US is already predicted nearly perfectly, so little or no error—or surprise—is introduced by the new CS component. Prior learning blocks learning to the new component.

To transition from Rescorla and Wagner's model to the TD model of classical conditioning (which we just call the TD model), we first recast their model in terms of the concepts that we are using throughout this book. Specifically, we match the notation we use for learning with linear function approximation (Section 9.4), and we think of the conditioning process as one of learning to predict the "magnitude of the US" on a trial on the basis of the compound CS presented on that trial, where the magnitude of a US Y is the R_{Y} of the Rescorla–Wagner model as given above. We also introduce states. Because the Rescorla–Wagner model is a *trial-level* model, meaning that it deals with how associative strengths change from trial to trial without considering any details about what happens within and between trials, we do not have to consider how states change during a trial until we present the full TD model in the following section. Instead, here we simply think of a state as a way of labeling a trial in terms of the collection of component CSs that are present on the trial.

Therefore, assume that trial-type, or state, s is described by a real-valued vector of features $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_d(s))^\top$ where $x_i(s) = 1$ if CS_i, the i^{th} component of a

compound CS, is present on the trial and 0 otherwise. Then if the d-dimensional vector of associative strengths is $\mathbf{w}$, the aggregate associative strength for trial-type s is

$$\hat{v}(s,\mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s). \tag{14.1}$$

This corresponds to a *value estimate* in reinforcement learning, and we think of it as the *US prediction.*

Now temporally let t denote the number of a complete trial and not its usual meaning as a time step (we revert to t's usual meaning when we extend this to the TD model below), and assume that S_t is the state corresponding to trial t. Conditioning trial t updates the associative strength vector $\mathbf{w}_t$ to $\mathbf{w}_{t+1}$ as follows:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\delta_t \mathbf{x}(S_t), \tag{14.2}$$

where α is the step-size parameter, and—because here we are describing the Rescorla–Wagner model—δ_t is the *prediction error*

$$\delta_t = R_t - \hat{v}(S_t,\mathbf{w}_t). \tag{14.3}$$

R_t is the target of the prediction on trial t, that is, the magnitude of the US, or in Rescorla and Wagner's terms, the associative strength that the US on the trial can support. Note that because of the factor $\mathbf{x}(S_t)$ in (14.2), only the associative strengths of CS components present on a trial are adjusted as a result of that trial. You can think of the prediction error as a measure of surprise, and the aggregate associative strength as the animal's expectation that is violated when it does not match the target US magnitude.

From the perspective of machine learning, the Rescorla–Wagner model is an error-correction supervised learning rule. It is essentially the same as the Least Mean Square (LMS), or Widrow-Hoff, learning rule (Widrow and Hoff, 1960) that finds the weights—here the associative strengths—that make the average of the squares of all the errors as close to zero as possible. It is a "curve-fitting," or regression, algorithm that is widely used in engineering and scientific applications (see Section 9.4).[3]

The Rescorla–Wagner model was very influential in the history of animal learning theory because it showed that a "mechanistic" theory could account for the main facts about blocking without resorting to more complex cognitive theories involving, for example, an animal's explicit recognition that another stimulus component had been added and then scanning its short-term memory backward to reassess the predictive relationships involving the US. The Rescorla–Wagner model showed how traditional contiguity theories of conditioning—that temporal contiguity of stimuli was a necessary and sufficient condition for learning—could be adjusted in a simple way to account for blocking (Moore and Schmajuk, 2008).

The Rescorla–Wagner model provides a simple account of blocking and some other features of classical conditioning, but it is not a complete or perfect model of classical

[3]The only differences between the LMS rule and the Rescorla–Wagner model are that for LMS the input vectors $\mathbf{x}_t$ can have any real numbers as components, and—at least in the simplest version of the LMS rule—the step-size parameter α does not depend on the input vector or the identity of the stimulus setting the prediction target.

conditioning. Different ideas account for a variety of other observed effects, and progress is still being made toward understanding the many subtleties of classical conditioning. The TD model, which we describe next, though also not a complete or perfect model model of classical conditioning, extends the Rescorla–Wagner model to address how within-trial and between-trial timing relationships among stimuli can influence learning and how higher-order conditioning might arise.

14.2.3 The TD Model

The TD model is a *real-time* model, as opposed to a trial-level model like the Rescorla–Wagner model. A single step t in the Rescorla–Wagner model represents an entire conditioning trial. The model does not apply to details about what happens during the time a trial is taking place, or what might happen between trials. Within each trial an animal might experience various stimuli whose onsets occur at particular times and that have particular durations. These timing relationships strongly influence learning. The Rescorla–Wagner model also does not include a mechanism for higher-order conditioning, whereas for the TD model, higher-order conditioning is a natural consequence of the bootstrapping idea that is at the base of TD algorithms.

To describe the TD model we begin with the formulation of the Rescorla–Wagner model above, but t now labels time steps within or between trials instead of complete trials. Think of the time between t and $t+1$ as a small time interval, say .01 second, and think of a trial as a sequences of states, one associated with each time step, where the state at step t now represents details of how stimuli are represented at t instead of just a label for the CS components present on a trial. In fact, we can completely abandon the idea of trials. From the point of view of the animal, a trial is just a fragment of its continuing experience interacting with its world. Following our usual view of an agent interacting with its environment, imagine that the animal is experiencing an endless sequence of states s, each represented by a feature vector $\mathbf{x}(s)$. That said, it is still often convenient to refer to trials as fragments of time during which patterns of stimuli repeat in an experiment.

State features are not restricted to describing the external stimuli that an animal experiences; they can describe neural activity patterns that external stimuli produce in an animal's brain, and these patterns can be history-dependent, meaning that they can be persistent patterns produced by sequences of external stimuli. Of course, we do not know exactly what these neural activity patterns are, but a real-time model like the TD model allows one to explore the consequences on learning of different hypotheses about the internal representations of external stimuli. For these reasons, the TD model does not commit to any particular state representation. In addition, because the TD model includes discounting and eligibility traces that span time intervals between stimuli, the model also makes it possible to explore how discounting and eligibility traces interact with stimulus representations in making predictions about the results of classical conditioning experiments.

Below we describe some of the state representations that have been used with the TD model and some of their implications, but for the moment we stay agnostic about

the representation and just assume that each state s is represented by a feature vector $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_n(s))^\top$. Then the aggregate associative strength corresponding to a state s is given by (14.1), the same as for the Rescorla-Wagner model, but the TD model updates the associative strength vector, $\mathbf{w}$, differently. With t now labeling a time step instead of a complete trial, the TD model governs learning according to this update:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t, \tag{14.4}$$

which replaces $\mathbf{x}_t(S_t)$ in the Rescorla–Wagner update (14.2) with $\mathbf{z}_t$, a vector of eligibility traces, and instead of the δ_t of (14.3), here δ_t is a TD error:

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t), \tag{14.5}$$

where γ is a discount factor (between 0 and 1), R_t is the prediction target at time t, and $\hat{v}(S_{t+1}, \mathbf{w}_t)$ and $\hat{v}(S_t, \mathbf{w}_t)$ are aggregate associative strengths at $t+1$ and t as defined by (14.1).

Each component i of the eligibility-trace vector $\mathbf{z}_t$ increments or decrements according to the component $x_i(S_t)$ of the feature vector $\mathbf{x}(S_t)$, and otherwise decays with a rate determined by $\gamma\lambda$:

$$\mathbf{z}_t = \gamma\lambda \mathbf{z}_{t-1} + \mathbf{x}(S_t). \tag{14.6}$$

Here λ is the usual eligibility trace decay parameter.

Note that if $\gamma = 0$, the TD model reduces to the Rescorla–Wagner model with the exceptions that: the meaning of t is different in each case (a trial number for the Rescorla–Wagner model and a time step for the TD model), and in the TD model there is a one-time-step lead in the prediction target R. The TD model is equivalent to the backward view of the semi-gradient TD(λ) algorithm with linear function approximation (Chapter 12), except that R_t in the model does not have to be a reward signal as it does when the TD algorithm is used to learn a value function for policy-improvement.

14.2.4 TD Model Simulations

Real-time conditioning models like the TD model are interesting primarily because they make predictions for a wide range of situations that cannot be represented by trial-level models. These situations involve the timing and durations of conditionable stimuli, the timing of these stimuli in relation to the timing of the US, and the timing and shapes of CRs. For example, the US generally must begin after the onset of a neutral stimulus for conditioning to occur, with the rate and effectiveness of learning depending on the inter-stimulus interval, or ISI, the interval between the onsets of the CS and the US. When CRs appear, they generally begin before the appearance of the US and their temporal profiles change during learning. In conditioning with compound CSs, the component stimuli of the compound CSs may not all begin and end at the same time, sometimes forming what is called a *serial compound* in which the component stimuli occur in a sequence over time. Timing considerations like these make it important to consider how

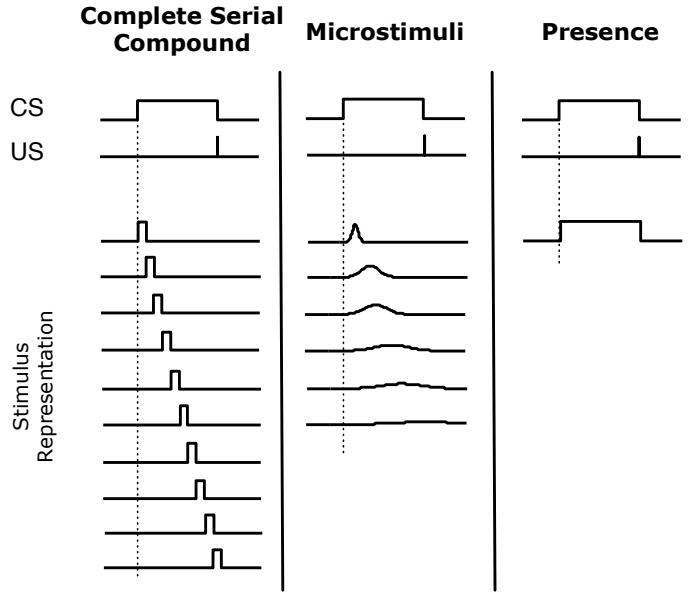


Figure 14.1: Three stimulus representations (in columns) sometimes used with the TD model. Each row represents one element of the stimulus representation. The three representations vary along a temporal generalization gradient, with no generalization between nearby time points in the complete serial compound (left column) and complete generalization between nearby time points in the presence representation (right column). The microstimulus representation occupies a middle ground. The degree of temporal generalization determines the temporal granularity with which US predictions are learned. Adapted with minor changes from *Learning & Behavior*, Evaluating the TD Model of Classical Conditioning, volume 40, 2012, p. 311, E. A. Ludvig, R. S. Sutton, E. J. Kehoe. With permission of Springer.

stimuli are represented, how these representations unfold over time during and between trials, and how they interact with discounting and eligibility traces.

Figure 14.1 shows three of the stimulus representations that have been used in exploring the behavior of the TD model: the *complete serial compound* (CSC), the *microstimulus* (MS), and the *presence* representations (Ludvig, Sutton, and Kehoe, 2012). These representations differ in the degree to which they force generalization among nearby time points during which a stimulus is present.

The simplest of the representations shown in Figure 14.1 is the presence representation in the figure's right column. This representation has a single feature for each component CS present on a trial, where the feature has value 1 whenever that component is present, and 0 otherwise.[4] The presence representation is not a realistic hypothesis about how stimuli are represented in an animal's brain, but as we describe below, the TD model with this representation can produce many of the timing phenomena seen in classical conditioning.

[4]In our formalism, there is a different state, S_t, for each time step t during a trial, and for a trial in which a compound CS consists of n component CSs of various durations occurring at various times throughout the trial, there is a feature, x_i, for each component CS_i, $i = 1, \ldots, n$, where $x_i(S_t) = 1$ for all times t when the CS_i is present, and equals zero otherwise.

For the CSC representation (left column of Figure 14.1), the onset of each external stimulus initiates a sequence of precisely-timed short-duration internal signals that continues until the external stimulus ends.[5] This is like assuming the animal's nervous system has a clock that keeps precise track of time during stimulus presentations; it is what engineers call a "tapped delay line." Like the presence representation, the CSC representation is unrealistic as a hypothesis about how the brain internally represents stimuli, but Ludvig et al. (2012) call it a "useful fiction" because it can reveal details of how the TD model works when relatively unconstrained by the stimulus representation. The CSC representation is also used in most TD models of dopamine-producing neurons in the brain, a topic we take up in Chapter 15. The CSC representation is often viewed as an essential part of the TD model, although this view is mistaken.

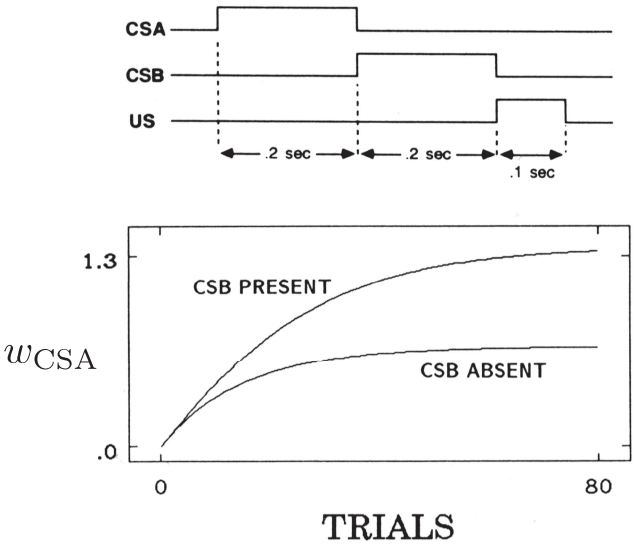
The MS representation (center column of Figure 14.1) is like the CSC representation in that each external stimulus initiates a cascade of internal stimuli, but in this case the internal stimuli—the microstimuli—are not of such limited and non-overlapping form; they are extended over time and overlap. As time elapses from stimulus onset, different sets of microstimuli become more or less active, and each subsequent microstimulus becomes progressively wider in time and reaches a lower maximal level. Of course, there are many MS representations depending on the nature of the microstimuli, and a number of examples of MS representations have been studied in the literature, in some cases along with proposals for how an animal's brain might generate them (see the Bibliographic and Historical Comments at the end of this chapter). MS representations are more realistic than the presence or CSC representations as hypotheses about neural representations of stimuli, and they allow the behavior of the TD model to be related to a broader collection of phenomena observed in animal experiments. In particular, by assuming that cascades of microstimuli are initiated by USs as well as by CSs, and by studying the significant effects on learning of interactions between microstimuli, eligibility traces, and discounting, the TD model is helping to frame hypotheses to account for many of the subtle phenomena of classical conditioning and how an animal's brain might produce them. We say more about this below, particularly in Chapter 15 where we discuss reinforcement learning and neuroscience.

Even with the simple presence representation, however, the TD model produces all the basic properties of classical conditioning that are accounted for by the Rescorla–Wagner model, plus features of conditioning that are beyond the scope of trial-level models. For example, as we have already mentioned, a conspicuous feature of classical conditioning is that the US generally must begin *after* the onset of a neutral stimulus for conditioning to occur, and that after conditioning, the CR begins *before* the appearance of the US. In other words, conditioning generally requires a positive ISI, and the CR generally anticipates the US. How the strength of conditioning (e.g., the percentage of CRs elicited by a CS) depends on the ISI varies substantially across species and response systems, but it typically has the following properties: it is negligible for a zero or negative ISI, i.e., when

[5] In our formalism, for each CS component $\mathrm{CS_i}$ present on a trial, and for each time step t during a trial, there is a separate feature x_i^t, where $x_i^t(S_{t'}) = 1$ if $t = t'$ for any t' at which $\mathrm{CS_i}$ is present, and equals 0 otherwise. This is different from the CSC representation in Sutton and Barto (1990) in which there are the same distinct features for each time step but no reference to external stimuli; hence the name complete serial compound.

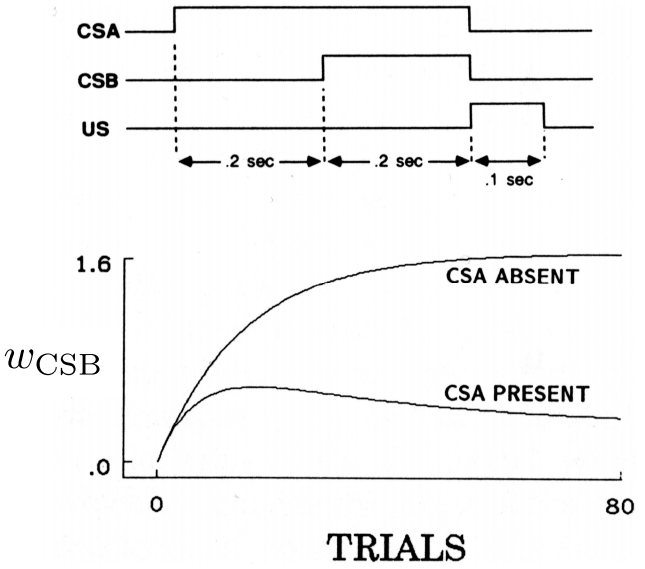
the US onset occurs simultaneously with, or earlier than, the CS onset (although research has found that associative strengths sometimes increase slightly or become negative with negative ISIs); it increases to a maximum at a positive ISI where conditioning is most effective; and it then decreases to zero after an interval that varies widely with response systems. The precise shape of this dependency for the TD model depends on the values of its parameters and details of the stimulus representation, but these basic features of ISI-dependency are core properties of the TD model.

One of the theoretical issues arising with serial-compound conditioning, that is, conditioning with a compound CS whose components occur in a sequence, concerns the facilitation of remote associations. It has been found that if the empty trace interval between a first CS (CSA) and the US is filled with a second CS (CSB) to form a serial-compound stimulus, then conditioning to CSA is facilitated. Shown to the right is the behavior of the TD model with the presence representation in a simulation of such an experiment whose timing details are shown above. Consistent with the experimental results (Kehoe, 1982), the model shows facilitation of both the rate of conditioning and the asymptotic level of conditioning of the first CS due to the presence of the second CS.



Facilitation of remote associations in the TD model

A well-known demonstration of the effects on conditioning of temporal relationships among stimuli within a trial is an experiment by Egger and Miller (1962) that involved two overlapping CSs in a delay configuration as shown to the right (top). Although CSB was in a better temporal relationship with the US, the presence of CSA substantially reduced conditioning to CSB as compared to controls in which CSA was absent. Directly to the right is shown the same result being generated by the TD model in a simulation of this experiment with the presence representation.



The Egger-Miller effect in the TD model

The TD model accounts for blocking because it is an error-correcting learning

rule like the Rescorla–Wagner model. Beyond accounting for basic blocking results, however, the TD model predicts (with the presence representation and more complex representations as well) that blocking is reversed if the blocked stimulus is moved earlier in time (like CSA in the diagram to the right) so that its onset occurs before the onset of the blocking stimulus. This feature of the TD model's behavior deserves attention because it had not been observed at the time of the model's introduction. Recall that in blocking, if an animal has already learned that one CS predicts a US, then learning that a newly-added second CS also predicts the US is much reduced, i.e., is blocked. But if the newly-added second CS begins earlier than the pretrained CS, then—according to the TD model—learning to the newly-added CS is not blocked. In fact, as training continues and the newly-added CS gains associative strength, the pretrained CS loses associative strength. The behavior of the TD model under these conditions is shown in the lower part of Figure 14.2.

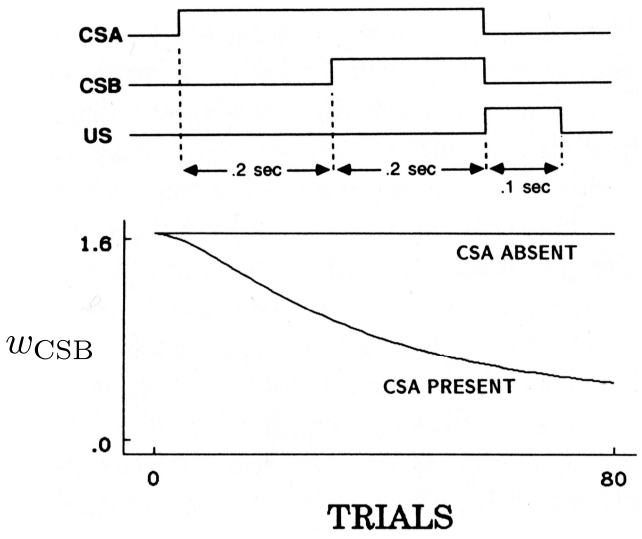


Figure 14.2: Temporal primacy overriding blocking in the TD model.

This simulation experiment differed from the Egger-Miller experiment (bottom of the preceding page) in that the shorter CS with the later onset was given prior training until it was fully associated with the US. This surprising prediction led Kehoe, Schreurs, and Graham (1987) to conduct the experiment using the well-studied rabbit nictitating membrane preparation. Their results confirmed the model's prediction, and they noted that non-TD models have considerable difficulty explaining their data.

With the TD model, an earlier predictive stimulus takes precedence over a later predictive stimulus because, like all the prediction methods described in this book, the TD model is based on the backing-up or bootstrapping idea: updates to associative strengths shift the strengths at a particular state toward the strength at later states. Another consequence of bootstrapping is that the TD model provides an account of higher-order conditioning, a feature of classical conditioning that is beyond the scope of the Rescorla-Wagner and similar models. As we described above, higher-order conditioning is the phenomenon in which a previously-conditioned CS can act as a US in conditioning another initially neutral stimulus. Figure 14.3 shows the behavior of the TD model (again with the presence representation) in a higher-order conditioning experiment—in this case it is second-order conditioning. In the first phase (not shown in the figure), CSB is trained to predict a US so that its associative strength increases, here to 1.65. In the second phase, CSA is paired with CSB in the absence of the US, in the sequential arrangement shown at the top of the figure. CSA acquires associative strength even though it is never paired with the US. With continued training, CSA's associative strength reaches a peak and then decreases because the associative strength of CSB, the secondary reinforcer, decreases so that it loses its ability to provide secondary reinforcement. CSB's associative

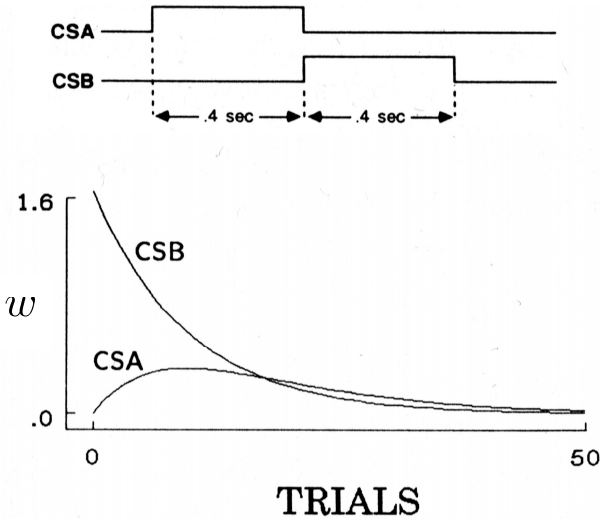


Figure 14.3: Second-order conditioning with the TD model.

strength decreases because the US does not occur in these higher-order conditioning trials. These are *extinction trials* for CSB because its predictive relationship to the US is disrupted so that its ability to act as a reinforcer decreases. This same pattern is seen in animal experiments. This extinction of conditioned reinforcement in higher-order conditioning trials makes it difficult to demonstrate higher-order conditioning unless the original predictive relationships are periodically refreshed by occasionally inserting first-order trials.

The TD model produces an analog of second- and higher-order conditioning because $\gamma\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t)$ appears in the TD error δ_t (14.5). Due to the first phase of learning, $\gamma\hat{v}(S_{t+1},\mathbf{w}_t)$ may differ from $\hat{v}(S_t,\mathbf{w}_t)$, making δ_t non-zero (a temporal difference). This difference has the same status as R_{t+1} in (14.5), implying that as far as learning is concerned there is no difference between a temporal difference and the occurrence of a US. In fact, this feature of the TD algorithm is one of the major reasons for its development, which we now understand through its connection to dynamic programming as described in Chapter 6. Bootstrapping values is intimately related to second-order, and higher-order, conditioning.

In the examples of the TD model's behavior described above, we examined only the changes in the associative strengths of the CS components; we did not look at what the model predicts about properties of an animal's conditioned responses (CRs): their timing, shape, and how they develop over conditioning trials. These properties depend on the species, the response system being observed, and parameters of the conditioning trials, but in many experiments with different animals and different response systems, the magnitude of the CR, or the probability of a CR, increases as the expected time of the US approaches. For example, in classical conditioning of a rabbit's nictitating membrane response that we mentioned above, over conditioning trials the delay from CS onset to when the nictitating membrane begins to move across the eye decreases over trials, and the amplitude of this anticipatory closure gradually increases over the interval between the CS and the US until the membrane reaches maximal closure at the expected time of the US. The timing and shape of this CR is critical to its adaptive significance—covering the eye too early reduces vision (even though the nictitating membrane is translucent), while covering it too late is of little protective value. Capturing CR features like these is challenging for models of classical conditioning.

The TD model does not include as part of its definition any mechanism for translating the time course of the US prediction, $\hat{v}(S_t,\mathbf{w}_t)$, into a profile that can be compared

with the properties of an animal's CR. The simplest choice is to let the time course of a simulated CR equal the time course of the US prediction. In this case, features of simulated CRs and how they change over trials depend only on the stimulus representation chosen and the values of the model's parameters α, γ, and λ.

Figure 14.4 shows the time courses of US predictions at different points during learning with the three representations shown in Figure 14.1. For these simulations the US occurred 25 time steps after the onset of the CS, and $\alpha = .05$, $\lambda = .95$ and $\gamma = .97$. With the CSC representation (Figure 14.4 left), the curve of the US prediction formed by the TD model increases exponentially throughout the interval between the CS and the US until it reaches a maximum exactly when the US occurs (at time step 25). This exponential increase is the result of discounting in the TD model learning rule. With the presence representation (Figure 14.4 middle), the US prediction is nearly constant while the stimulus is present because there is only one weight, or associative strength, to be learned for each stimulus. Consequently, the TD model with the presence representation cannot recreate many features of CR timing. With an MS representation (Figure 14.4 right), the development of the TD model's US prediction is more complicated. After 200 trials the prediction's profile is a reasonable approximation of the US prediction curve produced with the CSC representation.

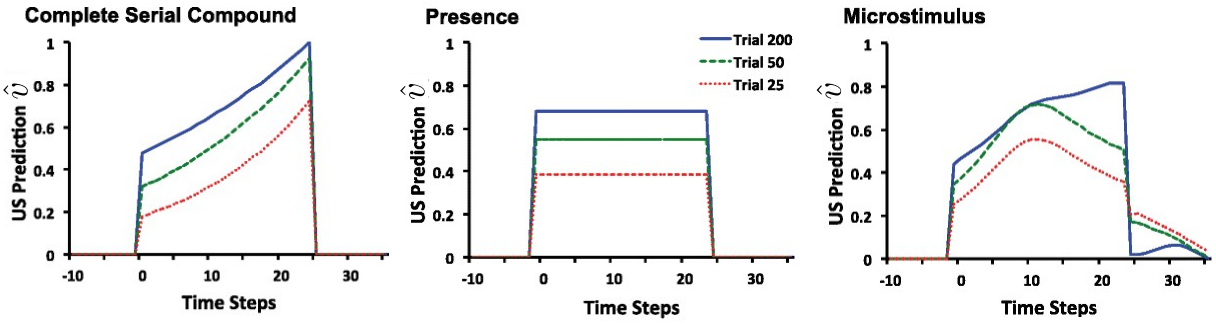


Figure 14.4: Time course of US prediction over the course of acquisition for the TD model with three different stimulus representations. Left: With the complete serial compound (CSC), the US prediction increases exponentially through the interval, peaking at the time of the US. At asymptote (trial 200), the US prediction peaks at the US intensity (1 in these simulations). Middle: With the presence representation, the US prediction converges to an almost constant level. This constant level is determined by the US intensity and the length of the CS–US interval. Right: With the microstimulus representation, at asymptote, the TD model approximates the exponentially increasing time course depicted with the CSC through a linear combination of the different microstimuli. Adapted with minor changes from *Learning & Behavior*, Evaluating the TD Model of Classical Conditioning, volume 40, 2012, E. A. Ludvig, R. S. Sutton, E. J. Kehoe. With permission of Springer.

The US prediction curves shown in Figure 14.4 were not intended to precisely match profiles of CRs as they develop during conditioning in any particular animal experiment, but they illustrate the strong influence that the stimulus representation has on predictions derived from the TD model. Further, although we can only mention it here, how the

stimulus representation interacts with discounting and eligibility traces is important in determining properties of the US prediction profiles produced by the TD model. Another dimension beyond what we can discuss here is the influence of different response-generation mechanisms that translate US predictions into CR profiles; the profiles shown in Figure 14.4 are "raw" US prediction profiles. Even without any special assumption about how an animal's brain might produce overt responses from US predictions, however, the profiles in Figure 14.4 for the CSC and MS representations increase as the time of the US approaches and reach a maximum at the time of the US, as is seen in many animal conditioning experiments.

The TD model, when combined with particular stimulus representations and response-generation mechanisms, is able to account for a surprisingly wide range of phenomena observed in animal classical conditioning experiments, but it is far from being a perfect model. To generate other details of classical conditioning the model needs to be extended, perhaps by adding model-based elements and mechanisms for adaptively altering some of its parameters. Other approaches to modeling classical conditioning depart significantly from the Rescorla–Wagner-style error-correction process. Bayesian models, for example, work within a probabilistic framework in which experience revises probability estimates. All of these models usefully contribute to our understanding of classical conditioning.

Perhaps the most notable feature of the TD model is that it is based on a theory—the theory we have described in this book—that suggests an account of what an animal's nervous system is *trying to do* while undergoing conditioning: it is trying to form accurate *long-term predictions*, consistent with the limitations imposed by the way stimuli are represented and how the nervous system works. In other words, it suggests a *normative account* of classical conditioning in which long-term, instead of immediate, prediction is a key feature.

The development of the TD model of classical conditioning is one instance in which the explicit goal was to model some of the details of animal learning behavior. In addition to its standing as an *algorithm*, then, TD learning is also the basis of this *model* of aspects of biological learning. As we discuss in Chapter 15, TD learning has also turned out to underlie an influential model of the activity of neurons that produce dopamine, a chemical in the brain of mammals that is deeply involved in reward processing. These are instances in which reinforcement learning theory makes detailed contact with animal behavioral and neural data.

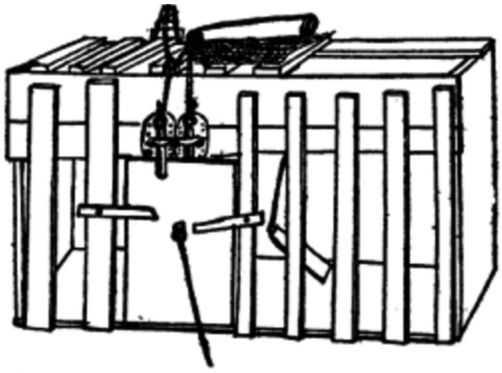
We now turn to considering correspondences between reinforcement learning and animal behavior in instrumental conditioning experiments, the other major type of laboratory experiment studied by animal learning psychologists.

14.3 Instrumental Conditioning

In *instrumental conditioning* experiments learning depends on the consequences of behavior: the delivery of a reinforcing stimulus is contingent on what the animal does. In classical conditioning experiments, in contrast, the reinforcing stimulus—the US—is delivered independently of the animal's behavior. Instrumental conditioning is usually considered to be the same as *operant conditioning*, the term B. F. Skinner (1938, 1963)

introduced for experiments with behavior-contingent reinforcement, though the experiments and theories of those who use these two terms differ in a number of ways, some of which we touch on below. We will exclusively use the term instrumental conditioning for experiments in which reinforcement is contingent upon behavior. The roots of instrumental conditioning go back to experiments performed by the American psychologist Edward Thorndike one hundred years before publication of the first edition of this book.

Thorndike observed the behavior of cats when they were placed in "puzzle boxes," such as the one at the right, from which they could escape by appropriate actions. For example, a cat could open the door of one box by performing a sequence of three separate actions: depressing a platform at the back of the box, pulling a string by clawing at it, and pushing a bar up or down. When first placed in a puzzle box, with food visible outside, all but a few of Thorndike's cats displayed "evident signs of discomfort" and extraordinarily vigorous activity "to strive instinctively to escape from confinement" (Thorndike, 1898).



One of Thorndike's puzzle boxes.
Reprinted from Thorndike, Animal Intelligence: An Experimental Study of the Associative Processes in Animals, *The Psychological Review, Series of Monograph Supplements* II(4), Macmillan, New York, 1898.

In experiments with different cats and boxes with different escape mechanisms, Thorndike recorded the amounts of time each cat took to escape over multiple experiences in each box. He observed that the time almost invariably decreased with successive experiences, for example, from 300 seconds to 6 or 7 seconds. He described cats' behavior in a puzzle box like this:

> The cat that is clawing all over the box in her impulsive struggle will probably claw the string or loop or button so as to open the door. And gradually all the other non-successful impulses will be stamped out and the particular impulse leading to the successful act will be stamped in by the resulting pleasure, until, after many trials, the cat will, when put in the box, immediately claw the button or loop in a definite way. (Thorndike 1898, p. 13)

These and other experiments (some with dogs, chicks, monkeys, and even fish) led Thorndike to formulate a number of "laws" of learning, the most influential being the *Law of Effect*. This law describes what is generally known as learning by trial and error. As we mentioned in Chapter 1, many aspects of the Law of Effect have generated controversy, and its details have been modified over the years. Still the law—in one form or another—expresses an enduring principle of learning.

Essential features of reinforcement learning algorithms correspond to features of animal learning described by the Law of Effect. First, reinforcement learning algorithms are *selectional*, meaning that they try alternatives and select among them by comparing their consequences. Second, reinforcement learning algorithms are *associative*, meaning that the alternatives found by selection are associated with particular situations, or states, to form the agent's policy. Like learning described by the Law of Effect, reinforcement

learning is not just the process of *finding* actions that produce a lot of reward, but also of *connecting* these actions to situations or states. Thorndike used the phrase learning by "selecting and connecting" (Hilgard, 1956). Natural selection in evolution is a prime example of a selectional process, but it is not associative (at least as it is commonly understood); supervised learning is associative, but it is not selectional because it relies on instructions that directly tell the agent how to change its behavior.

In computational terms, the Law of Effect describes an elementary way of combining *search* and *memory*: search in the form of trying and selecting among many actions in each situation, and memory in the form of associations linking situations with the actions found—so far—to work best in those situations. Search and memory are essential components of all reinforcement learning algorithms, whether memory takes the form of an agent's policy, value function, or environment model.

A reinforcement learning algorithm's need to search means that it has to explore in some way. Animals clearly explore as well, and early animal learning researchers disagreed about the degree of guidance an animal uses in selecting its actions in situations like Thorndike's puzzle boxes. Are actions the result of "absolutely random, blind groping" (Woodworth, 1938, p. 777), or is there some degree of guidance, either from prior learning, reasoning, or other means? Although some thinkers, including Thorndike, seem to have taken the former position, others favored more deliberate exploration. Reinforcement learning algorithms allow wide latitude for how much guidance an agent can employ in selecting actions. The forms of exploration we have used in the algorithms presented in this book, such as ε-greedy and upper-confidence-bound action selection, are merely among the simplest. More sophisticated methods are possible, with the only stipulation being that there has to be *some* form of exploration for the algorithms to work effectively.

The feature of our treatment of reinforcement learning allowing the set of actions available at any time to depend on the environment's current state echoes something Thorndike observed in his cats' puzzle-box behaviors. The cats selected actions from those that they instinctively perform in their current situation, which Thorndike called their "instinctual impulses." First placed in a puzzle box, a cat instinctively scratches, claws, and bites with great energy: a cat's instinctual responses to finding itself in a confined space. Successful actions are selected from these and not from every possible action or activity. This is like the feature of our formalism where the action selected from a state s belongs to a set of admissible actions, $\mathcal{A}(s)$. Specifying these sets is an important aspect of reinforcement learning because it can radically simplify learning. They are like an animal's instinctual impulses. On the other hand, Thorndike's cats might have been exploring according to an instinctual context-specific *ordering* over actions rather than by just selecting from a set of instinctual impulses. This is another way to make reinforcement learning easier.

Among the most prominent animal learning researchers influenced by the Law of Effect were Clark Hull (e.g., Hull, 1943) and B. F. Skinner (e.g., Skinner, 1938). At the center of their research was the idea of selecting behavior on the basis of its consequences. Reinforcement learning has features in common with Hull's theory, which included eligibility-like mechanisms and secondary reinforcement to account for the ability to learn when there is a significant time interval between an action and the consequent reinforcing

stimulus (see Section 14.4). Randomness also played a role in Hull's theory through what he called "behavioral oscillation" to introduce exploratory behavior.

Skinner did not fully subscribe to the memory aspect of the Law of Effect. Being averse to the idea of associative linkages, he instead emphasized selection from spontaneously-emitted behavior. He introduced the term "operant" to emphasize the key role of an action's effects on an animal's environment. Unlike the experiments of Thorndike and others, which consisted of sequences of separate trials, Skinner's operant conditioning experiments allowed animal subjects to behave for extended periods of time without interruption. He invented the operant conditioning chamber, now called a "Skinner box," the most basic version of which contains a lever or key that an animal can press to obtain a reward, such as food or water, which would be delivered according to a well-defined rule, called a reinforcement schedule. By recording the cumulative number of lever presses as a function of time, Skinner and his followers could investigate the effect of different reinforcement schedules on the animal's rate of lever-pressing. Modeling results from experiments likes these using the reinforcement learning principles we present in this book is not well developed, but we mention some exceptions in the Bibliographic and Historical Remarks section at the end of this chapter.

Another of Skinner's contributions resulted from his recognition of the effectiveness of training an animal by reinforcing successive approximations of the desired behavior, a process he called *shaping*. Although this technique had been used by others, including Skinner himself, its significance was impressed upon him when he and colleagues were attempting to train a pigeon to bowl by swiping a wooden ball with its beak. After waiting for a long time without seeing any swipe that they could reinforce, they

> ... decided to reinforce any response that had the slightest resemblance to a swipe—perhaps, at first, merely the behavior of looking at the ball—and then to select responses which more closely approximated the final form. The result amazed us. In a few minutes, the ball was caroming off the walls of the box as if the pigeon had been a champion squash player. (Skinner, 1958, p. 94)

Not only did the pigeon learn a behavior that is unusual for pigeons, it learned quickly through an interactive process in which its behavior and the reinforcement contingencies changed in response to each other. Skinner compared the process of altering reinforcement contingencies to the work of a sculptor shaping clay into a desired form. Shaping is a powerful technique for computational reinforcement learning systems as well. When it is difficult for an agent to receive any non-zero reward signal at all, either due to sparseness of rewarding situations or their inaccessibility given initial behavior, starting with an easier problem and incrementally increasing its difficulty as the agent learns can be an effective, and sometimes indispensable, strategy.

A concept from psychology that is especially relevant in the context of instrumental conditioning is *motivation*, which refers to processes that influence the direction and strength, or vigor, of behavior. Thorndike's cats, for example, were motivated to escape from puzzle boxes because they wanted the food that was sitting just outside. Obtaining this goal was rewarding to them and reinforced the actions allowing them to escape. It is difficult to link the concept of motivation, which has many dimensions, in a precise

way to reinforcement learning's computational perspective, but there are clear links with some of its dimensions.

In one sense, a reinforcement learning agent's reward signal is at the base of its motivation: the agent is motivated to maximize the total reward it receives over the long run. A key facet of motivation, then, is what makes an agent's experience rewarding. In reinforcement learning, reward signals depend on the state of the reinforcement learning agent's environment and the agent's actions. Further, as pointed out in Chapter 1, the state of the agent's environment not only includes information about what is external to the machine, like an organism or a robot, that houses the agent, but also what is internal to this machine. Some internal state components correspond to what psychologists call an animal's *motivational state*, which influences what is rewarding to the animal. For example, an animal will be more rewarded by eating when it is hungry than when it has just finished a satisfying meal. The concept of state dependence is broad enough to allow for many types of modulating influences on the generation of reward signals.

Value functions provide a further link to psychologists' concept of motivation. If the most basic motive for selecting an action is to obtain as much reward as possible, for a reinforcement learning agent that selects actions using a value function, a more proximal motive is to *ascend the gradient of its value function*, that is, to select actions expected to lead to the most highly-valued next states (or what is essentially the same thing, to select actions with the greatest action-values). For these agents, value functions are the main driving force determining the direction of their behavior.

Another dimension of motivation is that an animal's motivational state not only influences learning, but also influences the strength, or vigor, of the animal's behavior after learning. For example, after learning to find food in the goal box of a maze, a hungry rat will run faster to the goal box than one that is not hungry. This aspect of motivation does not link so cleanly to the reinforcement learning framework we present here, but in the Bibliographical and Historical Remarks section at the end of this chapter we cite several publications that propose theories of behavioral vigor based on reinforcement learning.

We turn now to the subject of learning when reinforcing stimuli occur well after the events they reinforce. The mechanisms used by reinforcement learning algorithms to enable learning with delayed reinforcement—eligibility traces and TD learning—closely correspond to psychologists' hypotheses about how animals can learn under these conditions.

14.4 Delayed Reinforcement

The Law of Effect requires a backward effect on connections, and some early critics of the law could not conceive of how the present could affect something that was in the past. This concern was amplified by the fact that learning can even occur when there is a considerable delay between an action and the consequent reward or penalty. Similarly, in classical conditioning, learning can occur when US onset follows CS offset by a non-negligible time interval. We call this the problem of delayed reinforcement, which is related to what Minsky (1961) called the "credit-assignment problem for learning systems": how do you

distribute credit for success among the many decisions that may have been involved in producing it? The reinforcement learning algorithms presented in this book include two basic mechanisms for addressing this problem. The first is the use of eligibility traces, and the second is the use of TD methods to learn value functions that provide nearly immediate evaluations of actions (in tasks like instrumental conditioning experiments) or that provide immediate prediction targets (in tasks like classical conditioning experiments). Both of these methods correspond to similar mechanisms proposed in theories of animal learning.

Pavlov (1927) pointed out that every stimulus must leave a trace in the nervous system that persists for some time after the stimulus ends, and he proposed that stimulus traces make learning possible when there is a temporal gap between the CS offset and the US onset. To this day, conditioning under these conditions is called *trace conditioning* (page 344). Assuming a trace of the CS remains when the US arrives, learning occurs through the simultaneous presence of the trace and the US. We discuss some proposals for trace mechanisms in the nervous system in Chapter 15.

Stimulus traces were also proposed as a means for bridging the time interval between actions and consequent rewards or penalties in instrumental conditioning. In Hull's influential learning theory, for example, "molar stimulus traces" accounted for what he called an animal's *goal gradient*, a description of how the maximum strength of an instrumentally-conditioned response decreases with increasing delay of reinforcement (Hull, 1932, 1943). Hull hypothesized that an animal's actions leave internal stimuli whose traces decay exponentially as functions of time since an action was taken. Looking at the animal learning data available at the time, he hypothesized that the traces effectively reach zero after 30 to 40 seconds.

The eligibility traces used in the algorithms described in this book are like Hull's traces: they are decaying traces of past state visitations, or of past state–action pairs. Eligibility traces were introduced by Klopf (1972) in his neuronal theory in which they are temporally-extended traces of past activity at synapses, the connections between neurons. Klopf's traces are more complex than the exponentially-decaying traces our algorithms use, and we discuss this more when we take up his theory in Section 15.9.

To account for goal gradients that extend over longer time periods than spanned by stimulus traces, Hull (1943) proposed that longer gradients result from conditioned reinforcement passing backwards from the goal, a process acting in conjunction with his molar stimulus traces. Animal experiments showed that if conditions favor the development of conditioned reinforcement during a delay period, learning does not decrease with increased delay as much as it does under conditions that obstruct secondary reinforcement. Conditioned reinforcement is favored if there are stimuli that regularly occur during the delay interval. Then it is as if reward is not actually delayed because there is more immediate conditioned reinforcement. Hull therefore envisioned that there is a primary gradient based on the delay of the primary reinforcement mediated by stimulus traces, and that this is progressively modified, and lengthened, by conditioned reinforcement.

Algorithms presented in this book that use both eligibility traces and value functions to enable learning with delayed reinforcement correspond to Hull's hypothesis about how animals are able to learn under these conditions. The actor–critic architecture discussed

in Sections 13.5, 15.7, and 15.8 illustrates this correspondence most clearly. The critic uses a TD algorithm to learn a value function associated with the system's current behavior, that is, to predict the current policy's return. The actor updates the current policy based on the critic's predictions, or more exactly, on changes in the critic's predictions. The TD error produced by the critic acts as a conditioned reinforcement signal for the actor, providing an immediate evaluation of performance even when the primary reward signal itself is considerably delayed. Algorithms that estimate action-value functions, such as Q-learning and Sarsa, similarly use TD learning principles to enable learning with delayed reinforcement by means of conditioned reinforcement. The close parallel between TD learning and the activity of dopamine producing neurons that we discuss in Chapter 15 lends additional support to links between reinforcement learning algorithms and this aspect of Hull's learning theory.

14.5 Cognitive Maps

Model-based reinforcement learning algorithms use environment models that have elements in common with what psychologists call *cognitive maps*. Recall from our discussion of planning and learning in Chapter 8 that by an environment model we mean anything an agent can use to predict how its environment will respond to its actions in terms of state transitions and rewards, and by planning we mean any process that computes a policy from such a model. Environment models consist of two parts: the state-transition part encodes knowledge about the effect of actions on state changes, and the reward-model part encodes knowledge about the reward signals expected for each state or each state–action pair. A model-based algorithm selects actions by using a model to predict the consequences of possible courses of action in terms of future states and the reward signals expected to arise from those states. The simplest kind of planning is to compare the predicted consequences of collections of "imagined" sequences of decisions.

Questions about whether or not animals use environment models, and if so, what are the models like and how are they learned, have played influential roles in the history of animal learning research. Some researchers challenged the then-prevailing stimulus-response (S–R) view of learning and behavior, which corresponds to the simplest model-free way of learning policies, by demonstrating *latent learning*. In the earliest latent learning experiment, two groups of rats were run in a maze. For the experimental group, there was no reward during the first stage of the experiment, but food was suddenly introduced into the goal box of the maze at the start of the second stage. For the control group, food was in the goal box throughout both stages. The question was whether or not rats in the experimental group would have learned anything during the first stage in the absence of food reward. Although the experimental rats did not *appear* to learn much during the first, unrewarded, stage, as soon as they discovered the food that was introduced in the second stage, they rapidly caught up with the rats in the control group. It was concluded that "during the non-reward period, the rats [in the experimental group] were developing a latent learning of the maze which they were able to utilize as soon as reward was introduced" (Blodgett, 1929).

Latent learning is most closely associated with the psychologist Edward Tolman, who interpreted this result, and others like it, as showing that animals could learn a "cognitive map of the environment" in the absence of rewards or penalties, and that they could use the map later when they were motivated to reach a goal (Tolman, 1948). A cognitive map could also allow a rat to plan a route to the goal that was different from the route the rat had used in its initial exploration. Explanations of results like these led to the enduring controversy lying at the heart of the behaviorist/cognitive dichotomy in psychology. In modern terms, cognitive maps are not restricted to models of spatial layouts but are more generally environment models, or models of an animal's "task space" (e.g., Wilson, Takahashi, Schoenbaum, and Niv, 2014). The cognitive map explanation of latent learning experiments is analogous to the claim that animals use model-based algorithms, and that environment models can be learned even without explicit rewards or penalties. Models are then used for planning when the animal is motivated by the appearance of rewards or penalties.

Tolman's account of how animals learn cognitive maps was that they learn stimulus-stimulus, or S–S, associations by experiencing successions of stimuli as they explore an environment. In psychology this is called *expectancy theory*: given S–S associations, the occurrence of a stimulus generates an expectation about the stimulus to come next. This is much like what control engineers call *system identification*, in which a model of a system with unknown dynamics is learned from labeled training examples. In the simplest discrete-time versions, training examples are S–S′ pairs, where S is a state and S′, the subsequent state, is the label. When S is observed, the model creates the "expectation" that S′ will be observed next. Models more useful for planning involve actions as well, so that examples look like SA–S′, where S′ is expected when action A is executed in state S. It is also useful to learn how the environment generates rewards. In this case, examples are of the form S–R or SA–R, where R is a reward signal associated with S or the SA pair. These are all forms of supervised learning by which an agent can acquire cognitive-like maps whether or not it receives any non-zero reward signals while exploring its environment.

14.6 Habitual and Goal-directed Behavior

The distinction between model-free and model-based reinforcement learning algorithms corresponds to the distinction psychologists make between *habitual* and *goal-directed* control of learned behavioral patterns. Habits are behavior patterns triggered by appropriate stimuli and then performed more-or-less automatically. Goal-directed behavior, according to how psychologists use the phrase, is purposeful in the sense that it is controlled by knowledge of the value of goals and the relationship between actions and their consequences. Habits are sometimes said to be controlled by antecedent stimuli, whereas goal-directed behavior is said to be controlled by its consequences (Dickinson, 1980, 1985). Goal-directed control has the advantage that it can rapidly change an animal's behavior when the environment changes its way of reacting to the animal's actions. While habitual behavior responds quickly to input from an accustomed environment, it is unable

to quickly adjust to changes in the environment. The development of goal-directed behavioral control was likely a major advance in the evolution of animal intelligence.

Figure 14.5 illustrates the difference between model-free and model-based decision strategies in a hypothetical task in which a rat has to navigate a maze that has distinctive goal boxes, each delivering an associated reward of the magnitude shown (Figure 14.5 top). Starting at S_1, the rat has to first select left (L) or right (R) and then has to select L or R again at S_2 or S_3 to reach one of the goal boxes. The goal boxes are the terminal states of each episode of the rat's episodic task. A model-free strategy (Figure 14.5 lower left) relies on stored values for state–action pairs. These action values are estimates of the highest return the rat can expect for each action taken from each (nonterminal) state. They are obtained over many trials of running the maze from start to finish. When the action values have become good enough estimates of the optimal returns, the rat just has

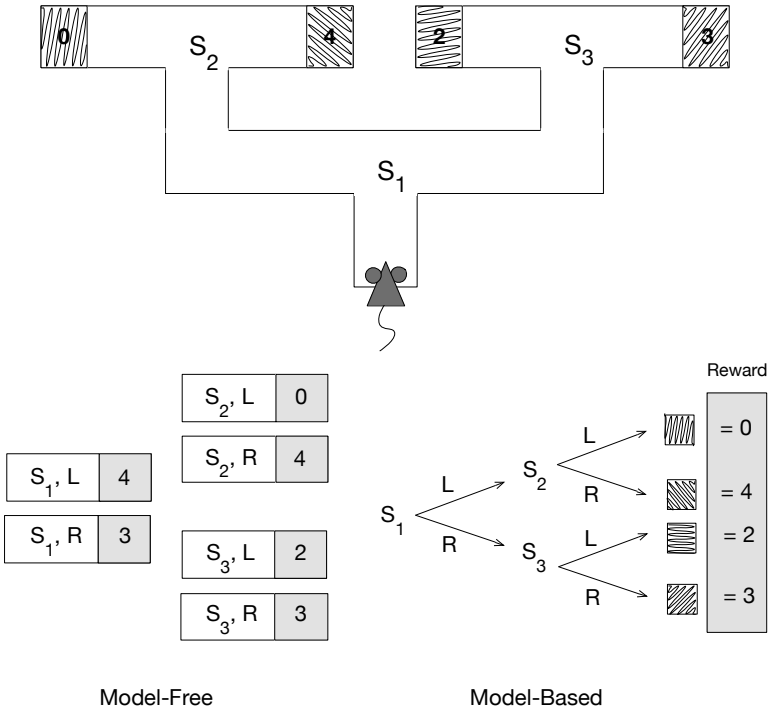


Figure 14.5: Model-based and model-free strategies to solve a hypothetical sequential action-selection problem. Top: a rat navigates a maze with distinctive goal boxes, each associated with a reward having the value shown. Lower left: a model-free strategy relies on stored action values for all the state–action pairs obtained over many learning trials. To make decisions the rat just has to select at each state the action with the largest action value for that state. Lower right: in a model-based strategy, the rat learns an environment model, consisting of knowledge of state–action-next-state transitions and a reward model consisting of knowledge of the reward associated with each distinctive goal box. The rat can decide which way to turn at each state by using the model to simulate sequences of action choices to find a path yielding the highest return. Adapted from *Trends in Cognitive Science*, volume 10, number 8, Y. Niv, D. Joel, and P. Dayan, A Normative Perspective on Motivation, p. 376, 2006, with permission from Elsevier.

to select at each state the action with the largest action value in order to make optimal decisions. In this case, when the action-value estimates become accurate enough, the rat selects L from S_1 and R from S_2 to obtain the maximum return of 4. A different model-free strategy might simply rely on a cached policy instead of action values, making direct links from S_1 to L and from S_2 to R. In neither of these strategies do decisions rely on an environment model. There is no need to consult a state-transition model, and no connection is required between the features of the goal boxes and the rewards they deliver.

Figure 14.5 (lower right) illustrates a model-based strategy. It uses an environment model consisting of a state-transition model and a reward model. The state-transition model is shown as a decision tree, and the reward model associates the distinctive features of the goal boxes with the rewards to be found in each. (The rewards associated with states S_1, S_2, and S_3 are also part of the reward model, but here they are zero and are not shown.) A model-based agent can decide which way to turn at each state by using the model to simulate sequences of action choices to find a path yielding the highest return. In this case the return is the reward obtained from the outcome at the end of the path. Here, with a sufficiently accurate model, the rat would select L and then R to obtain a return of 4. Comparing the predicted returns of simulated paths is a simple form of planning, which can be done in a variety of ways as discussed in Chapter 8.

When the environment of a model-free agent changes the way it reacts to the agent's actions, the agent has to acquire new experience in the changed environment during which it can update its policy and/or value function. In the model-free strategy shown in Figure 14.5 (lower left), for example, if one of the goal boxes were to somehow shift to delivering a different reward, the rat would have to traverse the maze, possibly many times, to experience the new reward upon reaching that goal box, all the while updating either its policy or its action-value function (or both) based on this experience. The key point is that for a model-free agent to change the action its policy specifies for a state, or to change an action value associated with a state, it has to move to that state, act from it, possibly many times, and experience the consequences of its actions.

A model-based agent can accommodate changes in its environment without this kind of 'personal experience' with the states and actions affected by the change. A change in its model automatically (through planning) changes its policy. Planning can determine the consequences of changes in the environment that have never been linked together in the agent's own experience. For example, again referring to the maze task of Figure 14.5, imagine that a rat with a previously learned transition and reward model is placed directly in the goal box to the right of S_2 to find that the reward available there now has value 1 instead of 4. The rat's reward model will change even though the action choices required to find that goal box in the maze were not involved. The planning process will bring knowledge of the new reward to bear on maze running without the need for additional experience in the maze; in this case changing the policy to right turns at both S_1 and S_3 to obtain a return of 3.

Exactly this logic is the basis of *outcome-devaluation experiments* with animals. Results from these experiments provide insight into whether an animal has learned a habit or if its behavior is under goal-directed control. Outcome-devaluation experiments are like latent-learning experiments in that the reward changes from one stage to the next. After

an initial rewarded stage of learning, the reward value of an outcome is changed, including being shifted to zero or even to a negative value.

An early important experiment of this type was conducted by Adams and Dickinson (1981). They trained rats via instrumental conditioning until the rats energetically pressed a lever for sucrose pellets in a training chamber. The rats were then placed in the same chamber with the lever retracted and allowed non-contingent food, meaning that pellets were made available to them independently of their actions. After 15-minutes of this free-access to the pellets, rats in one group were injected with the nausea-inducing poison lithium chloride. This was repeated for three sessions, in the last of which none of the injected rats consumed any of the non-contingent pellets, indicating that the reward value of the pellets had been decreased—the pellets had been devalued. In the next stage taking place a day later, the rats were again placed in the chamber and given a session of extinction training, meaning that the response lever was back in place but disconnected from the pellet dispenser so that pressing it did not release pellets. The question was whether the rats that had the reward value of the pellets decreased would lever-press less than rats that did not have the reward value of the pellets decreased, even without experiencing the devalued reward as a result of lever-pressing. It turned out that the injected rats had significantly lower response rates than the non-injected rats *right from the start of the extinction trials.*

Adams and Dickinson concluded that the injected rats associated lever pressing with consequent nausea by means of a cognitive map linking lever pressing to pellets, and pellets to nausea. Hence, in the extinction trials, the rats "knew" that the consequences of pressing the lever would be something they did not want, and so they reduced their lever-pressing right from the start. The important point is that they reduced lever-pressing without ever having experienced lever-pressing directly followed by being sick: no lever was present when they were made sick. They seemed able to combine knowledge of the outcome of a behavioral choice (pressing the lever will be followed by getting a pellet) with the reward value of the outcome (pellets are to be avoided) and hence could alter their behavior accordingly. Not every psychologist agrees with this "cognitive" account of this kind of experiment, and it is not the only possible way to explain these results, but the model-based planning explanation is widely accepted.

Nothing prevents an agent from using both model-free and model-based algorithms, and there are good reasons for using both. We know from our own experience that with enough repetition, goal-directed behavior tends to turn into habitual behavior. Experiments show that this happens for rats too. Adams (1982) conducted an experiment to see if extended training would convert goal-directed behavior into habitual behavior. He did this by comparing the effect of outcome devaluation on rats that experienced different amounts of training. If extended training made the rats less sensitive to devaluation compared to rats that received less training, this would be evidence that extended training made the behavior more habitual. Adams' experiment closely followed the Adams and Dickinson (1981) experiment just described. Simplifying a bit, rats in one group were trained until they made 100 rewarded lever-presses, and rats in the other group—the overtrained group—were trained until they made 500 rewarded lever-presses. After this training, the reward value of the pellets was decreased (using lithium chloride injections) for rats

in both groups. Then both groups of rats were given a session of extinction training. Adams' question was whether devaluation would effect the rate of lever-pressing for the overtrained rats less than it would for the non-overtrained rats, which would be evidence that extended training reduces sensitivity to outcome devaluation. It turned out that devaluation strongly decreased the lever-pressing rate of the non-overtrained rats. For the overtrained rats, in contrast, devaluation had little effect on their lever-pressing; in fact, if anything, it made it more vigorous. (The full experiment included control groups showing that the different amounts of training did not by themselves significantly effect lever-pressing rates after learning.) This result suggested that while the non-overtrained rats were acting in a goal-directed manner sensitive to their knowledge of the outcome of their actions, the overtrained rats had developed a lever-pressing habit.

Viewing this and other results like it from a computational perspective provides insight as to why one might expect animals to behave habitually in some circumstances but in a goal-directed way in others, and why they shift from one mode of control to another as they continue to learn. While animals undoubtedly use algorithms that do not exactly match those we have presented in this book, one can gain insight into animal behavior by considering the tradeoffs that various reinforcement learning algorithms imply. An idea developed by computational neuroscientists Daw, Niv, and Dayan (2005) is that animals use both model-free and model-based processes. Each process proposes an action, and the action chosen for execution is the one proposed by the process judged to be the more trustworthy of the two as determined by measures of confidence that are maintained throughout learning. Early in learning the planning process of a model-based system is more trustworthy because it chains together short-term predictions which can become accurate with less experience than long-term predictions of the model-free process. But with continued experience, the model-free process becomes more trustworthy because planning is prone to making mistakes due to model inaccuracies and short-cuts necessary to make planning feasible, such as various forms of "tree-pruning": the removal of unpromising search tree branches. According to this idea one would expect a shift from goal-directed behavior to habitual behavior as more experience accumulates. Other ideas have been proposed for how animals arbitrate between goal-directed and habitual control, and both behavioral and neuroscience research continues to examine this and related questions.

The distinction between model-free and model-based algorithms is proving to be useful for this research. One can examine the computational implications of these types of algorithms in abstract settings that expose basic advantages and limitations of each type. This serves both to suggest and to sharpen questions that guide the design of experiments necessary for increasing psychologists' understanding of habitual and goal-directed behavioral control.

14.7 Summary

Our goal in this chapter has been to discuss correspondences between reinforcement learning and the experimental study of animal learning in psychology. We emphasized at the outset that reinforcement learning as described in this book is not intended

to model details of animal behavior. It is an abstract computational framework that explores idealized situations from the perspective of artificial intelligence and engineering. But many of the basic reinforcement learning algorithms were inspired by psychological theories, and in some cases, these algorithms have contributed to the development of new animal learning models. This chapter described the most conspicuous of these correspondences.

The distinction in reinforcement learning between algorithms for prediction and algorithms for control parallels animal learning theory's distinction between classical, or Pavlovian, conditioning and instrumental conditioning. The key difference between instrumental and classical conditioning experiments is that in the former the reinforcing stimulus is contingent upon the animal's behavior, whereas in the latter it is not. Learning to predict via a TD algorithm corresponds to classical conditioning, and we described the *TD model of classical conditioning* as one instance in which reinforcement learning principles account for some details of animal learning behavior. This model generalizes the influential Rescorla–Wagner model by including the temporal dimension where events within individual trials influence learning, and it provides an account of second-order conditioning, where predictors of reinforcing stimuli become reinforcing themselves. It also is the basis of an influential view of the activity of dopamine neurons in the brain, something we take up in Chapter 15.

Learning by trial and error is at the base of the control aspect of reinforcement learning. We presented some details about Thorndike's experiments with cats and other animals that led to his *Law of Effect*, which we discussed here and in Chapter 1 (page 15). We pointed out that in reinforcement learning, exploration does not have to be limited to "blind groping"; trials can be generated by sophisticated methods using innate and previously learned knowledge as long as there is *some* exploration. We discussed the training method B. F. Skinner called *shaping* in which reward contingencies are progressively altered to train an animal to successively approximate a desired behavior. Shaping is not only indispensable for animal training, it is also an effective tool for training reinforcement learning agents. There is also a connection to the idea of an animal's motivational state, which influences what an animal will approach or avoid and what events are rewarding or punishing for the animal.

The reinforcement learning algorithms presented in this book include two basic mechanisms for addressing the problem of delayed reinforcement: eligibility traces and value functions learned via TD algorithms. Both mechanisms have antecedents in theories of animal learning. Eligibility traces are similar to stimulus traces of early theories, and value functions correspond to the role of secondary reinforcement in providing nearly immediate evaluative feedback.

The next correspondence the chapter addressed is that between reinforcement learning's environment models and what psychologists call *cognitive maps*. Experiments conducted in the mid 20th century purported to demonstrate the ability of animals to learn cognitive maps as alternatives to, or as additions to, state–action associations, and later use them to guide behavior, especially when the environment changes unexpectedly. Environment models in reinforcement learning are like cognitive maps in that they can be learned by supervised learning methods without relying on reward signals, and then they can be used later to plan behavior.

Reinforcement learning's distinction between *model-free* and *model-based* algorithms corresponds to the distinction in psychology between *habitual* and *goal-directed* behavior. Model-free algorithms make decisions by accessing information that has been stored in a policy or an action-value function, whereas model-based methods select actions as the result of planning ahead using a model of the agent's environment. Outcome-devaluation experiments provide information about whether an animal's behavior is habitual or under goal-directed control. Reinforcement learning theory has helped clarify thinking about these issues.

Animal learning clearly informs reinforcement learning, but as a type of machine learning, reinforcement learning is directed toward designing and understanding effective learning algorithms, not toward replicating or explaining details of animal behavior. We focused on aspects of animal learning that relate in clear ways to methods for solving prediction and control problems, highlighting the fruitful two-way flow of ideas between reinforcement learning and psychology without venturing deeply into many of the behavioral details and controversies that have occupied the attention of animal learning researchers. Future development of reinforcement learning theory and algorithms will likely exploit links to many other features of animal learning as the computational utility of these features becomes better appreciated. We expect that a flow of ideas between reinforcement learning and psychology will continue to bear fruit for both disciplines.

Many connections between reinforcement learning and areas of psychology and other behavioral sciences are beyond the scope of this chapter. We largely omit discussing links to the psychology of decision making, which focuses on how actions are selected, or how decisions are made, *after* learning has taken place. We also do not discuss links to ecological and evolutionary aspects of behavior studied by ethologists and behavioral ecologists: how animals relate to one another and to their physical surroundings, and how their behavior contributes to evolutionary fitness. Optimization, MDPs, and dynamic programming figure prominently in these fields, and our emphasis on agent interaction with dynamic environments connects to the study of agent behavior in complex "ecologies." Multi-agent reinforcement learning, omitted in this book, has connections to social aspects of behavior. Despite the lack of treatment here, reinforcement learning should by no means be interpreted as dismissing evolutionary perspectives. Nothing about reinforcement learning implies a *tabula rasa* view of learning and behavior. Indeed, experience with engineering applications has highlighted the importance of building into reinforcement learning systems knowledge that is analogous to what evolution provides to animals.

Bibliographical and Historical Remarks

Ludvig, Bellemare, and Pearson (2011) and Shah (2012) review reinforcement learning in the contexts of psychology and neuroscience. These publications are useful companions to this chapter and the following chapter on reinforcement learning and neuroscience.

14.1 Dayan, Niv, Seymour, and Daw (2006) focused on interactions between classical and instrumental conditioning, particularly situations where classically-conditioned and instrumental responses are in conflict. They proposed a Q-learning framework for modeling aspects of this interaction. Modayil and Sutton (2014) used a mobile robot to demonstrate the effectiveness of a control method combining a fixed response with online prediction learning. Calling this *Pavlovian control*, they emphasized that it differs from the usual control methods of reinforcement learning, being based on predictively executing fixed responses and not on reward maximization. The electro-mechanical machine of Ross (1933) and especially the learning version of Walter's turtle (Walter, 1951) were very early illustrations of Pavlovian control.

14.2.1 Kamin (1968) first reported blocking, now commonly known as Kamin blocking, in classical conditioning. Moore and Schmajuk (2008) provide an excellent summary of the blocking phenomenon, the research it stimulated, and its lasting influence on animal learning theory. Gibbs, Cool, Land, Kehoe, and Gormezano (1991) describe second-order conditioning of the rabbit's nictitating membrane response and its relationship to conditioning with serial-compound stimuli. Finch and Culler (1934) reported obtaining fifth-order conditioning of a dog's foreleg withdrawal "when the *motivation* of the animal is maintained through the various orders."

14.2.2 The idea built into the Rescorla–Wagner model that learning occurs when animals are surprised is derived from Kamin (1969). Models of classical conditioning other than Rescorla and Wagner's include the models of Klopf (1988), Grossberg (1975), Mackintosh (1975), Moore and Stickney (1980), Pearce and Hall (1980), and Courville, Daw, and Touretzky (2006). Schmajuk (2008) reviews models of classical conditioning. Wagner (2008) provides a modern psychological perspective on the Rescorla-Wagner model and similar elemental theories of learning.

14.2.3 An early version of the TD model of classical conditioning appeared in Sutton and Barto (1981a), which also included the early model's prediction that temporal primacy overrides blocking, later shown by Kehoe, Schreurs, and Graham (1987) to occur in the rabbit nictitating membrane preparation. Sutton and Barto (1981a) contains the earliest recognition of the near identity between the Rescorla–Wagner model and the Least-Mean-Square (LMS), or Widrow-Hoff, learning rule (Widrow and Hoff, 1960). This early model was revised following Sutton's development of the TD algorithm (Sutton, 1984, 1988) and was first presented as the TD model in Sutton and Barto (1987) and more completely in Sutton and Barto (1990), upon which this section is largely based. Additional exploration

of the TD model and its possible neural implementation was conducted by Moore and colleagues (Moore, Desmond, Berthier, Blazis, Sutton, and Barto, 1986; Moore and Blazis, 1989; Moore, Choi, and Brunzell, 1998; Moore, Marks, Castagna, and Polewan, 2001). Klopf's (1988) drive-reinforcement theory of classical conditioning extends the TD model to address additional experimental details, such as the S-shape of acquisition curves. In some of these publications TD is taken to mean Time Derivative instead of Temporal Difference.

14.2.4 Ludvig, Sutton, and Kehoe (2012) evaluated the performance of the TD model in previously unexplored tasks involving classical conditioning and examined the influence of various stimulus representations, including the microstimulus representation that they introduced earlier (Ludvig, Sutton, and Kehoe, 2008). Earlier investigations of the influence of various stimulus representations and their possible neural implementations on response timing and topography in the context of the TD model are those of Moore and colleagues cited above. Although not in the context of the TD model, representations like the microstimulus representation of Ludvig et al. (2012) have been proposed and studied by Grossberg and Schmajuk (1989), Brown, Bullock, and Grossberg (1999), Buhusi and Schmajuk (1999), and Machado (1997). The figures on pages 353–355 are adapted from Sutton and Barto (1990).

14.3 Section 1.7 includes comments on the history of trial-and-error learning and the Law of Effect. The idea that Thorndike's cats might have been exploring according to an instinctual context-specific ordering over actions rather than by just selecting from a set of instinctual impulses was suggested by Peter Dayan (personal communication). Selfridge, Sutton, and Barto (1985) illustrated the effectiveness of shaping in a pole-balancing reinforcement learning task. Other examples of shaping in reinforcement learning are Gullapalli and Barto (1992), Mahadevan and Connell (1992), Mataric (1994), Dorigo and Colombette (1994), Saksida, Raymond, and Touretzky (1997), and Randløv and Alstrøm (1998). Ng (2003) and Ng, Harada, and Russell (1999) used the term shaping in a sense somewhat different from Skinner's, focusing on the problem of how to alter the reward signal without altering the set of optimal policies.

Dickinson and Balleine (2002) discuss the complexity of the interaction between learning and motivation. Wise (2004) provides an overview of reinforcement learning and its relation to motivation. Daw and Shohamy (2008) link motivation and learning to aspects of reinforcement learning theory. See also McClure, Daw, and Montague (2003), Niv, Joel, and Dayan (2006), Rangel, Camerer, and Montague (2008), and Dayan and Berridge (2014). McClure et al. (2003), Niv, Daw, and Dayan (2006), and Niv, Daw, Joel, and Dayan (2007) present theories of behavioral vigor related to the reinforcement learning framework.

14.4 Spence, Hull's student and collaborator at Yale, elaborated the role of higher-order reinforcement in addressing the problem of delayed reinforcement (Spence, 1947). Learning over very long delays, as in taste-aversion conditioning with

delays up to several hours, led to interference theories as alternatives to decaying-trace theories (e.g., Revusky and Garcia, 1970; Boakes and Costa, 2014). Other views of learning under delayed reinforcement invoke roles for awareness and working memory (e.g., Clark and Squire, 1998; Seo, Barraclough, and Lee, 2007).

14.5 Thistlethwaite (1951) provides an extensive review of latent learning experiments up to the time of its publication. Ljung (1998) provides an overview of model learning, or system identification, techniques in engineering. Gopnik, Glymour, Sobel, Schulz, Kushnir, and Danks (2004) present a Bayesian theory about how children learn models.

14.6 Connections between habitual and goal-directed behavior and model-free and model-based reinforcement learning were first proposed by Daw, Niv, and Dayan (2005). The hypothetical maze task used to explain habitual and goal-directed behavioral control is based on the explanation of Niv, Joel, and Dayan (2006). Dolan and Dayan (2013) review four generations of experimental research related to this issue and discuss how it can move forward on the basis of reinforcement learning's model-free/model-based distinction. Dickinson (1980, 1985) and Dickinson and Balleine (2002) discuss experimental evidence related to this distinction. Donahoe and Burgos (2000) alternatively argue that model-free processes can account for the results of outcome-devaluation experiments. Dayan and Berridge (2014) argue that classical conditioning involves model-based processes. Rangel, Camerer, and Montague (2008) review many of the outstanding issues involving habitual, goal-directed, and Pavlovian modes of control.

Comments on Terminology— The traditional meaning of *reinforcement* in psychology is the strengthening of a pattern of behavior (by increasing either its intensity or frequency) as a result of an animal receiving a stimulus (or experiencing the omission of a stimulus) in an appropriate temporal relationship with another stimulus or with a response. Reinforcement produces changes that remain in future behavior. Sometimes in psychology reinforcement refers to the process of producing lasting changes in behavior, whether the changes strengthen or weaken a behavior pattern (Mackintosh, 1983). Letting reinforcement refer to weakening in addition to strengthening is at odds with the everyday meaning of reinforce, and its traditional use in psychology, but it is a useful extension that we have adopted here. In either case, a stimulus considered to be the cause of the behavioral change is called a *reinforcer*.

Psychologists do not generally use the specific phrase *reinforcement learning* as we do. Animal learning pioneers probably regarded reinforcement and learning as being synonymous, so it would be redundant to use both words. Our use of the phrase follows its use in computational and engineering research, influenced mostly by Minsky (1961). But the phrase is lately gaining currency in psychology and neuroscience, likely because strong parallels have surfaced between reinforcement learning algorithms and animal learning—parallels described in this chapter and the next.

According to common usage, a *reward* is an object or event that an animal will approach and work for. A reward may be given to an animal in recognition of its 'good'

behavior, or given in order to make the animal's behavior 'better.' Similarly, a *penalty* is an object or event that the animal usually avoids and that is given as a consequence of 'bad' behavior, usually in order to change that behavior. *Primary reward* is reward due to machinery built into an animal's nervous system by evolution to improve its chances of survival and reproduction, for example, reward produced by the taste of nourishing food, sexual contact, successful escape, and many other stimuli and events that predicted reproductive success over the animal's ancestral history. As explained in Section 14.2.1, *higher-order reward* is reward delivered by stimuli that predict primary reward, either directly or indirectly by predicting other stimuli that predict primary reward. Reward is *secondary* if its rewarding quality is the result of directly predicting primary reward.

In this book we call R_t the 'reward signal at time t,' or sometimes just the 'reward at time t,' but we do not think of it as an object or event in the agent's environment. Because R_t is a number—not an object or an event—it is more like a reward signal in neuroscience, which is a signal internal to the brain, like the activity of neurons, that influences decision making and learning. This signal might be triggered when the animal perceives an attractive (or an aversive) object, but it can also be triggered by things that do not physically exist in the animal's external environment, such as memories, ideas, or hallucinations. Because our R_t can be positive, negative, or zero, it might be better to call a negative R_t a penalty, and an R_t equal to zero a neutral signal, but for simplicity we generally avoid these terms.

In reinforcement learning, the process that generates all the R_ts defines the problem the agent is trying to solve. The agent's objective is to keep the magnitude of R_t as large as possible over time. In this respect, R_t is like primary reward for an animal if we think of the problem the animal faces as the problem of obtaining as much primary reward as possible over its lifetime (and thereby, through the prospective "wisdom" of evolution, improve its chances of solving its real problem, which is to pass its genes on to future generations). However, as we suggest in Chapter 15, it is unlikely that there is a single "master" reward signal like R_t in an animal's brain.

Not all reinforcers are rewards or penalties. Sometimes reinforcement is not the result of an animal receiving a stimulus that evaluates its behavior by labeling the behavior good or bad. A behavior pattern can be reinforced by a stimulus that arrives to an animal no matter how the animal behaved. As described in Section 14.1, whether the delivery of reinforcement depends, or does not depend, on preceding behavior is the defining difference between instrumental, or operant, conditioning experiments and classical, or Pavlovian, conditioning experiments. Reinforcement is at work in both types of experiments, but only in the former is it feedback that evaluates past behavior. (Though it has often been pointed out that even when the reinforcing US in a classical conditioning experiment is not contingent on the subject's preceding behavior, its reinforcing value can be influenced by this behavior, an example being that a closed eye makes an air puff to the eye less aversive.)

The distinction between reward signals and reinforcement signals is a crucial point when we discuss neural correlates of these signals in the next chapter. Like a reward signal, for us, the reinforcement signal at any specific time is a positive or negative number, or zero. A reinforcement signal is the major factor directing changes a learning algorithm

makes in an agent's policy, value estimates, or environment models. The definition that makes the most sense to us is that a reinforcement signal at any time is a number that multiplies (possibly along with some constants) a vector to determine parameter updates in some learning algorithm.

For some algorithms, the reward signal alone is the critical multiplier in the parameter-update equation. For these algorithms the reinforcement signal is the same as the reward signal. But for most of the algorithms we discuss in this book, reinforcement signals include terms in addition to the reward signal, an example being a TD error $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$, which is the reinforcement signal for TD state-value learning (and analogous TD errors for action-value learning). In this reinforcement signal, R_{t+1} is the *primary reinforcement* contribution, and the temporal difference in predicted values, $\gamma V(S_{t+1}) - V(S_t)$ (or an analogous temporal difference for action values), is the *conditioned reinforcement* contribution. Thus, whenever $\gamma V(S_{t+1}) - V(S_t) = 0$, δ_t signals 'pure' primary reinforcement; and whenever $R_{t+1} = 0$, it signals 'pure' conditioned reinforcement, but it often signals a mixture of these. Note as we mentioned in Section 6.1, this δ_t is not available until time $t + 1$. We therefore think of δ_t as the reinforcement signal at time $t + 1$, which is fitting because it reinforces predictions and/or actions made earlier at step t.

A possible source of confusion is the terminology used by the famous psychologist B. F. Skinner and his followers. For Skinner, positive reinforcement occurs when the consequences of an animal's behavior increase the frequency of that behavior; punishment occurs when the behavior's consequences decrease that behavior's frequency. Negative reinforcement occurs when behavior leads to the removal of an aversive stimulus (that is, a stimulus the animal does not like), thereby increasing the frequency of that behavior. Negative punishment, on the other hand, occurs when behavior leads to the removal of an appetitive stimulus (that is, a stimulus the animal likes), thereby decreasing the frequency of that behavior. We find no critical need for these distinctions because our approach is more abstract than this, with both reward and reinforcement signals allowed to take on both positive and negative values. (But note especially that when our reinforcement signal is negative, it is not the same as Skinner's negative reinforcement.)

On the other hand, it has often been pointed out that using a single number as a reward or a penalty signal, depending only on its sign, is at odds with the fact that animals' appetitive and aversive systems have qualitatively different properties and involve different brain mechanisms. This points to a direction in which the reinforcement learning framework might be developed in the future to exploit computational advantages of separate appetitive and aversive systems, but for now we are passing over these possibilities.

Another discrepancy in terminology is how we use the word *action*. To many cognitive scientists, an action is purposeful in the sense of being the result of an animal's knowledge about the relationship between the behavior in question and the consequences of that behavior. An action is goal-directed and the result of a decision, whereas a response is triggered by a stimulus and is the result of a reflex or a habit. We use the word action without differentiating among what others call actions, decisions, and responses. These are important distinctions, but for us they are encompassed by differences between

model-free and model-based reinforcement learning algorithms, which we discussed above in relation to habitual and goal-directed behavior in Section 14.6. Dickinson (1985) discusses the distinction between responses and actions.

A term used a lot in this book is *control*. What we mean by control is entirely different from what it means to animal learning psychologists. By control we mean that an agent influences its environment to bring about states or events that the agent prefers: the agent exerts control over its environment. This is the sense of control used by control engineers. In psychology, on the other hand, control typically means that an animal's behavior is influenced by—is controlled by—the stimuli the animal receives (stimulus control) or the reinforcement schedule it experiences. Here the environment is controlling the agent. Control in this sense is the basis of behavior modification therapy. Of course, both of these directions of control are at play when an agent interacts with its environment, but our focus is on the agent as controller, not the environment as controller. A view equivalent to ours, and perhaps more illuminating, is that the agent is actually controlling the input it receives from its environment (Powers, 1973). This is *not* what psychologists mean by stimulus control.

Sometimes reinforcement learning is understood to refer solely to learning policies directly from rewards (and penalties) without the involvement of value functions or environment models. This is what psychologists call stimulus-response, or S-R, learning. But for us, along with most of today's psychologists, reinforcement learning is much broader than this, including in addition to S-R learning, methods involving value functions, environment models, planning, and other processes that are commonly thought to belong to the more cognitive side of mental functioning.

Chapter 15

Neuroscience

Neuroscience is the multidisciplinary study of nervous systems: how they regulate bodily functions; control behavior; change over time as a result of development, learning, and aging; and how cellular and molecular mechanisms make these functions possible. One of the most exciting aspects of reinforcement learning is the mounting evidence from neuroscience that the nervous systems of humans and many other animals implement algorithms that correspond in striking ways to reinforcement learning algorithms. The main objective of this chapter is to explain these parallels and what they suggest about the neural basis of reward-related learning in animals.

The most remarkable point of contact between reinforcement learning and neuroscience involves dopamine, a chemical deeply involved in reward processing in the brains of mammals. Dopamine appears to convey temporal-difference (TD) errors to brain structures where learning and decision making take place. This parallel is expressed by the *reward prediction error hypothesis of dopamine neuron activity*, a hypothesis that resulted from the convergence of computational reinforcement learning and results of neuroscience experiments. In this chapter we discuss this hypothesis, the neuroscience findings that led to it, and why it is a significant contribution to understanding brain reward systems. We also discuss parallels between reinforcement learning and neuroscience that are less striking than this dopamine/TD-error parallel but that provide useful conceptual tools for thinking about reward-based learning in animals. Other elements of reinforcement learning have the potential to impact the study of nervous systems, but their connections to neuroscience are still relatively undeveloped. We discuss several of these evolving connections that we think will grow in importance over time.

As we outlined in the history section of this book's introductory chapter (Section 1.7), many aspects of reinforcement learning were influenced by neuroscience. A second objective of this chapter is to acquaint readers with ideas about brain function that have contributed to our approach to reinforcement learning. Some elements of reinforcement learning are easier to understand when seen in light of theories of brain function. This is particularly true for the idea of the eligibility trace, one of the basic mechanisms of reinforcement learning, that originated as a conjectured property of synapses, the structures by which nerve cells—neurons—communicate with one another.

In this chapter we do not delve very deeply into the enormous complexity of the neural systems underlying reward-based learning in animals: this chapter is too short, and we are not neuroscientists. We do not try to describe—or even to name—the very many brain structures and pathways, or any of the molecular mechanisms, believed to be involved in these processes. We also do not do justice to hypotheses and models that are alternatives to those that align so well with reinforcement learning. It should not be surprising that there are differing views among experts in the field. We can only provide a glimpse into this fascinating and developing story. We hope, though, that this chapter convinces you that a very fruitful channel has emerged connecting reinforcement learning and its theoretical underpinnings to the neuroscience of reward-based learning in animals.

Many excellent publications cover links between reinforcement learning and neuroscience, some of which we cite in this chapter's final section. Our treatment differs from most of these because we assume familiarity with reinforcement learning as presented in the earlier chapters of this book, but we do not assume knowledge of neuroscience. We begin with a brief introduction to the neuroscience concepts needed for a basic understanding of what is to follow.

15.1 Neuroscience Basics

Some basic information about nervous systems is helpful for following what we cover in this chapter. Terms that we refer to later are italicized. Skipping this section will not be a problem if you already have an elementary knowledge of neuroscience.

Neurons, the main components of nervous systems, are cells specialized for processing and transmitting information using electrical and chemical signals. They come in many forms, but a neuron typically has a cell body, *dendrites*, and a single *axon*. Dendrites are structures that branch from the cell body to receive input from other neurons (or to also receive external signals in the case of sensory neurons). A neuron's axon is a fiber that carries the neuron's output to other neurons (or to muscles or glands). A neuron's output consists of sequences of electrical pulses called *action potentials* that travel along the axon. Action potentials are also called *spikes*, and a neuron is said to *fire* when it generates a spike. In models of neural networks it is common to use real numbers to represent a neuron's *firing rate*, the average number of spikes per some unit of time.

A neuron's axon can branch widely so that the neuron's action potentials reach many targets. The branching structure of a neuron's axon is called the neuron's *axonal arbor*. Because the conduction of an action potential is an active process, not unlike the burning of a fuse, when an action potential reaches an axonal branch point it "lights up" action potentials on all of the outgoing branches (although propagation to a branch can sometimes fail). As a result, the activity of a neuron with a large axonal arbor can influence many target sites.