GENERALIZATION IN REINFORCEMENT

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.

ADP methods, it might be possible to handle 10,000 states or more.

Backgammon and chess are tiny subsets of the real world, yet their state spaces contain on the order of 10^{20} and 10^{40} states, respectively.

one must visit all these states many times in order to learn how to play the game!

One way to handle such problems is to use **function approximation**, which simply means using any sort of representation for the Q-function other than a lookup table

For example, we described an **evaluation function** for chess that is represented as a weighted linear function of a set of **features** (or **basis functions**) $f_1, \ldots, f_n$

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

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.

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

Function approximation makes it practical to represent utility functions for very large state spaces

The compression achieved by a function approximator allows the learning agent to generalize from states it has visited to states it has not visited

There is a tradeoff 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

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

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

Thus, if $(\theta 0, \theta 1, \theta 2)=(0.5, 0.2, 0.1)$,

then $\hat{U}\ \theta(1, 1)=0.8$

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.

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

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha\,(u_j(s) - \hat{U}_\theta(s)) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} \,. \tag{21.11}$$

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 (21.10), we get three simple update rules:

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

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 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.

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 new versions of the TD and Q-learning equations are given by

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

for utilities and

$$\theta_i \leftarrow \theta_i + \alpha\,[R(s) + \gamma\,\max_{a'} \hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a)]\frac{\partial \hat{Q}_\theta(s, a)}{\partial \theta_i} \tag{21.13}$$

for Q-values.

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.

Some practical examples are

21.6.1 Applications to game playing

Gerry Tesauro's backgammon program TD-GAMMON (1992) forcefully illustrates the potential of reinforcement learning techniques.

21.6.2 Application to robot control

The setup for the famous **cart–pole** balancing problem, also known as the **inverted pendulum**, is shown in Figure 21.9.

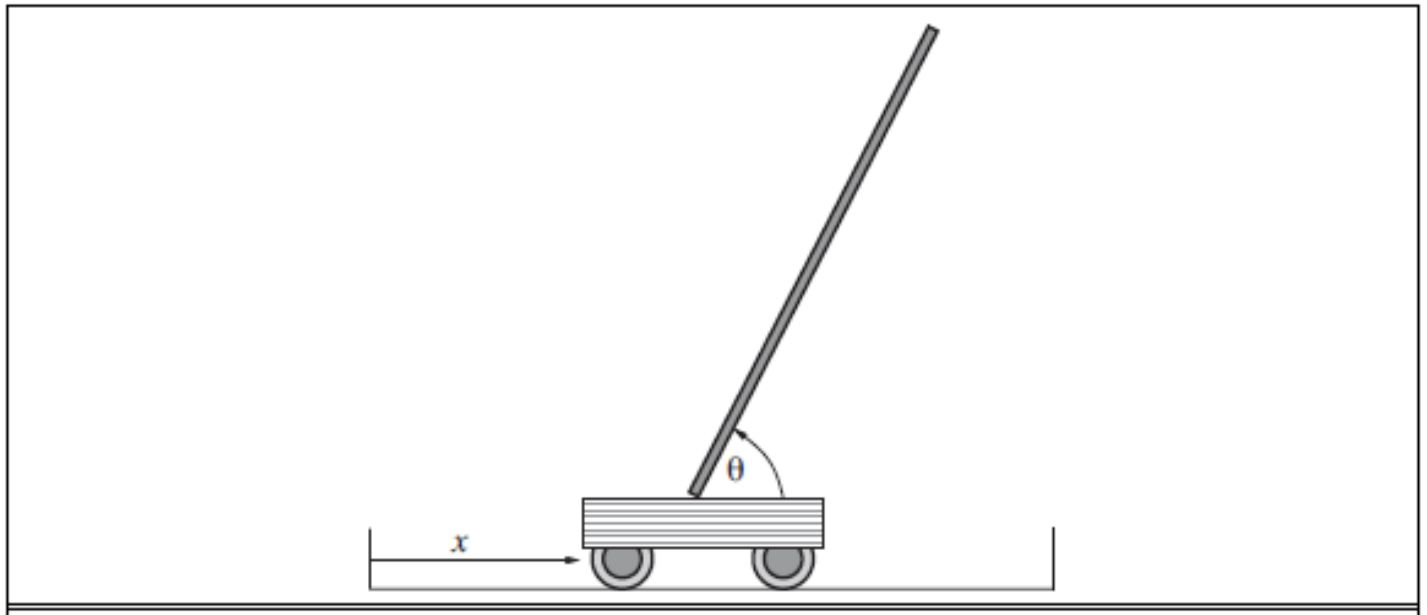

Figure 21.9 Setup for the problem of balancing a long pole on top of a moving cart. The cart can be jerked left or right by a controller that observes x, θ, $\dot{x}$, and $\dot{\theta}$.

Still more impressive is the application of reinforcement learning to helicopter flight (Figure 21.10). This work has generally used policy search (Bagnell and Schneider, 2001) as well as the PEGASUS algorithm with simulation based on a learned transition model

Figure 21.10 Superimposed time-lapse images of an 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. 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.)