CHAPTER 13

REINFORCEMENT LEARNING

Reinforcement learning addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals. This very generic problem covers tasks such as learning to control a mobile robot, learning to optimize operations in factories, and learning to play board games. Each time the agent performs an action in its environment, a trainer may provide a reward or penalty to indicate the desirability of the resulting state. For example, when training an agent to play a game the trainer might provide a positive reward when the game is won, negative reward when it is lost, and zero reward in all other states. The task of the agent is to learn from this indirect, delayed reward, to choose sequences of actions that produce the greatest cumulative reward. This chapter focuses on an algorithm called Q learning that can acquire optimal control strategies from delayed rewards, even when the agent has no prior knowledge of the effects of its actions on the environment. Reinforcement learning algorithms are related to dynamic programming algorithms frequently used to solve optimization problems.

13.1 INTRODUCTION

Consider building a learning robot. The robot, or *agent*, has a set of sensors to observe the *state* of its environment, and a set of *actions* it can perform to alter this state. For example, a mobile robot may have sensors such as a camera and sonars, and actions such as "move forward" and "turn." Its task is to learn a control strategy, or *policy*, for choosing actions that achieve its goals. For example, the robot may have a goal of docking onto its battery charger whenever its battery level is low.

This chapter is concerned with how such agents can learn successful control policies by experimenting in their environment. We assume that the goals of the agent can be defined by a *reward* function that assigns a numerical value—an immediate payoff—to each distinct action the agent may take from each distinct state. For example, the goal of docking to the battery charger can be captured by assigning a positive reward (e.g., +100) to state-action transitions that immediately result in a connection to the charger and a reward of zero to every other state-action transition. This reward function may be built into the robot, or known only to an external teacher who provides the reward value for each action performed by the robot. The task of the robot is to perform sequences of actions, observe their consequences, and learn a control policy. The control policy we desire is one that, from any initial state, chooses actions that maximize the reward accumulated over time by the agent. This general setting for robot learning is summarized in Figure 13.1.

As is apparent from Figure 13.1, the problem of learning a control policy to maximize cumulative reward is very general and covers many problems beyond robot learning tasks. In general the problem is one of learning to control sequential processes. This includes, for example, manufacturing optimization problems in which a sequence of manufacturing actions must be chosen, and the reward to be maximized is the value of the goods produced minus the costs involved. It includes sequential scheduling problems such as choosing which taxis to send for passengers in a large city, where the reward to be maximized is a function of the wait time of the passengers and the total fuel costs of the taxi fleet. In general, we are interested in any type of agent that must learn to choose actions that alter the state of its environment and where a cumulative reward function is used to define the quality of any given action sequence. Within this class of problems we will consider specific settings, including settings in which the actions have deterministic or nondeterministic outcomes, and settings in which the agent

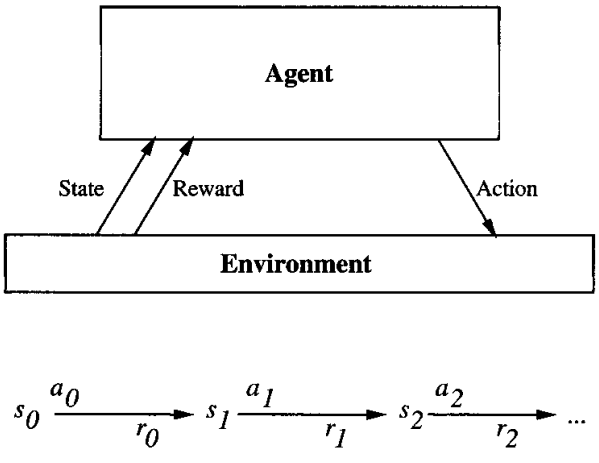


FIGURE 13.1
An agent interacting with its environment. The agent exists in an environment described by some set of possible states S. It can perform any of a set of possible actions A. Each time it performs an action a_t in some state s_t the agent receives a real-valued reward r_t that indicates the immediate value of this state-action transition. This produces a sequence of states s_i, actions a_i, and immediate rewards r_i as shown in the figure. The agent's task is to learn a control policy, $\pi : S \rightarrow A$, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.

has or does not have prior knowledge about the effects of its actions on the environment.

Note we have touched on the problem of learning to control sequential processes earlier in this book. In Section 11.4 we discussed explanation-based learning of rules to control search during problem solving. There the problem is for the agent to choose among alternative actions at each step in its search for some goal state. The techniques discussed here differ from those of Section 11.4, in that here we consider problems where the actions may have nondeterministic outcomes and where the learner lacks a domain theory that describes the outcomes of its actions. In Chapter 1 we discussed the problem of learning to choose actions while playing the game of checkers. There we sketched the design of a learning method very similar to those discussed in this chapter. In fact, one highly successful application of the reinforcement learning algorithms of this chapter is to a similar game-playing problem. Tesauro (1995) describes the TD-GAMMON program, which has used reinforcement learning to become a world-class backgammon player. This program, after training on 1.5 million self-generated games, is now considered nearly equal to the best human players in the world and has played competitively against top-ranked players in international backgammon tournaments.

The problem of learning a control policy to choose actions is similar in some respects to the function approximation problems discussed in other chapters. The target function to be learned in this case is a control policy, $\pi : S \rightarrow A$, that outputs an appropriate action a from the set A, given the current state s from the set S. However, this reinforcement learning problem differs from other function approximation tasks in several important respects.

- *Delayed reward.* The task of the agent is to learn a target function π that maps from the current state s to the optimal action $a = \pi(s)$. In earlier chapters we have always assumed that when learning some target function such as π, each training example would be a pair of the form $\langle s, \pi(s) \rangle$. In reinforcement learning, however, training information is not available in this form. Instead, the trainer provides only a sequence of immediate reward values as the agent executes its sequence of actions. The agent, therefore, faces the problem of *temporal credit assignment*: determining which of the actions in its sequence are to be credited with producing the eventual rewards.
- *Exploration.* In reinforcement learning, the agent influences the distribution of training examples by the action sequence it chooses. This raises the question of which experimentation strategy produces most effective learning. The learner faces a tradeoff in choosing whether to favor *exploration* of unknown states and actions (to gather new information), or *exploitation* of states and actions that it has already learned will yield high reward (to maximize its cumulative reward).
- *Partially observable states.* Although it is convenient to assume that the agent's sensors can perceive the entire state of the environment at each time step, in many practical situations sensors provide only partial information. For example, a robot with a forward-pointing camera cannot see what is

behind it. In such cases, it may be necessary for the agent to consider its previous observations together with its current sensor data when choosing actions, and the best policy may be one that chooses actions specifically to improve the observability of the environment.

- *Life-long learning.* Unlike isolated function approximation tasks, robot learning often requires that the robot learn several related tasks within the same environment, using the same sensors. For example, a mobile robot may need to learn how to dock on its battery charger, how to navigate through narrow corridors, and how to pick up output from laser printers. This setting raises the possibility of using previously obtained experience or knowledge to reduce sample complexity when learning new tasks.

13.2 THE LEARNING TASK

In this section we formulate the problem of learning sequential control strategies more precisely. Note there are many ways to do so. For example, we might assume the agent's actions are deterministic or that they are nondeterministic. We might assume that the agent can predict the next state that will result from each action, or that it cannot. We might assume that the agent is trained by an expert who shows it examples of optimal action sequences, or that it must train itself by performing actions of its own choice. Here we define one quite general formulation of the problem, based on Markov decision processes. This formulation of the problem follows the problem illustrated in Figure 13.1.

In a Markov decision process (MDP) the agent can perceive a set S of distinct states of its environment and has a set A of actions that it can perform. At each discrete time step t, the agent senses the current state s_t, chooses a current action a_t, and performs it. The environment responds by giving the agent a reward $r_t = r(s_t, a_t)$ and by producing the succeeding state $s_{t+1} = \delta(s_t, a_t)$. Here the functions δ and r are part of the environment and are not necessarily known to the agent. In an MDP, the functions $\delta(s_t, a_t)$ and $r(s_t, a_t)$ depend only on the current state and action, and not on earlier states or actions. In this chapter we consider only the case in which S and A are finite. In general, δ and r may be nondeterministic functions, but we begin by considering only the deterministic case.

The task of the agent is to learn a *policy*, $\pi : S \rightarrow A$, for selecting its next action a_t based on the current observed state s_t; that is, $\pi(s_t) = a_t$. How shall we specify precisely which policy π we would like the agent to learn? One obvious approach is to require the policy that produces the greatest possible cumulative reward for the robot over time. To state this requirement more precisely, we define the cumulative value $V^{\pi}(s_t)$ achieved by following an arbitrary policy π from an arbitrary initial state s_t as follows:

$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$

$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{13.1}$$

where the sequence of rewards r_{t+i} is generated by beginning at state s_t and by repeatedly using the policy π to select actions as described above (i.e., $a_t = \pi(s_t)$, $a_{t+1} = \pi(s_{t+1})$, etc.). Here $0 \leq \gamma < 1$ is a constant that determines the relative value of delayed versus immediate rewards. In particular, rewards received i time steps into the future are discounted exponentially by a factor of γ^i. Note if we set $\gamma = 0$, only the immediate reward is considered. As we set γ closer to 1, future rewards are given greater emphasis relative to the immediate reward.

The quantity $V^\pi(s)$ defined by Equation (13.1) is often called the *discounted cumulative reward* achieved by policy π from initial state s. It is reasonable to discount future rewards relative to immediate rewards because, in many cases, we prefer to obtain the reward sooner rather than later. However, other definitions of total reward have also been explored. For example, *finite horizon* reward, $\sum_{i=0}^{h} r_{t+i}$, considers the undiscounted sum of rewards over a finite number h of steps. Another possibility is *average reward*, $\lim_{h\to\infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}$, which considers the average reward per time step over the entire lifetime of the agent. In this chapter we restrict ourselves to considering discounted reward as defined by Equation (13.1). Mahadevan (1996) provides a discussion of reinforcement learning when the criterion to be optimized is average reward.

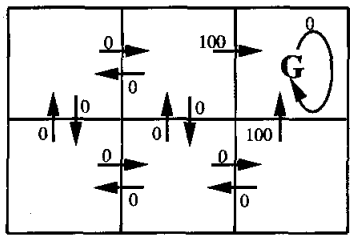
We are now in a position to state precisely the agent's learning task. We require that the agent learn a policy π that maximizes $V^\pi(s)$ for all states s. We will call such a policy an *optimal policy* and denote it by π^*.

$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}}\, V^\pi(s), (\forall s) \tag{13.2}$$

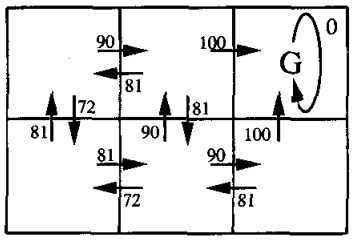
To simplify notation, we will refer to the value function $V^{\pi^*}(s)$ of such an optimal policy as $V^*(s)$. $V^*(s)$ gives the maximum discounted cumulative reward that the agent can obtain starting from state s; that is, the discounted cumulative reward obtained by following the optimal policy beginning at state s.

To illustrate these concepts, a simple grid-world environment is depicted in the topmost diagram of Figure 13.2. The six grid squares in this diagram represent six possible states, or locations, for the agent. Each arrow in the diagram represents a possible action the agent can take to move from one state to another. The number associated with each arrow represents the immediate reward $r(s, a)$ the agent receives if it executes the corresponding state-action transition. Note the immediate reward in this particular environment is defined to be zero for all state-action transitions except for those leading into the state labeled **G**. It is convenient to think of the state **G** as the goal state, because the only way the agent can receive reward, in this case, is by entering this state. Note in this particular environment, the only action available to the agent once it enters the state **G** is to remain in this state. For this reason, we call **G** an *absorbing* state.

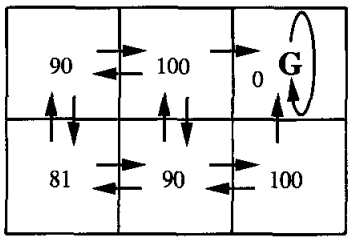
Once the states, actions, and immediate rewards are defined, and once we choose a value for the discount factor γ, we can determine the optimal policy π^* and its value function $V^*(s)$. In this case, let us choose $\gamma = 0.9$. The diagram at the bottom of the figure shows one optimal policy for this setting (there are others as well). Like any policy, this policy specifies exactly one action that the



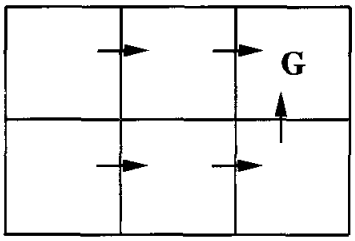
$r(s, a)$ (immediate reward) values



$Q(s, a)$ values



$V^*(s)$ values



One optimal policy

FIGURE 13.2
A simple deterministic world to illustrate the basic concepts of Q-learning. Each grid square represents a distinct state, each arrow a distinct action. The immediate reward function, $r(s, a)$ gives reward 100 for actions entering the goal state **G**, and zero otherwise. Values of $V^*(s)$ and $Q(s, a)$ follow from $r(s, a)$, and the discount factor $\gamma = 0.9$. An optimal policy, corresponding to actions with maximal Q values, is also shown.

agent will select in any given state. Not surprisingly, the optimal policy directs the agent along the shortest path toward the state **G**.

The diagram at the right of Figure 13.2 shows the values of V^* for each state. For example, consider the bottom right state in this diagram. The value of V^* for this state is 100 because the optimal policy in this state selects the "move up" action that receives immediate reward 100. Thereafter, the agent will remain in the absorbing state and receive no further rewards. Similarly, the value of V^* for the bottom center state is 90. This is because the optimal policy will move the agent from this state to the right (generating an immediate reward of zero), then upward (generating an immediate reward of 100). Thus, the discounted future reward from the bottom center state is

$$0 + \gamma 100 + \gamma^2 0 + \gamma^3 0 + \cdots = 90$$

Recall that V^* is defined to be the sum of discounted future rewards over the infinite future. In this particular environment, once the agent reaches the absorbing state **G** its infinite future will consist of remaining in this state and receiving rewards of zero.

13.3 Q LEARNING

How can an agent learn an optimal policy π^* for an arbitrary environment? It is difficult to learn the function $\pi^* : S \rightarrow A$ directly, because the available training data does not provide training examples of the form $\langle s, a \rangle$. Instead, the only training information available to the learner is the sequence of immediate rewards $r(s_i, a_i)$ for $i = 0, 1, 2, \ldots$. As we shall see, given this kind of training information it is easier to learn a numerical evaluation function defined over states and actions, then implement the optimal policy in terms of this evaluation function.

What evaluation function should the agent attempt to learn? One obvious choice is V^*. The agent should prefer state s_1 over state s_2 whenever $V^*(s_1) > V^*(s_2)$, because the cumulative future reward will be greater from s_1. Of course the agent's policy must choose among actions, not among states. However, it can use V^* in certain settings to choose among actions as well. The optimal action in state s is the action a that maximizes the sum of the immediate reward $r(s, a)$ plus the value V^* of the immediate successor state, discounted by γ.

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s, a) + \gamma V^*(\delta(s, a))] \tag{13.3}$$

(recall that $\delta(s, a)$ denotes the state resulting from applying action a to state s.) Thus, the agent can acquire the optimal policy by learning V^*, *provided it has perfect knowledge of the immediate reward function r and the state transition function δ*. When the agent knows the functions r and δ used by the environment to respond to its actions, it can then use Equation (13.3) to calculate the optimal action for any state s.

Unfortunately, learning V^* is a useful way to learn the optimal policy *only* when the agent has perfect knowledge of δ and r. This requires that it be able to perfectly predict the immediate result (i.e., the immediate reward and immediate successor) for every possible state-action transition. This assumption is comparable to the assumption of a perfect domain theory in explanation-based learning, discussed in Chapter 11. In many practical problems, such as robot control, it is impossible for the agent or its human programmer to predict in advance the exact outcome of applying an arbitrary action to an arbitrary state. Imagine, for example, the difficulty in describing δ for a robot arm shoveling dirt when the resulting state includes the positions of the dirt particles. In cases where either δ or r is unknown, learning V^* is unfortunately of no use for selecting optimal actions because the agent cannot evaluate Equation (13.3). What evaluation function should the agent use in this more general setting? The evaluation function Q, defined in the following section, provides one answer.

13.3.1 The Q Function

Let us define the evaluation function $Q(s, a)$ so that its value is the maximum discounted cumulative reward that can be achieved starting from state s and applying action a as the first action. In other words, the value of Q is the reward received immediately upon executing action a from state s, plus the value (discounted by γ) of following the optimal policy thereafter.

$$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a)) \tag{13.4}$$

Note that $Q(s, a)$ is exactly the quantity that is maximized in Equation (13.3) in order to choose the optimal action a in state s. Therefore, we can rewrite Equation (13.3) in terms of $Q(s, a)$ as

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s, a) \tag{13.5}$$

Why is this rewrite important? Because it shows that if the agent learns the Q function instead of the V^* function, it will be able to select optimal actions *even when it has no knowledge of the functions* r *and* δ. As Equation (13.5) makes clear, it need only consider each available action a in its current state s and choose the action that maximizes $Q(s, a)$.

It may at first seem surprising that one can choose globally optimal action sequences by reacting repeatedly to the local values of Q for the current state. This means the agent can choose the optimal action without ever conducting a lookahead search to explicitly consider what state results from the action. Part of the beauty of Q learning is that the evaluation function is defined to have precisely this property—the value of Q for the current state and action summarizes in a single number all the information needed to determine the discounted cumulative reward that will be gained in the future if action a is selected in state s.

To illustrate, Figure 13.2 shows the Q values for every state and action in the simple grid world. Notice that the Q value for each state-action transition equals the r value for this transition plus the V^* value for the resulting state discounted by γ. Note also that the optimal policy shown in the figure corresponds to selecting actions with maximal Q values.

13.3.2 An Algorithm for Learning Q

Learning the Q function corresponds to learning the optimal policy. How can Q be learned?

The key problem is finding a reliable way to estimate training values for Q, given only a sequence of immediate rewards r spread out over time. This can be accomplished through iterative approximation. To see how, notice the close relationship between Q and V^*,

$$V^*(s) = \max_{a'} Q(s, a')$$

which allows rewriting Equation (13.4) as

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a') \tag{13.6}$$

This recursive definition of Q provides the basis for algorithms that iteratively approximate Q (Watkins 1989). To describe the algorithm, we will use the symbol $\hat{Q}$ to refer to the learner's estimate, or hypothesis, of the actual Q function. In this algorithm the learner represents its hypothesis $\hat{Q}$ by a large table with a separate entry for each state-action pair. The table entry for the pair $\langle s, a \rangle$ stores the value for $\hat{Q}(s, a)$—the learner's current hypothesis about the actual but unknown value $Q(s, a)$. The table can be initially filled with random values (though it is easier to understand the algorithm if one assumes initial values of zero). The agent repeatedly observes its current state s, chooses some action a, executes this action, then observes the resulting reward $r = r(s, a)$ and the new state $s' = \delta(s, a)$. It then updates the table entry for $\hat{Q}(s, a)$ following each such transition, according to the rule:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a') \tag{13.7}$$

Note this training rule uses the agent's current $\hat{Q}$ values for the new state s' to refine its estimate of $\hat{Q}(s, a)$ for the previous state s. This training rule is motivated by Equation (13.6), although the training rule concerns the agent's approximation $\hat{Q}$, whereas Equation (13.6) applies to the actual Q function. Note although Equation (13.6) describes Q in terms of the functions $\delta(s, a)$ and $r(s, a)$, the agent does not need to know these general functions to apply the training rule of Equation (13.7). Instead it executes the action in its environment and then observes the resulting new state s' and reward r. Thus, it can be viewed as sampling these functions at the current values of s and a.

The above Q learning algorithm for deterministic Markov decision processes is described more precisely in Table 13.1. Using this algorithm the agent's estimate $\hat{Q}$ converges in the limit to the actual Q function, provided the system can be modeled as a deterministic Markov decision process, the reward function r is

---

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s, a)$ to zero.
Observe the current state s
Do forever:

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s, a)$ as follows:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

- $s \leftarrow s'$

TABLE 13.1
Q learning algorithm, assuming deterministic rewards and actions. The discount factor γ may be any constant such that $0 \leq \gamma < 1$.

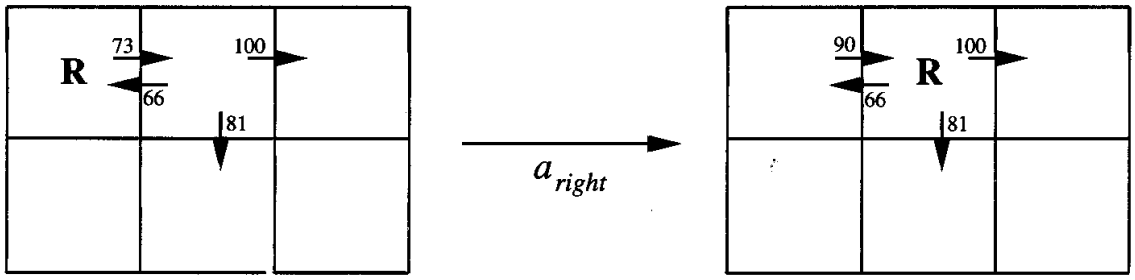
bounded, and actions are chosen so that every state-action pair is visited infinitely often.

13.3.3 An Illustrative Example

To illustrate the operation of the Q learning algorithm, consider a single action taken by an agent, and the corresponding refinement to $\hat{Q}$ shown in Figure 13.3. In this example, the agent moves one cell to the right in its grid world and receives an immediate reward of zero for this transition. It then applies the training rule of Equation (13.7) to refine its estimate $\hat{Q}$ for the state-action transition it just executed. According to the training rule, the new $\hat{Q}$ estimate for this transition is the sum of the received reward (zero) and the highest $\hat{Q}$ value associated with the resulting state (100), discounted by γ (.9).

Each time the agent moves forward from an old state to a new one, Q learning propagates $\hat{Q}$ estimates *backward* from the new state to the old. At the same time, the immediate reward received by the agent for the transition is used to augment these propagated values of $\hat{Q}$.

Consider applying this algorithm to the grid world and reward function shown in Figure 13.2, for which the reward is zero everywhere, except when entering the goal state. Since this world contains an absorbing goal state, we will assume that training consists of a series of *episodes*. During each episode, the agent begins at some randomly chosen state and is allowed to execute actions until it reaches the absorbing goal state. When it does, the episode ends and



$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$
$$\leftarrow 0 + 0.9 \max\{66, 81, 100\}$$
$$\leftarrow 90$$

FIGURE 13.3
The update to $\hat{Q}$ after executing a single action. The diagram on the left shows the initial state s_1 of the robot (**R**) and several relevant $\hat{Q}$ values in its initial hypothesis. For example, the value $\hat{Q}(s_1, a_{right}) = 72.9$, where a_{right} refers to the action that moves **R** to its right. When the robot executes the action a_{right}, it receives immediate reward $r = 0$ and transitions to state s_2. It then updates its estimate $\hat{Q}(s_1, a_{right})$ based on its $\hat{Q}$ estimates for the new state s_2. Here $\gamma = 0.9$.

the agent is transported to a new, randomly chosen, initial state for the next episode.

How will the values of $\hat{Q}$ evolve as the Q learning algorithm is applied in this case? With all the $\hat{Q}$ values initialized to zero, the agent will make no changes to any $\hat{Q}$ table entry until it happens to reach the goal state and receive a nonzero reward. This will result in refining the $\hat{Q}$ value for the single transition leading into the goal state. On the next episode, if the agent passes through this state adjacent to the goal state, its nonzero $\hat{Q}$ value will allow refining the value for some transition two steps from the goal, and so on. Given a sufficient number of training episodes, the information will propagate from the transitions with nonzero reward back through the entire state-action space available to the agent, resulting eventually in a $\hat{Q}$ table containing the Q values shown in Figure 13.2.

In the next section we prove that under certain assumptions the Q learning algorithm of Table 13.1 will converge to the correct Q function. First consider two general properties of this Q learning algorithm that hold for any deterministic MDP in which the rewards are non-negative, assuming we initialize all $\hat{Q}$ values to zero. The first property is that under these conditions the $\hat{Q}$ values never decrease during training. More formally, let $\hat{Q}_n(s, a)$ denote the learned $\hat{Q}(s, a)$ value after the nth iteration of the training procedure (i.e., after the nth state-action transition taken by the agent). Then

$$(\forall s, a, n) \quad \hat{Q}_{n+1}(s, a) \geq \hat{Q}_n(s, a)$$

A second general property that holds under these same conditions is that throughout the training process every $\hat{Q}$ value will remain in the interval between zero and its true Q value.

$$(\forall s, a, n) \quad 0 \leq \hat{Q}_n(s, a) \leq Q(s, a)$$

13.3.4 Convergence

Will the algorithm of Table 13.1 converge toward a $\hat{Q}$ equal to the true Q function? The answer is yes, under certain conditions. First, we must assume the system is a deterministic MDP. Second, we must assume the immediate reward values are bounded; that is, there exists some positive constant c such that for all states s and actions a, $|r(s, a)| < c$. Third, we assume the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often. By this third condition we mean that if action a is a legal action from state s, then over time the agent must execute action a from state s repeatedly and with nonzero frequency as the length of its action sequence approaches infinity. Note these conditions are in some ways quite general and in others fairly restrictive. They describe a more general setting than illustrated by the example in the previous section, because they allow for environments with arbitrary positive or negative rewards, and for environments where any number of state-action transitions may produce nonzero rewards. The conditions are also restrictive in that they require the agent to visit every distinct state-action transition infinitely often. This is a very strong assumption in large (or continuous!) domains. We will discuss stronger

convergence results later. However, the result described in this section provides the basic intuition for understanding why Q learning works.

The key idea underlying the proof of convergence is that the table entry $\hat{Q}(s, a)$ with the largest error must have its error reduced by a factor of γ whenever it is updated. The reason is that its new value depends only in part on error-prone $\hat{Q}$ estimates, with the remainder depending on the error-free observed immediate reward r.

Theorem 13.1. Convergence of Q learning for deterministic Markov decision processes. Consider a Q learning agent in a deterministic MDP with bounded rewards $(\forall s, a)|r(s, a)| \leq c$. The Q learning agent uses the training rule of Equation (13.7), initializes its table $\hat{Q}(s, a)$ to arbitrary finite values, and uses a discount factor γ such that $0 \leq \gamma < 1$. Let $\hat{Q}_n(s, a)$ denote the agent's hypothesis $\hat{Q}(s, a)$ following the nth update. If each state-action pair is visited infinitely often, then $\hat{Q}_n(s, a)$ converges to $Q(s, a)$ as $n \to \infty$, for all s, a.

Proof. Since each state-action transition occurs infinitely often, consider consecutive intervals during which each state-action transition occurs at least once. The proof consists of showing that the maximum error over all entries in the $\hat{Q}$ table is reduced by at least a factor of γ during each such interval. $\hat{Q}_n$ is the agent's table of estimated Q values after n updates. Let Δ_n be the maximum error in $\hat{Q}_n$; that is

$$\Delta_n \equiv \max_{s,a} |\hat{Q}_n(s, a) - Q(s, a)|$$

Below we use s' to denote $\delta(s, a)$. Now for any table entry $\hat{Q}_n(s, a)$ that is updated on iteration $n + 1$, the magnitude of the error in the revised estimate $\hat{Q}_{n+1}(s, a)$ is

$$\begin{aligned}
|\hat{Q}_{n+1}(s, a) - Q(s, a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s', a')) - (r + \gamma \max_{a'} Q(s', a'))| \\
&= \gamma |\max_{a'} \hat{Q}_n(s', a') - \max_{a'} Q(s', a')| \\
&\leq \gamma \max_{a'} |\hat{Q}_n(s', a') - Q(s', a')| \\
&\leq \gamma \max_{s'', a'} |\hat{Q}_n(s'', a') - Q(s'', a')| \\
|\hat{Q}_{n+1}(s, a) - Q(s, a)| &\leq \gamma \Delta_n
\end{aligned}$$

The third line above follows from the second line because for any two functions f_1 and f_2 the following inequality holds

$$|\max_a f_1(a) - \max_a f_2(a)| \leq \max_a |f_1(a) - f_2(a)|$$

In going from the third line to the fourth line above, note we introduce a new variable s'' over which the maximization is performed. This is legitimate because the maximum value will be at least as great when we allow this additional variable to vary. Note that by introducing this variable we obtain an expression that matches the definition of Δ_n.

Thus, the updated $\hat{Q}_{n+1}(s, a)$ for any s, a is at most γ times the maximum error in the $\hat{Q}_n$ table, Δ_n. The largest error in the initial table, Δ_0, is bounded because values of $\hat{Q}_0(s, a)$ and $Q(s, a)$ are bounded for all s, a. Now after the first interval

during which each s, a is visited, the largest error in the table will be at most $\gamma \Delta_0$. After k such intervals, the error will be at most $\gamma^k \Delta_0$. Since each state is visited infinitely often, the number of such intervals is infinite, and $\Delta_n \rightarrow 0$ as $n \rightarrow \infty$. This proves the theorem. ■

13.3.5 Experimentation Strategies

Notice the algorithm of Table 13.1 does not specify how actions are chosen by the agent. One obvious strategy would be for the agent in state s to select the action a that maximizes $\hat{Q}(s, a)$, thereby exploiting its current approximation $\hat{Q}$. However, with this strategy the agent runs the risk that it will overcommit to actions that are found during early training to have high $\hat{Q}$ values, while failing to explore other actions that have even higher values. In fact, the convergence theorem above requires that each state-action transition occur infinitely often. This will clearly not occur if the agent always selects actions that maximize its current $\hat{Q}(s, a)$. For this reason, it is common in Q learning to use a probabilistic approach to selecting actions. Actions with higher $\hat{Q}$ values are assigned higher probabilities, but every action is assigned a nonzero probability. One way to assign such probabilities is

$$P(a_i|s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}}$$

where $P(a_i|s)$ is the probability of selecting action a_i, given that the agent is in state s, and where $k > 0$ is a constant that determines how strongly the selection favors actions with high $\hat{Q}$ values. Larger values of k will assign higher probabilities to actions with above average $\hat{Q}$, causing the agent to *exploit* what it has learned and seek actions it believes will maximize its reward. In contrast, small values of k will allow higher probabilities for other actions, leading the agent to *explore* actions that do not currently have high $\hat{Q}$ values. In some cases, k is varied with the number of iterations so that the agent favors exploration during early stages of learning, then gradually shifts toward a strategy of exploitation.

13.3.6 Updating Sequence

One important implication of the above convergence theorem is that Q learning need not train on optimal action sequences in order to converge to the optimal policy. In fact, it can learn the Q function (and hence the optimal policy) while training from actions chosen completely at random at each step, as long as the resulting training sequence visits every state-action transition infinitely often. This fact suggests changing the sequence of training example transitions in order to improve training efficiency without endangering final convergence. To illustrate, consider again learning in an MDP with a single absorbing goal state, such as the one in Figure 13.1. Assume as before that we train the agent with a sequence of episodes. For each episode, the agent is placed in a random initial state and is allowed to perform actions and to update its $\hat{Q}$ table until it reaches the absorbing goal state. A new training episode is then begun by removing the agent from the

goal state and placing it at a new random initial state. As noted earlier, if we begin with all $\hat{Q}$ values initialized to zero, then after the first full episode only one entry in the agent's $\hat{Q}$ table will have been changed: the entry corresponding to the final transition into the goal state. Note that if the agent happens to follow the same sequence of actions from the same random initial state in its second full episode, then a second table entry would be made nonzero, and so on. If we run repeated identical episodes in this fashion, the frontier of nonzero $\hat{Q}$ values will creep backward from the goal state at the rate of one new state-action transition per episode. Now consider training on these same state-action transitions, but in reverse chronological order for each episode. That is, we apply the same update rule from Equation (13.7) for each transition considered, but perform these updates in reverse order. In this case, after the first full episode the agent will have updated its $\hat{Q}$ estimate for every transition along the path it took to the goal. This training process will clearly converge in fewer iterations, although it requires that the agent use more memory to store the entire episode before beginning the training for that episode.

A second strategy for improving the rate of convergence is to store past state-action transitions, along with the immediate reward that was received, and retrain on them periodically. Although at first it might seem a waste of effort to retrain on the same transition, recall that the updated $\hat{Q}(s, a)$ value is determined by the values $\hat{Q}(s', a)$ of the successor state $s' = \delta(s, a)$. Therefore, if subsequent training changes one of the $\hat{Q}(s', a)$ values, then retraining on the transition $\langle s, a \rangle$ may result in an altered value for $\hat{Q}(s, a)$. In general, the degree to which we wish to replay old transitions versus obtain new ones from the environment depends on the relative costs of these two operations in the specific problem domain. For example, in a robot domain with navigation actions that might take several seconds to perform, the delay in collecting a new state-action transition from the external world might be several orders of magnitude more costly than internally replaying a previously observed transition. This difference can be very significant given that Q learning can often require thousands of training iterations to converge.

Note throughout the above discussion we have kept our assumption that the agent does not know the state-transition function $\delta(s, a)$ used by the environment to create the successor state $s' = \delta(s, a)$, or the function $r(s, a)$ used to generate rewards. If it does know these two functions, then many more efficient methods are possible. For example, if performing external actions is expensive the agent may simply ignore the environment and instead simulate it internally, efficiently generating simulated actions and assigning the appropriate simulated rewards. Sutton (1991) describes the DYNA architecture that performs a number of simulated actions after each step executed in the external world. Moore and Atkeson (1993) describe an approach called *prioritized sweeping* that selects promising states to update next, focusing on predecessor states when the current state is found to have a large update. Peng and Williams (1994) describe a similar approach. A large number of efficient algorithms from the field of dynamic programming can be applied when the functions δ and r are known. Kaelbling et al. (1996) survey a number of these.

13.4 NONDETERMINISTIC REWARDS AND ACTIONS

Above we considered Q learning in deterministic environments. Here we consider the nondeterministic case, in which the reward function $r(s, a)$ and action transition function $\delta(s, a)$ may have probabilistic outcomes. For example, in Tesauro's (1995) backgammon playing program, action outcomes are inherently probabilistic because each move involves a roll of the dice. Similarly, in robot problems with noisy sensors and effectors it is often appropriate to model actions and rewards as nondeterministic. In such cases, the functions $\delta(s, a)$ and $r(s, a)$ can be viewed as first producing a probability distribution over outcomes based on s and a, and then drawing an outcome at random according to this distribution. When these probability distributions depend solely on s and a (e.g., they do not depend on previous states or actions), then we call the system a nondeterministic Markov decision process.

In this section we extend the Q learning algorithm for the deterministic case to handle nondeterministic MDPs. To accomplish this, we retrace the line of argument that led to the algorithm for the deterministic case, revising it where needed.

In the nondeterministic case we must first restate the objective of the learner to take into account the fact that outcomes of actions are no longer deterministic. The obvious generalization is to redefine the value V^π of a policy π to be the *expected value* (over these nondeterministic outcomes) of the discounted cumulative reward received by applying this policy

$$V^\pi(s_t) \equiv E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right]$$

where, as before, the sequence of rewards r_{t+i} is generated by following policy π beginning at state s. Note this is a generalization of Equation (13.1), which covered the deterministic case.

As before, we define the optimal policy π^* to be the policy π that maximizes $V^\pi(s)$ for all states s. Next we generalize our earlier definition of Q from Equation (13.4), again by taking its expected value.

$$\begin{aligned} Q(s, a) &\equiv E[r(s, a) + \gamma V^*(\delta(s, a))] \\ &= E[r(s, a)] + \gamma E[V^*(\delta(s, a))] \\ &= E[r(s, a)] + \gamma \sum_{s'} P(s'|s, a) V^*(s') \end{aligned} \tag{13.8}$$

where $P(s'|s, a)$ is the probability that taking action a in state s will produce the next state s'. Note we have used $P(s'|s, a)$ here to rewrite the expected value of $V^*(\delta(s, a))$ in terms of the probabilities associated with the possible outcomes of the probabilistic δ.

As before we can re-express Q recursively

$$Q(s, a) = E[r(s, a)] + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a') \tag{13.9}$$

which is the generalization of the earlier Equation (13.6). To summarize, we have simply redefined $Q(s, a)$ in the nondeterministic case to be the expected value of its previously defined quantity for the deterministic case.

Now that we have generalized the definition of Q to accommodate the nondeterministic environment functions r and δ, a new training rule is needed. Our earlier training rule derived for the deterministic case (Equation 13.7) fails to converge in this nondeterministic setting. Consider, for example, a nondeterministic reward function $r(s, a)$ that produces different rewards each time the transition $\langle s, a\rangle$ is repeated. In this case, the training rule will repeatedly alter the values of $\hat{Q}(s, a)$, even if we initialize the $\hat{Q}$ table values to the correct Q function. In brief, this training rule does not converge. This difficulty can be overcome by modifying the training rule so that it takes a decaying weighted average of the current $\hat{Q}$ value and the revised estimate. Writing $\hat{Q}_n$ to denote the agent's estimate on the nth iteration of the algorithm, the following revised training rule is sufficient to assure convergence of $\hat{Q}$ to Q:

$$\hat{Q}_n(s, a) \leftarrow (1 - \alpha_n)\hat{Q}_{n-1}(s, a) + \alpha_n[r + \gamma \max_{a'} \hat{Q}_{n-1}(s', a')] \tag{13.10}$$

where

$$\alpha_n = \frac{1}{1 + visits_n(s, a)} \tag{13.11}$$

where s and a here are the state and action updated during the nth iteration, and where $visits_n(s, a)$ is the total number of times this state-action pair has been visited up to and including the nth iteration.

The key idea in this revised rule is that revisions to $\hat{Q}$ are made more gradually than in the deterministic case. Notice if we were to set α_n to 1 in Equation (13.10) we would have exactly the training rule for the deterministic case. With smaller values of α, this term is now averaged in with the current $\hat{Q}(s, a)$ to produce the new updated value. Notice that the value of α_n in Equation (13.11) decreases as n increases, so that updates become smaller as training progresses. By reducing α at an appropriate rate during training, we can achieve convergence to the correct Q function. The choice of α_n given above is one of many that satisfy the conditions for convergence, according to the following theorem due to Watkins and Dayan (1992).

Theorem 13.2. Convergence of Q learning for nondeterministic Markov decision processes. Consider a Q learning agent in a nondeterministic MDP with bounded rewards $(\forall s, a)|r(s, a)| \leq c$. The Q learning agent uses the training rule of Equation (13.10), initializes its table $\hat{Q}(s, a)$ to arbitrary finite values, and uses a discount factor γ such that $0 \leq \gamma < 1$. Let $n(i, s, a)$ be the iteration corresponding to the ith time that action a is applied to state s. If each state-action pair is visited infinitely often, $0 \leq \alpha_n < 1$, and

$$\sum_{i=1}^{\infty} \alpha_{n(i,s,a)} = \infty, \quad \sum_{i=1}^{\infty} [\alpha_{n(i,s,a)}]^2 < \infty$$

then for all s and a, $\hat{Q}_n(s, a) \rightarrow Q(s, a)$ as $n \rightarrow \infty$, with probability 1.

While Q learning and related reinforcement learning algorithms can be proven to converge under certain conditions, in practice systems that use Q learning often require many thousands of training iterations to converge. For example, Tesauro's TD-GAMMON discussed earlier trained for 1.5 million backgammon games, each of which contained tens of state-action transitions.

13.5 TEMPORAL DIFFERENCE LEARNING

The Q learning algorithm learns by iteratively reducing the discrepancy between Q value estimates for adjacent states. In this sense, Q learning is a special case of a general class of *temporal difference* algorithms that learn by reducing discrepancies between estimates made by the agent at different times. Whereas the training rule of Equation (13.10) reduces the difference between the estimated $\hat{Q}$ values of a state and its immediate successor, we could just as well design an algorithm that reduces discrepancies between this state and more distant descendants or ancestors.

To explore this issue further, recall that our Q learning training rule calculates a training value for $\hat{Q}(s_t, a_t)$ in terms of the values for $\hat{Q}(s_{t+1}, a_{t+1})$ where s_{t+1} is the result of applying action a_t to the state s_t. Let $Q^{(1)}(s_t, a_t)$ denote the training value calculated by this one-step lookahead

$$Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$$

One alternative way to compute a training value for $Q(s_t, a_t)$ is to base it on the observed rewards for two steps

$$Q^{(2)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)$$

or, in general, for n steps

$$Q^{(n)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \cdots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n \max_a \hat{Q}(s_{t+n}, a)$$

Sutton (1988) introduces a general method for blending these alternative training estimates, called TD(λ). The idea is to use a constant $0 \leq \lambda \leq 1$ to combine the estimates obtained from various lookahead distances in the following fashion

$$Q^{\lambda}(s_t, a_t) \equiv (1 - \lambda) \left[Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t) + \cdots \right]$$

An equivalent recursive definition for Q^{λ} is

$$Q^{\lambda}(s_t, a_t) = r_t + \gamma [\ (1 - \lambda) \max_a \hat{Q}(s_t, a_t) + \lambda\ Q^{\lambda}(s_{t+1}, a_{t+1})]$$

Note if we choose $\lambda = 0$ we have our original training estimate $Q^{(1)}$, which considers only one-step discrepancies in the $\hat{Q}$ estimates. As λ is increased, the algorithm places increasing emphasis on discrepancies based on more distant lookaheads. At the extreme value $\lambda = 1$, only the observed r_{t+i} values are considered,

with no contribution from the current $\hat{Q}$ estimate. Note when $\hat{Q} = Q$, the training values given by Q^λ will be identical for all values of λ such that $0 \leq \lambda \leq 1$.

The motivation for the TD(λ) method is that in some settings training will be more efficient if more distant lookaheads are considered. For example, when the agent follows an optimal policy for choosing actions, then Q^λ with $\lambda = 1$ will provide a perfect estimate for the true Q value, regardless of any inaccuracies in $\hat{Q}$. On the other hand, if action sequences are chosen suboptimally, then the r_{t+i} observed far into the future can be misleading.

Peng and Williams (1994) provide a further discussion and experimental results showing the superior performance of Q^λ in one problem domain. Dayan (1992) shows that under certain assumptions a similar TD(λ) approach applied to learning the V^* function converges correctly for any λ such that $0 \leq \lambda \leq 1$. Tesauro (1995) uses a TD(λ) approach in his TD-GAMMON program for playing backgammon.

13.6 GENERALIZING FROM EXAMPLES

Perhaps the most constraining assumption in our treatment of Q learning up to this point is that the target function is represented as an explicit lookup table, with a distinct table entry for every distinct input value (i.e., state-action pair). Thus, the algorithms we discussed perform a kind of rote learning and make no attempt to estimate the Q value for unseen state-action pairs by generalizing from those that have been seen. This rote learning assumption is reflected in the convergence proof, which proves convergence only if every possible state-action pair is visited (infinitely often!). This is clearly an unrealistic assumption in large or infinite spaces, or when the cost of executing actions is high. As a result, more practical systems often combine function approximation methods discussed in other chapters with the Q learning training rules described here.

It is easy to incorporate function approximation algorithms such as BACKPROPAGATION into the Q learning algorithm, by substituting a neural network for the lookup table and using each $\hat{Q}(s, a)$ update as a training example. For example, we could encode the state s and action a as network inputs and train the network to output the target values of $\hat{Q}$ given by the training rules of Equations (13.7) and (13.10). An alternative that has sometimes been found to be more successful in practice is to train a separate network for each action, using the state as input and $\hat{Q}$ as output. Another common alternative is to train one network with the state as input, but with one $\hat{Q}$ output for each action. Recall that in Chapter 1, we discussed approximating an evaluation function over checkerboard states using a linear function and the LMS algorithm.

In practice, a number of successful reinforcement learning systems have been developed by incorporating such function approximation algorithms in place of the lookup table. Tesauro's successful TD-GAMMON program for playing backgammon used a neural network and the BACKPROPAGATION algorithm together with a TD(λ) training rule. Zhang and Dietterich (1996) use a similar combination of BACKPROPAGATION and TD(λ) for job-shop scheduling tasks. Crites and Barto (1996) describe

a neural network reinforcement learning approach for an elevator scheduling task. Thrun (1996) reports a neural network based approach to Q learning to learn basic control procedures for a mobile robot with sonar and camera sensors. Mahadevan and Connell (1991) describe a Q learning approach based on clustering states, applied to a simple mobile robot control problem.

Despite the success of these systems, for other tasks reinforcement learning fails to converge once a generalizing function approximator is introduced. Examples of such problematic tasks are given by Boyan and Moore (1995), Baird (1995), and Gordon (1995). Note the convergence theorems discussed earlier in this chapter apply only when $\hat{Q}$ is represented by an explicit table. To see the difficulty, consider using a neural network rather than an explicit table to represent $\hat{Q}$. Note if the learner updates the network to better fit the training Q value for a particular transition $\langle s_i, a_i \rangle$, the altered network weights may also change the $\hat{Q}$ estimates for arbitrary other transitions. Because these weight changes may increase the error in $\hat{Q}$ estimates for these other transitions, the argument proving the original theorem no longer holds. Theoretical analyses of reinforcement learning with generalizing function approximators are given by Gordon (1995) and Tsitsiklis (1994). Baird (1995) proposes gradient-based methods that circumvent this difficulty by directly minimizing the sum of squared discrepancies in estimates between adjacent states (also called Bellman residual errors).

13.7 RELATIONSHIP TO DYNAMIC PROGRAMMING

Reinforcement learning methods such as Q learning are closely related to a long line of research on dynamic programming approaches to solving Markov decision processes. This earlier work has typically assumed that the agent possesses perfect knowledge of the functions $\delta(s, a)$ and $r(s, a)$ that define the agent's environment. Therefore, it has primarily addressed the question of how to compute the optimal policy using the least computational effort, assuming the environment could be perfectly simulated and no direct interaction was required. The novel aspect of Q learning is that it assumes the agent does *not* have knowledge of $\delta(s, a)$ and $r(s, a)$, and that instead of moving about in an internal mental model of the state space, it must move about the real world and observe the consequences. In this latter case our primary concern is usually the number of real-world actions that the agent must perform to converge to an acceptable policy, rather than the number of computational cycles it must expend. The reason is that in many practical domains such as manufacturing problems, the costs in time and in dollars of performing actions in the external world dominate the computational costs. Systems that learn by moving about the real environment and observing the results are typically called *online* systems, whereas those that learn solely by simulating actions within an internal model are called *offline* systems.

The close correspondence between these earlier approaches and the reinforcement learning problems discussed here is apparent by considering Bellman's equation, which forms the foundation for many dynamic programming approaches

to solving MDPs. Bellman's equation is

$$(\forall s \in S) V^*(s) = E[r(s, \pi(s)) + \gamma V^*(\delta(s, \pi(s)))]$$

Note the very close relationship between Bellman's equation and our earlier definition of an optimal policy in Equation (13.2). Bellman (1957) showed that the optimal policy π^* satisfies the above equation and that any policy π satisfying this equation is an optimal policy. Early work on dynamic programming includes the Bellman-Ford shortest path algorithm (Bellman 1958; Ford and Fulkerson 1962), which learns paths through a graph by repeatedly updating the estimated distance to the goal for each graph node, based on the distances for its neighbors. In this algorithm the assumption that graph edges and the goal node are known is equivalent to our assumption that $\delta(s, a)$ and $r(s, a)$ are known. Barto et al. (1995) discuss the close relationship between reinforcement learning and dynamic programming.

13.8 SUMMARY AND FURTHER READING

The key points discussed in this chapter include:

- Reinforcement learning addresses the problem of learning control strategies for autonomous agents. It assumes that training information is available in the form of a real-valued reward signal given for each state-action transition. The goal of the agent is to learn an action policy that maximizes the total reward it will receive from any starting state.
- The reinforcement learning algorithms addressed in this chapter fit a problem setting known as a Markov decision process. In Markov decision processes, the outcome of applying any action to any state depends only on this action and state (and not on preceding actions or states). Markov decision processes cover a wide range of problems including many robot control, factory automation, and scheduling problems.
- Q learning is one form of reinforcement learning in which the agent learns an evaluation function over states and actions. In particular, the evaluation function $Q(s, a)$ is defined as the maximum expected, discounted, cumulative reward the agent can achieve by applying action a to state s. The Q learning algorithm has the advantage that it can be employed even when the learner has no prior knowledge of how its actions affect its environment.
- Q learning can be proven to converge to the correct Q function under certain assumptions, when the learner's hypothesis $\hat{Q}(s, a)$ is represented by a lookup table with a distinct entry for each $\langle s, a \rangle$ pair. It can be shown to converge in both deterministic and nondeterministic MDPs. In practice, Q learning can require many thousands of training iterations to converge in even modest-sized problems.
- Q learning is a member of a more general class of algorithms, called temporal difference algorithms. In general, temporal difference algorithms learn

by iteratively reducing the discrepancies between the estimates produced by the agent at different times.

- Reinforcement learning is closely related to dynamic programming approaches to Markov decision processes. The key difference is that historically these dynamic programming approaches have assumed that the agent possesses knowledge of the state transition function $\delta(s, a)$ and reward function $r(s, a)$. In contrast, reinforcement learning algorithms such as Q learning typically assume the learner lacks such knowledge.

The common theme that underlies much of the work on reinforcement learning is to iteratively reduce the discrepancy between evaluations of successive states. Some of the earliest work on such methods is due to Samuel (1959). His checkers learning program attempted to learn an evaluation function for checkers by using evaluations of later states to generate training values for earlier states. Around the same time, the Bellman-Ford, single-destination, shortest-path algorithm was developed (Bellman 1958; Ford and Fulkerson 1962), which propagated distance-to-goal values from nodes to their neighbors. Research on optimal control led to the solution of Markov decision processes using similar methods (Bellman 1961; Blackwell 1965). Holland's (1986) bucket brigade method for learning classifier systems used a similar method for propagating credit in the face of delayed rewards. Barto et al. (1983) discussed an approach to temporal credit assignment that led to Sutton's paper (1988) defining the TD(λ) method and proving its convergence for $\lambda = 0$. Dayan (1992) extended this result to arbitrary values of λ. Watkins (1989) introduced Q learning to acquire optimal policies when the reward and action transition functions are unknown. Convergence proofs are known for several variations on these methods. In addition to the convergence proofs presented in this chapter see, for example, (Baird 1995; Bertsekas 1987; Tsitsiklis 1994, Singh and Sutton 1996).

Reinforcement learning remains an active research area. McCallum (1995) and Littman (1996), for example, discuss the extension of reinforcement learning to settings with hidden state variables that violate the Markov assumption. Much current research seeks to scale up these methods to larger, more practical problems. For example, Maclin and Shavlik (1996) describe an approach in which a reinforcement learning agent can accept imperfect advice from a trainer, based on an extension to the KBANN algorithm (Chapter 12). Lin (1992) examines the role of teaching by providing suggested action sequences. Methods for scaling up by employing a hierarchy of actions are suggested by Singh (1993) and Lin (1993). Dietterich and Flann (1995) explore the integration of explanation-based methods with reinforcement learning, and Mitchell and Thrun (1993) describe the application of the EBNN algorithm (Chapter 12) to Q learning. Ring (1994) explores continual learning by the agent over multiple tasks.

Recent surveys of reinforcement learning are given by Kaelbling et al. (1996); Barto (1992); Barto et al. (1995); Dean et al. (1993).