So far, we have assumed that the utility functions and Q-functions learned by the agents are represented in tabular form with one output value for each input tuple. Such an approach works reasonably well for small state spaces, but the time to convergence and (for ADP) the time per iteration increase rapidly as the space gets larger. With carefully controlled, approximate ADP methods, it might be possible to handle 10,000 states or more. This suffices for two-dimensional maze-like environments, but more realistic worlds are out of the question. Chess and backgammon are tiny subsets of the real world, yet their state spaces contain on the order of 10^{50} to 10^{120} states. It would be absurd to suppose that one must visit all these states in order to learn how to play the game!

FUNCTION APPROXIMATION

One way to handle such problems is to use **function approximation**, which simply means using any sort of representation for the function other than a table. The representation is viewed as approximate because it might not be the case that the *true* utility function or Q-function can be represented in the chosen form. For example, in Chapter 6 we described an **evaluation function** for chess that is represented as a weighted linear function of a set of **features** (or **basis functions**) $f_1, \ldots, f_n$:

BASIS FUNCTIONS

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

A reinforcement learning algorithm can learn values for the parameters $\theta = \theta_1, \ldots, \theta_n$ such that the evaluation function $\hat{U}_\theta$ approximates the true utility function. Instead of, say, 10^{120} values in a table, this function approximator is characterized by, say, $n = 20$ parameters—an *enormous* compression. Although no one knows the true utility function for chess, no one believes that it can be represented exactly in 20 numbers. If the approximation is good enough, however, the agent might still play excellent chess.[4]

Function approximation makes it practical to represent utility functions for very large state spaces, but that is not its principal benefit. *The compression achieved by a function approximator allows the learning agent to generalize from states it has visited to states it has not visited.* That is, the most important aspect of function approximation is not that it

[4] We do know that the exact utility function can be represented in a page or two of Lisp, Java, or C++. That is, it can be represented by a program that solves the game exactly every time it is called. We are interested only in function approximators that use a *reasonable* amount of computation. It might in fact be better to learn a very simple function approximator and combine it with a certain amount of look-ahead search. The trade-offs involved are currently not well understood.

requires less space, but that it allows for inductive generalization over input states. To give you some idea of the power of this effect: by examining only one in every 10^{44} of the possible backgammon states, it is possible to learn a utility function that allows a program to play as well as any human (Tesauro, 1992).

On the flip side, of course, there is the problem that there could fail to be any function in the chosen hypothesis space that approximates the true utility function sufficiently well. As in all inductive learning, there is a trade-off between the size of the hypothesis space and the time it takes to learn the function. A larger hypothesis space increases the likelihood that a good approximation can be found, but also means that convergence is likely to be delayed.

Let us begin with the simplest case, which is direct utility estimation. (See Section 21.2.) With function approximation, this is an instance of **supervised learning**. For example, suppose we represent the utilities for the 4×3 world using a simple linear function. The features of the squares are just their x and y coordinates, so we have

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y \,. \tag{21.9}$$

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 20.)

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

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha (u_j(s) - \hat{U}_\theta(s)) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} \,. \tag{21.10}$$

WIDROW–HOFF RULE
DELTA RULE

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 (21.9), we get three simple update rules:

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

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. θ_0, θ_1, and θ_2 are all decreased by 0.4α, which reduces the error for (1,1). Notice that changing the θ_i*s 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.

We expect that the agent will learn faster if it uses a function approximator, provided that the hypothesis space is not too large, but includes some functions that are a reasonably good fit to the true utility function. Exercise 21.7 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). This world is well suited for a linear utility function because the true utility function is smooth and nearly linear. (See Exercise 21.10.) If we put the +1 reward at (5,5), the true utility is more like a pyramid and the function approximator in Equation (21.9) will fail miserably. All is not lost, however! Remember that what matters for linear function approximation is that the function be linear in the *parameters*—the features themselves can be arbitrary nonlinear functions of the state variables. Hence, we can include a term such as $\theta_3\sqrt{(x-x_g)^2+(y-y_g)^2}$ that measures the distance to the goal.

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 (21.3 and 21.8) are

$$\theta_i \leftarrow \theta_i + \alpha\,[R(s) + \gamma\,\hat{U}_\theta(s') - \hat{U}_\theta(s)]\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} \tag{21.11}$$

for utilities and

$$\theta_i \leftarrow \theta_i + \alpha\,[R(s) + \gamma \max_{a'} \hat{Q}_\theta(a', s') - \hat{Q}_\theta(a, s)]\frac{\partial \hat{Q}_\theta(a, s)}{\partial \theta_i} \tag{21.12}$$

for Q-values. These update rules can be shown to converge to the closest possible[5] approximation to the true function when the function approximator is *linear* in the parameters. Unfortunately, all bets are off when *nonlinear* functions—such as neural networks—are used. There are some very simple cases in which the parameters can go off to infinity 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.

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 Chapter 18 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. For example, if the state is defined by n Boolean variables, we will need to learn n Boolean functions to predict all the variables. For a *partially observable* environment, the learning problem is much more difficult. 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 20. Inventing the hidden variables and learning the model structure are still open problems.

We now turn to examples of large-scale applications of reinforcement learning. We will see that, in cases where a utility function (and hence a model) is used, the model is usually taken as given. For example, in learning an evaluation function for backgammon, it is normally assumed that the legal moves and their effects are known in advance.

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

Applications to game-playing

The first significant application of reinforcement learning was also the first significant learning program of any kind—the checker-playing program written by Arthur Samuel (1959, 1967). Samuel first used a weighted linear function for the evaluation of positions, using up to 16 terms at any one time. He applied a version of Equation (21.11) to update the weights. There were some significant differences, however, between his program and current methods. First, he updated the weights using the difference between the current state and the backed-up value generated by full look-ahead in the search tree. This works fine, because it amounts to viewing the state space at a different granularity. A second difference was that the program did *not* use any observed rewards! That is, the values of terminal states were ignored. This means that it is quite possible for Samuel's program not to converge, or to converge on a strategy designed to lose rather than to win. He managed to avoid this fate by insisting that the weight for material advantage should always be positive. Remarkably, this was sufficient to direct the program into areas of weight space corresponding to good checker play.

Gerry Tesauro's TD-Gammon system (1992) forcefully illustrates the potential of reinforcement learning techniques. In earlier work (Tesauro and Sejnowski, 1989), Tesauro tried learning a neural network representation of $Q(a, s)$ directly from examples of moves labeled with relative values by a human expert. This approach proved extremely tedious for the expert. It resulted in a program, called NEUROGAMMON, that was strong by computer standards, but not competitive with human experts. The TD-Gammon project was an attempt to learn from self-play alone. The only reward signal was given at the end of each game. The evaluation function was represented by a fully connected neural network with a single hidden layer containing 40 nodes. Simply by repeated application of Equation (21.11), TD-Gammon learned to play considerably better than Neurogammon, even though the input representation contained just the raw board position with no computed features. This took about 200,000 training games and two weeks of computer time. Although that may seem like a lot of games, it is only a vanishingly small fraction of the state space. When precomputed features were added to the input representation, a network with 80 hidden units was able, after 300,000 training games, to reach a standard of play comparable to that of the top three human players worldwide. Kit Woolsey, a top player and analyst, said that "There is no question in my mind that its positional judgment is far better than mine."

Application to robot control

CART–POLE
INVERTED PENDULUM

The setup for the famous **cart–pole** balancing problem, also known as the **inverted pendulum**, is shown in Figure 21.9. The problem is to control the position x of the cart so that the pole stays roughly upright ($\theta \approx \pi/2$), while staying within the limits of the cart track as shown. Over two thousand papers in reinforcement learning and control theory have been published on this seemingly simple problem. The cart–pole problem differs from the problems described earlier in that the state variables x, θ, $\dot{x}$, and $\dot{\theta}$ are continuous. The actions are usually discrete: jerk left or jerk right, the so-called **bang-bang control** regime.

BANG-BANG CONTROL

The earliest work on learning for this problem was carried out by Michie and Chambers (1968). Their BOXES algorithm was able to balance the pole for over an hour after only

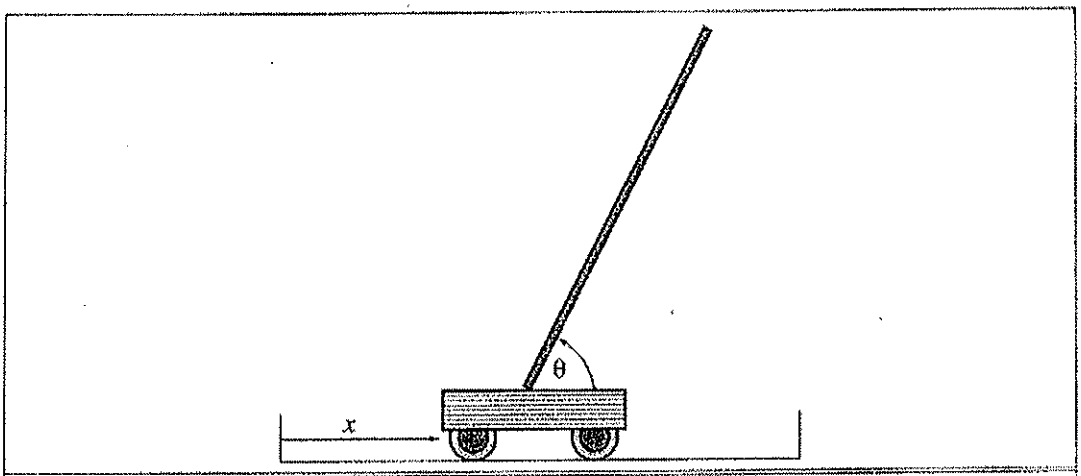

Figure 21.9 Setup for the problem of balancing a long pole on top of a moving cart. The cart can be jerked left or right by a controller that observes x, θ, $\dot{x}$, and $\dot{\theta}$.

about 30 trials. Moreover, unlike many subsequent systems, BOXES was implemented with a real cart and pole, not a simulation. The algorithm first discretized the four-dimensional state space into boxes—hence the name. It then ran trials until the pole fell over or the cart hit the end of the track. Negative reinforcement was associated with the final action in the final box and then propagated back through the sequence. It was found that the discretization caused some problems when the apparatus was initialized in a position different from those used in training, suggesting that generalization was not perfect. 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. Nowadays, balancing a *triple* inverted pendulum is a common exercise—a feat far beyond the capabilities of most humans.