OPTIMIZING WHILE LEARNING

We are finally ready to tackle the problem of searching for good policies while simultaneously trying to produce good value function approximations. Our discussion is restricted to problems where the policy is based on the value function approximation, since chapter 7 has already addressed the strategy of direct policy search. The guiding principle in this chapter is that we can find good policies if we can find good value function approximations.

The statistical tools presented in chapters 8 and 9 focused on finding the best statistical fit within a particular approximation architecture. We actually did not address whether we had chosen a good architecture. This is particularly true of our linear models, where we used the tools of stochastic optimization and linear regression to ensure that we obtain the best fit given a model, without regard to whether it was a good model.

In this chapter, we return to the problem of trying to find the best policy, where we assume throughout that our policies are of the form

$$A^\pi(S) = \arg \max_{a \in A} \left( C(S, a) + \gamma \mathbb{E} V(S, a, W) \right),$$

if we have an infinite horizon problem with discrete actions. Or, we may consider finite-horizon problems with vector-valued decisions $x_t$, where a policy would look like

$$X^\pi_t(S_t) = \arg \max_{x_t \in X_t} \left( C(S_t, x_t) + \gamma \mathbb{E} V_t(S_t, x_t, W_{t+1}) \right).$$
The point is that the policy depends on some sort of value function approximation. When we write our generic optimization problem

$$\max_{\pi} \mathbb{E} \sum_{t=0}^{T} \gamma^t C(S_t, A_t^\pi(S_t)),$$

the maximization over policies can mean choosing an architecture for $V_t(S_t)$, and choosing the parameters that control the architecture. For example, we might be choosing between a myopic policy (see equation (6.4) in chapter 6), or perhaps a simple linear architecture with one basis function

$$V(S) = \theta_0 + \theta_1 S,$$

or perhaps a linear architecture with two basis functions,

$$V(S) = \theta_0 + \theta_1 S + \theta_2 S^2.$$ (10.2)

We might even use a nonlinear architecture such as

$$V(S) = \frac{e^{\theta_0 + \theta_1 S}}{1 + e^{\theta_0 + \theta_1 S}}.$$

Optimizing over policies may consist of choosing a value function approximation such as (10.1) or (10.2), but then we still have to choose the best parameter vector within each class.

We begin our presentation with an overview of the basic algorithmic strategies that we cover in this chapter, all of which are based on using value function approximations that are intended to approximate the value of being in a state. The remainder of the chapter is organized around covering the following strategies:

**Approximate value iteration** - These are policies that iteratively update the value function approximation, and then immediately update the policy. We strive to find a value function approximation that estimates the value of being in each state while following a (near) optimal policy, but only in the limit. We intermingle the treatment of finite and infinite horizon problems. Variations include

- **Lookup table representations** - Here we introduce three major strategies that reflect the use of the pre-decision state, state-action pairs, and the post-decision state:
  - **AVI for pre-decision state** - Approximate value iteration using the classical pre-decision state variable.
  - **Q-learning** - Estimating the value of state-action pairs.
  - **AVI for the post-decision state** - Approximate value iteration where value function approximations are approximated around the post-decision state.

**Parametric architectures** - We discuss the issues that arise when trying to use linear models in the context of approximate value iteration.

**Approximate policy iteration** - These are policies that attempt to explicitly approximate the value of a policy to some level of accuracy within an inner loop, within which the policy is held fixed.
API using lookup tables - We use this setting to present the basic idea.

API using linear models - This is perhaps one of the most important areas of research in approximate dynamic programming.

API using nonparametric models - This is a relatively young area of research, and we summarize some recent results.

The linear programming method - The linear programming method, first introduced in chapter 3, can be adapted to exploit value function approximations.

10.1 OVERVIEW OF ALGORITHMIC STRATEGIES

The algorithmic strategies that we examine in this chapter are based on the principles of value iteration and policy iteration, first introduced in chapter 3. We continue to adapt our algorithms to finite and infinite horizons. Basic value iteration for finite horizon problems work by solving

\[ V_t(S_t) = \max_{a_t} \left\{ C(S_t, a_t) + \gamma E\{V_{t+1}(S_{t+1})|S_t\} \right\}. \]  (10.3)

Equation (10.3) works by stepping backward in time, where \( V_t(S_t) \) is computed for each (presumably discrete) state \( S_t \). This is classical “backward” dynamic programming which suffers from the well known curse of dimensionality, because we typically are unable to “loop over all the states.”

Approximate dynamic programming approaches finite horizon problems by solving problems of the form

\[ \hat{v}_t^n = \max_{a_t} \left\{ C(S_t^n, a_t) + \gamma \hat{V}_{t+1}^{n-1} (S_M, a_{t+1}) \right\}. \]  (10.4)

Here, we have formed the value function approximation around the post-decision state. We execute the equations by stepping forward in time. If \( a_{t+1}^n \) is the action that optimizes (10.4), then we compute our next state using \( S_{t+1}^n = S_M(S_t^n, a_{t+1}^n, W_{t+1}) \) where \( W_{t+1} \) is sampled from some distribution. The process runs until we reach the end of our horizon, at which point we return to the beginning of the horizon and repeat the process.

Classical value iteration for infinite horizon problems is centered on the basic iteration

\[ V^n(S) = \max_a \left\{ C(S, a) + \gamma E\{V^{n-1}(S')|S\} \right\}. \]  (10.5)

Again, equation (10.5) has to be executed for each state \( S \). After each iteration, the new estimate \( V^n \) replaces the old estimate \( V^{n-1} \) on the right, after which \( n \) is incremented.

When we use approximate methods, we might observe an estimate of the value of being in a state using

\[ \hat{v}^n = \max_a \left\{ C(S^n, a) + \gamma \hat{V}^{n-1}(S_M, a) \right\}. \]  (10.6)

We then use the observed state-value pair \((S^n, \hat{v}^n)\) to update the value function approximation.

When we use approximate value iteration, \( \hat{v}^n \) (or \( \hat{v}_t^n \)) cannot be viewed as a noisy but unbiased observation of the value of being in a state. These observations are calculated as a function of the value function \( \hat{V}^{n-1}(s) \). While we hope the value function approximation
converges to something, we generally cannot say anything about the function prior to convergence. This means that $\hat{v}^n$ does not have any particular property. We are simply guided by the basic value iteration update in equation (10.5) (or (10.3)), which suggests that if we repeat this step often enough, we may eventually learn the right value function for the right policy. Unfortunately, we have only limited guarantees that this is the case when we depend on approximations.

Approximate value iteration imbeds a policy approximation loop within an outer loop where policies are updated. Assume we fix our policy using

$$A^{n,n}(S) = \arg \max_{a \in A} \left( C(S, a) + \gamma \nabla^{n-1}(S, a) \right),$$  \hspace{1cm} (10.7)

Now perform the loop over $m = 1, \ldots, M$

$$\hat{v}^{n,m} = \max_{a \in A} \left( C(S^{n,m}, a) + \gamma \nabla^{n-1}(a(S^{n,m}), a) \right)$$

where $S^{n,m+1} = S^M(S^{n,m}, x^{n,m}, W^{n+1})$. Note that the value function $\nabla^{n-1}(s)$ remains constant within this inner loop. After executing this loop, we take the series of observations $\hat{v}^{n,1}, \ldots, \hat{v}^{n,M}$ and use them to update $\nabla^{n-1}(s)$ to obtain $\nabla^{n}(s)$. Typically, $\nabla^{n}(s)$ does not depend on $\nabla^{n-1}(s)$, other than to influence the calculation of $\hat{v}^{n,m}$. If $M$ is large enough, $\nabla^{n}(s)$ will represent an accurate approximation of the value of being in state $s$ while following the policy in equation (10.7). In fact, it is specifically because of this ability to approximate a policy that approximate policy iteration is emerging as a powerful algorithmic strategy for approximate dynamic programming. However, the cost of using the inner policy evaluation loop can be significant, and for this reason approximate value iteration and its variants remain popular.

10.2 APPROXIMATE VALUE ITERATION AND Q-LEARNING USING LOOKUP TABLES

Arguably the most natural and elementary approach for approximate dynamic programming uses approximate value iteration. In this section we explore variations of approximate value iteration and Q-learning.

10.2.1 Value iteration using a pre-decision state variable

Classical value iteration (for a finite-horizon problem) estimates the value of being in a specific state $S^n_t$ using

$$\hat{v}^n_t = \max_{a_t} \left( C(S^n_t, a_t) + \gamma E \{ \nabla^{n-1}_t(S_{t+1}|S^n_t) \} \right),$$  \hspace{1cm} (10.8)

where $S_{t+1} = S^M(S^n_t, x^n_t, W^n_{t+1})$, and $S^n_t$ is the state that we are in at time $t$, iteration $n$. We assume that we are following a sample path $\omega^n$, where we compute $W^n_{t+1} = W_{t+1}(\omega^n)$.

After computing $\hat{v}^n_t$, we update the value function using the standard equation

$$\nabla^n_t(S^n_t) = (1 - \alpha_n-1) \nabla^{n-1}_t(S^n_t) + \alpha_n-1 \hat{v}^n_t.$$  \hspace{1cm} (10.9)

If we sample states at random (rather than following the trajectory) and repeat equations (10.8) and (10.9), we will eventually converge to the correct value of being in each state.
Step 0. Initialization:

Step 0a. Initialize $V^0_t$, $t \in T$.
Step 0b. Set $n = 1$.
Step 0c. Initialize $S^0$. 

Step 1. Sample $\omega^n$.

Step 2. Do for $t = 0, 1, \ldots, T$:

Step 2a. Choose $\hat{\Omega}^n \subseteq \Omega$ and solve:
$$
\hat{v}^n_t = \max_{a_t} \left[ C_t(S^n_t, a_t) + \gamma \sum_{\omega \in \hat{\Omega}^n} p^n(\omega) V^{n-1}_{t+1}(S^M(S^n_t, a_t, W_{t+1}(\omega))) \right]
$$
and let $a^n_t$ be the value of $a_t$ that solves the maximization problem.

Step 2b. Compute:
$$
S_{t+1}^n = S^M(S^n_t, a^n_t, W_{t+1}(\omega^n)).
$$

Step 2c. Update the value function:
$$
V^n_t \leftarrow U^V(V^{n-1}_t, S^n_t, \hat{v}^n_t).
$$

Step 3. Increment $n$. If $n \leq N$, go to Step 1.

Step 4. Return the value functions $(V^n_t)_{t=1}^T$.

Figure 10.1 Approximate dynamic programming using a pre-decision state variable.

Note that we are assuming a finite-horizon model, and that we can compute the expectation exactly. When we can compute the expectation exactly, this is very close to classical value iteration, with the only exception that we are not looping over all the states at every iteration.

One reason to use the pre-decision state variable is that for some problems, computing the expectation is easy. For example, $W_{t+1}$ might be a binomial random variable (did a customer arrive, did a component fail) which makes the expectation especially easy. If this is not the case, then we have to approximate the expectation. For example, we might use

$$
\hat{v}^n_t = \max_{a_t} \left[ C(S^n_t, a_t) + \gamma \sum_{\omega \in \hat{\Omega}^n} p^n(\omega) V^{n-1}_{t+1}(S^M(S^n_t, a_t, W_{t+1}(\omega))) \right].
$$

Either way, using a lookup table representation we can update the value of being in state $S^n_t$ using
$$
V^n_t(S^n_t) = (1 - \alpha_{n-1}) V^{n-1}_t(S^n_t) + \alpha_{n-1} \hat{v}^n_t.
$$

Keep in mind that if we can compute an expectation (or if we approximate it using a large sample $\hat{\Omega}$), then the stepsize should be much larger than when we are using a single sample realization (as we did with the post-decision formulation). An outline of the overall algorithm is given by figure 10.1.

At this point a reasonable question to ask is: Does this algorithm work? The answer is possibly, but not in general. Before we get an algorithm that will work (at least in theory), we need to deal with what is known as the exploration-exploitation problem.
10.2.2 On-policy, off-policy and the exploration-exploitation problem

The algorithm in figure 10.1 uses a kind of default logic for determining the next state to visit. Specifically, we solve the optimization problem in equation (10.10) and from this, we not only determine \( \hat{v}_t^n \) which we use to update the value of being in a state, we also determine an action \( a_t^n \). Then, in Step 2b of the algorithm, we use this action to help determine the next state to visit using the transition function

\[
S_{t+1}^n = S^M(S_t^n, a_t^n, W_{t+1}^n).
\]

Using the action \( a_t^n \), which is the action determined by the policy we are trying to optimize, means that we are using a concept known as trajectory following. The policy that determines the action we would like to take is known in the reinforcement learning community as the target policy, but we prefer the alternate term learning policy. When we are optimizing policies, what we are doing is trying to improve the target policy. When we are approximating the value of being in a state while following a fixed policy, we are evaluating the target policy, which is sometimes referred to as the learning policy.

We can encounter serious problems if we use the learning policy to determine the next state to visit. Consider the two-stage dynamic program illustrated in figure 10.2. Assume we start in state 1, and further assume that we initialize the value of being in each of the two states to \( V_0(1) = V_0(2) = 0 \). We see a negative contribution of -$5 to move from state 1 to 2, but a contribution of $0 to stay in state 1. We do not see the contribution of $20 to move from state 2 back to state 1, to it appears to be best to stay in state 1.

We need some way to force the system to visit state 2 so that we discover the contribution of $20. One way to do this is to adopt logic that forces the system to explore by choosing actions at random. For example, we may flip a coin and choose an action with probability \( \epsilon \), or choose the action \( a_t^n \) determined by the learning policy with probability \( 1 - \epsilon \). This policy is known in the literature as epsilon greedy.

The policy that determines which action to use to determine the next state to visit, if it is different than the learning policy, is known as the behavior policy or the sampling policy. The name “behavior policy” arises when we are modeling a real system such as a human playing a game or a factory assembling components. The behavior policy is, literally, the policy that describes how the system behaves. By contrast, if we are simply designing an algorithm, we feel that the term “sampling policy” more accurately describes the actual function being served by this policy. We also note that while it is common to implement a sampling policy through the choice of action, we may also simply choose a state at random.
Step 0. Initialization:

Step 0a. Initialize an approximation for the value function $\tilde{Q}_0^t(S_t, a_t)$ for all states $S_t$ and decisions $a_t \in A_t$, $t = \{0, 1, \ldots, T\}$.

Step 0b. Set $n = 1$.

Step 0c. Initialize $S^1_0$.

Step 1. Choose a sample path $\omega^n$.

Step 2. Do for $t = 0, 1, \ldots, T$:

Step 2a. Determine the action using $\epsilon$-greedy. With probability $\epsilon$, choose an action $a^n$ at random from $A$. With probability $1 - \epsilon$, choose $a^n$ using $a^n_t = \arg\max_{a_t \in A_t} \tilde{Q}^{n-1}_t(S^n_t, a_t)$.

Step 2b. Sample $W_{t+1}^n = W_{t+1}(\omega^n)$ and compute the next state $S^n_{t+1} = S^M(S^n_t, a^n_t, W_{t+1}^n)$.

Step 2c. Compute:

$\hat{q}^n_t = C(S^n_t, a^n_t) + \gamma \max_{a_{t+1} \in A_{t+1}} \tilde{Q}^{n-1}_{t+1}(S^n_{t+1}, a_{t+1})$.

Step 2d. Update $\tilde{Q}^{n-1}_t$ and $V^{n-1}_t$ using:

$\tilde{Q}^n_t(S^n_t, a^n_t) = (1 - \alpha_{n-1})\tilde{Q}^{n-1}_t(S^n_t, a^n_t) + \alpha_{n-1}\hat{q}^n_t$.

Step 3. Increment $n$. If $n \leq N$ go to Step 1.

Step 4. Return the Q-factors $(\tilde{Q}^n_t)_{t=1}^T$.

Figure 10.3 A Q-learning algorithm.

If the learning policy also determines the next state we visit, then we say that the algorithm is on policy. If the sampling policy is different than the learning policy, then we say that the algorithm is off policy, which means that the policy that we use to determine the next state to visit does not follow the policy we are trying to optimize.

In the remainder of this chapter, we are going to distinguish on-policy and off-policy algorithms. However, we defer to chapter 12 a more complete discussion of the different policies that can be used.

10.2.3 Q-learning

We first introduced Q-learning and its cousin SARSA in chapter 4, and we refer the reader back to this presentation for an introduction to this important algorithmic strategy. For completeness, we present a version of the full algorithm in figure 10.3 for a time-dependent, finite-horizon problem.

10.2.4 Value iteration using a post-decision state variable

For the many applications that lend themselves to a compact post-decision state variable, it is possible to adapt approximate value iteration to value functions estimated around the post-decision state variable. At the heart of the algorithm we choose actions (and estimate the value of being in state $S^n_t$) using

$\tilde{v}_t^n = \arg\max_{a_t \in A_t} \left( C(S^n_t, a_t) + \gamma \tilde{Q}_{t-1}^{n-1}(S^M, a_t) \right)$.
Step 0. Initialization:

Step 0a. Initialize an approximation for the value function \( V^0_t(S^n_t) \) for all post-decision states \( S^n_t, t = \{0, 1, \ldots, T\} \).

Step 0b. Set \( n = 1 \).

Step 0c. Initialize \( S^n_{0,1} \).

Step 1. Choose a sample path \( \omega^n \).

Step 2. Do for \( t = 0, 1, \ldots, T \):

Step 2a: Determine the action using \( \epsilon \)-greedy. With probability \( 1 - \epsilon \), choose an action \( \alpha^n \) at random from \( \mathcal{A} \). With probability \( \epsilon \), choose \( \alpha^n \) using

\[
\hat{v}^n_t = \arg \max_{a_t \in \mathcal{A}_t} \left( C(S^n_t, a_t) + \gamma V^{n-1}_{t-1}(S^{M,a}_{n-1}(S^n_t, a_t)) \right).
\]

Let \( \alpha^n_t \) be the action that solves the maximization problem.

Step 2b. Update \( V^{n-1}_t(S^n_{t-1}) \) using:

\[
V^{n-1}_t(S^n_{t-1}) = (1 - \alpha_{n-1}) V^{n-1}_{t-1}(S^n_{t-1}) + \alpha_{n-1} \hat{v}^n_t.
\]

Step 2c. Sample \( W^n_{t+1} = W_{t+1}(\omega^n) \) and compute the next state \( S^n_{t+1} = S^M(S^n_t, \alpha^n_t, W^n_{t+1}) \).

Step 3. Increment \( n \). If \( n \leq N \) go to Step 1.

Step 4. Return the value functions \( (V^n_t)_{t=1}^T \).

Figure 10.4 Approximate value iteration for finite horizon problems using the post-decision state variable.

The distinguishing feature when we use the post-decision state variable is that the maximization problem is now deterministic. The key step is how we update the value function approximation. Instead of using \( \hat{v}^n_t \) to update a pre-decision value function approximation \( V^{n-1}_t(S^n_t) \), we use \( \hat{v}^n_t \) to update a post-decision value function approximation around the previous post-decision state \( S^n_{t-1} \). This is done using

\[
V^{n-1}_t(S^n_{t-1}) = (1 - \alpha_{n-1}) V^{n-1}_{t-1}(S^n_{t-1}) + \alpha_{n-1} \hat{v}^n_t.
\]

The post-decision state not only allows us to solve deterministic optimization problems, there are many applications where the post-decision state has either the same dimensionality as the pre-decision state, or, for some applications, a much lower dimensionality. Examples were given in section 4.6. A complete summary of the algorithm is given in figure 10.4.

\( Q \)-learning shares certain similarities with dynamic programming using a post-decision value function. In particular, both require the solution of a deterministic optimization problem to make a decision. However, \( Q \)-learning accomplishes this goal by creating an artificial post-decision state given by the state/action pair \((S, a)\). We then have to learn the value of being in \((S, a)\), rather than the value of being in state \( S \) alone (which is already very hard for most problems).

If we compute the value function approximation \( V^n(S^n) \) around the post-decision state \( S^n = S^{M,a}(S, a) \), we can create \( Q \)-factors directly from the contribution function and the post-decision value function using

\[
Q^n(S, a) = C(S, a) + \gamma V^n_t(S^{M,a}(S, a)).
\]
Viewed this way, approximate value iteration using value functions estimated around a post-decision state variable is equivalent to Q-learning. However, if the post-decision state is compact, then estimating $\hat{V}(S^n)$ is much easier than estimating $\hat{Q}(S,a)$.

10.2.5 Value iteration using a backward pass

Classical approximate value iteration, which is equivalent to temporal difference learning with $\lambda = 0$ (also known as TD(0)), can be implemented using a pure forward pass, which enhances its simplicity. However, there are problems where it is useful to simulate decisions moving forward in time, and then updating value functions moving backward in time. This is also known as temporal difference learning with $\lambda = 1$, but we find “backward pass” to be more descriptive. The algorithm is depicted in figure 10.5.

In this algorithm, we step forward through time creating a trajectory of states, actions, and outcomes. We then step backwards through time, updating the value of being in a state using information from the same trajectory in the future. We are going to use this algorithm to also illustrate ADP for a time-dependent, finite horizon problem. In addition, we are going to illustrate a form of policy evaluation. Pay careful attention to how variables are indexed.

The idea of stepping backward through time to produce an estimate of the value of being in a state was first introduced in the control theory community under the name of backpropagation through time (BTT). The result of our backward pass is $\hat{v}^{n}_{t}$, which is the contribution from the sample path $\omega^{n}$ and a particular policy. Our policy is, quite literally, the set of decisions produced by the value function approximation $\hat{V}^{n-1}$. Unlike our forward-pass algorithm (where $\hat{v}^{n}_{t}$ depends on the approximation $\hat{V}^{n-1}_{t}(S^{n}_{t})$), $\hat{v}^{n}_{t}$ is a valid, unbiased estimate of the value of being in state $S^{n}_{t}$ at time $t$ and following the policy produced by $\hat{V}^{n-1}$. We introduce an inner loop so that rather than updating the value function approximation with a single $\hat{v}^{n}_{t}$, we average across a set of samples to create a more stable estimate, $\bar{v}^{n}_{t}$.

These two strategies are easily illustrated using our simple asset selling problem. For this illustration, we are going to slightly simplify the model we provided earlier, where we assumed that the change in price, $\hat{p}_{t}$, was the exogenous information. If we use this model, we have to retain the price $p_{t}$ in our state variable (even the post-decision state variable). For our illustration, we are going to assume that the exogenous information is the price itself, so that $p_{t} = \hat{p}_{t}$. We further assume that $\hat{p}_{t}$ is independent of all previous prices (a pretty strong assumption). For this model, the pre-decision state is $S_{t} = (R_{t}, p_{t})$ while the post-decision state variable is simply $S^{n}_{t} = R^{n}_{t} = R_{t} - a_{t}$ which indicates whether we are holding the asset or not. Further, $S_{t+1} = S^{n}_{t}$ since the resource transition function is deterministic.

With this model, a single-pass algorithm (approximate value iteration) is performed by stepping forward through time, $t = 1, 2, \ldots, T$. At time $t$, we first sample $\hat{p}_{t}$ and we find

$$\hat{v}_{t}^{n} = \max_{a_{t} \in \{0,1\}} \left( \hat{p}_{t}^{n} a_{t} + (1 - a_{t})(-c_{t} + \bar{v}_{t+1}^{n-1}) \right).$$

Assume that the holding cost $c_{t} = 2$ for all time periods.

Table 10.1 illustrates three iterations of a single-pass algorithm for a three-period problem. We initialize $\bar{v}_{t}^{0} = 0$ for $t = 0, 1, 2, 3$. Our first decision is $a_{1}$ after we see $\hat{p}_{1}$. The first column shows the iteration counter, while the second shows the stepsize $\alpha_{n-1} = 1/n$. For the first iteration, we always choose to sell because $\bar{v}_{t}^{0} = 0$, which means that $\hat{v}_{1}^{1} = \hat{p}_{1}^{1}$. Since our stepsize is 1.0, this produces $\bar{v}_{t-1}^{1} = \hat{p}_{t}^{1}$ for each time period.
Step 0. Initialization:

Step 0a. Initialize $V^n_t$, $t \in T$.
Step 0b. Initialize $S^n_1$.
Step 0c. Choose an initial policy $A^{\pi,0}$.
Step 0d. Set $n = 1$.

Step 1. Repeat for $m = 1, 2, \ldots, M$:

Step 1. Choose a sample path $\omega^m$.
Step 2: Do for $t = 0, 1, 2, \ldots, T$:

Step 2a: Find
\[ a^n_{t,m} = A^{\pi,n-1}(S^n_t) \]
Step 2b: Update the state variable
\[ S^{n,m}_{t+1} = S^M(S^{n,m}_t, a^{n,m}_t, W_{t+1}(\omega^n)) \]
Step 3: Set $\hat{v}^{n,m}_{T+1} = 0$ and do for $t = T, T-1, \ldots, 1$:
\[ \hat{v}^{n,m}_t = C(S^{n,m}_t, a^{n,m}_t) + \gamma \hat{v}^{n,m}_{t+1} \]
Step 4: Compute the average value from starting in state $S^n_1$:
\[ \bar{v}^n_0 = \frac{1}{M} \sum_{m=1}^{M} v^{n,m}_0. \]
Step 5. Update the value function approximation by using the average values:
\[ V^n_0 \leftarrow U^{V}(T^n_{0}, S^n_0, \bar{v}^n_0). \]
Step 6. Update the policy
\[ A^{\pi,n}(S) = \arg \max_{a \in A} (C(S^n_0, a) + \gamma V^n_0(S^M(S^n_0, a))) \]
Step 6. Increment $n$. If $n \leq N$ go to Step 1.
Step 7. Return the value functions $(V^n_t)_{t=1}^T$.

Figure 10.5 Double-pass version of the approximate dynamic programming algorithm for a finite horizon problem.

In the second iteration, our first decision problem is
\[ \hat{v}^2_1 = \max\{p^2_1 - c_1 + \bar{v}^1_1\} \]
\[ = \max\{24, -2 + 34\} \]
\[ = 32, \]
which means $a^2_1 = 0$ (since we are holding). We then use $\hat{v}^2_1$ to update $\bar{v}^2_0$ using
\[ \bar{v}^2_0 = (1 - \alpha_1)\bar{v}^1_0 + \alpha_1 \hat{v}^2_1 \]
\[ = (0.5)30.0 + (0.5)32.0 \]
\[ = 31.0 \]
Repeating this logic, we hold again for $t = 2$ but we always sell at $t = 3$ since this is the last time period. In the third pass, we again sell in the first time period, but hold for the second time period.

It is important to realize that this problem is quite simple, and we do not have to deal with exploration issues. If we sell, we are no longer holding the asset and the forward pass should stop (more precisely, we should continue to simulate the process given that we have sold the asset). Instead, even if we sell the asset, we step forward in time and continue to evaluate the state that we are holding the asset (the value of the state where we are not holding the asset is, of course, zero). Normally, we evaluate only the states that we transition to (see step 2b), but for this problem, we are actually visiting all the states (since there is, in fact, only one state that we really need to evaluate).

Now consider a double-pass algorithm. Table 10.2 illustrates the forward pass, followed by the backward pass, where for simplicity we are going to use only a single inner iteration ($M = 1$). Each line of the table only shows the numbers determined during the forward or backward pass. In the first pass, we always sell (since the value of the future is zero), which means that at each time period the value of holding the asset is the price in that period.

In the second pass, it is optimal to hold for two periods until we sell in the last period. The value $\hat{v}_2^3$ for each time period is the contribution of the rest of the trajectory which, in this case, is the price we receive in the last time period. So, since $a_1 = a_2 = 0$ followed by $a_3 = 1$, the value of holding the asset at time 3 is the $27$ price we receive for selling in that time period. The value of holding the asset at time $t = 2$ is the holding cost of $-2$ plus $\hat{v}_2^3$, giving $\hat{v}_2^3 = -2 + \hat{v}_3^3 = -2 + 27 = 25$. Similarly, holding the asset at time 1 means $\hat{v}_1^3 = -2 + \hat{v}_2^3 = -2 + 25 = 23$. The smoothing of $\hat{v}_n^t$ with $\bar{v}_{n-1}^{t-1}$ to produce $\bar{v}_{n-1}^{t-1}$ is the same as for the single pass algorithm.
The value of implementing the double-pass algorithm depends on the problem. For example, imagine that our asset is an expensive piece of replacement equipment for a jet aircraft. We hold the part in inventory until it is needed, which could literally be years for certain parts. This means there could be hundreds of time periods (if each time period is a day) where we are holding the part. Estimating the value of the part now (which would determine whether we order the part to hold in inventory) using a single-pass algorithm could produce extremely slow convergence. A double-pass algorithm would work dramatically better. But if the part is used frequently, staying in inventory for only a few days, then the single-pass algorithm will work fine.

10.2.6 Value iteration for multidimensional decision vectors

In the previous section we saw that the use of the post-decision state variable meant that the process of choosing the best action required solving a deterministic optimization problem. This opens the door to considering problems where the decision is a vector $x_t$. For example, imagine that we have a problem of assigning an agent in a multiskill call center to customers requiring help with their computers. An agent of type $i$ might have a particular set of language and technical skills. A customer has answered a series of automated questions, which we capture with a label $j$. Let

\[ R_{ti} = \text{The number of agents available at time } t \text{ with skill set } i, \]
\[ D_{tj} = \text{The number of customers waiting at time } t \text{ whose queries are characterized by } j. \]

We let $R_t = (R_{ti})$ and $D_t = (D_{tj})$, and finally let our state variable be $S_t = (R_t, D_t)$. Let our decision vector be defined using

\[ x_{tij} = \text{Number of agents of type } i \text{ who are assigned to customers of type } j \text{ at time } t, \]
\[ x_t = (x_{tij}). \]

For realistic problems, it is easy to create vectors $x_t$ with hundreds or thousands of dimensions. Finally, let

\[ c_{ij} = \text{Estimated time required for an agent of type } i \text{ to serve a customer of type } j. \]

The state variables evolve according to

\[ R_{t+1,i} = R_{ti} - \sum_j x_{tij} + \hat{R}_{t+1,i}, \]
\[ D_{t+1,j} = D_{tj} - \sum_i x_{tij} + \hat{D}_{t+1,j}. \]

Here, $\hat{R}_{t+1,i}$ represents the number of agents that were busy but which became idle between $t$ and $t + 1$ because they completed their previous assignment. $\hat{D}_{t+1,j}$ represents arrival of new customers. We note that $R_t$ and $D_t$ are pre-decision state variables. Their post-decision counterparts are given by

\[ R_{t+1,i}^p = R_{ti} - \sum_j x_{tij}, \]  
\[ D_{t+1,j}^p = D_{tj} - \sum_i x_{tij}. \]
We are going to have to create a value function approximation \( V(S_t^x) \). For the purpose of illustrating the basic idea, assume we use a separable approximation, which we can write using
\[
V_t(R_{ti}^x, D_{tj}^x) = \sum_i V_t^{R_i}(R_{ti}^x) + \sum_j V_t^{D_j}(D_{tj}^x).
\]

Since we are minimizing, we intend to create scalar, convex approximations for \( V_t^{R_i}(R_{ti}^x) \) and \( V_t^{D_j}(D_{tj}^x) \). Now, our VFA policy requires solving a linear (or nonlinear) math programming problem of the form
\[
\min_{x_t} \sum_i \sum_j c_{ij} x_{tij} + \sum_i V_{t-1,i}(R_{ti}^x) + \sum_j V_{t-1,j}(D_{tj}^x)
\]
(10.14)
where \( R_{ti}^x \) and \( D_{tj}^x \) are given by (10.12) and (10.13), respectively. Also, our optimization problem has to be solved subject to the constraints
\[
\sum_i x_{tij} \leq R_{ti}, \quad (10.15)
\]
\[
\sum_i x_{tij} \leq D_{tj}, \quad (10.16)
\]
\[
x_{tij} \geq 0. \quad (10.17)
\]

The optimization problem described by equations (10.14) - (10.17) is a linear or nonlinear optimization problem. If the scalar, separable value function approximations are convex, this is generally fairly easy to solve, even when \( x_t \) has hundreds or thousands of dimensions.

Elsewhere, we would let \( \hat{v}_t^x \) be the value of the optimal objective function, which we then use to update the value function approximation. For this problem class, we are going to take advantage of the fact that the optimal solution will yield dual variables for the constraints (10.15) and (10.16). Call these dual variables \( \hat{v}_t^{R_i} \) and \( \hat{v}_t^{D_j} \), respectively. Thus, we can interpret \( \hat{v}_t^{R_i} \) as an approximation of the marginal value of \( R_{ti} \), while \( \hat{v}_t^{D_j} \) is an approximation of the marginal value of \( D_{tj} \). We can use these marginal values to update our value function approximations. However, we would use \( \hat{v}_t^{R_i} \) and \( \hat{v}_t^{D_j} \) to update the value function approximations \( V_{t-1,i}(R_{ti}^{x-1}) \) and \( V_{t-1,j}(D_{tj}^{x-1}) \), which means we are using information from the problem we solve at time \( t \) to update value function approximations at time \( t-1 \) around the previous, post-decision state variable.

This logic is described in much greater detail in chapters 13 and 14. Our goal here is simply to illustrate the ability to use the post-decision value function to handle optimization problems with high-dimensional decision vectors.

### 10.3 STATISTICAL BIAS IN THE MAX OPERATOR

A subtle type of bias arises when we are optimizing because we are taking the maximum over a set of random variables. In algorithms such as Q-learning or approximate value iteration, we are computing \( \hat{q}_n \) by choosing the best of a set of decisions which depend on \( \bar{Q}_n(S, a) \). The problem is that the estimates \( \bar{Q}_n(S, a) \) are random variables. In the best of circumstances, assume that \( \bar{Q}_n(S, a) \) is an unbiased estimate of the true value \( V_t(S^a) \) of being in (post-decision) state \( S^a \). Because it is still a statistical estimate with
some degree of variation, some of the estimates will be too high while others will be too
low. If a particular action takes us to a state where the estimate just happens to be too high
(due to statistical variation), then we are more likely to choose this as the best action and
use it to compute $\hat{q}^n$.

To illustrate, assume we have to choose an action $a \in A$, where $C(S, a)$ is the contribu-
tion earned by using decision $a$ (given that we are in state $S$) which then takes us to state
$S^a(S, a)$ where we receive an estimated value $V(S^a(S, a))$. Normally, we would update
the value of being in state $S$ by computing

$$
\hat{v}^n = \max_{a \in A} \left( C(S, a) + V^{n-1}(S, a) \right).
$$

We would then update the value of being in state $S$ using our standard update formula

$$
V^n(S) = (1 - \alpha_n) V^{n-1}(S) + \alpha_n \hat{v}^n.
$$

Since $V^{n-1}(S^a(S, a))$ is a random variable, sometimes it will overestimate the true value
of being in state $S^a(S, a)$ while other times it will underestimate the true value. Of course,
we are more likely to choose an action that takes us to a state where we have overestimated
the value.

We can quantify the error due to statistical bias as follows. Fix the iteration counter $n$
(so that we can ignore it), and let

$$
U_a = C(S, a) + V(S^a(S, a))
$$

be the estimated value of using action $a$. The statistical error, which we represent as $\beta$, is
given by

$$
\beta = \mathbb{E}\{\max_{a \in A} U_a\} - \max_{a \in A} \mathbb{E}U_a.
$$

The first term on the right-hand side of (10.18) is the expected value of $V(S)$, which is
computed based on the best observed value. The second term is the correct answer (which
we can only find if we know the true mean). We can get an estimate of the difference by
using a strategy known as the “plug-in principle.” We assume that $\mathbb{E}U_a = V(S^a(S, a))$,
which means that we assume that the estimates $\hat{V}(S^a(S, a))$ are correct, and then try to
estimate $\mathbb{E}\{\max_{a \in A} U_a\}$. Thus, computing the second term in (10.18) is easy.

The challenge is computing $\mathbb{E}\{\max_{a \in A} U_a\}$. We assume that while we have been com-
puting $\hat{V}(S^a(S, a))$, we have also been computing $\bar{\sigma}^2(a) = \text{Var}(U_a) = \text{Var}(\hat{V}(S^a(S, a)))$.
Using the plug-in principle, we are going to assume that the estimates $\bar{\sigma}^2(a)$ represent the
true variances of the value function approximations. Computing $\mathbb{E}\{\max_{a \in A} U_a\}$ for more
than a few decisions is computationally intractable, but we can use a technique called
the Clark approximation to provide an estimate. This strategy finds the exact mean and
variance of the maximum of two normally distributed random variables, and then assumes
that this maximum is also normally distributed. Assume the decisions can be ordered so
that $A = \{1, 2, \ldots, |A|\}$. Now let

$$
\bar{U}_2 = \max\{U_1, U_2\}.
$$

We can compute the mean and variance of $\bar{U}_2$ as follows. First compute

$$
\text{Var}(\bar{U}_2) = \sigma_1^2 + \sigma_2^2 - 2\sigma_1\sigma_2 \rho_{12}
$$
where \( \sigma_1^2 = \text{Var}(U_1) \), \( \sigma_2^2 = \text{Var}(U_2) \), and \( \rho_{12} \) is the correlation coefficient between \( U_1 \) and \( U_2 \) (we allow the random variables to be correlated, but shortly we are going to approximate them as being independent). Next find

\[
z = \frac{\mu_1 - \mu_2}{\sqrt{\text{Var}(\bar{U}_2)}}.
\]

where \( \mu_1 = \mathbb{E}U_1 \) and \( \mu_2 = \mathbb{E}U_2 \). Now let \( \Phi(z) \) be the cumulative standard normal distribution (that is, \( \Phi(z) = \mathbb{P}[Z \leq z] \) where \( Z \) is normally distributed with mean 0 and variance 1), and let \( \phi(z) \) be the standard normal density function. If we assume that \( U_1 \) and \( U_2 \) are normally distributed (a reasonable assumption when they represent sample estimates of the value of being in a state), then it is a straightforward exercise to show that

\[
\mathbb{E}\bar{U}_2 = \mu_1 \Phi(z) + \mu_2 \Phi(-z) + \sqrt{\text{Var}(\bar{U}_2)} \phi(z)
\]

\[
\text{Var}(\bar{U}_2) = \left( \mu_1^2 + \sigma_1^2 \right) \Phi(z) + \left( \mu_2^2 + \sigma_2^2 \right) \Phi(-z) + (\mu_1 + \mu_2) \sqrt{\text{Var}(\bar{U}_2)} \phi(z)
\]

(10.19)

(10.20)

Now assume that we have a third random variable, \( U_3 \), where we wish to find \( \mathbb{E}\max\{U_1, U_2, U_3\} \). The Clark approximation solves this by using

\[
\bar{U}_3 = \mathbb{E}\max\{U_1, U_2, U_3\} \\
\approx \mathbb{E}\max\{U_3, \bar{U}_2\},
\]

where we assume that \( \bar{U}_2 \) is normally distributed with mean given by (10.19) and variance given by (10.20). For our setting, it is unlikely that we would be able to estimate the correlation coefficient \( \rho_{12} \) (or \( \rho_{23} \)), so we are going to assume that the random estimates are independent. This idea can be repeated for large numbers of decisions by using

\[
\bar{U}_a = \mathbb{E}\max\{U_1, U_2, \ldots, U_a\} \\
\approx \mathbb{E}\max\{U_a, \bar{U}_{a-1}\}.
\]

We can apply this repeatedly until we find the mean of \( \bar{U}_{|A|} \), which is an approximation of \( \mathbb{E}\max_{a \in A} U_a \). This, in turn, allows us to compute an estimate of the statistical bias \( \beta \) given by equation (10.18).

Figure 10.6 plots \( \beta = \mathbb{E}\max_a U_a - \max_a \mathbb{E}U_a \) as it is being computed for 100 decisions, averaged over 30 sample realizations. The standard deviation of each \( U_a \) was fixed at \( \sigma = 20 \). The plot shows that the error increases steadily until the set \( A \) reaches about 20 or 25 decisions, after which it grows much more slowly. Of course, in an approximate dynamic programming application, each \( U_a \) would have its own standard deviation which would tend to decrease as we sample a decision repeatedly (a behavior that the approximation above captures nicely).

This brief analysis suggests that the statistical bias in the max operator can be significant. However, it is highly data dependent. If there is a single dominant decision, then the error will be negligible. The problem only arises when there are many (as in 10 or more) decisions that are competitive, and where the standard deviation of the estimates is not small relative to the differences between the means. Unfortunately, this is likely to be the case in most large-scale applications (if a single decision is dominant, then it suggests that the solution is probably obvious).
The relative magnitudes of value iteration bias over statistical bias will depend on the nature of the problem. If we are using a pure forward pass (TD(0)), and if the value of being in a state at time $t$ reflects rewards earned over many periods into the future, then the value iteration bias can be substantial (especially if the stepsize is too small). Value iteration bias has long been recognized in the dynamic programming community. By contrast, statistical bias appears to have received almost no attention, and as a result we are not aware of any research addressing this problem. We suspect that statistical bias is likely to inflate value function approximations fairly uniformly, which means that the impact on the policy may be quite small. However, if the goal is to obtain the value function itself (for example, to estimate the value of an asset or a contract), then the bias can distort the results.

### 10.4 APPROXIMATE VALUE ITERATION AND Q-LEARNING USING LINEAR MODELS

Approximate value iteration, $Q$-learning and temporal difference learning (with $\lambda = 0$) are clearly the simplest methods for updating an estimate of the value of being in a state. Linear models are the simplest methods for approximating a value function. Not surprisingly, then, there has been considerable interest in putting these two strategies together.

Figure 10.7 depicts a basic adaptation of approximate value iteration using a linear model to approximate the value function. The strategy is popular because it is so easy to implement. As of this writing, the design of provably convergent algorithms based on approximate value iteration using linear models is an active area of research. However, general purpose algorithms such as the one sketched in figure 10.7 are not convergent, and may behave quite badly. This is true even if the true value function can be perfectly represented using a set of basis functions.

The problem with approximate value iteration is that the update of the coefficient vector $\theta^m$ can be noisy, and this immediately impacts the policy, which further contributions to the noise. In short, the strategy is fundamentally unstable.
Step 0. Initialization:

Step 0a. Initialize $V^0$.
Step 0b. Initialize $S^1$.
Step 0c. Set $n = 1$.

Step 1. Solve

$$
\hat{v}^n = \max_{a \in A^n} \left( C(S^n, a) + \gamma \sum_f \theta_f^{n-1} \phi_f(S^M(S^n, a)) \right)
$$

(10.21)

and let $a^n$ be the value of $a$ that solves (10.21).

Step 2. Update the value function recursively using equations (9.24)-(9.28) from chapter 9 to obtain $\theta^n$.

Step 3. Choose a sample $W^{n+1} = W(\omega^{n+1})$ and determine the next state using some policy such as $S^n = S^M(S^n, a^n, W^{n+1})$.

Step 3. Increment $n$. If $n \leq N$ go to Step 1.

Step 4. Return the value functions $V^N$.

Figure 10.7 Approximate value iteration using a linear model.

It is possible to design algorithms using approximate value iteration that are provably convergent, but these require special structure. For example, chapters 13 and 14 describe the use of approximate value iteration in the context of resource allocation problems, where we can exploit the property of concavity. This is an incredibly powerful property and allows us to obtain algorithms that are both practical and, under certain conditions, provably convergent. However, without these properties, there are few guarantees. It is for this reason that the research community has focused more on approximate policy iteration (see section 10.5 below).

Despite the potentially poor performance of this algorithm, it remains a popular strategy because its simplicity. In addition, while it may not work, it might work quite well! The important point here is that it is a strategy that may be worth trying, but caution needs to be used. Given these observations, this section is aimed at providing some guidance to improve the chances that the algorithm will work.

The most important step whenever a linear model is used, regardless of the setting, is to choose the basis functions carefully so that the linear model has a chance of representing the true value function accurately. The biggest strength of a linear model is also its biggest weakness. A large error can distort the update of $\theta^n$ which then impacts the accuracy of the entire approximation. Since the value function approximation determines the policy (see Step 1), a poor approximation leads to poor policies, which then distorts the observations $\hat{v}^n$. This can be a vicious circle from which the algorithm may never recover.

A second step is in the specific choice of recursive least squares updating. Figure 10.7 refers to the classic recursive least squares updating formulas in equations (9.24)-(9.28). However, buried in these formulas is the implicit use of a stepsize rule of $1/n$. We show in chapter 11 that a stepsize $1/n$ is particularly bad for approximate value iteration (as well as $Q$-learning and TD(0) learning). While this stepsize can work well (indeed, it is optimal) for stationary data, it is very poorly suited for the backward learning that arises in approximate value iteration. Fortunately, the problem is easily fixed if we replace the
updating equations for $B^n$ and $\gamma$, which are given as

$$
B^n = B^{n-1} - \frac{1}{\gamma^n}(B^{n-1}\phi^n(\phi^n)^T B^{n-1}),
$$

$$
\gamma^n = 1 + (\phi^n)^T B^{n-1}\phi^n,
$$
in equations (9.27) and (9.28) with

$$
B^n = \frac{1}{\lambda} \left( B^{n-1} - \frac{1}{\gamma^n}(B^{n-1}\phi^n(\phi^n)^T B^{n-1}) \right),
$$

$$
\gamma^n = \lambda + (\phi^n)^T B^{n-1}\phi^n,
$$
in equations (9.30) and (9.31). Here, $\lambda$ discounts older errors. $\lambda = 1$ produces the original recursive formulas. When used with approximate value iteration, it is important to use $\lambda < 1$. In section 9.3.2, we argue that if you choose a stepsize rule for $\alpha_n$ such as $\alpha_n = a/(a + n - 1)$, you should set $\lambda_n$ at iteration $n$ using

$$
\lambda_n = \alpha_{n-1} \left( \frac{1 - \alpha_n}{\alpha_n} \right).
$$

The last issue that needs special care is the rule for determining the next state to visit. As we discussed in section 9.4, temporal difference learning (TD(0) is approximate value iteration for a fixed policy) is only guaranteed to converge while using on-policy learning, and can diverge when using off-policy learning. Off-policy learning is a cornerstone of convergence proofs when using lookup tables, but it appears to cause considerable problems when using a linear model (it gets worse if our model is nonlinear in the parameters). At the same time, exploration should not be needed if our basis functions are carefully chosen (for a fixed policy). Interestingly, it was recently shown that approximate value iteration may converge if we start with a policy that is close enough to the optimal policy. However, this condition is relaxed as the discount factor is decreased, suggesting that a good strategy is to start with a smaller discount factor, find a good policy for that discount factor and then increase the discount factor toward the desired level.

### 10.5 APPROXIMATE POLICY ITERATION

One of the most important tools in the toolbox for approximate dynamic programming is approximate policy iteration. This algorithm is neither simpler nor more elegant than approximate value iteration, but it can offer convergence guarantees while using linear models to approximate the value function.

In this section we review several flavors of approximate policy iteration, including

a) Finite horizon problems using lookup tables.

b) Finite horizon problems using basis functions.

c) Infinite horizon problems using basis functions.

Finite horizon problems allow us to obtain Monte Carlo estimates of the value of a policy by simulating the policy until the end of the horizon. Note that a “policy” here always refers to decisions that are determined by value function approximations. We use the finite horizon
setting to illustrate approximating value function approximations using lookup tables and basis functions, which allows us to highlight the strengths and weaknesses of the transition to basis functions.

We then present an algorithm based on least squares temporal differences (LSTD) and contrast the steps required for finite horizon and infinite horizon problems when using basis functions.

10.5.1 Finite horizon problems using lookup tables

A fairly general purpose version of an approximate policy iteration algorithm is given in figure 10.8 for an infinite horizon problem. This algorithm helps to illustrate the choices that can be made when designing a policy iteration algorithm in an approximate setting. The algorithm features three nested loops. The innermost loop steps forward and backward in time from an initial state \( S_n^0 \). The purpose of this loop is to obtain an estimate of the value of a path. Normally, we would choose \( T \) large enough so that \( \gamma^T \) is quite small (thereby approximating an infinite path). The next outer loop repeats this process \( M \) times to obtain a statistically reliable estimate of the value of a policy (determined by \( V^{\pi,n} \)). The third loop, representing the outer loop, performs policy updates (in the form of updating the value function). In a more practical implementation, we might choose states at random rather than looping over all states.

Readers should note that we have tried to index variables in a way that shows how they are changing (do they change with outer iteration \( n \)? inner iteration \( m \)? the forward look-ahead counter \( t \)?)? This does not mean that it is necessary to store, for example, each state or decision for every \( n \), \( m \), and \( t \). In an actual implementation, the software should be designed to store only what is necessary.

We can create different variations of approximate policy iteration by our choice of parameters. First, if we let \( T \to \infty \), we are evaluating a true infinite horizon policy. If we simultaneously let \( M \to \infty \), then \( \hat{v}^n \) approaches the exact, infinite horizon value of the policy \( \pi \) determined by \( V^{\pi,n} \). Thus, for \( M = T = \infty \), we have a Monte Carlo-based version of exact policy iteration.

We can choose a finite value of \( T \) that produces values \( \hat{v}^{n,m} \) that are close to the infinite horizon results. We can also choose finite values of \( M \), including \( M = 1 \). When we use finite values of \( M \), this means that we are updating the policy before we have fully evaluated the policy. This variant is known in the literature as optimistic policy iteration because rather than wait until we have a true estimate of the value of the policy, we update the policy after each sample (presumably, although not necessarily, producing a better policy). We may also think of this as a form of partial policy evaluation, not unlike the hybrid value/policy iteration described in section 3.6.

10.5.2 Finite horizon problems using basis functions

The simplest demonstration of approximate policy iteration using basis functions is in the setting of a finite horizon problem. Figure 10.9 provides an adaption of the algorithm using lookup tables when we are using basis functions. There is an outer loop over \( n \) where we fix the policy using

\[
A^\pi(S_t) = \arg\max_a \left( C(S_t, a) + \gamma \sum_{f} \theta^\pi_{tf} \phi_f(S_t) \right). \tag{10.22}
\]
Step 0. Initialization:

Step 0a. Initialize $\nabla^{\pi,0}$.  
Step 0b. Set a look-ahead parameter $T$ and inner iteration counter $M$.  
Step 0c. Set $n = 1$.  

Step 1. Sample a state $S^n_0$ and then do:  

Step 2. Do for $m = 1, 2, \ldots, M$:  

Step 3. Choose a sample path $\omega^m$ (a sample realization over the lookahead horizon $T$).  

Step 4. Do for $t = 0, 1, \ldots, T$:  

Step 4a. Compute
$$\alpha_{n,m}^t = \arg\max_{a_{n,m}^t \in \mathcal{A}_{n,m}^t} \left( C(S^n_{t,m}, a_{n,m}^t) + \gamma \nabla^{\pi,n-1}(S^M_{t,M}(S^n_{t+m-1,m}, a_{t,m})) \right).$$  

Step 4b. Compute
$$S^n_{t+1,m} = S^M(S^n_{t,m}, a_{n,m}^t, W_{t+1}(\omega^m)).$$  

Step 5. Initialize $\hat{\alpha}_{T+1,m}^n = 0$.  

Step 6. Do for $t = T, T - 1, \ldots, 0$:  

Step 6a. Accumulate $\hat{\alpha}_{n,m}^t$:  

$$\hat{\alpha}_{n,m}^t = C(S^n_{t,m}, a_{n,m}^t) + \gamma \hat{\alpha}_{t+1,m}^n.$$  

Step 6b. Update the approximate value of the policy:  

$$\hat{\alpha}_{n,m}^t = \left( \frac{m - 1}{m} \right) \hat{\alpha}_{n,m-1}^t + \frac{1}{m} \hat{\alpha}_{n,m}^0.$$  

Step 8. Update the value function at $S^n$:  

$$\nabla^{\pi,n} = (1 - \alpha_{n-1})\hat{\alpha}_{n-1}^n + \alpha_{n-1}\hat{\alpha}_{0}^n.$$  

Step 9. Set $n = n + 1$. If $n < N$, go to Step 1.  

Step 10. Return the value functions ($\nabla^{\pi,N}$).

---

Figure 10.8  A policy iteration algorithm for infinite horizon problems

We are assuming that the basis functions are not themselves time-dependent, although they depend on the state variable $S_t$ which, of course, is time dependent. The policy is determined by the parameters $\theta^{\pi,n}_{t,f}$.  

We update the policy $A^{\pi}_t(s)$ by performing repeated simulations of the policy in an inner loop that runs $m = 1, \ldots, M$. Within this inner loop, we use recursive least squares to update a parameter vector $\theta^{\pi,n}_{t,f}$. This step replaces step 6b in figure 10.8.  

If we let $M \to \infty$, then the parameter vector $\theta^{\pi,M}_{t,f}$ approaches the best possible fit for the policy $A^{\pi}_t(s)$ determined by $\theta^{\pi,n-1}$. However, it is very important to realize that this is not equivalent to performing a perfect evaluation of a policy using a lookup table representation. The problem is that (for discrete states), lookup tables have the potential for perfectly approximating a policy, whereas this is not generally true when we use basis functions. If we have a poor choice of basis functions, we may be able find the best possible value of $\theta^{\pi,n}_{t,f}$ as $m$ goes to infinity, but we may still have a terrible approximation of the policy produced by $\theta^{\pi,n-1}$.
**Step 0.** Initialization:

**Step 0a.** Fix the basis functions \( \phi_f(s) \).

**Step 0b.** Initialize \( \theta_{\pi}^{n,0} \) for all \( t \). This determines the policy we simulate in the inner loop.

**Step 0c.** Set \( n = 1 \).

**Step 1.** Sample an initial starting state \( S_0^m \):

**Step 2.** Initialize \( \theta_{\pi}^{1,0} \) (if \( n > 1 \), use \( \theta_{\pi}^{n,0} = \theta_{\pi}^{n-1} \)), which is used to estimate the value of policy \( \pi \) produced by \( \theta_{\pi}^{n,0} \). \( \theta_{\pi}^{n,0} \) is used to approximate the value of following policy \( \pi \) determined by \( \theta_{\pi}^{n,0} \).

**Step 3.** Do for \( m = 1, 2, \ldots, M \):

**Step 4.** Choose a sample path \( \omega^m \).

**Step 5.** Do for \( t = 0, 1, \ldots, T \):

**Step 5a.** Compute

\[
\hat{a}_{t+1}^{n,m} = \arg \max_{a_t \in A_t} \left( C(S_t^{n,m}, a_t) + \gamma \sum_f \theta_{\pi}^{n-1} \phi_f(S_{t+1}^{n,m}, a_t) \right).
\]

**Step 5b.** Compute

\[
S_{t+1}^{n,m} = S_M(S_t^{n,m}, a_t^{n,m}, W_{t+1}(\omega^m)).
\]

**Step 6.** Initialize \( \hat{v}_{T+1}^{n,m} = 0 \).

**Step 7.** Do for \( t = T, T - 1, \ldots, 0 \):

\[
\hat{v}_t^{n,m} = C(S_t^{n,m}, a_t^{n,m}) + \gamma \hat{v}_{t+1}^{n,m}.
\]

**Step 8.** Update \( \theta_{\pi}^{n,0} \) using recursive least squares to obtain \( \theta_{\pi}^{n,0} \) (see section 9.3).

**Step 9.** Set \( n = n + 1 \). If \( n < N \), go to Step 1.

**Step 10.** Return the value functions \( (V_{\pi}^{n,N}) \).

---

**Figure 10.9** A policy iteration algorithm for finite horizon problems using basis functions.

### 10.5.3 LSTD for infinite horizon problems using basis functions

We have built the foundation for approximate policy iteration using lookup tables and basis functions for finite horizon problems. We now make the transition to infinite horizon problems using basis functions, where we introduce the dimension of projecting contributions over an infinite horizon. There are several ways of accomplishing this (see section 9.1.2). We use least squares temporal differencing, since it represents the most natural extension of classical policy iteration for infinite horizon problems.

To begin, we let a sample realization of a one-period contribution, given state \( S^m \), action \( a^m \) and random information \( W^{m+1} \) be given by

\[
\hat{C}^m = C(S^m, a^m, W^{m+1}).
\]

As in the past, we let \( \phi^m = \phi(S^m) \) be the column vector of basis functions evaluated at state \( S^m \). We next fix a policy which chooses actions greedily based on a value function approximation given by \( \hat{V}^n(s) = \sum_f \theta^n_f \phi_f(s) \) (see equation (10.22)). Imagine that we have simulated this policy over a set of iterations \( i = (0, 1, \ldots, m) \), giving us a sequence of contributions \( \hat{C}^i, i = 1, \ldots, m \). Drawing on the foundation provided in section 9.5, we
can use standard linear regression to estimate $\theta^m$ using

$$\theta^m = \left[ \frac{1}{1 + m} \sum_{i=0}^{m} \phi_i (\phi^i - \gamma \phi^{i+1})^T \right]^{-1} \left[ \frac{1}{1 + m} \sum_{i=1}^{m} \phi^i \hat{C}^i \phi^i \right].$$  (10.23)

As a reminder, the term $\phi_i (\phi^i)^T - \gamma \phi_i (\phi^{i+1})^T$ can be viewed as a simulated, sample realization of $I - \gamma P$ projected onto the feature space. Just as we would use $(I - \gamma P)^{-1}$ in our basic policy iteration to project the infinite-horizon value of a policy $\pi$ (for a review, see section 3.5), we are using the term

$$\left[ \frac{1}{1 + m} \sum_{i=0}^{m} \phi_i (\phi^i - \gamma \phi^{i+1})^T \right]^{-1} \left[ \frac{1}{1 + m} \sum_{i=1}^{m} \phi^i \hat{C}^i \phi^i \right].$$

Equation (10.23) requires solving a matrix inverse for every observation. It is much more efficient to use recursive least squares, which is done by using

$$e^m = \hat{C}^m - (\phi^m - \gamma \phi^{m+1})^T \theta^{m-1},$$
$$B^m = B^{m-1} - \frac{B^m \phi^m (\phi^m - \gamma \phi^{m+1})^T B^{m-1}}{1 + (\phi^m - \gamma \phi^{m+1})^T B^{m-1} \phi^m},$$
$$\theta^m = \phi^{m-1} + \frac{e^m B^{m-1} \phi^m}{1 + (\phi^m - \gamma \phi^{m+1})^T B^{m-1} \phi^m}.$$

Figure 10.10 provides a detailed summary of the complete algorithm. The algorithm has some nice properties if we are willing to assume that there is a vector $\theta^*$ such that the true value function $V(s) = \sum_{f \in \mathcal{F}} \theta^*_f \phi_f(s)$ (admittedly, a pretty strong assumption). First, if the inner iteration limit $M$ increases as a function of $n$ so that the quality of the approximation of the policy gets better and better, then the overall algorithm will converge to the true optimal policy. Of course, this means letting $M \to \infty$, but from a practical perspective, it means that the algorithm can find a policy arbitrarily close to the optimal policy.

Second, the algorithm can be used with vector-valued and continuous actions (we normally use the notation $x_t$ in this case). There are several features of the algorithm that allow this. First, computing the policy $A^\pi(s|\theta^m)$ requires solving a deterministic optimization problem. If we are using discrete actions, it means simply enumerating the actions and choosing the best one. If we have continuous actions, we need to solve a nonlinear programming problem. The only practical issue is that we may not be able to guarantee that the objective function is concave (or convex if we are minimizing). Second, note that we are using trajectory following (also known as on-policy training) in Step 6c, without an explicit exploration step. The algorithm does not require exploration, but it does require that we be able to solve the recursive least squares equations.

We can avoid exploration as long as there is enough variation in the states we visit that allows us to compute $\theta^m$ in equation (10.23). When we use lookup tables, we require exploration to guarantee that we eventually will visit every state infinitely often. When we
Step 0. Initialization:

**Step 0a.** Initialize $\theta^0$.

**Step 0b.** Set the initial policy:

$$A^\pi(s|\theta^0) = \arg \max_{a \in A} \{ C(s,a) + \gamma \phi(S^M(s,a))^T \theta^0 \}.$$

**Step 0c.** Set $n = 1$.

**Step 1.** Do for $n = 1, \ldots, N$.

**Step 2.** Initialize $S^n_0$.

**Step 3.** Do for $m = 0, 1, \ldots, M$:

**Step 4.** Initialize $\theta^{n,m}$.

**Step 5.** Sample $W^{m+1}$.

**Step 6.** Do the following:

**Step 6a.** Computing the action $a^{n,m} = A^\pi(S^m|\theta^{n-1})$.

**Step 6b.** Compute the post-decision state $S^{n,m} = S^M(a^{n,m}, S^n_m, W^{m+1})$.

**Step 6c.** Compute the next pre-decision state $S^{n,m+1} = S^M(S^n_m, a^{n,m}, W^{m+1})$.

**Step 6d.** Compute the input variable $\phi(S^n_m) - \gamma \phi(S^{n,m+1})$ for equation (10.23).

**Step 7.** Do the following:

**Step 7a.** Compute the response variable $\hat{C}^m = C(S^n_m, a^{n,m}, W^{m+1})$.

**Step 7b.** Compute $\theta^{n,m}$ using equation (10.23).

**Step 8.** Update $\theta^n$ and the policy:

$$\theta^{n+1} = \theta^{n,m}$$

$$A^{\pi,n+1}(s) = \arg \max_{a \in A} \{ C(s,a) + \gamma \phi(S^M(s,a)) \theta^{n+1} \}.$$

**Step 9.** Return the $A^\pi(s|\theta^N)$ and parameter $\theta^N$.

---

**Figure 10.10** Approximate policy iteration for infinite horizon problems using least squares temporal differencing.

Use basis functions, we only need to visit states with sufficient diversity that we can estimate the parameter vector $\theta^m$. In the language of statistics, the issue is one of identification (that is, the ability to estimate $\theta$) rather than exploration. This is a much easier requirement to satisfy, and one of the major advantages of parametric models.

### 10.6 THE ACTOR-CRITIC PARADIGM

It is very popular in some communities to view approximate dynamic programming in terms of an “actor” and a “critic.” In this setting, a decision function that chooses a decision given the state is known as an actor. The process that determines the contribution (cost or reward) from a decision is known as the critic, from which we can compute a value function. The interaction of making decisions and updating the value function is referred to as an actor-critic framework. The slight change in vocabulary brings out the observation that the techniques of approximate dynamic programming closely mimic human behavior. This is especially true when we drop any notion of costs or contributions and simply work in terms of succeeding (or winning) and failing (or losing).
Step 0. Initialization:

Step 0a. Initialize $V^{n,0}_t, t \in T$.

Step 0b. Set $n = 1$.

Step 0c. Initialize $S^1_0$.

Step 1. Do for $n = 1, 2, \ldots, N$:

Step 2. Do for $m = 1, 2, \ldots, M$:

Step 3. Choose a sample path $\omega^m$.

Step 4: Initialize $\hat{v}^m = 0$

Step 5: Do for $t = 0, 1, \ldots, T$:

Step 5a. Solve:

$$a^{n,m}_t = \arg \max_{a_t \in A^{n,m}_t} \left( C_t(S^{n,m}_t, a_t) + \gamma V^{n,n-1}_{t-1}(S^{M,a}(S^{n,m}_t, a_t)) \right) \quad (10.24)$$

Step 5b. Compute:

$$S^{n,m}_t = S^{M,a}(S^{n,m}_t, a^{n,m}_t)$$

$$S^{n,m}_{t+1} = S^M(S^{n,m}_{t+1}, a^{n,m}, W_{t+1}(\omega^m))$$

Step 6. Do for $t = T-1, \ldots, 0$:

Step 6a. Accumulate the path cost (with $\hat{v}^m_T = 0$)

$$\hat{v}^m_t = C_t(S^{n,m}_t, a^{m}_t) + \gamma \hat{v}^m_{t+1}$$

Step 6b. Update approximate value of the policy starting at time $t$:

$$\hat{V}^{n,m}_{t-1} \leftarrow U^V(\hat{V}^{n,m-1}_{t-1}, S^{n,m}_{t-1}, \hat{v}^m_t) \quad (10.25)$$

where we typically use $\alpha_{m-1} = 1/m$.

Step 7. Update the policy value function

$$V^{\pi,n}_t(S^m_t) = \hat{V}^{n,M}_t(S^m_t) \quad \forall t = 0, 1, \ldots, T$$

Step 8. Return the value functions $(V^{\pi,n}_t)_{t=1}^T$.

Figure 10.11 Approximate policy iteration using value function-based policies.

The policy iteration algorithm in figure 10.11 provides one illustration of the actor-critic paradigm. The decision function is equation (10.24), where $V^{\pi,n-1}$ determines the policy (in this case). This is the actor. Equation (10.25), where we update our estimate of the value of the policy, is the critic. We fix the actor for a period of time and perform repeated iterations where we try to estimate value functions given a particular actor (policy). From time to time, we stop and use our value function to modify our behavior (something critics like to do). In this case, we update the behavior by replacing $V^\pi$ with our current $\hat{V}$.

In other settings, the policy is a rule or function that does not directly use a value function (such as $V^\pi$ or $\overline{V}$). For example, if we are driving through a transportation network (or traversing a graph) the policy might be of the form “when at node $i$, go next to node $j$.” As we update the value function, we may decide the right policy at node $i$ is to traverse to node $k$. Once we have updated our policy, the policy itself does not directly depend on a value function.

Another example might arise when determining how much of a resource we should have on hand. We might solve the problem by maximizing a function of the form $f(x) =$
\[ \beta_0 - \beta_1 (x - \beta_2)^2. \] Of course, \( \beta_0 \) does not affect the optimal quantity. We might use the value function to update \( \beta_0 \) and \( \beta_1 \). Once these are determined, we have a function that does not itself directly depend on a value function.

### 10.7 POLICY GRADIENT METHODS

Perhaps the cleanest illustration of the actor-critic framework arises when we parameterize both the value of being in a state as well as the policy. We use a standard strategy from the literature which uses \( Q \)-factors, and where the goal is to maximize the average contribution per time period (see section 3.7 for a brief introduction using the classical derivation based on transition matrices). Our presentation here represents only a streamlined sketch of an idea that is simple in principle but which involves some fairly advanced principles.

We assume that the \( Q \)-factors are parameterized using

\[
Q(s,a|\theta) = \sum_f \theta_f \phi_f(s,a).
\]

The policy is represented using a function such as

\[
A^\pi(s|\eta) = \frac{e^{\eta \phi(s,a)}}{\sum_{a'} e^{\eta \phi(s,a')}}.
\]

This choice of policy has the important feature that the probability that an action is chosen is greater than zero. Also, \( A^\pi(s|\eta) \) is differentiable in the policy parameter vector \( \eta \).

In the language of actor-critic algorithms, \( Q(s,a|\theta) \) is an approximation of the critic parameterized by \( \theta \), while \( A^\pi(s|\eta) \) is an approximate policy parameterized by \( \eta \). We can update \( \theta \) and \( \eta \) using standard stochastic gradient methods. We begin by defining

\[
\psi_\theta(s,a) = \nabla_\theta \ln \pi_\theta(a|s) = \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)}.
\]

Since we are maximizing the average reward per time period, we begin by estimating the average reward per time period using

\[
e^{n+1} = (1 - \alpha_n)e^n + \alpha_n C(S^{n+1}, a^{n+1}).
\]

We then compute the temporal difference in terms of the difference between the contribution of a state-action pair, and the average contribution, using

\[
\delta^n = C(S^n, a^n) - e^n + (\theta^n)^T \phi_{\theta^n}(S^{n+1}, a^{n+1}) - (\theta^n)^T \phi_{\theta^n}(S^n, a^n).
\]

Assume we are using a TD(\( \lambda \)) updating procedure where we assume \( 0 < \lambda < 1 \). We compute the eligibility trace using

\[
Z^{n+1} = \lambda Z^n + \phi_{\theta^n}(S^{n+1}, a^{n+1})
\]

We can now present the updating equations for the actor (the policy) and the critic (the \( Q \)-factors) in a simple and compact way. The actor update is given by

\[
\eta^{n+1} = \eta^n - \beta^n \Gamma(\theta^n) (\theta^n)^T \phi_{\theta^n}(S^{n+1}, a^{n+1}) \psi_\theta(S^{n+1}, a^{n+1}).
\]  

(10.26)
The critic update is given by

\[ \theta^{n+1} = \theta^n + \alpha_n \delta^n Z^n. \] (10.27)

Equations (10.26) and (10.27) provide an elegant and compact illustration of an actor-critic updating equation, where both the value function and the policy are approximated using parametric models. This method is likely to be of limited practical value, largely because of the form of the policy which requires a positive probability that any action may be chosen.

10.8 THE LINEAR PROGRAMMING METHOD USING BASIS FUNCTIONS

In section 3.8, we showed that the determination of the value of being in each state can be found by solving the following linear program

\[ \min_v \sum_{s \in S} \beta_s v(s) \] (10.28)

subject to

\[ v(s) \geq C(s, x) + \gamma \sum_{s' \in S} p(s'|s, x) v(s') \quad \text{for all } s \text{ and } x. \] (10.29)

The problem with this formulation arises because it requires that we enumerate the state space to create the value function vector \((v(s))_{s \in S}\). Furthermore, we have a constraint for each state-action pair, a set that will be huge even for relatively small problems.

We can partially solve this problem by replacing the discrete value function with a regression function such as

\[ \nabla(s|\theta) = \sum_{f \in F} \theta_f \phi_f(s), \]

where \((\phi_f)_{f \in F}\) is an appropriately designed set of basis functions. This produces a revised linear programming formulation

\[ \min_{\theta} \sum_{s \in S} \beta_s \sum_{f \in F} \theta_f \phi_f(s) \]

subject to:

\[ v(s) \geq C(s, x) + \gamma \sum_{s' \in S} p(s'|s, x) \sum_{f \in F} \theta_f \phi_f(s') \quad \text{for all } s \text{ and } x. \]

This is still a linear program, but now the decision variables are \((\theta_f)_{f \in F}\) instead of \((v(s))_{s \in S}\). Note that rather than use a stochastic iterative algorithm, we obtain \(\theta\) directly by solving the linear program.

We still have a problem with a huge number of constraints. Since we no longer have to determine \(|S|\) decision variables (in (10.28)-(10.29) the parameter vector \((v(s))_{s \in S}\) represents our decision variables), it is not surprising that we do not actually need all the constraints. One strategy that has been proposed is to simply choose a random sample of
states and actions. Given a state space $S$ and set of actions (decisions) $\mathcal{X}$, we can randomly choose states and actions to create a smaller set of constraints.

Some care needs to be exercised when generating this sample. In particular, it is important to generate states roughly in proportion to the probability that they will actually be visited. Then, for each state that is generated, we need to randomly sample one or more actions. The best strategy for doing this is going to be problem-dependent.

This technique has been applied to the problem of managing a network of queues. Figure 10.12 shows a queueing network with three servers and eight queues. A server can serve only one queue at a time. For example, server $A$ might be a machine that paints components one of three colors (say, red, green, and blue). It is best to paint a series of parts red before switching over to blue. There are customers arriving exogenously (denoted by the arrival rates $\lambda_1$ and $\lambda_2$). Other customers arrive from other queues (for example, departures from queue 1 become arrivals to queue 2). The problem is to determine which queue a server should handle after each service completion.

If we assume that customers arrive according to a Poisson process and that all servers have negative exponential service times (which means that all processes are memoryless), then the state of the system is given by

$$S_t = R_t = (R_{ti})_{i=1}^8,$$

where $R_{ti}$ is the number of customers in queue $i$. Let $\mathcal{K} = \{1, 2, 3\}$ be our set of servers, and let $a_t$ be the attribute vector of a server given by $a_t = (k, q_t)$, where $k$ is the identity of the server and $q_t$ is the queue being served at time $t$. Each server can only serve a subset of queues (as shown in figure 10.12). Let $\mathcal{D} = \{1, 2, \ldots, 8\}$ represent a decision to serve a particular queue, and let $\mathcal{D}_a$ be the decisions that can be used for a server with attribute $a$. Finally, let $x_{tad} = 1$ if we decide to assign a server with attribute $a$ to serve queue $d \in \mathcal{D}_a$.

The state space is effectively infinite (that is, too large to enumerate). But we can still sample states at random. Research has shown that it is important to sample states roughly in proportion to the probability they are visited. We do not know the probability a state will be visited, but it is known that the probability of having a queue with $r$ customers (when there are Poisson arrivals and negative exponential servers) follows a geometric distribution. For
this reason, it was chosen to sample a state with \( r = \sum_{i} R_{ti} \) customers with probability \( (1 - \gamma)\gamma^r \), where \( \gamma \) is a discount factor (a value of 0.95 was used).

Further complicating this problem class is that we also have to sample actions. Let \( X \) be the set of all feasible values of the decision vector \( x \). The number of possible decisions for each server is equal to the number of queues it serves, so the total number of values for the vector \( x \) is \( 3 \times 2 \times 3 = 18 \). In the experiments for this illustration, only 5,000 states were sampled (in portion to \( (1 - \gamma)\gamma^r \)), but all the actions were sampled for each state, producing 90,000 constraints.

Once the value function is approximated, it is possible to simulate the policy produced by this value function approximation. The results were compared against two myopic policies: serving the longest queue, and first-in, first-out (that is, serve the customer who had arrived first). The costs produced by each policy are given in table 10.3, showing that the ADP-based strategy significantly outperforms these other policies.

Considerably more numerical work is needed to test this strategy on more realistic systems. For example, for systems that do not exhibit Poisson arrivals or negative exponential service times, it is still possible that sampling states based on geometric distributions may work quite well. More problematic is the rapid growth in the feasible region \( X \) as the number of servers, and queues per server, increases.

An alternative to using constraint sampling is an advanced technique known as column generation. Instead of generating a full linear program which enumerates all decisions (that is, \( v(s) \) for each state), and all constraints (equation (10.29)), it is possible to generate sequences of larger and larger linear programs, adding rows (constraints) and columns (decisions) as needed. These techniques are beyond the scope of our presentation, but readers need to be aware of the range of techniques available for this problem class.

### 10.9 APPROXIMATE POLICY ITERATION USING KERNEL REGRESSION*

We build on the foundation provided in section 9.8 that describes the use of kernel regression in the context of least squares temporal difference (LSTD) learning. As we have done earlier, we let the one-period contribution be given by

\[
\hat{C}^m = C(S^{n,m}, a^{n,m}, W^{m+1}).
\]

Let \( S^{n,i}, i = 1, \ldots, m \) be the sample-path of post-decision states produced by following a policy. Let \( k(S^{n,i}, S^{n,j}) \) be the normalized kernel function given by

\[
k(S^{n,i}, S^{n,j}) = \frac{K_h(S^{n,i}, S^{n,j})}{\sum_{i=0}^{m-1} K_h(S^{n,i}, S^{n,j})}.
\]
which means that $\sum_{i=0}^{m-1} k(S^{a,i}, S^{a,j}) = 1$. Then, let $P^{\pi,m}$ be a $M \times M$ matrix where the $(i,j)$th entry is given by

$$P^{\pi,m}_{i,j} = k(S^{a,i-1}, S^{a,j}).$$

By construction, $P^{\pi,m}$ is a stochastic matrix (its rows sum to 1), which means that $I - \gamma P^{\pi,m}$ is invertible.

Define the kernel-based approximation of Bellman’s operator for a fixed policy $\hat{M}^{\pi,m}$ from the sample path of post-decision states $S^{a,0}, \ldots, S^{a,m+1}$ using

$$\hat{M}^{\pi,m}V(s) = \sum_{i=0}^{m-1} k(S^{a,i}, s)(\hat{C}i + \gamma V(S^{M,a}(S^{i}, a^i, W^{i+1}))).$$

We would like to find the fixed point of the kernel-based Bellman equation defined by

$$\hat{V}^{\pi} = \hat{M}^{\pi,m}\hat{V}^{\pi} = P^{\pi}[c^{\pi} + \gamma \hat{V}^{\pi}] = [I - \gamma P^{\pi}]^{-1} c^{\pi}.$$ 

We can avoid the matrix inversion by using a value iteration approximation

$$\hat{V}^{\pi,k+1} = P^{\pi}(c^{\pi} + \gamma \hat{V}^{\pi,k}).$$

The vector $\hat{V}^{\pi}$ has an element $\hat{V}^{\pi}(S^{a,i})$ for each of the (post-decision) states $S^{a,i}$ that we have visited. We then extrapolate from this vector of calculated values for the states we have visited, giving us the continuous function

$$V^{\pi}(s) = \sum_{i=0}^{m-1} k(S^{a,i}, s)\left(\hat{C}i + \gamma \hat{V}^{\pi}(S^{a,i+1})\right).$$

This approximation forms the basis of our approximate policy iteration. The full algorithm is given in figure 10.13.

**10.10FINITE HORIZON APPROXIMATIONS FOR STEADY-STATE APPLICATIONS**

It is easy to assume that if we have a problem with stationary data (that is, all random information is coming from a distribution that is not changing over time), then we can solve the problem as an infinite horizon problem, and use the resulting value function to produce a policy that tells us what to do in any state. If we can, in fact, find the optimal value function for every state, this is true.

There are many applications of infinite horizon models to answer policy questions. Do we have enough doctors? What if we increase the buffer space for holding customers in a queue? What is the impact of lowering transaction costs on the amount of money a mutual fund holds in cash? What happens if a car rental company changes the rules allowing rental offices to give customers a better car if they run out of the type of car that a customer reserved? These are all dynamic programs controlled by a constraint (the size of a buffer or the number of doctors), a parameter (the transaction cost), or the rules governing the physics of the problem (the ability to substitute cars). We may be interested in understanding the
Step 0. Initialization:

Step 0a. Initialize the policy $A^n(s)$.

Step 0b. Choose the kernel function $K_h(s, s')$.

Step 0b. Set $n = 1$.

Step 1. Do for $n = 1, \ldots, N$.

Step 2. Choose an initial state $S^n_0$.

Step 3. Do for $m = 0, 1, \ldots, M$:

Step 4: Let $a^{n,m} = A^n(S^{n,m})$.

Step 5: Sample $W^{m+1}$.

Step 6: Compute the post-decision state $S^m_{a,m} = S^{M,a}(S^n_0, a^{n,m}, W^{m+1})$ and the next state $S^{m+1} = S^M(S^n_0, a^{n,m}, W^{m+1})$.

Step 7: Let $C^n(m)$ be a vector of dimensionality $M$ with element $C^n(m) = C(S^n_0, a^{n,m}, W^{m+1})$, $m = 1, \ldots, M$.

Step 8: Let $P^n(m) = (I - \gamma P^n(m))^{-1}C^n$, where $P^n(m)$ is an $m$-dimensional vector with $i$th element $P^n(m)_{ij}$ for $i = 1, \ldots, m$. This can be approximated using value iteration.

Step 9: Let $V^n(s) = \sum_{m=0}^{M-1} K_h(s, S^n_0, a^{n,m}, W^{m+1})$ be our kernel-based value function approximation.

Step 10: Update the policy:

$$A^{n+1}(s) = \arg \max_a (C^n(s, a) + \gamma V^n(S^{M,a}(s, a)))$$

Step 11. Return the $A^n(N)(s)$ and parameter $\theta^N$.

Figure 10.13 Approximate policy iteration using least squares temporal differencing and kernel regression.

Behavior of such a system as these variables are adjusted. For infinite horizon problems that are too complex to solve exactly, ADP offers a way to approximate these solutions.

Infinite horizon models also have applications in operational settings. Assume that we have a problem governed by stationary processes. We could solve the steady-state version of the problem, and use the resulting value function to define a policy that would work from any starting state. This works if we have, in fact, found at least a close approximation of the optimal value function for any starting state. However, if you have made it this far in this book, then that means you are interested in working on problems where the optimal value function cannot be found for all states. Typically, we are forced to approximate the value function, and it is always the case that we do the best job of fitting the value function around states that we visit most of the time.

When we are working in an operational setting, then we start with some known initial state $S_0$. From this state, there are a range of “good” decisions, followed by random information, that will take us to a set of states $S_1$ that is typically heavily influenced by our starting state. Figure 10.14 illustrates the phenomenon. Assume that our true, steady-state value function approximation looks like the sine function. At time $t = 1$, the probability distribution of the state $S_t$ that we can reach is shown as the shaded area. Assume that we have chosen to fit a quadratic function of the value function, using observations of $S_t$ that we generate through Monte Carlo sampling. We might obtain the dotted curve labeled...
Figure 10.14  Exact value function (sine curve) and value function approximations for $t = 1, 2, 3$, which change with the probability distribution of the states that we can reach from $S_0$.

as $\overline{V}_1(S_1)$, which closely fits the true value function around the states $S_1$ that we have observed.

For times $t = 2$ and $t = 3$, the distribution of states $S_2$ and $S_3$ that we actually observe grows wider and wider. As a result, the best fit of a quadratic function spreads as well. So, even though we have a steady-state problem, the best value function approximation depends on the initial state $S_0$ and how many time periods into the future that we are projecting. Such problems are best modeled as finite horizon problems, but only because we are forced to approximate the problem.

10.11 BIBLIOGRAPHIC NOTES

Section 10.2 - Approximate value iteration using lookup tables encompasses the family of algorithms that depend on an approximation of the value of a future state to estimate the value of being in a state now, which includes $Q$-learning and temporal-difference learning. These methods represent the foundation of approximate dynamic programming and reinforcement learning, and we already saw an introduction to lookup table versions of these algorithms in chapter 4.

Section 10.4 - The problems with the use of linear models in the context of approximate value iteration (TD learning) are well known in the research literature. Good discussions of these issues are found in Bertsekas & Tsitsiklis (1996), Tsitsiklis et al. (1997), Baird (1995) and Precup et al. (2001), to name a few. Munos & Szepesvari
(2008) proves convergence for an approximate value iteration algorithm using a linear model, but imposes technical conditions such as requiring that the algorithm start with a policy that is close to the optimal policy. This paper provides insights into the types of restrictions that may be required for approximate value iteration to work.

Section 10.5 - Bradtke & Barto (1996) first introduced least squares temporal differencing, which is a way of approximating the one-period contribution using a linear model, and then projecting the infinite horizon performance. Lagoudakis & Parr (2003) describes the least squares policy iteration algorithm (LSPI) which uses a linear model to approximate the Q-factors, which is then imbedded in a model-free algorithm.

Section 10.6 - There is a long history of referring to policies as “actors” and value functions as “critics” (see, for example, Barto et al. (1983), Williams & Baird (1990), Bertsekas & Tsitsiklis (1996) and Sutton & Barto (1998)). Borkar & Konda (1997) and Konda & Borkar (1999) analyze actor-critic algorithms as an updating process with two time-scales, one for the inner iteration to evaluate a policy, and one for the outer iteration where the policy is updated. Konda & Tsitsiklis (2003) discusses actor-critic algorithms using linear models to represent both the actor and the critic, using bootstrapping for the critic. Bhatnagar et al. (2009) suggest several new variations of actor-critic algorithms, and proves convergence when both the actor and the critic use bootstrapping.

Section 10.7 - Policy gradient methods have received considerable attention in the reinforcement learning community. The material in this section is based on Konda & Tsitsiklis (2003), but we have provided only a streamlined presentation, and we urge readers to consult the original article before attempting to implement the equations given in this section. One of the earliest policy-gradient algorithms is given in Williams (1992). Marbach & Tsitsiklis (2001) provides gradient-based algorithms for optimizing Markov reward processes, which is a mathematically equivalent problem. Sutton et al. (2000) provides a version of a policy-gradient algorithm, but in a form which is difficult to compute. Sutton (1983) compares several policy gradient algorithms. Szepesvari (2010) provides a recent summary of policy gradient algorithms.

Section 10.8 - Schweitzer & Seidmann (1985) describes the use of basis functions in the context of the linear programming method. The idea is further developed in Farias & van Roy (2003) which also develops performance guarantees. Farias & Roy (2001) investigates the use of constraint sampling and proves results on the number of samples that are needed.

Section 10.9 - This material is based on Ma & Powell (2010).

Exercises - Exercise 10.1 is due to de Farias & Van Roy (2000).

PROBLEMS

10.1 Consider a Markov chain with two states, 1 and 2 and just one policy. There is a reward from being in state 1 of $1, and if in state 1 there is a probability 0.2 of staying in state 1, and a probability 0.8 of transitioning to state 2. The reward for being in state 2 is $2 with a probability 0.2 of transitioning to state 1, and a probability of 0.8 of staying in state 2. Let $c^n$ be the column vector of rewards for being in each state, and let $P$ be the...
one-step transition matrix. Let the basis functions be given by $\phi(1) = 1$ and $\phi(2) = 2$, and let $\Phi = [1, 2]^T$. Let $\Pi$ be the projection operator (derived earlier). If we represent the value function using $\Phi \theta$, then Bellman’s equation would be written

$$
\Phi \theta = \Pi M^\pi \Phi \theta = \Pi (c^\pi + \gamma P \Phi \theta),
$$

where $\gamma = 5/5.4$.

a) First using value iteration and generate 5 iterations of

$$
v^{n+1} = c^\pi + \gamma P v^n,
$$

starting with $v^0 = 0$. What can you say about the limiting performance of $v^n$ if we were to repeat this indefinitely?

b) Now consider what happens when we use value iteration with basis functions, where

$$
\theta = (\Phi^T \Phi)^{-1} \Phi^T (c^\pi + \gamma P \Phi \theta).
$$

Use this to show that value iteration for this problem would reduce to $\theta^{n+1} = 1 + \theta^n$. What can you say about the limiting behavior of approximate value iteration using basis functions?