CHAPTER 23

REINFORCEMENT LEARNING

In which we see how experiencing rewards and punishments can teach an agent how to maximize rewards in the future.

With **supervised learning**, an agent learns by passively observing example input/output pairs provided by a "teacher." In this chapter, we will see how agents can actively learn from their own experience, without a teacher, by considering their own ultimate success or failure.

23.1 Learning from Rewards

Consider the problem of learning to play chess. Let's imagine treating this as a supervised learning problem using the methods of Chapters 19, 21, and 22. The chess-playing agent function takes as input a board position and returns a move, so we train this function by supplying examples of chess positions, each labeled with the correct move. Now, it so happens that we have available databases of several million grandmaster games, each a sequence of positions and moves. The moves made by the winner are, with few exceptions, assumed to be good, if not always perfect. Thus, we have a promising training set. The problem is that there are relatively few examples (about 10^8) compared to the space of all possible chess positions (about 10^{40}). In a new game, one soon encounters positions that are significantly different from those in the database, and the trained agent function is likely to fail miserably—not least because it has no idea of what its moves are supposed to achieve (checkmate) or even what effect the moves have on the positions of the pieces. And of course chess is a tiny part of the real world. For more realistic problems, we would need much vaster grandmaster databases, and they simply don't exist.[1]

Reinforcement learning

An alternative is **reinforcement learning** (RL), in which an agent interacts with the world and periodically receives **rewards** (or, in the terminology of psychology, **reinforcements**) that reflect how well it is doing. For example, in chess the reward is 1 for winning, 0 for losing, and $\frac{1}{2}$ for a draw. We have already seen the concept of rewards in Chapter 16 for **Markov decision processes** (MDPs). Indeed, the goal is the same in reinforcement learning: maximize the expected sum of rewards. Reinforcement learning differs from "just solving an MDP" because the agent is not *given* the MDP as a problem to solve; the agent is *in* the MDP. It may not know the transition model or the reward function, and it has to act in order to learn more. Imagine playing a new game whose rules you don't know; after a hundred or so moves, the referee tells you "You lose." That is reinforcement learning in a nutshell.

From our point of view as designers of AI systems, providing a reward signal to the agent is usually much easier than providing labeled examples of how to behave. First, the reward

[1] As Yann LeCun and Alyosha Efros have pointed out, "the AI revolution will not be supervised."

function is often (as we saw for chess) very concise and easy to specify: it requires only a few lines of code to tell the chess agent if it has won or lost the game or to tell the car-racing agent that it has won or lost the race or has crashed. Second, we don't have to be experts, capable of supplying the correct action in any situation, as would be the case if we tried to apply supervised learning.

It turns out, however, that a little bit of expertise can go a long way in reinforcement learning. The two examples in the preceding paragraph—the win/loss rewards for chess and racing—are what we call **sparse** rewards, because in the vast majority of states the agent is given no informative reward signal at all. In games such as tennis and cricket, we can easily supply additional rewards for each point won or for each run scored. In car racing, we could reward the agent for making progress around the track in the right direction. When learning to crawl, any forward motion is an achievement. These intermediate rewards make learning much easier.

Sparse

As long as we can provide the correct reward signal to the agent, reinforcement learning provides a very general way to build AI systems. This is particularly true for *simulated* environments, where there is no shortage of opportunities to gain experience. The addition of deep learning as a tool within RL systems has also made new applications possible, including learning to play Atari video games from raw visual input (Mnih *et al.*, 2013), controlling robots (Levine *et al.*, 2016), and playing poker (Brown and Sandholm, 2017).

Literally hundreds of different reinforcement learning algorithms have been devised, and many of them can employ as tools a wide range of learning methods from Chapters 19, 21, and 22. In this chapter, we cover the basic ideas and give some sense of the variety of approaches through a few examples. We categorize the approaches as follows:

- **Model-based reinforcement learning**: In these approaches the agent uses a transition model of the environment to help interpret the reward signals and to make decisions about how to act. The model may be initially unknown, in which case the agent learns the model from observing the effects of its actions, or it may already be known—for example, a chess program may know the rules of chess even if it does not know how to choose good moves. In partially observable environments, the transition model is also useful for **state estimation** (see Chapter 14). Model-based reinforcement learning systems often learn a **utility function** $U(s)$, defined (as in Chapter 16) in terms of the sum of rewards from state s onward.[2]

  Model-based reinforcement learning

- **Model-free reinforcement learning**: In these approaches the agent neither knows nor learns a transition model for the environment. Instead, it learns a more direct representation of how to behave. This comes in one of two varieties:

  Model-free reinforcement learning

  - **Action-utility learning**: We introduced action-utility functions in Chapter 16. The most common form of action-utility learning is **Q-learning**, where the agent learns a **Q-function**, or quality-function, $Q(s,a)$, denoting the sum of rewards from state s onward if action a is taken. Given a Q-function, the agent can choose what to do in s by finding the action with the highest Q-value.

    Action-utility learning
    Q-learning
    Q-function

  - **Policy search**: The agent learns a policy $\pi(s)$ that maps directly from states to actions. In the terminology of Chapter 2, this a **reflex agent**.

    Policy search

[2] In the RL literature, which draws more on operations research than economics, utility functions are often called **value functions** and denoted $V(s)$.

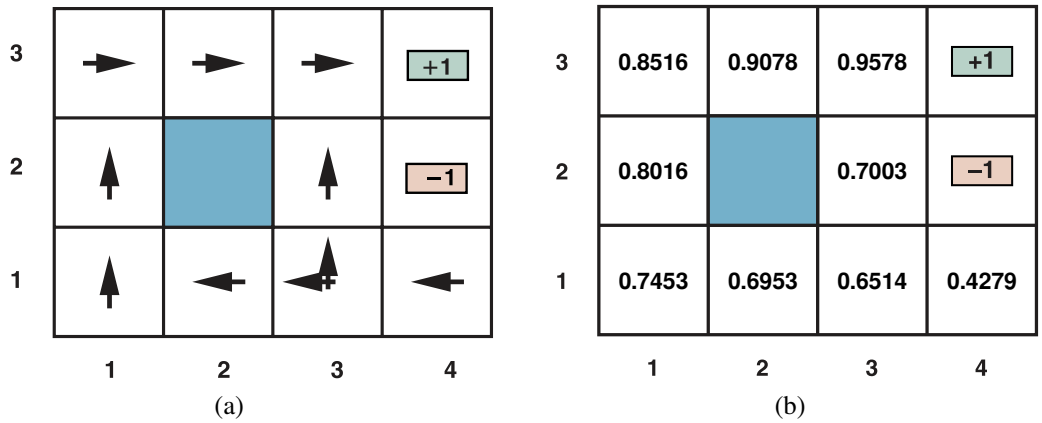


Figure 23.1 (a) The optimal policies for the stochastic environment with $R(s,a,s') = -0.04$ for transitions between nonterminal states. There are two policies because in state (3,1) both *Left* and *Up* are optimal. We saw this before in Figure 16.2. (b) The utilities of the states in the 4×3 world, given policy π.

Passive reinforcement learning

Active reinforcement learning

We begin in Section 23.2 with **passive reinforcement learning**, where the agent's policy is fixed and the task is to learn the utilities of states (or of state–action pairs); this could also involve learning a model of the environment. (An understanding of Markov decision processes, as described in Chapter 16, is essential for this section.) Section 23.3 covers **active reinforcement learning**, where the agent must also figure out what to do. The principal issue is **exploration**: an agent must experience as much as possible of its environment in order to learn how to behave in it. Section 23.4 discusses how an agent can use inductive learning (including deep learning methods) to learn much faster from its experiences. We also discuss other approaches that can help scale up RL to solve real problems, including providing intermediate pseudorewards to guide the learner and organizing behavior into a hierarchy of actions. Section 23.5 covers methods for policy search. In Section 23.6, we explore **apprenticeship learning**: training a learning agent using demonstrations rather than reward signals. Finally, Section 23.7 reports on applications of reinforcement learning.

23.2 Passive Reinforcement Learning

We start with the simple case of a fully observable environment with a small number of actions and states, in which an agent already has a fixed policy $\pi(s)$ that determines its actions. The agent is trying to learn the utility function $U^{\pi}(s)$—the expected total discounted reward if policy π is executed beginning in state s. We call this a **passive learning agent**.

Passive learning agent

The passive learning task is similar to the **policy evaluation** task, part of the policy iteration algorithm described in Section 16.2.2. The difference is that the passive learning agent does not know the transition model $P(s' \mid s,a)$, which specifies the probability of reaching state s' from state s after doing action a; nor does it know the reward function $R(s,a,s')$, which specifies the reward for each transition.

We will use as our example the 4×3 world introduced in Chapter 16. Figure 23.1 shows the optimal policies for that world and the corresponding utilities. The agent executes a set

of **trials** in the environment using its policy π. In each trial, the agent starts in state (1,1) and experiences a sequence of state transitions until it reaches one of the terminal states, (4,2) or (4,3). Its percepts supply both the current state and the reward received for the transition that just occurred to reach that state. Typical trials might look like this: Trial

$$(1,1) \xrightarrow[Up]{\text{-.04}} (1,2) \xrightarrow[Up]{\text{-.04}} (1,3) \xrightarrow[Right]{\text{-.04}} (1,2) \xrightarrow[Up]{\text{-.04}} (1,3) \xrightarrow[Right]{\text{-.04}} (2,3) \xrightarrow[Right]{\text{-.04}} (3,3) \xrightarrow[Right]{\text{+1}} (4,3)$$
$$(1,1) \xrightarrow[Up]{\text{-.04}} (1,2) \xrightarrow[Up]{\text{-.04}} (1,3) \xrightarrow[Right]{\text{-.04}} (2,3) \xrightarrow[Right]{\text{-.04}} (3,3) \xrightarrow[Right]{\text{-.04}} (3,2) \xrightarrow[Up]{\text{-.04}} (3,3) \xrightarrow[Right]{\text{+1}} (4,3)$$
$$(1,1) \xrightarrow[Up]{\text{-.04}} (1,2) \xrightarrow[Up]{\text{-.04}} (1,3) \xrightarrow[Right]{\text{-.04}} (2,3) \xrightarrow[Right]{\text{-.04}} (3,3) \xrightarrow[Right]{\text{-.04}} (3,2) \xrightarrow[Up]{\text{-1}} (4,2)$$

Note that each transition is annotated with both the action taken and the reward received at the next state. The object is to use the information about rewards to learn the expected utility $U^\pi(s)$ associated with each nonterminal state s. The utility is defined to be the expected sum of (discounted) rewards obtained if policy π is followed. As in Equation (16.2) on page 557, we write

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1})\right], \tag{23.1}$$

where $R(S_t, \pi(S_t), S_{t+1})$ is the reward received when action $\pi(S_t)$ is taken in state S_t and reaches state S_{t+1}. Note that S_t is a random variable denoting the state reached at time t when executing policy π, starting from state $S_0 = s$. We will include a **discount factor** γ in all of our equations, but for the 4×3 world we will set $\gamma = 1$, which means no discounting.

23.2.1 Direct utility estimation

The idea of **direct utility estimation** is that the utility of a state is defined as the expected total reward from that state onward (called the expected **reward-to-go**), and that each trial provides a *sample* of this quantity for each state visited. For example, the first of the three trials shown earlier provides a sample total reward of 0.76 for state (1,1), two samples of 0.80 and 0.88 for (1,2), two samples of 0.84 and 0.92 for (1,3), and so on. Thus, at the end of each sequence, the algorithm calculates the observed reward-to-go for each state and updates the estimated utility for that state accordingly, just by keeping a running average for each state in a table. In the limit of infinitely many trials, the sample average will converge to the true expectation in Equation (23.1). Direct utility estimation Reward-to-go

This means that we have reduced reinforcement learning to a standard supervised learning problem in which each example is a (*state*, *reward-to-go*) pair. We have a lot of powerful algorithms for supervised learning, so this approach seems promising, but it ignores an important constraint: *The utility of a state is determined by the reward and the expected utility of the successor states.* More specifically, the utility values obey the Bellman equations for a fixed policy (see also Equation (16.14)):

$$U_i(s) = \sum_{s'} P(s' \mid s, \pi_i(s))[R(s, \pi_i(s), s') + \gamma U_i(s')]. \tag{23.2}$$

By ignoring the connections between states, direct utility estimation misses opportunities for learning. For example, the second of the three trials given earlier reaches the state (3,2), which has not previously been visited. The next transition reaches (3,3), which is known from the first trial to have a high utility. The Bellman equation suggests immediately that (3,2) is also likely to have a high utility, because it leads to (3,3), but direct utility estimation

learns nothing until the end of the trial. More broadly, we can view direct utility estimation as searching for U in a hypothesis space that is much larger than it needs to be, in that it includes many functions that violate the Bellman equations. For this reason, the algorithm often converges very slowly.

23.2.2 Adaptive dynamic programming

Adaptive dynamic programming

An **adaptive dynamic programming** (or ADP) agent takes advantage of the constraints among the utilities of states by learning the transition model that connects them and solving the corresponding Markov decision process using dynamic programming. For a passive learning agent, this means plugging the learned transition model $P(s' \mid s, \pi(s))$ and the observed rewards $R(s, \pi(s), s')$ into Equation (23.2) to calculate the utilities of the states. As we remarked in our discussion of policy iteration in Chapter 16, these Bellman equations are linear when the policy π is fixed, so they can be solved using any linear algebra package.

Alternatively, we can adopt the approach of **modified policy iteration** (see page 568), using a simplified value iteration process to update the utility estimates after each change to the learned model. Because the model usually changes only slightly with each observation, the value iteration process can use the previous utility estimates as initial values and typically converge very quickly.

Learning the transition model is easy, because the environment is fully observable. This means that we have a supervised learning task where the input for each training example is a state–action pair, (s,a), and the output is the resulting state, s'. The transition model $P(s' \mid s,a)$ is represented as a table and it is estimated directly from the counts that are accumulated in $N_{s' \mid sa}$. The counts record how often state s' is reached when executing a in s. For example, in the three trials given on page 843, *Right* is executed four times in (3,3) and the resulting state is (3,2) twice and (4,3) twice, so $P((3,2) \mid (3,3), \mathit{Right})$ and $P((4,3) \mid (3,3), \mathit{Right})$ are both estimated to be $\frac{1}{2}$.

The full agent program for a passive ADP agent is shown in Figure 23.2. Its performance on the 4×3 world is shown in Figure 23.3. In terms of how quickly its value estimates improve, the ADP agent is limited only by its ability to learn the transition model. In this sense, it provides a standard against which to measure any other reinforcement learning algorithms. It is, however, intractable for large state spaces. In backgammon, for example, it would involve solving roughly 10^{20} equations in 10^{20} unknowns.

23.2.3 Temporal-difference learning

Solving the underlying MDP as in the preceding section is not the only way to bring the Bellman equations to bear on the learning problem. Another way is to use the observed transitions to adjust the utilities of the observed states so that they agree with the constraint equations. Consider, for example, the transition from (1,3) to (2,3) in the second trial on page 843. Suppose that as a result of the first trial, the utility estimates are $U^\pi(1,3) = 0.88$ and $U^\pi(2,3) = 0.96$. Now, if this transition from (1,3) to (2,3) occurred all the time, we would expect the utilities to obey the equation

$$U^\pi(1,3) = -0.04 + U^\pi(2,3),$$

so $U^\pi(1,3)$ would be 0.92. Thus, its current estimate of 0.88 might be a little low and should be increased. More generally, when a transition occurs from state s to state s' via action $\pi(s)$,

function PASSIVE-ADP-LEARNER(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r
  persistent: π, a fixed policy
    mdp, an MDP with model P, rewards R, actions A, discount γ
    U, a table of utilities for states, initially empty
    $N_{s'|s,a}$, a table of outcome count vectors indexed by state and action, initially zero
    s, a, the previous state and action, initially null

  if s' is new **then** $U[s'] \leftarrow 0$
  if s is not null **then**
    increment $N_{s'|s,a}[s,a][s']$
    $R[s, a, s'] \leftarrow r$
    add a to $A[s]$
    $\mathbf{P}(\cdot \mid s,a) \leftarrow$ NORMALIZE($N_{s'|s,a}[s, a]$)
    $U \leftarrow$ POLICYEVALUATION(π, U, mdp)
  $s, a \leftarrow s', \pi[s']$
  return a

Figure 23.2 A passive reinforcement learning agent based on adaptive dynamic programming. The agent chooses a value for γ and then incrementally computes the P and R values of the MDP. The POLICY-EVALUATION function solves the fixed-policy Bellman equations, as described on page 567.

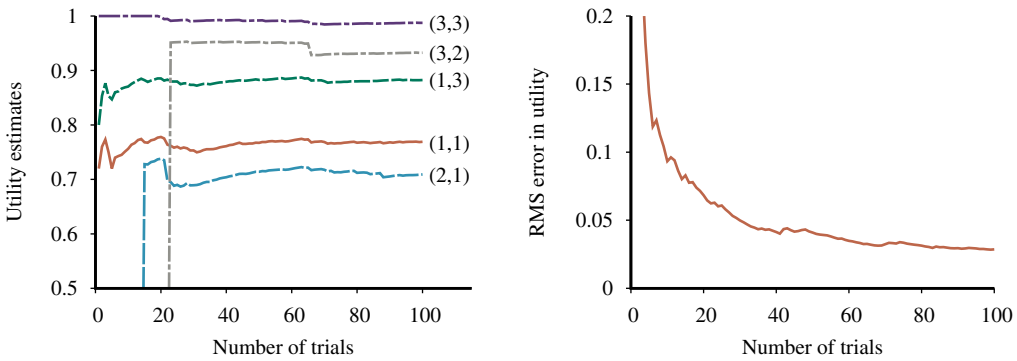


Figure 23.3 The passive ADP learning curves for the 4×3 world, given the optimal policy shown in Figure 23.1. (a) The utility estimates for a selected subset of states, as a function of the number of trials. Notice that it takes 14 and 23 trials respectively before the rarely visited states (2,1) and (3,2) "discover" that they connect to the +1 exit state at (4,3). (b) The root-mean-square error (see Appendix A) in the estimate for $U(1,1)$, averaged over 50 runs of 100 trials each.

```
function PASSIVE-TD-LEARNER(percept) returns an action
  inputs: percept, a percept indicating the current state s' and reward signal r
  persistent: π, a fixed policy
              s, the previous state, initially null
              U, a table of utilities for states, initially empty
              N_s, a table of frequencies for states, initially zero

  if s' is new then U[s'] ← 0
  if s is not null then
    increment N_s[s]
    U[s] ← U[s] + α(N_s[s]) × (r + γ U[s'] - U[s])
  s ← s'
  return π[s']
```

Figure 23.4 A passive reinforcement learning agent that learns utility estimates using temporal differences. The step-size function $\alpha(n)$ is chosen to ensure convergence.

we apply the following update to $U^\pi(s)$:

$$U^\pi(s) \leftarrow U^\pi(s) + \alpha[R(s,\pi(s),s') + \gamma U^\pi(s') - U^\pi(s)]\,. \tag{23.3}$$

Here, α is the **learning rate** parameter. Because this update rule uses the difference in utilities between successive states (and thus successive times), it is often called the **temporal-difference** (TD) equation. Just as in the weight update rules from Chapter 19 (e.g., Equation (19.6) on page 698), the TD term $R(s,\pi(s),s') + \gamma U^\pi(s') - U^\pi(s)$ is effectively an error signal, and the update is intended to reduce the error.

Temporal-difference

All temporal-difference methods work by adjusting the utility estimates toward the ideal equilibrium that holds locally when the utility estimates are correct. In the case of passive learning, the equilibrium is given by Equation (23.2). Now Equation (23.3) does in fact cause the agent to reach the equilibrium given by Equation (23.2), but there is some subtlety involved. First, notice that the update involves only the observed successor s', whereas the actual equilibrium conditions involve all possible next states. One might think that this causes an improperly large change in $U^\pi(s)$ when a very rare transition occurs; but, in fact, because rare transitions occur only rarely, the *average value* of $U^\pi(s)$ will converge to the correct quantity in the limit, even if the value itself continues to fluctuate.

Furthermore, if we turn the parameter α into a function that decreases as the number of times a state has been visited increases, as shown in Figure 23.4, then $U^\pi(s)$ itself will converge to the correct value.[3] Figure 23.5 illustrates the performance of the passive TD agent on the 4×3 world. It does not learn quite as fast as the ADP agent and shows much higher variability, but it is much simpler and requires much less computation per observation. Notice that *TD does not need a transition model to perform its updates.* The environment itself supplies the connection between neighboring states in the form of observed transitions.

The ADP and TD approaches are closely related. Both try to make local adjustments to the utility estimates in order to make each state "agree" with its successors. One difference is

[3] The technical conditions are given on page 702. In Figure 23.5 we have used $\alpha(n) = 60/(59+n)$, which satisfies the conditions.

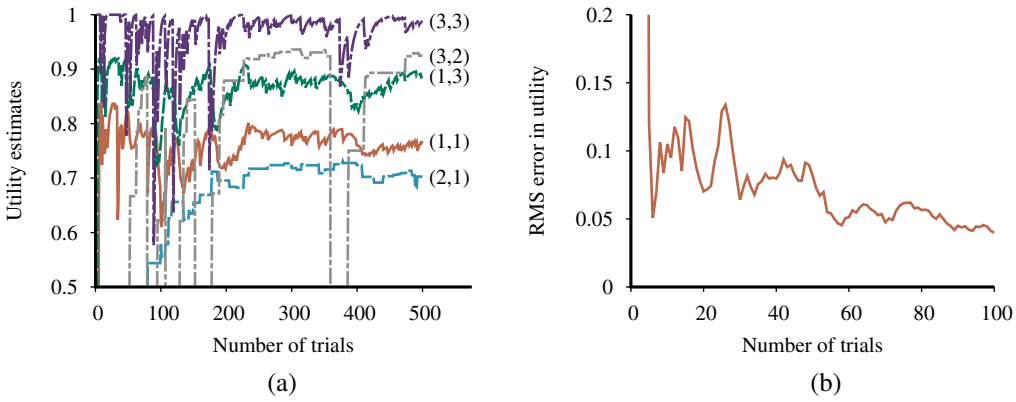


Figure 23.5 The TD learning curves for the 4×3 world. (a) The utility estimates for a selected subset of states, as a function of the number of trials, for a single run of 500 trials. Compare with the run of 100 trials in Figure 23.3(a). (b) The root-mean-square error in the estimate for $U(1,1)$, averaged over 50 runs of 100 trials each.

that TD adjusts a state to agree with its *observed* successor (Equation (23.3)), whereas ADP adjusts the state to agree with *all* of the successors that might occur, weighted by their probabilities (Equation (23.2)). This difference disappears when the effects of TD adjustments are averaged over a large number of transitions, because the frequency of each successor in the set of transitions is approximately proportional to its probability. A more important difference is that whereas TD makes a single adjustment per observed transition, ADP makes as many as it needs to restore consistency between the utility estimates U and the transition model P. Although the observed transition makes only a local change in P, its effects might need to be propagated throughout U. Thus, TD can be viewed as a crude but efficient first approximation to ADP.

Each adjustment made by ADP could be seen, from the TD point of view, as a result of a **pseudoexperience** generated by simulating the current transition model. It is possible to extend the TD approach to use a transition model to generate several pseudoexperiences—transitions that the TD agent can imagine *might* happen, given its current model. For each observed transition, the TD agent can generate a large number of imaginary transitions. In this way, the resulting utility estimates will approximate more and more closely those of ADP—of course, at the expense of increased computation time.

Pseudoexperience

In a similar vein, we can generate more efficient versions of ADP by directly approximating the algorithms for value iteration or policy iteration. Even though the value iteration algorithm is efficient, it is intractable if we have, say, 10^{100} states. However, many of the necessary adjustments to the state values on each iteration will be extremely tiny. One possible approach to generating reasonably good answers quickly is to bound the number of adjustments made after each observed transition. One can also use a heuristic to rank the possible adjustments so as to carry out only the most significant ones. The **prioritized sweeping** heuristic prefers to make adjustments to states whose *likely* successors have just undergone a *large* adjustment in their own utility estimates.

Prioritized sweeping

Using heuristics like this, approximate ADP algorithms can learn roughly as fast as full ADP, in terms of the number of training sequences, but can be orders of magnitude more efficient in terms of total computation (see Exercise 23.PRSW). This enables them to handle state spaces that are far too large for full ADP. Approximate ADP algorithms have an additional advantage: in the early stages of learning a new environment, the transition model P often will be far from correct, so there is little point in calculating an exact utility function to match it. An approximation algorithm can use a minimum adjustment size that decreases as the transition model becomes more accurate. This eliminates the very long runs of value iteration that can occur early in learning due to large changes in the model.

23.3 Active Reinforcement Learning

A passive learning agent has a fixed policy that determines its behavior. An **active learning agent** gets to decide what actions to take. Let us begin with the adaptive dynamic programming (ADP) agent and consider how it can be modified to take advantage of this new freedom.

First, the agent will need to learn a complete transition model with outcome probabilities for *all* actions, rather than just the model for the fixed policy. The learning mechanism used by PASSIVE-ADP-AGENT will do just fine for this. Next, we need to take into account the fact that the agent has a choice of actions. The utilities it needs to learn are those defined by the *optimal* policy; they obey the Bellman equations (which we repeat here):

$$U(s) = \max_{a \in A(s)} \sum_{s'} P(s' \mid s,a)[R(s,a,s') + \gamma U(s')] . \tag{23.4}$$

These equations can be solved to obtain the utility function U using the value iteration or policy iteration algorithms from Chapter 16.

The final issue is what to do at each step. Having obtained a utility function U that is optimal for the learned model, the agent can extract an optimal action by one-step look-ahead to maximize the expected utility; alternatively, if it uses policy iteration, the optimal policy is already available, so it could simply execute the action the optimal policy recommends. But should it?

23.3.1 Exploration

Figure 23.6 shows the results of one sequence of trials for an ADP agent that follows the recommendation of the optimal policy for the learned model at each step. The agent *does not* learn the true utilities or the true optimal policy! What happens instead is that in the third trial, it finds a policy that reaches the +1 reward along the lower route via (2,1), (3,1), (3,2), and (3,3). (See Figure 23.6(b).) After experimenting with minor variations, from the eighth trial onward it sticks to that policy, never learning the utilities of the other states and never finding the optimal route via (1,2), (1,3), and (2,3). We will call this agent a **greedy agent**, because it greedily takes the action that it currently believes to be optimal at each step. Sometimes greed pays off and the agent converges to the optimal policy, but often it does not.

Greedy agent

How can it be that choosing the optimal action leads to suboptimal results? The answer is that the learned model is not the same as the true environment; what is optimal in the learned model can therefore be suboptimal in the true environment. Unfortunately, the agent does not know what the true environment is, so it cannot compute the optimal action for the true environment. What, then, should it do?

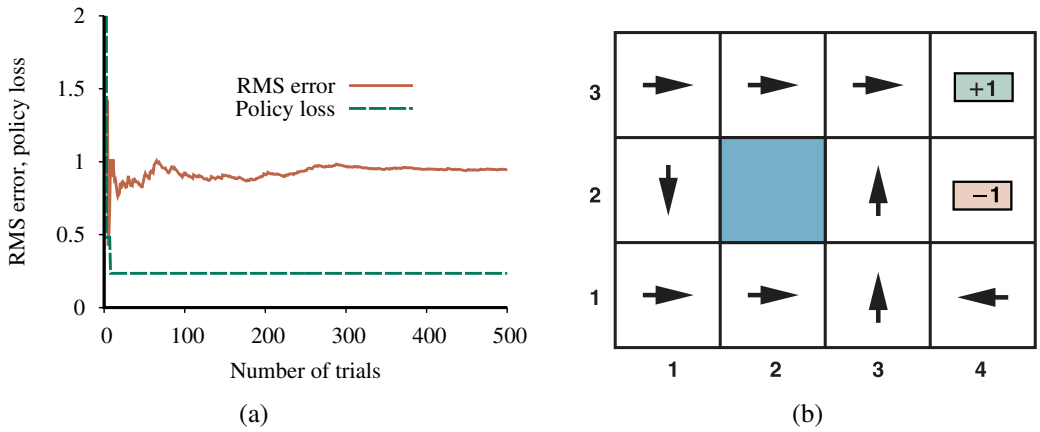


Figure 23.6 Performance of a greedy ADP agent that executes the action recommended by the optimal policy for the learned model. (a) The root mean square (RMS) error averaged across all nine nonterminal squares and the policy loss in (1,1). We see that the policy converges quickly, after just eight trials, to a suboptimal policy with a loss of 0.235. (b) The suboptimal policy to which the greedy agent converges in this particular sequence of trials. Notice the *Down* action in (1,2).

The greedy agent has overlooked the fact that actions do more than provide *rewards*; they also provide *information* in the form of percepts in the resulting states. As we saw with **bandit problems** in Section 16.3, an agent must make a tradeoff between **exploitation** of the current best action to maximize its short-term reward and **exploration** of previously unknown states to gain information that can lead to a change in policy (and to greater rewards in the future). In the real world, one constantly has to decide between continuing in a comfortable existence, versus striking out into the unknown in the hopes of a better life.

Although bandit problems are difficult to solve exactly to obtain an *optimal* exploration scheme, it is nonetheless possible to come up with a scheme that will eventually discover an optimal policy, even if it might take longer to do so than is optimal. Any such scheme should not be greedy in terms of the immediate next move, but should be what is called "greedy in the limit of infinite exploration," or **GLIE**. A GLIE scheme must try each action in each state GLIE
an unbounded number of times to avoid having a finite probability that an optimal action is missed. An ADP agent using such a scheme will eventually learn the true transition model, and can then operate under exploitation.

There are several GLIE schemes; one of the simplest is to have the agent choose a random action at time step t with probability $1/t$ and to follow the greedy policy otherwise. While this does eventually converge to an optimal policy, it can be slow. A better approach would give some weight to actions that the agent has not tried very often, while tending to avoid actions that are believed to be of low utility (as we did with Monte Carlo tree search in Section 6.4). This can be implemented by altering the constraint equation (23.4) so that it assigns a higher utility estimate to relatively unexplored state–action pairs.

This amounts to an optimistic prior over the possible environments and causes the agent to behave initially as if there were wonderful rewards scattered all over the place. Let us use $U^+(s)$ to denote the optimistic estimate of the utility (i.e., the expected reward-to-go) of the

state s, and let $N(s,a)$ be the number of times action a has been tried in state s. Suppose we are using value iteration in an ADP learning agent; then we need to rewrite the update equation (Equation (16.10) on page 563) to incorporate the optimistic estimate:

$$U^+(s) \leftarrow \max_a f\left(\sum_{s'} P(s' \mid s,a)[R(s,a,s') + \gamma U^+(s')],\ N(s,a)\right). \tag{23.5}$$

Exploration function

Here, f is the **exploration function**. The function $f(u,n)$ determines how greed (preference for high values of the utility u) is traded off against curiosity (preference for actions that have not been tried often and have a low count n). The function should be increasing in u and decreasing in n. Obviously, there are many possible functions that fit these conditions. One particularly simple definition is

$$f(u,n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise,} \end{cases}$$

where R^+ is an optimistic estimate of the best possible reward obtainable in any state and N_e is a fixed parameter. This will have the effect of making the agent try each state–action pair at least N_e times. The fact that U^+ rather than U appears on the right-hand side of Equation (23.5) is very important. As exploration proceeds, the states and actions near the start state might well be tried a large number of times. If we used U, the more pessimistic utility estimate, then the agent would soon become disinclined to explore further afield. The use of U^+ means that the benefits of exploration are propagated back from the edges of unexplored regions, so that actions that lead *toward* unexplored regions are weighted more highly, rather than just actions that are themselves unfamiliar.

The effect of this exploration policy can be seen clearly in Figure 23.7(b), which shows a rapid convergence toward zero policy loss, unlike with the greedy approach. A very nearly optimal policy is found after just 18 trials. Notice that the RMS error in the utility estimates does not converge as quickly. This is because the agent stops exploring the unrewarding parts of the state space fairly soon, visiting them only "by accident" thereafter. However, it makes perfect sense for the agent not to care about the exact utilities of states that it knows are undesirable and can be avoided. There is not much point in learning about the best radio station to listen to while falling off a cliff.

23.3.2 Safe exploration

So far we have assumed that an agent is free to explore as it wishes—that any negative rewards serve only to improve its model of the world. That is, if we play a game of chess and lose, we suffer no damage (except perhaps to our pride), and whatever we learned will make us a better player in the next game. Similarly, in a simulation environment for a self-driving car, we could explore the limits of the car's performance, and any accidents give us more information. If the car crashes, we just hit the reset button.

Unfortunately, the real world is less forgiving. If you are a baby sunfish, your probability of surviving to adulthood is about 0.00000001. Many actions are **irreversible**, in the sense defined for online search agents in Section 4.5: no subsequent sequence of actions can restore the state to what it was before the irreversible action was taken. In the worst case, the agent enters an **absorbing state** where no actions have any effect and no rewards are received.

Absorbing state

In many practical settings, we cannot afford to have our agents taking irreversible actions or entering absorbing states. For example, an agent learning to drive in a real car should avoid

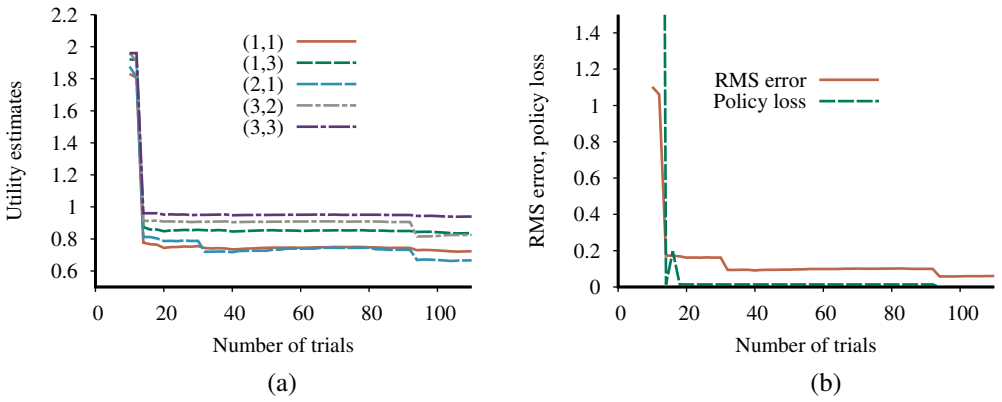


Figure 23.7 Performance of the exploratory ADP agent using $R^+ = 2$ and $N_e = 5$. (a) Utility estimates for selected states over time. (b) The RMS error in utility values and the associated policy loss.

taking actions that might lead to any of the following:

- states with large negative rewards, such as serious car crashes;
- states from which there is no escape, such as driving the car into a deep ditch;
- states that permanently limit future rewards, such as damaging the car's engine so that its maximum speed is reduced.

We can end up in a bad state either because our model is *unknown*, and we actively choose to explore in a direction that turns out to be bad, or because our model is *incorrect* and we don't know that a given action can have a disastrous result. Note that the algorithm in Figure 23.2 is using maximum-likelihood estimation (see Chapter 21) to learn the transition model; moreover, by choosing a policy based solely on the *estimated* model, it is acting *as if* the model were correct. This is not necessarily a good idea! For example, a taxi agent that didn't know how traffic lights work might ignore a red light once or twice with no ill effects and then formulate a policy to ignore all red lights from then on.

A better idea would be to choose a policy that works reasonably well for the whole range of models that have a reasonable chance of being the true model, even if the policy happens to be suboptimal for the maximum-likelihood model. There are three mathematical approaches that have this flavor.

Bayesian reinforcement learning

The first approach, **Bayesian reinforcement learning**, assumes a prior probability $P(h)$ over hypotheses h about what the true model is; the posterior probability $P(h \mid \mathbf{e})$ is obtained in the usual way by Bayes' rule given the observations to date. Then, if the agent has decided to stop learning, the optimal policy is the one that gives the highest expected utility. Let U_h^π be the expected utility, averaged over all possible start states, obtained by executing policy π in model h. Then we have

$$\pi^* = \operatorname*{argmax}_\pi \sum_h P(h \mid \mathbf{e}) U_h^\pi .$$

In some special cases, this policy can even be computed! If the agent will continue learning in the future, however, then finding an optimal policy becomes considerably more difficult,

Exploration POMDP

because the agent must consider the effects of future observations on its beliefs about the transition model. The problem becomes an **exploration POMDP** whose belief states are distributions over models. In principle, this exploration POMDP can be formulated and solved before the agent ever sets foot in the world. (Exercise 23.EPOM asks you to do this for the Minesweeper game to find the best first move.) The result is a complete strategy that tells the agent what to do next given any possible percept sequence. Solving the exploration POMDP is usually wildly intractable, but the concept provides an analytical foundation for understanding the exploration problem described in Section 23.3.

It is worth noting that being perfectly Bayesian will not protect the agent from an untimely death. Unless the prior gives some indication of percepts that suggest danger, there is nothing to prevent the agent from taking an exploratory action that leads to an absorbing state. For example, it used to be thought that human infants had an innate fear of heights and would not crawl off a cliff, but this turns out not to be true (Adolph *et al.*, 2014).

Robust control theory

The second approach, derived from **robust control theory**, allows for a set of possible models $\mathcal{H}$ without assigning probabilities to them, and defines an optimal robust policy as one that gives the best outcome in the *worst case* over $\mathcal{H}$:

$$\pi^* = \operatorname*{argmax}_{\pi} \min_{h} U_h^{\pi} .$$

Often, the set $\mathcal{H}$ will be the set of models that exceed some likelihood threshold on $P(h \mid \mathbf{e})$, so the robust and Bayesian approaches are related.

The robust control approach can be considered as a game between the agent and an adversary, where the adversary gets to pick the worst possible result for any action, and the policy we get is the minimax solution for the game. Our logical wumpus agent (see Section 7.7) is a robust control agent in this way: it considers all models that are logically possible, and does not explore any locations that could possibly contain a pit or a wumpus, so it is finding the action with maximum utility in the worst case over all possible hypotheses.

The problem with the worst-case assumption is that it results in overly conservative behavior. A self-driving car that assumes that every other driver *will try to collide with it* has no choice but to stay in the garage. Real life is full of such risk–reward tradeoffs.

Although one reason for venturing into reinforcement learning was to escape the need for a human teacher (as in supervised learning), it turns out that human knowledge can help keep a system safe. One way is to record a series of actions by an experienced teacher, so that the system will act reasonably from the start, and can learn to improve from there. A second way is for a human to write down constraints on what a system can do, and have a program outside of the reinforcement learning system enforce those constraints. For example, when training an autonomous helicopter, a partial policy can be provided that takes over control when the helicopter enters a state from which any further unsafe actions would lead to an irrecoverable state—one in which the safety controller cannot guarantee that the absorbing state will be avoided. In all other states, the learning agent is free to do as it pleases.

23.3.3 Temporal-difference Q-learning

Now that we have an active ADP agent, let us consider how to construct an active temporal-difference (TD) learning agent. The most obvious change is that the agent will have to learn a transition model so that it can choose an action based on $U(s)$ via one-step look-ahead. The model acquisition problem for the TD agent is identical to that for the ADP agent, and the

TD update rule remains unchanged. Once again, it can be shown that the TD algorithm will converge to the same values as ADP, as the number of training sequences tends to infinity.

The **Q-learning** method avoids the need for a model by learning an action-utility function $Q(s,a)$ instead of a utility function $U(s)$. $Q(s,a)$ denotes the expected total discounted reward if the agent takes action a in s and acts optimally thereafter. Knowing the Q-function enables the agent to act optimally simply by choosing $\arg\max_a Q(s,a)$, with no need for look-ahead based on a transition model.

We can also derive a model-free TD update for the Q-values. We begin with the Bellman equation for $Q(s,a)$, repeated here from Equation (16.8):

$$Q(s,a) = \sum_{s'} P(s' \mid s,a)[R(s,a,s') + \gamma \max_{a'} Q(s',a')] \tag{23.6}$$

From this, we can write down the Q-learning TD update, by analogy to the TD update for utilities in Equation (23.3):

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)]\,. \tag{23.7}$$

This update is calculated whenever action a is executed in state s leading to state s'. As in Equation (23.3), the term $R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)$ represents an error that the update is trying to minimize.

The important part of this equation is what it does not contain: *a TD Q-learning agent does not need a transition model, $P(s' \mid s,a)$, either for learning or for action selection.* As noted at the beginning of the chapter, model-free methods can be applied even in very complex domains because no model need be provided or learned. On the other hand, the Q-learning agent has no means of looking into the future, so it may have difficulty when rewards are sparse and long action sequences must be constructed to reach them.

The complete agent design for an exploratory TD Q-learning agent is shown in Figure 23.8. Notice that it uses exactly the same exploration function f as that used by the exploratory ADP agent—hence the need to keep statistics on actions taken (the table N). If a simpler exploration policy is used—say, acting randomly on some fraction of steps, where the fraction decreases over time—then we can dispense with the statistics.

Q-learning has a close relative called **SARSA** (for state, action, reward, state, action). SARSA
The update rule for SARSA is very similar to the Q-learning update rule (Equation (23.7)), except that SARSA updates with the Q-value of the action a' that is actually taken:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a,s') + \gamma\, Q(s',a') - Q(s,a)]\,, \tag{23.8}$$

The rule is applied at the end of each s, a, r, s', a' quintuplet—hence the name. The difference from Q-learning is quite subtle: whereas Q-learning backs up the Q-value from the best action in s', SARSA waits until an action is taken and backs up the Q-value for that action. If the agent is greedy and always takes the action with the best Q-value, the two algorithms are identical. When exploration is happening, however, they differ: if the exploration yields a negative reward, SARSA penalizes the action, while Q-learning does not.

Q-learning is an **off-policy** learning algorithm, because it learns Q-values that answer the Off-policy
question "What would this action be worth in this state, assuming that I stop using whatever policy I am using now, and start acting according to a policy that chooses the best action (according to my estimates)?" SARSA is an **on-policy** algorithm: it learns Q-values that On-policy
answer the question "What would this action be worth in this state, assuming I stick with my

function Q-LEARNING-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r
  persistent: Q, a table of action values indexed by state and action, initially zero
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    s, a, the previous state and action, initially null

  if s is not null **then**
    increment $N_{sa}[s,a]$
    $Q[s,a] \leftarrow Q[s,a] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
  $s,a \leftarrow s', \text{argmax}_{a'} f(Q[s',a'], N_{sa}[s',a'])$
  return a

Figure 23.8 An exploratory Q-learning agent. It is an active learner that learns the value $Q(s,a)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model.

policy?" Q-learning is more flexible than SARSA, in the sense that a Q-learning agent can learn how to behave well when under the control of a wide variety of exploration policies. On the other hand, SARSA is appropriate if the overall policy is even partly controlled by other agents or programs, in which case it is better to learn a Q-function for what will actually happen rather than what would happen if the agent got to pick estimated best actions. Both Q-learning and SARSA learn the optimal policy for the 4×3 world, but they do so at a much slower rate than the ADP agent. This is because the local updates do not enforce consistency among all the Q-values via the model.

23.4 Generalization in Reinforcement Learning

So far, we have assumed that utility functions and Q-functions are represented in tabular form with one output value for each state. This works for state spaces with up to about 10^6 states, which is more than enough for our toy two-dimensional grid environments. But in real-world environments with many more states, convergence will be too slow. Backgammon is simpler than most real-world applications, yet it has about 10^{20} states. We cannot easily visit them all in order to learn how to play the game.

Function approximation

Chapter 6 introduced the idea of an **evaluation function** as a compact measure of desirability for potentially vast state spaces. In the terminology of this chapter, the evaluation function is an approximate utility function; we use the term **function approximation** for the process of constructing a compact approximation of the true utility function or Q-function. For example, we might approximate the utility function using a weighted linear combination of **features** $f_1, \ldots, f_n$:

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s) .$$

Instead of learning 10^{20} state values in a table, a reinforcement learning algorithm can learn, say, 20 values for the parameters $\theta = \theta_1, \ldots, \theta_{20}$ that make $\hat{U}_\theta$ a good approximation to the true utility function. Sometimes this approximate utility function is combined with look-ahead search to produce more accurate decisions. Adding look-ahead search means that effective

behavior can be generated from a much simpler utility function approximator that is learnable from far fewer experiences.

Function approximation makes it practical to represent utility (or Q) functions for very large state spaces, but more importantly, it allows for inductive generalization: the agent can generalize from states it has visited to states it has not yet visited. Tesauro (1992) used this technique to build a backgammon-playing program that played at human champion level, even though it explored only a trillionth of the complete state space of backgammon.

23.4.1 Approximating direct utility estimation

The method of direct utility estimation (Section 23.2) generates trajectories in the state space and extracts, for each state, the sum of rewards received from that state onward until termination. The state and the sum of rewards received constitute a training example for a **supervised learning** algorithm. For example, suppose we represent the utilities for the 4×3 world using a simple linear function, where the features of the squares are just their x and y coordinates. In that case, we have

$$\hat{U}_\theta(x,y) = \theta_0 + \theta_1 x + \theta_2 y . \tag{23.9}$$

Thus, if $(\theta_0, \theta_1, \theta_2) = (0.5, 0.2, 0.1)$, then $\hat{U}_\theta(1,1) = 0.8$. Given a collection of trials, we obtain a set of sample values of $\hat{U}_\theta(x,y)$, and we can find the best fit, in the sense of minimizing the squared error, using standard linear regression (see Chapter 19).

For reinforcement learning, it makes more sense to use an *online* learning algorithm that updates the parameters after each trial. Suppose we run a trial and the total reward obtained starting at (1,1) is 0.4. This suggests that $\hat{U}_\theta(1,1)$, currently 0.8, is too large and must be reduced. How should the parameters be adjusted to achieve this? As with neural-network learning, we write an error function and compute its gradient with respect to the parameters. If $u_j(s)$ is the observed total reward from state s onward in the jth trial, then the error is defined as (half) the squared difference of the predicted total and the actual total: $E_j(s) = (\hat{U}_\theta(s) - u_j(s))^2/2$. The rate of change of the error with respect to each parameter θ_i is $\partial E_j / \partial \theta_i$, so to move the parameter in the direction of decreasing the error, we want

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha [u_j(s) - \hat{U}_\theta(s)] \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} . \tag{23.10}$$

This is called the **Widrow–Hoff rule**, or the **delta rule**, for online least-squares. For the linear function approximator $\hat{U}_\theta(s)$ in Equation (23.9), we get three simple update rules:

Widrow–Hoff rule
Delta rule

$$\begin{aligned}
\theta_0 &\leftarrow \theta_0 + \alpha [u_j(s) - \hat{U}_\theta(s)] , \\
\theta_1 &\leftarrow \theta_1 + \alpha [u_j(s) - \hat{U}_\theta(s)] x , \\
\theta_2 &\leftarrow \theta_2 + \alpha [u_j(s) - \hat{U}_\theta(s)] y .
\end{aligned}$$

We can apply these rules to the example where $\hat{U}_\theta(1,1)$ is 0.8 and $u_j(1,1)$ is 0.4. Parameters θ_0, θ_1, and θ_2 are all decreased by 0.4α, which reduces the error for (1,1). Notice that *changing the parameters θ_i in response to an observed transition between two states also changes the values of $\hat{U}_\theta$ for every other state!* This is what we mean by saying that function approximation allows a reinforcement learner to generalize from its experiences.

The agent will learn faster if it uses a function approximator, provided that the hypothesis space is not too large and includes some functions that are a reasonably good fit to the true utility function. Exercise 23.APLM asks you to evaluate the performance of direct utility

estimation, both with and without function approximation. The improvement in the 4×3 world is noticeable but not dramatic, because this is a very small state space to begin with. The improvement is much greater in a 10×10 world with a +1 reward at (10,10).

The 10×10 world is well suited for a linear utility function because the true utility function is smooth and nearly linear: it is basically a diagonal slope with its lower corner at (1,1) and its upper corner at (10,10). (See Exercise 23.TENX.) On the other hand, if we put the +1 reward at (5,5), the true utility is more like a pyramid and the function approximator in Equation (23.9) will fail miserably.

All is not lost, however! Remember that what matters for linear function approximation is that the function be linear in the features. But we can choose the features to be arbitrary nonlinear functions of the state variables. Hence, we can include a feature such as $f_3(x,y) = \sqrt{(x-x_g)^2 + (y-y_g)^2}$ that measures the distance to the goal. With this new feature, the linear function approximator does well.

23.4.2 Approximating temporal-difference learning

We can apply these ideas equally well to temporal-difference learners. All we need do is adjust the parameters to try to reduce the temporal difference between successive states. The new versions of the TD and Q-learning equations (23.3 on page 846 and 23.7 on page 853) are given by

$$\theta_i \leftarrow \theta_i + \alpha\,[R(s,a,s') + \gamma\,\hat{U}_\theta(s') - \hat{U}_\theta(s)]\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} \tag{23.11}$$

for utilities and

$$\theta_i \leftarrow \theta_i + \alpha\,[R(s,a,s') + \gamma \max_{a'} \hat{Q}_\theta(s',a') - \hat{Q}_\theta(s,a)]\frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i} \tag{23.12}$$

for Q-values. For passive TD learning, the update rule can be shown to converge to the closest possible approximation to the true function when the function approximator is *linear* in the features.[4] With active learning and *nonlinear* functions such as neural networks, nearly all bets are off: there are some very simple cases in which the parameters can go off to infinity with these update rules, even though there are good solutions in the hypothesis space. There are more sophisticated algorithms that can avoid these problems, but at present reinforcement learning with general function approximators remains a delicate art.

Catastrophic forgetting

In addition to parameters diverging to infinity, there is a more surprising problem called **catastrophic forgetting**. Suppose you are training an autonomous vehicle to drive along (simulated) roads safely without crashing. You assign a high negative reward for crossing the edge of the road, and you use quadratic features of the road position so that the car can learn that the utility of being in the middle of the road is higher than being close to the edge. All goes well, and the car learns to drive perfectly down the middle of the road. After a few minutes of this, you are starting to get bored and are about to halt the simulation and write up the excellent results. All of a sudden, the vehicle swerves off the road and crashes. Why? What has happened is that the car has learned *too well*: because it has learned to steer away from the edge, it has learned that the entire central region of the road is a safe place to be, and it has forgotten that the region closer to the edge is dangerous. The central region therefore

[4] The definition of distance between utility functions is rather technical; see Tsitsiklis and Van Roy (1997).

has a flat value function, so the quadratic features get zero weight; then, any nonzero weight on the linear features causes the car to slide off the road to one side or the other.

Experience replay

One solution to this problem, called **experience replay**, ensures that the car keeps reliving its youthful crashing behavior at regular intervals. The learning algorithm can retain trajectories from the entire learning process and replay those trajectories to ensure that its value function is still accurate for parts of the state space it no longer visits.

For model-based reinforcement learning systems, function approximation can also be very helpful for learning a model of the environment. Remember that learning a model for an *observable* environment is a *supervised* learning problem, because the next percept gives the outcome state. Any of the supervised learning methods in Chapters 19, 21, and 22 can be used, with suitable adjustments for the fact that we need to predict a complete state description rather than just a Boolean classification or a single real value. With a learned model, the agent can do a look-ahead search to improve its decisions and can carry out internal simulations to improve its approximate representations of U or Q rather than requiring slow and potentially expensive real-world experiences.

For a *partially observable* environment, the learning problem is much more difficult because the next percept is no longer a label for the state prediction problem. If we know what the hidden variables are and how they are causally related to each other and to the observable variables, then we can fix the structure of a dynamic Bayesian network and use the EM algorithm to learn the parameters, as was described in Chapter 21. Learning the internal structure of dynamic Bayesian networks and creating new state variables is still considered a difficult problem. Deep recurrent neural networks (Section 22.6) have in some cases been successful at inventing the hidden structure.

23.4.3 Deep reinforcement learning

There are two reasons why we need to go beyond linear function approximators: first, there may be no good linear function that comes close to approximating the utility function or the Q-function; second, we may not be able to invent the necessary features, particularly in new domains. If you think about it, these are really the same reason: it is always *possible* to represent U or Q as linear combinations of features, especially if we have features such as $f_1(s) = U(s)$ or $f_2(s,a) = Q(s,a)$, but unless we can come up with such features (in an efficiently computable form) the linear function approximator may be insufficient.

For these reasons (or reason), researchers have explored more complex, nonlinear function approximators since the earliest days of reinforcement learning. Currently, deep neural networks (Chapter 22) are very popular in this role and have proved to be effective even when the input is a raw image with no human-designed feature extraction at all. If all goes well, the deep neural network in effect discovers the useful features for itself. And if the final layer of the network is linear, then we can see what features the network is using to build its own linear function approximator. A reinforcement learning system that uses a deep network as a function approximator is called a deep reinforcement learning system.

Just as in Equation (23.9), the deep network is a function parameterized by θ, except that now the function is much more complicated. The parameters are all the weights in all the layers of the network. Nonetheless, the gradients required for Equations (23.11) and (23.12) are just the same as the gradients required for supervised learning, and they can be computed by the same back-propagation process described in Section 22.4.

As we explain in Section 23.7, deep RL has achieved very significant results, including learning to play a wide range of video games at an expert level, defeating the human world champion at Go, and training robots to perform complex tasks.

Despite its impressive successes, deep RL still faces significant obstacles: it is often difficult to get good performance and the trained system may behave very unpredictably if the environment differs even a little from the training data. Compared to other applications of deep learning, deep RL is rarely applied in commercial settings. It is, nonetheless, a very active area of research.

23.4.4 Reward shaping

As noted in the introduction to this chapter, real-world environments may have very sparse rewards: many primitive actions are required to achieve any nonzero reward. For example, a soccer-playing robot might send a hundred thousand motor control commands to its various joints before conceding a goal. Now it has to work out what it did wrong. The technical term for this is the **credit assignment** problem. Other than playing trillions of soccer games so that the negative reward eventually propagates back to the actions responsible for it, is there a good solution?

Credit assignment

Reward shaping

Pseudoreward

One common method, originally used in animal training, is called **reward shaping**. This involves supplying the agent with additional rewards, called **pseudorewards**, for "making progress." For example, we might give pseudorewards to the robot for making contact with the ball or for advancing it toward the goal. Such rewards can speed up learning enormously and are simple to provide, but there is a risk that the agent will learn to maximize the pseudorewards rather than the true rewards; for example, standing next to the ball and "vibrating" causes many contacts with the ball.

In Chapter 16 (page 559), we saw a way to modify the reward function without changing the optimal policy. For any potential function $\Phi(s)$ and any reward function R, we can create a new reward function R' as follows:

$$R'(s,a,s') = R(s,a,s') + \gamma\Phi(s') - \Phi(s)\,.$$

The potential fuction Φ can be constructed to reflect any desirable aspects of the state, such as achievement of subgoals or distance to a desired terminal state. For example, Φ for the soccer-playing robot could add a constant bonus for states where the robot's team has possession and another bonus for reducing the distance of the ball from the opponents' goal. This will result in faster learning overall, but will not prevent the robot from, say, learning to pass back to the goalkeeper when danger threatens.

23.4.5 Hierarchical reinforcement learning

Another way to cope with very long action sequences is to break them up into a few smaller pieces, and then break those into smaller pieces still, and so on until the action sequences are short enough to make learning easy. This approach is called **hierarchical reinforcement learning** (HRL), and it has much in common with the **HTN planning** methods described in Chapter 11. For example, scoring a goal in soccer can be broken down into obtaining possession, passing to a teammate, receiving the ball from a team-mate, dribbling toward the goal, and shooting; each of these can be broken down further into lower-level motor behaviors. Obviously, there are multiple ways of obtaining possession and shooting, multiple

Hierarchical reinforcement learning

teammates one could pass to, and so on, so each higher-level action may have many different lower-level implementations.

To illustrate these ideas, we'll use a simplified soccer game called **keepaway**, in which one team of three players tries to keep possession of the ball for as long as possible by dribbling and passing amongst themselves while the other team of two players tries to take possession by intercepting a pass or tackling a player in possession.[5] The game is implemented within the RoboCup 2D simulator, which provides detailed continuous-state motion models with 100ms time steps and has proved to be a good testbed for RL systems. Keepaway

A hierarchical reinforcement learning agent begins with a **partial program** that outlines a hierarchical structure for the agent's behavior. The partial-programming language for agent programs extends any ordinary programming language by adding primitives for unspecified choices that must be filled in by learning. (Here, we use pseudocode for the programming language.) The partial program can be arbitrarily complicated, as long as it terminates. Partial program

It is easy to see that HRL includes ordinary RL as a special case. We simply provide the trivial partial program that allows the agent to keep choosing any action from $A(s)$, the set of actions that can be executed in the current state s:

while *true* **do**
  choose$(A(s))$.

The **choose** operator allows the agent to choose any element of the specified set. The learning process converts the partial agent program into a complete program by learning how each choice should be made. For example, the learning process might associate a Q-function with each choice; once the Q-functions are learned, the program produces behavior by choosing the option with the highest Q-value each time it encounters a choice.

The agent programs for keepaway are more interesting. We'll look at the partial program for a single player on the "keeper" team. The choice of what to do at the top level depends mainly on whether the player has the ball or not:

while not IS-TERMINAL(s) **do**
  if BALL-IN-MY-POSSESSION(s) **then choose**$(\{$PASS, HOLD, DRIBBLE$\})$
  else choose$(\{$STAY, MOVE, INTERCEPT-BALL$\})$.

Each of these choices invokes a subroutine that may itself make further choices, all the way down to primitive actions that can be executed directly. For example, the high-level action PASS chooses a teammate to pass to, but also has the choice to do nothing and return control to the higher level if appropriate (e.g., if there is no one to pass to):

choose$(\{$PASS-TO(**choose**(TEAMMATES(s))), **return**$\})$.

The PASS-TO routine then has to choose a speed and direction for the pass. While it is relatively easy for a human—even one with little expertise in soccer—to provide this kind of high-level advice to the learning agent, it would be difficult, if not impossible, to write down the rules for determining the speed and direction of the kick to maximize the probability of maintaining possession. Similarly, it is far from obvious how to choose the right teammate to receive the ball or where to move in order to make oneself available to receive the ball. The partial program provides general know-how—overall scaffolding and structural organization for complex behaviors—and the learning process works out all the details.

[5] Rumors that keepaway was inspired by the real-world tactics of Barcelona FC are probably unfounded.

Joint state space

The theoretical foundations of HRL are based on the concept of the **joint state space**, in which each state (s,m) is composed of a physical state s and a machine state m. The machine state is defined by the current internal state of the agent program: the program counter for each subroutine on the current call stack, the values of the arguments, and the values of all local and global variables. For example, if the agent program has chosen to pass to teammate Ali and is in the middle of calculating the speed of the pass, then the fact that Ali is the argument of PASS-TO is part of the current machine state. A **choice state** $\sigma = (s,m)$ is one in which the program counter for m is at a choice point in the agent program. Between two choice states, any number of computational transitions and physical actions may occur, but they are all preordained, so to speak: by definition, the agent isn't making any choices in between choice states. Essentially, the hierarchical RL agent is solving a Markovian decision problem with the following elements:

Choice state

- The states are the choice states σ of the joint state space.
- The actions at σ are the choices c available in σ according to the partial program.
- The reward function $\rho(\sigma,c,\sigma')$ is the expected sum of rewards for all physical transitions occurring between the choice states σ and σ'.
- The transition model $\tau(\sigma,c,\sigma')$ is defined in the obvious way: if c invokes a physical action a, then τ borrows from the physical model $P(s' \mid s,a)$; if c invokes a computational transition, such as calling a subroutine, then the transition deterministically modifies the computational state m according to the rules of the programming language.[6]

By solving this decision problem, the agent finds the optimal policy that is consistent with original partial program.

Hierarchical RL can be a very effective method for learning complex behaviors. In keepaway, an HRL agent based on the partial program sketched above learns a policy that keeps possession forever against the standard taker policy—a significant improvement on the previous record of about 10 seconds. One important characteristic is that the lower-level skills are not fixed subroutines in the usual sense; their choices are sensitive to the entire internal state of the agent program, so they behave differently depending on where they are invoked within that program and what is going on at the time. If necessary, the Q-functions for the lower-level choices can be initialized by a separate training process with its own reward function, and then integrated into the overall system so they can be adapted to function well in the context of the whole agent.

In the preceding section we saw that shaping rewards can be helpful for learning complex behaviors. In HRL, the fact that learning takes place in the joint state space provides additional opportunities for shaping. For example, to help with learning the Q-function for accurate passing within the PASS-TO routine, we can provide a shaping reward that depends on the location of the intended recipient and the proximity of opponents to that player: the ball should be close to the recipient and far from the opponents. That seems entirely obvious; but *the identity of the intended recipient for a pass is not part of the physical state of the*

[6] Because more than one physical action may be executed before the next choice state is reached, the problem is technically a semi-Markov decision process, which allows actions to have different durations, including stochastic durations. If the discount factor $\gamma < 1$, then the action duration affects the discounting applied to the reward obtained during the action, which means that some extra discount bookkeeping has to be done and the transition model includes the duration distribution.

world. The physical state consists only of the positions, orientations, and velocities of the players and the ball. There is no "passing" and no "recipient" in the physical world; these are entirely internal constructs. This means that there is no way to provide such sensible advice to a standard RL system.

Additive decomposition

The hierarchical structure of behavior also provides a natural **additive decomposition** of the overall utility function. Remember that utility is the sum of rewards over time, and consider a sequence of, say, ten time steps with rewards $[r_1, r_2, \ldots, r_{10}]$. Suppose that for the first five time steps the agent is doing PASS-TO(*Ali*) and for the remaining five steps it is doing MOVE-INTO-SPACE. Then the utility for the initial state is the sum of the total reward during PASS-TO and the total reward during MOVE-INTO-SPACE. The former depends only on whether the ball gets to Ali with enough time and space for Ali to retain possession, and the latter depends only on whether the agent reaches a good location to receive the ball. In other words, the overall utility decomposes into several terms, each of which depends on only a few variables. This, in turns, means that learning occurs much more quickly than if we try to learn a single utility function that depends on all the variables. This is somewhat analogous to the representation theorems underlying the conciseness of Bayes nets (Chapter 13).

23.5 Policy Search

Policy search

The final approach we will consider for reinforcement learning problems is called **policy search**. In some ways, policy search is the simplest of all the methods in this chapter: the idea is to keep twiddling the policy as long as its performance improves, then stop.

Let us begin with the policies themselves. Remember that a policy π is a function that maps states to actions. We are interested primarily in *parameterized* representations of π that have far fewer parameters than there are states in the state space (just as in the preceding section). For example, we could represent π by a collection of parameterized Q-functions, one for each action, and take the action with the highest predicted value:

$$\pi(s) = \operatorname*{argmax}_a \hat{Q}_\theta(s,a)\,. \tag{23.13}$$

Each Q-function could be a linear function, as in Equation (23.9), or it could be a nonlinear function such as a deep neural network. Policy search will then adjust the parameters θ to improve the policy. Notice that if the policy is represented by Q-functions, then policy search results in a process that learns Q-functions. *This process is not the same as Q-learning!*

In Q-learning with function approximation, the algorithm finds a value of θ such that $\hat{Q}_\theta$ is "close" to Q^*, the optimal Q-function. Policy search, on the other hand, finds a value of θ that results in good performance; the values found by the two methods may differ very substantially. (For example, the approximate Q-function defined by $\hat{Q}_\theta(s,a) = Q^*(s,a)/100$ gives optimal performance, even though it is not at all close to Q^*.) Another clear instance of the difference is the case where $\pi(s)$ is calculated using, say, depth-10 look-ahead search with an approximate utility function $\hat{U}_\theta$. A value of θ that gives good results may be a long way from making $\hat{U}_\theta$ resemble the true utility function.

One problem with policy representations of the kind given in Equation (23.13) is that the policy is a *discontinuous* function of the parameters when the actions are discrete. That is, there will be values of θ such that an infinitesimal change in θ causes the policy to switch from one action to another. This means that the value of the policy may also change dis-

Stochastic policy

continuously, which makes gradient-based search difficult. For this reason, policy search methods often use a **stochastic policy** representation $\pi_\theta(s,a)$, which specifies the *probability* of selecting action a in state s. One popular representation is the **softmax** function:

$$\pi_\theta(s,a) = \frac{e^{\beta \hat{Q}_\theta(s,a)}}{\sum_{a'} e^{\beta \hat{Q}_\theta(s,a')}} . \tag{23.14}$$

The parameter $\beta > 0$ modulates the softness of the softmax: for values of β that are large compared to the separations between Q-values, the softmax approaches a hard max, whereas for values of β close to zero the softmax approaches a uniform random choice among the actions. For all finite values of β, the softmax provides a differentiable function of θ; hence, the value of the policy (which depends continuously on the action-selection probabilities) is a differentiable function of θ.

Policy value

Policy gradient

Now let us look at methods for improving the policy. We start with the simplest case: a deterministic policy and a deterministic environment. Let $\rho(\theta)$ be the **policy value**, that is, the expected reward-to-go when π_θ is executed. If we can derive an expression for $\rho(\theta)$ in closed form, then we have a standard optimization problem, as described in Chapter 4. We can follow the **policy gradient** vector $\nabla_\theta \rho(\theta)$, provided $\rho(\theta)$ is differentiable. Alternatively, if $\rho(\theta)$ is not available in closed form, we can evaluate π_θ simply by executing it and observing the accumulated reward. We can follow the **empirical gradient** by hill climbing—that is, evaluating the change in policy value for small increments in each parameter. With the usual caveats, this process will converge to a local optimum in policy space.

When the environment (or the policy) is nondeterministic, things get more difficult. Suppose we are trying to do hill climbing, which requires comparing $\rho(\theta)$ and $\rho(\theta + \Delta\theta)$ for some small $\Delta\theta$. The problem is that the total reward for each trial may vary widely, so estimates of the policy value from a small number of trials will be quite unreliable; trying to compare two such estimates will be even more unreliable. One solution is simply to run lots of trials, measuring the sample variance and using it to determine that enough trials have been run to get a reliable indication of the direction of improvement for $\rho(\theta)$. Unfortunately, this is impractical for many real problems in which trials may be expensive, time-consuming, and perhaps even dangerous.

For the case of a nondeterministic policy $\pi_\theta(s,a)$, it is possible to obtain an unbiased estimate of the gradient at θ, $\nabla_\theta \rho(\theta)$, directly from the results of trials executed at θ. For simplicity, we will derive this estimate for the simple case of an episodic environment in which each action a obtains reward $R(s_0,a,s_0)$ and the environment restarts in s_0. In this case, the policy value is just the expected value of the reward, and we have

$$\nabla_\theta \rho(\theta) = \nabla_\theta \sum_a R(s_0,a,s_0)\pi_\theta(s_0,a) = \sum_a R(s_0,a,s_0)\nabla_\theta \pi_\theta(s_0,a) .$$

Now we perform a simple trick so that this summation can be approximated by samples generated from the probability distribution defined by $\pi_\theta(s_0,a)$. Suppose that we have N trials in all, and the action taken on the jth trial is a_j. Then

$$\begin{aligned} \nabla_\theta \rho(\theta) &= \sum_a \pi_\theta(s_0,a) \cdot \frac{R(s_0,a,s_0)\nabla_\theta \pi_\theta(s_0,a)}{\pi_\theta(s_0,a)} \\ &= \approx \frac{1}{N} \sum_{j=1}^{N} \frac{R(s_0,a_j,s_0)\nabla_\theta \pi_\theta(s_0,a_j)}{\pi_\theta(s_0,a_j)} . \end{aligned}$$

Thus, the true gradient of the policy value is approximated by a sum of terms involving the gradient of the action-selection probability in each trial. For the sequential case, this generalizes to

$$\nabla_\theta \rho(\theta) \approx \frac{1}{N} \sum_{j=1}^{N} \frac{u_j(s) \nabla_\theta \pi_\theta(s, a_j)}{\pi_\theta(s, a_j)}$$

for each state s visited, where a_j is executed in s on the jth trial and $u_j(s)$ is the total reward received from state s onward in the jth trial. The resulting algorithm, called REINFORCE, is due to Ron Williams (1992); it is usually much more effective than hill climbing using lots of trials at each value of θ. However, it is still much slower than necessary.

Consider the following task: given two blackjack policies, determine which is best. The policies might have true net returns per hand of, say, -0.21% and $+0.06\%$, so finding out which is better is very important. One way to do this is to have each policy play against a standard "dealer" for a certain number of hands and then to measure their respective winnings. The problem with this, as we have seen, is that the winnings of each policy fluctuate wildly depending on whether it receives good or bad cards. One would need several million hands to have a reliable indication of which policy is better. The same issue arises when using random sampling to compare two adjacent policies in a hill-climbing algorithm.

A better solution for blackjack is to generate a certain number of hands in advance and *have each program play the same set of hands.* In this way, we eliminate the measurement error due to differences in the cards received. Only a few thousand hands are needed to determine which of the two blackjack policies is better.

This idea, called **correlated sampling**, can be applied to policy search in general, given an environment simulator in which the random-number sequences can be repeated. It was implemented in a policy-search algorithm called PEGASUS (Ng and Jordan, 2000), which was one of the first algorithms to achieve completely stable autonomous helicopter flight (see Figure 23.9(b)). It can be shown that the number of random sequences required to ensure that the value of *every* policy is well estimated depends only on the complexity of the policy space, and not at all on the complexity of the underlying domain.

Correlated sampling

23.6 Apprenticeship and Inverse Reinforcement Learning

Some domains are so complex that it is difficult to define a reward function for use in reinforcement learning. Exactly what do we want our self-driving car to do? Certainly it should not take too long to get to the destination, but it should not drive so fast as to incur undue risk or to get speeding tickets. It should conserve fuel/energy. It should avoid jostling or accelerating the passengers too much, but it can slam on the brakes in an emergency. And so on. Deciding how much weight to give to each of these factors is a difficult task. Worse still, there are almost certainly important factors we have forgotten, such as the obligation to behave with consideration for other drivers. Omitting a factor usually leads to behavior that assigns an extreme value to the omitted factor—in this case, extremely inconsiderate driving—in order to maximize the remaining factors.

One approach is to do extensive testing in simulation, notice problematic behaviors, and try to modify the reward function to eliminate those behaviors. Another approach is to seek additional sources of information about the appropriate reward function. One such source

is the behavior of agents who are already optimizing (or, let's say, nearly optimizing) that reward function—in this case, expert human drivers.

Apprenticeship learning

The general field of **apprenticeship learning** studies the process of learning how to behave well given observations of expert behavior. We show the algorithm examples of expert driving and tell it to "do it like that." There are (at least) two ways to approach the apprenticeship learning problem. The first is the one we discussed briefly at the beginning of the chapter: assuming the environment is observable, we apply supervised learning to the observed state–action pairs to learn a policy $\pi(s)$. This is called **imitation learning**. It has had some success in robotics (see page 973) but suffers from the the problem of brittleness: even small deviations from the training set lead to errors that grow over time and eventually to failure. Moreover, imitation learning will at best duplicate the teacher's performance, not exceed it. When humans learn by imitation, we sometimes use the pejorative term "aping" to describe what they are doing. (It's quite possible that apes use the term "humaning" amongst themselves, perhaps in an even more pejorative sense.) The implication is that the imitation learner doesn't understand *why* it should perform any given action.

Imitation learning

The second approach to apprenticeship learning is to understand *why*: to observe the expert's actions (and resulting states) and try to work out what reward function the expert is maximizing. Then we could derive an optimal policy with respect to that reward function. One expects that this approach will produce robust policies from relatively few examples of expert behavior; after all, the field of reinforcement learning is predicated on the idea that the reward function, rather than the policy or the value function, is the most succinct, robust, and transferable definition of the task. Furthermore, if the learner makes appropriate allowances for possible suboptimality on the part of the expert, then it may be able to do better than the expert by optimizing an accurate approximation to the true reward function. We call this approach **inverse reinforcement learning** (IRL): learning rewards by observing a policy, rather than learning a policy by observing rewards.

Inverse reinforcement learning

How do we find the expert's reward function, given the expert's actions? Let us begin by assuming that the expert was acting rationally. In that case, it seems we should be looking for a reward function R^* such that the total expected discounted reward under the expert's policy is higher than (or at least the same as) under any other possible policy.

Unfortunately, there will be many reward functions that satisfy this constraint; one of them is $R^*(s,a,s') = 0$, because any policy is rational when there are no rewards at all.[7] Another problem with this approach is that the assumption of a rational expert is unrealistic. It means, for example, that a robot observing Lee Sedol making what eventually turns out to be a losing move against ALPHAGO would have to assume that Lee Sedol was trying to lose the game.

To avoid the problem that $R^*(s,a,s') = 0$ explains any observed behavior, it helps to think in a Bayesian way. (See Section 21.1 for a reminder of what this means.) Suppose we observe data $\mathbf{d}$ and let h_R be the hypothesis that R is the true reward function. Then according to Bayes' rule, we have

$$P(h_R \mid \mathbf{d}) = \alpha P(\mathbf{d} \mid h_R) P(h_R) .$$

[7] According to Equation (16.9) on page 559, a reward function $R'(s,a,s') = R(s,a,s') + \gamma\Phi(s') - \Phi(s)$ has exactly the same optimal policies as $R(s,a,s')$, so we can recover the reward function only up to the possible addition of any shaping function $\Phi(s)$. This is not such a serious problem, because a robot using R' will behave just like a robot using the "correct" R.

Now, if the prior $P(h_R)$ is based on simplicity, then the hypothesis that $R=0$ scores fairly well, because 0 is certainly simple. On the other hand, the term $P(\mathbf{d}\,|\,h_R)$ is *infinitesimal* for the hypothesis that $R=0$, because it doesn't explain why the expert chose that particular behavior out of the vast space of behaviors that would be optimal if the hypothesis were true. On the other hand, for a reward function R that has a unique optimal policy or a relatively small equivalence class of optimal policies, $P(\mathbf{d}\,|\,h_R)$ will be far higher.

To allow for the occasional mistake by the expert, we simply allow $P(\mathbf{d}\,|\,h_R)$ to be nonzero even when $\mathbf{d}$ comes from behavior that is a little bit suboptimal according to R. A typical assumption—made, it must be said, more for mathematical convenience than faithfulness to actual human data—is that an agent whose true Q-function is $Q(s,a)$ chooses not according to the deterministic policy $\pi(s)=\arg\max_a Q(s,a)$ but instead according to a stochastic policy defined by the softmax distribution from Equation (23.14). This is sometimes called **Boltzmann rationality** because, in statistical mechanics, the state occupation probabilities in a Boltzmann distribution depend exponentially on their energy levels.

Boltzmann rationality

There are dozens of inverse RL algorithms in the literature. One of the simplest is called **feature matching**. It assumes that the reward function can be written as a weighted linear combination of features:

Feature matching

$$R_\theta(s,a,s') = \sum_{i=1}^{n} \theta_i f_i(s,a,s') = \theta \cdot \mathbf{f}.$$

For example, the features in the driving domain might include speed, speed in excess of the speed limit, acceleration, proximity to nearest obstacle, etc.

Recall from Equation (16.2) on page 557 that the utility of executing a policy π, starting in state s_0, is defined to be

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1})\right],$$

where the expectation E is with respect to the probability distribution over state sequences determined by s and π. Because R is assumed to be a linear combination of feature values, we can rewrite this as follows:

$$\begin{aligned} U^\pi(s) &= E\left[\sum_{t=0}^{\infty} \gamma^t \sum_{i=1}^{n} \theta_i f_i(S_t, \pi(S_t), S_{t+1})\right] \\ &= \sum_{i=1}^{n} \theta_i E\left[\sum_{t=0}^{\infty} \gamma^t f_i(S_t, \pi(S_t), S_{t+1})\right] \\ &= \sum_{i=1}^{n} \theta_i \mu_i(\pi) = \theta \cdot \mu(\pi) \end{aligned}$$

where we have defined the **feature expectation** $\mu_i(\pi)$ as the expected discounted value of the feature f_i when policy π is executed. For example, if f_i is the excess speed of the vehicle (above the speed limit), then $\mu_i(\pi)$ is the (time-discounted) average excess speed over the entire trajectory. The key point about feature expectations is the following: *if a policy π produces feature expectations $\mu_i(\pi)$ that match those of the expert's policy π_E, then π is as good as the expert's policy according to the expert's own reward function.* Now, we cannot measure the exact values for the feature expectations of the expert's policy, but we can approximate them using the average values on the observed trajectories. Thus, we need

Feature expectation

to find values for the parameters θ_i such that the feature expectations of the policy induced by the parameter values match those of the expert policy on the observed trajectories. The following algorithm achieves this with any desired error bound.

- Pick an initial default policy $\pi^{(0)}$.
- For $j = 1, 2, \ldots$ until convergence:
  - Find parameters $\theta^{(j)}$ such that the expert's policy maximally outperforms the policies $\pi^{(0)}, \ldots, \pi^{(j-1)}$ according to the expected utility $\theta^{(j)} \cdot \mu(\pi)$.
  - Let $\pi^{(j)}$ be the optimal policy for the reward function $R^{(j)} = \theta^{(j)} \cdot \mathbf{f}$.

This algorithm converges to a policy that is close in value to the expert's, according to the expert's own reward function. It requires only $O(n \log n)$ iterations and $O(n \log n)$ expert demonstrations, where n is the number of features.

A robot can use inverse reinforcement learning to learn a good policy for itself, by understanding the actions of an expert. In addition, the robot can learn the policies used by other agents in a multiagent domain, whether they be adversarial or cooperative. And finally, inverse reinforcement learning can be used for scientific inquiry (without any thought of agent design), to better understand the behavior of humans and other animals.

A key assumption in inverse RL is that the "expert" is behaving optimally, or nearly optimally, with respect to some reward function in a single-agent MDP. This is a reasonable assumption if the learner is watching the expert through a one-way mirror while the expert goes about his or her business unawares. It is not a reasonable assumption if the expert is aware of the learner. For example, suppose a robot is in medical school, learning to be a surgeon by watching a human expert. An inverse RL algorithm would assume that the human performs the surgery in the usual optimal way, as if the robot were not there. But that's not what would happen: the human surgeon is motivated to have the robot (like any other medical student) learn quickly and well, and so she will modify her behavior considerably. She might explain what she is doing as she goes along; she might point out mistakes to avoid, such as making the incision too deep or the stitches too tight; she might describe the contingency plans in case something goes wrong during surgery. None of these behaviors make sense when performing surgery in isolation, so inverse RL algorithms will not be able to interpret the underlying reward function. Instead, we need to understand this kind of situation as a two-person assistance game, as described in Section 17.2.5.

23.7 Applications of Reinforcement Learning

We now turn to applications of reinforcement learning. These include game playing, where the transition model is known and the goal is to learn the utility function, and robotics, where the model is initially unknown.

23.7.1 Applications in game playing

In Chapter 1 we described Arthur Samuel's early work on reinforcement learning for checkers, which began in 1952. A few decades passed before the challenge was taken up again, this time by Gerry Tesauro in his work on backgammon. Tesauro's first attempt (1990) was a system called NEUROGAMMON. The approach was an interesting variant on imitation learning. The input was a set of 400 games played by Tesauro against himself. Rather than learn a pol-

icy, NEUROGAMMON converted each move (s, a, s') into a set of training examples, each of which labeled s' as a better position than some other position s'' reachable from s by a different move. The network had two separate halves, one for s' and one for s'', and was constrained to choose which was better by comparing the outputs of the two halves. In this way, each half was forced to learn an evaluation function $\hat{U}_\theta$. NEUROGAMMON won the 1989 Computer Olympiad—the first learning program ever to win a computer game tournament—but never progressed past Tesauro's own intermediate level of play.

Tesauro's next system, TD-GAMMON (1992), adopted Sutton's recently published TD learning method—essentially returning to the approach explored by Samuel, but with much greater technical understanding of how to do it right. The evaluation function $\hat{U}_\theta$ was represented by a fully connected neural network with a single hidden layer containing 80 nodes. (It also used some manually designed input features borrowed from NEUROGAMMON.) After 300,000 training games, it reached a standard of play comparable to the top three human players in the world. Kit Woolsey, a top-ten player, said, "There is no question in my mind that its positional judgment is far better than mine."

The next challenge was to learn from raw perceptual inputs—something closer to the real world—rather than discrete game board representations. Beginning in 2012, a team at DeepMind developed the **deep Q-network (DQN)** system, the first modern deep RL system. DQN uses a deep neural network to represent the Q-function; otherwise it is a typical reinforcement learning system. DQN was trained separately on each of 49 different Atari video games. It learned to drive simulated race cars, shoot alien spaceships, and bounce balls with paddles. In each case, the agent learned a Q-function from raw image data with the reward signal being the game score. Overall, the system performed at roughly human expert level, although a few games gave it trouble. One game in particular, *Montezuma's Revenge*, proved far too difficult, because it required extended planning strategies, and the rewards were too sparse. Subsequent work produced deep RL systems that generated more extensive exploratory behaviors and were able to conquer *Montezuma's Revenge* and other difficult games.

Deep Q-network (DQN)

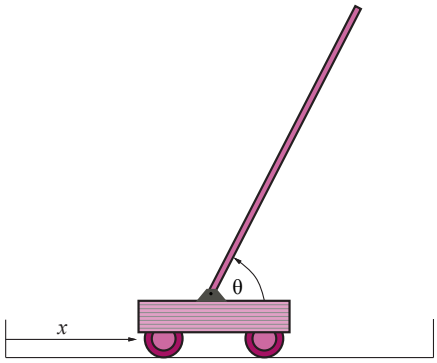
DeepMind's ALPHAGO system also used deep reinforcement learning to beat the best human players at the game of Go (see Chapter 6). Whereas a Q-function with no look-ahead suffices for Atari games, which are primarily reactive in nature, Go requires substantial look-ahead. For this reason, ALPHAGO learned both a value function and a Q-function that guided its search by predicting which moves are worth exploring. The Q-function, implemented as a convolutional neural network, is accurate enough by itself to beat most amateur human players without any search at all.

23.7.2 Application to robot control

The setup for the famous **cart–pole** balancing problem, also known as the **inverted pendulum**, is shown in Figure 23.9(a). The problem is to keep the pole roughly upright ($\theta \approx 90°$) by applying forces to move the cart right or left, while keeping the position x within the limits of the track. Several thousand papers in reinforcement learning and control theory have been published on this seemingly simple problem. One difficulty is that the state variables x, θ, $\dot{x}$, and $\dot{\theta}$ are continuous. The actions, however, are defined to be discrete: jerk left or jerk right, the so-called **bang-bang control** regime.

Cart–pole
Inverted pendulum
Bang-bang control

The earliest work on learning for this problem was carried out by Michie and Chambers (1968), using a real cart and pole, not a simulation. Their BOXES algorithm was able



(a)

(b)

Figure 23.9 (a) Setup for the problem of balancing a long pole on top of a moving cart. The cart can be jerked left or right by a controller that observes the cart's position x and velocity $\dot{x}$, as well as the pole's angle θ and rate of change of angle $\dot{\theta}$. (b) Six superimposed time-lapse images of a single autonomous helicopter performing a very difficult "nose-in circle" maneuver. The helicopter is under the control of a policy developed by the PEGASUS policy-search algorithm (Ng *et al.*, 2003). A simulator model was developed by observing the effects of various control manipulations on the real helicopter; then the algorithm was run on the simulator model overnight. A variety of controllers were developed for different maneuvers. In all cases, performance far exceeded that of an expert human pilot using remote control. (Image courtesy of Andrew Ng.)

to balance the pole for over an hour after 30 trials. The algorithm first discretized the four-dimensional state space into boxes—hence the name. It then ran trials until the pole fell over. Negative reinforcement was associated with the final action in the final box and then propagated back through the sequence. Improved generalization and faster learning can be obtained using an algorithm that *adaptively* partitions the state space according to the observed variation in the reward, or by using a continuous-state, nonlinear function approximator such as a neural network. Nowadays, balancing a *triple* inverted pendulum (three poles joined together end to end) is a common exercise—a feat far beyond the capabilities of most humans, but achievable using reinforcement learning.

Still more impressive is the application of reinforcement learning to radio-controlled helicopter flight (Figure 23.9(b)). This work has generally used policy search over large MDPs (Bagnell and Schneider, 2001; Ng *et al.*, 2003), often combined with imitation learning and inverse RL given observations of a human expert pilot (Coates *et al.*, 2009).

Inverse RL has also been applied successfully to interpret human behavior, including destination prediction and route selection by taxi drivers based on 100,000 miles of GPS data (Ziebart *et al.*, 2008) and detailed physical movements by pedestrians in complex environments based on hours of video observation (Kitani *et al.*, 2012). In the area of robotics, a single expert demonstration was enough for the LittleDog quadruped to learn a 25-feature reward function and nimbly traverse a previously unseen area of rocky terrain (Kolter *et al.*, 2008). For more on how RL and inverse RL are used in robotics, see Sections 26.7 and 26.8.

Summary

This chapter has examined the reinforcement learning problem: how an agent can become proficient in an unknown environment, given only its percepts and occasional rewards. Reinforcement learning is a very broadly applicable paradigm for creating intelligent systems. The major points of the chapter are as follows.

- The overall agent design dictates the kind of information that must be learned:
  - A **model-based reinforcement learning** agent acquires (or is equipped with) a transition model $P(s' \mid s,a)$ for the environment and learns a utility function $U(s)$.
  - A **model-free reinforcement learning** agent may learn an action-utility function $Q(s,a)$ or a policy $\pi(s)$.
- Utilities can be learned using several different approaches:
  - **Direct utility estimation** uses the total observed reward-to-go for a given state as direct evidence for learning its utility.
  - **Adaptive dynamic programming** (ADP) learns a model and a reward function from observations and then uses value or policy iteration to obtain the utilities or an optimal policy. ADP makes optimal use of the local constraints on utilities of states imposed through the neighborhood structure of the environment.
  - **Temporal-difference** (TD) methods adjust utility estimates to be more consistent with those of successor states. They can be viewed as simple approximations of the ADP approach that can learn without requiring a transition model. Using a learned model to generate pseudoexperiences can, however, result in faster learning.
- Action-utility functions, or Q-functions, can be learned by an ADP approach or a TD approach. With TD, **Q-learning** requires no model in either the learning or action-selection phase. This simplifies the learning problem but potentially restricts the ability to learn in complex environments, because the agent cannot simulate the results of possible courses of action.
- When the learning agent is responsible for selecting actions while it learns, it must trade off the estimated value of those actions against the potential for learning useful new information. An exact solution for the exploration problem is infeasible, but some simple heuristics do a reasonable job. An exploring agent must also take care to avoid premature death.
- In large state spaces, reinforcement learning algorithms must use an approximate functional representation of $U(s)$ or $Q(s,a)$ in order to generalize over states. **Deep reinforcement learning**—using deep neural networks as function approximators—has achieved considerable success on hard problems.
- **Reward shaping** and **hierarchical reinforcement learning** are helpful for learning complex behaviors, particularly when rewards are sparse and long action sequences are required to obtain them.
- **Policy-search** methods operate directly on a representation of the policy, attempting to improve it based on observed performance. The variation in the performance in a stochastic domain is a serious problem; for simulated domains this can be overcome by fixing the randomness in advance.

- **Apprenticeship learning** through observation of expert behavior can be an effective solution when a correct reward function is hard to specify. **Imitation learning** formulates the problem as supervised learning of a policy from the expert's state–action pairs. **Inverse reinforcement learning** infers reward information from the expert's behavior.

Reinforcement learning continues to be one of the most active areas of machine learning research. It frees us from manual construction of behaviors and from labeling the vast data sets required for supervised learning, or having to hand-code control strategies. Applications in robotics promise to be particularly valuable; these will require methods for handling continuous, high-dimensional, partially observable environments in which successful behaviors may consist of thousands or even millions of primitive actions.

We have presented a variety of approaches to reinforcement learning because there is (at least so far) no single best approach. The question of model-based versus model-free methods is, at its heart, a question about the best way to represent the agent function. This is an issue at the foundations of artificial intelligence. As we stated in Chapter 1, one of the key historical characteristics of much AI research is its (often unstated) adherence to the **knowledge-based** approach. This amounts to an assumption that the best way to represent the agent function is to build a representation of some aspects of the environment in which the agent is situated. Some argue that with access to sufficient data, model-free methods can succeed in any domain. Perhaps this is true in theory, but of course, the universe may not contain enough data to make it true in practice. (For example, it is not easy to imagine how a model-free approach would enable one to design and build, say, the LIGO gravity-wave detector.) Our intuition, for what it's worth, is that as the environment becomes more complex, the advantages of a model-based approach become more apparent.

Bibliographical and Historical Notes

It seems likely that the key idea of reinforcement learning—that animals do more of what they are rewarded for and less of what they are punished for—played a significant role in the domestication of dogs at least 15,000 years ago. The early foundations of our scientific understanding of reinforcement learning include the work of the Russian physiologist Ivan Pavlov, who won the Nobel Prize in 1904, and that of the American psychologist Edward Thorndike—particularly his book *Animal Intelligence* (1911). Hilgard and Bower (1975) provide a good survey.

Alan Turing (1948, 1950) proposed reinforcement learning as an approach for teaching computers; he considered it a partial solution, writing, "The use of punishments and rewards can at best be a part of the teaching process." Arthur Samuel's checkers program (1959, 1967) was the first successful use of machine learning of any kind. Samuel suggested most of the modern ideas in reinforcement learning, including temporal-difference learning and function approximation. He experimented with multilayer representations of value functions, similar to today's deep RL. In the end, he found that a simple linear evaluation function over handcrafted features worked best. This may have been a consequence of working with a computer roughly 100 billion times less powerful than a modern tensor processing unit.

Around the same time, researchers in adaptive control theory (Widrow and Hoff, 1960), building on work by Hebb (1949), were training simple networks using the delta rule. Thus,

reinforcement learning combines influences from animal psychology, neuroscience, operations research, and optimal control theory.

The connection between reinforcement learning and Markov decision processes was first made by Werbos (1977). (Work by Ian Witten (1977) described a TD-like process in the language of control theory.) The development of reinforcement learning in AI stems primarily from work at the University of Massachusetts in the early 1980s (Barto *et al.*, 1981). An influential paper by Rich Sutton (1988) provided a mathematical understanding of temporal-difference methods. The combination of temporal-difference learning with the model-based generation of simulated experiences was proposed in Sutton's DYNA architecture (Sutton, 1990). Q-learning was developed in Chris Watkins's Ph.D. thesis (1989), while SARSA appeared in a technical report by Rummery and Niranjan (1994). Prioritized sweeping was introduced independently by Moore and Atkeson (1993) and Peng and Williams (1993).

Function approximation in reinforcement learning goes back to Arthur Samuel's checkers program (1959). The use of neural networks to represent value functions was common in the 1980s and came to the fore in Gerry Tesauro's TD-Gammon program (Tesauro, 1992, 1995). Deep neural networks are currently the most popular choice for function approximators in reinforcement learning. Arulkumaran *et al.* (2017) and Francois-Lavet *et al.* (2018) give overviews of deep RL. The DQN system (Mnih *et al.*, 2015) uses a deep network to learn a Q-function, while ALPHAZERO (Silver *et al.*, 2018) learns both a value function for use with a known model and a Q-function for use in metalevel decisions that guide search. Irpan (2018) warns that deep RL systems can perform poorly if the actual environment is even slightly different from the training environment.

Weighted linear combinations of features and neural networks are factored representations for function approximation. It is also possible to apply reinforcement learning to *structured* representations; this is called **relational reinforcement learning** (Tadepalli *et al.*, 2004). The use of relational descriptions allows for generalization across complex behaviors involving different objects.

Analysis of the convergence properties of reinforcement learning algorithms using function approximation is an extremely technical subject. Results for TD learning have been progressively strengthened for the case of linear function approximators (Sutton, 1988; Dayan, 1992; Tsitsiklis and Van Roy, 1997), but several examples of divergence have been presented for nonlinear functions (see Tsitsiklis and Van Roy, 1997, for a discussion). Papavassiliou and Russell (1999) describe a type of reinforcement learning that converges with any form of function approximator, provided that the problem of fitting the hypothesis to the data is solvable. Liu *et al.* (2018) describe the family of **gradient TD** algorithms and provide extensive theoretical analysis of convergence and sample complexity.

A variety of exploration methods for sequential decision problems are discussed by Barto *et al.* (1995). Kearns and Singh (1998) and Brafman and Tennenholtz (2000) describe algorithms that explore unknown environments and are guaranteed to converge on near-optimal policies with a sample complexity that is polynomial in the number of states. Bayesian reinforcement learning (Dearden *et al.*, 1998, 1999) provides another angle on both model uncertainty and exploration.

The basic idea underlying imitation learning is to apply supervised learning to a training set of expert actions. This is an old idea in adaptive control, but first came to prominence in AI with the work of Sammut *et al.* (1992) on "Learning to Fly" in a flight simulator.

They called their method **behavioral cloning**. A few years later, the same research group reported that the method was much more fragile than had been reported initially (Camacho and Michie, 1995): even very small perturbations caused the learned policy to deviate from the desired trajectory, leading to compounding errors as the agent strayed further and further from the training set. (See also the discussion on page 973.) Work on apprenticeship learning aims to make the approach more robust, in part by including information about the desired outcomes rather than just the expert policy. Ng *et al.* (2003) and Coates *et al.* (2009) show how apprenticeship learning works for learning to fly an actual helicopter, as illustrated in Figure 23.9(b) on page 868.

Inverse reinforcement learning (IRL) was introduced by Russell (1998), and the first algorithms were developed by Ng and Russell (2000). (A similar problem has been studied in economics for much longer, under the heading of **structural estimation of MDPs** (Sargent, 1978).) The algorithm given in the chapter is due to Abbeel and Ng (2004). Baker *et al.* (2009) describe how the understanding of another agent's actions can be seen as inverse planning. Ho *et al.* (2017) show that agents can learn better from behaviors that are *instructive* rather than *optimal*. Hadfield-Menell *et al.* (2017a) extend IRL into a game-theoretic formulation that encompasses both observer and demonstrator, showing how teaching and learning behaviors emerge as solutions of the game.

García and Fernández (2015) give a comprehensive survey on safe reinforcement learning. Munos *et al.* (2017) describe an algorithm for safe off-policy (e.g., Q-learning) exploration. Hans *et al.* (2008) break the problem of safe exploration into two parts: defining a safety function to indicate which states to avoid, and defining a backup policy to lead the agent back to safety when it might otherwise enter an unsafe state. You *et al.* (2017) show how to train a deep reinforcement learning model to drive a car in simulation, and then use transfer learning to drive safely in the real world.

Thomas *et al.* (2017) offer an approach to learning that is guaranteed, with high probability, to do no worse than the current policy. Akametalu *et al.* (2014) describe a reachability-based approach, in which the learning process operates under the guidance of a control policy that ensures the agent never reaches an unsafe state. Saunders *et al.* (2018) demonstrate that a system can use human intervention to stop it from wandering out of the safe region, and can learn over time to need less intervention.

Policy search methods were brought to the fore by Williams (1992), who developed the REINFORCE family of algorithms, which stands for "REward Increment = Nonnegative Factor $\times$ Offset Reinforcement $\times$ Characteristic Eligibility." Later work by Marbach and Tsitsiklis (1998), Sutton *et al.* (2000), and Baxter and Bartlett (2000) strengthened and generalized the convergence results for policy search. Schulman *et al.* (2015b) describe **trust region policy optimization**, a theoretically well-founded and also practical policy search algorithm that has spawned many variants. The method of correlated sampling to reduce variance in Monte Carlo comparisons is due to Kahn and Marshall (1953); it is also one of a number of variance reduction methods explored by Hammersley and Handscomb (1964).

Early approaches to hierarchical reinforcement learning (HRL) attempted to construct hierarchies using **state abstraction**—that is, grouping states together into abstract states and then doing RL in the abstract state space (Dayan and Hinton, 1993). Unfortunately, the transition model for abstract states is typically non-Markovian, leading to divergent behavior of standard RL algorithms. The temporal abstraction approach in this chapter was developed in

the late 1990s (Parr and Russell, 1998; Andre and Russell, 2002; Sutton *et al.*, 2000) and extended to handle concurrent behaviors by Marthi *et al.* (2005). Dietterich (2000) introduced the notion of an additive decomposition of Q-functions induced by the subroutine hierarchy. Temporal abstraction is based on a much earlier result due to Forestier and Varaiya (1978), who showed that a large MDP can be decomposed into a two-layer system in which a supervisory layer chooses among low-level controllers, each of which returns control to the supervisor on completion. The problem of learning the abstraction hierarchy itself has been studied at least since the work of Peter Andreae (1985); for a recent exploration into learning robot motion primitives, see Frans *et al.* (2018). The keepaway game was introduced by Stone *et al.* (2005); the HRL solution given here is due to Bai and Russell (2017).

Neuroscience has often inspired reinforcement learning and confirmed the value of the approach. Research using single-cell recording suggests that the dopamine system in primate brains implements something resembling value-function learning (Schultz *et al.*, 1997). The neuroscience text by Dayan and Abbott (2001) describes possible neural implementations of temporal-difference learning; related research describes other neuroscientific and behavioral experiments (Dayan and Niv, 2008; Niv, 2009; Lee *et al.*, 2012).

Work in reinforcement learning has been accelerated by the availability of open-source simulation environments for developing and testing learning agents. The University of Alberta's Arcade Learning Environment (ALE) (Bellemare *et al.*, 2013) provided such a framework for 55 classic Atari video games. The pixels on the screen are provided to the agent as percepts, along with a hardwired score of the game so far. ALE was used by the DeepMind team to implement DQN learning and verify the generality of their system on a wide variety of games (Mnih *et al.*, 2015).

DeepMind in turn open-sourced several agent platforms, including the DeepMind Lab (Beattie *et al.*, 2016), the AI Safety Gridworlds (Leike *et al.*, 2017), the Unity game platform (Juliani *et al.*, 2018), and the DM Control Suite (Tassa *et al.*, 2018). Blizzard released the StarCraft II Learning Environment (SC2LE), to which DeepMind added the PySC2 component for machine learning in Python (Vinyals *et al.*, 2017a).

Facebook's AI Habitat simulation (Savva *et al.*, 2019) provides a photo-realistic virtual environment for indoor robotic tasks, and their HORIZON platform (Gauci *et al.*, 2018) enables reinforcement learning in large-scale production systems. The SYNTHIA system (Ros *et al.*, 2016) is a simulation environment designed for improving the computer vision capabilities of self-driving cars. The OpenAI Gym (Brockman *et al.*, 2016) provides several environments for reinforcement learning agents, and is compatible with other simulations such as the Google Football simulator.

Littman (2015) surveys reinforcement learning for a general scientific audience. The canonical text by Sutton and Barto (2018), two of the field's pioneers, shows how reinforcement learning weaves together the ideas of learning, planning, and acting. Kochenderfer (2015) takes a slightly less mathematical approach, with plenty of real-world examples. A short book by Szepesvari (2010) gives an overview of reinforcement learning algorithms. Bertsekas and Tsitsiklis (1996) provide a rigorous grounding in the theory of dynamic programming and stochastic convergence. Reinforcement learning papers are published frequently in the journals *Machine Learning* and *Journal of Machine Learning Research*, and in the the proceedings of the International Conference on Machine Learning (ICML) and the Neural Information Processing Systems (NeurIPS) conferences.