REINFORCEMENT LEARNING AND STOCHASTIC OPTIMIZATION
A unified framework for sequential decisions

Warren B. Powell

June 30, 2021
We are now ready to tackle the problem of searching for good policies while simultaneously trying to produce good value function approximations. The guiding principle in this chapter is that we can find good policies if we can find good value function approximations. The problem is that finding good value function approximations requires that we be simulating “good” policies (using the methods of chapter 16). It is the interaction between the two that creates all the complications.

The algorithmic strategies presented in this chapter are all based on algorithms we first presented in chapter 14, with two notable exceptions:

- We never take expectations - Random variables are always handled through either Monte Carlo simulation, historical trajectories, or direct field observations.

- We use machine learning to approximate functions - This means we have to deal with estimation errors due to noise, errors due to biased observations, and structural errors from the chosen approximating architecture.

The statistical tools presented in chapter 3 focused on finding the best statistical fit of a function that we can only observe with noise, but where we assumed that the observations are unbiased. In chapter 16, we saw that the sampled estimate $\hat{V}_n^p$ of the value of being in state $S_t^p$ could be biased for several reasons:

- If we are using approximate value iteration, the value functions have to steadily accumulate downstream values (recall the slow convergence illustrated in table 16.1).
• The sampled $\hat{v}_n^t$ might depend on downstream value function approximations, which might produce structural biases (e.g. if we use a linear approximation of a nonlinear function).

• $\hat{v}_n^t$ depends on the policies that are being used to make decisions in the future which in turn depend on value function approximations which are a) incorrect and b) changing over the iterations.

In all three cases, our observations of $\hat{v}_n^t$ are biased, but in a way that is also changing over iterations as we search for better policies.

When we write our generic optimization problem

$$\max_{\pi} \mathbb{E} \left\{ \sum_{t=0}^{T} \gamma^t C(S_t, X_{\pi}^t(S_t)) | S_0 \right\},$$  \hspace{1cm} (17.1)

the maximization over policies can mean choosing one of the approximation strategies for $V(S_t)$ from chapter 3, and choosing the parameters that control the approximation. A useful way to express this search is to let $f \in \mathcal{F}$ be the set of architectures (functions), and let $\theta \in \Theta_f$ be any tunable parameters for functions in class $f$, which means our policy $\pi$ is an element of $(f \in \mathcal{F}, \theta \in \Theta_f)$. Our search over policies is then the same as

$$\max_{\pi=(f \in \mathcal{F}, \theta \in \Theta_f)} \mathbb{E} \left\{ \sum_{t=0}^{T} \gamma^t C(S_t, X_{\pi}^t(S_t)) | S_0 \right\}.$$  \hspace{1cm} (17.1)

For example, we might be choosing between a myopic policy, or perhaps a simple linear architecture with one basis function

$$V_t(S_t) = \theta_0 + \theta_1 S_t,$$  \hspace{1cm} (17.2)

or perhaps a linear architecture with two basis functions,

$$V_t(S_t) = \theta_0 + \theta_1 S_t + \theta_2 S^2.$$  \hspace{1cm} (17.3)

We might even use a nonlinear architecture such as

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

We can try estimating value functions with each of these architectures (which still requires searching for $\theta$ for each function class), and then compare the performance of the resulting policies using the objective function in (17.1), which is how we would actually perform the search over function classes (admittedly this is ad hoc).

We begin our presentation with an overview of the basic algorithmic strategies that we cover in this chapter.

### 17.1 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 14. We continue to adapt our algorithms to finite and infinite horizons.
Basic value iteration for finite horizon problems work by solving

$$V_t(S_t) = \max_{x_t} \left( C(S_t, x_t) + \gamma \mathbb{E}\{V_{t+1}(S_{t+1})|S_t, x_t\} \right).$$

Equation (17.4) 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 (using the post-decision state)

$$\hat{v}_t^n = \max_{x_t} \left( C(S^n_t, x_t) + \gamma \hat{V}^{x,n-1}_{t+1}(S^{M,x}(S^n_t, x_t)) \right).$$

Here, we have formed the value function approximation around the post-decision state. We execute the equations by stepping forward in time which creates a natural state sampling procedure known in the reinforcement literature as trajectory following. If $x^n_t$ is the decision that optimizes (17.5), then we compute our next state using $S^n_{t+1} = S^{M}(S^n_t, x^n_t, W^n_{t+1})$ where $W^n_{t+1}$ 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

$$V^n(S) = \max_x \left( C(S, x) + \gamma \mathbb{E}\{V^{n-1}(S')|S\} \right).$$

Again, equation (17.6) 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

$$\hat{v}^n = \max_{x} \left( C(S^n, x) + \gamma \hat{V}^{x,n-1}(S^{M,x}(S^n, x^n)) \right).$$

We then use the observed state-value pair $(S^n, \hat{v}^n)$ to update the value function approximation using whatever architecture we have chosen.

Using $\hat{v}^n$ to update the value function approximation can introduce a significant level of noise, that is then translated to the behavior of the policy producing unpredictable effects (this is well known to experimentalists in the ADP community). One strategy for mitigating this noise is to imbed a policy approximation loop within an outer loop where policies are updated. Assume we fix our policy using

$$X^{\pi,n}(S) = \arg \max_{x \in \mathcal{X}} \left( C(S, x) + \gamma \hat{V}^{x,n-1}(S^{M,x}(S, x)) \right),$$

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

$$\hat{v}^{n,m} = \max_{x \in \mathcal{X}} \left( C(S^{n,m}, x) + \gamma \hat{V}^{x,n-1}(S^{M,x}(S^{n,m}, x)) \right)$$

where $S^{n+1,m} = S^{M}(S^{n,m}, x^{n,m}, W^{n+1,m})$. Note that the value function $\hat{V}^{x,n-1}(s)$ 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 $\hat{V}^{x,n-1}(s)$ to obtain $\hat{V}^{x,n}(s)$. 
Typically, $\overline{V}^{x,n}(s)$ does not depend on $\overline{V}^{x,n-1}(s)$, other than to influence the calculation of $\hat{\nu}^n,m$. If $M$ is large enough, $\overline{V}^{x,n}(s)$ will represent an accurate approximation of the value of being in state $s$ while following the policy in equation (17.8). 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.

Repeated evaluations of a policy helps reduce the noise, but does not eliminate the errors in the approximation itself, possibly due to the choice of architecture, or possibly due to the reality that our observations $\hat{\nu}^n$ are based on approximations which means that our policy is suboptimal, biasing the estimates $\hat{\nu}^n$. In other words, there is a lot going on that distorts the trajectory of the algorithm.

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 (by using the updated value function approximation). 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-decision 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-decision pairs.
  - AVI for the post-decision state - Approximate value iteration where value function approximations are approximated around the post-decision state.

- Parametric architectures - We summarize some of the extensive literature which depends on linear models (basis functions), and touch on nonlinear models.

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 strategy continues to attract attention because of its simplicity.

- API using nonparametric models - Nonparametric models offer significantly greater flexibility, but the price is that they are less stable (they can respond much more quickly to random variations) and require considerably more observations.

The linear programming method - The linear programming method, first introduced in chapter 14, can be adapted to exploit value function approximations.
Arguably the most natural and elementary approach for approximate dynamic programming uses approximate value iteration. In this section we explore the following topics related to this important algorithmic strategy:

- Value iteration using a pre-decision state variable.
- On-policy, off-policy and the exploration-exploitation problem.
- Q-learning.
- Value iteration using a post-decision state variable.
- Value iteration using a backward pass.

### 17.2.1 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 $S^n_t$

$$v^n_t = \max_{x_t} \left( C(S^n_t, x_t) + \gamma \mathbb{E}\{V_{t+1}(S^n_{t+1})|S^n_t}\right),$$

(17.9)

where $S_{t+1} = S^M(S^n_t, x_t, W^n_{t+1})$, 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 $W^n_{t+1} = W^n_{t+1}(\omega^n)$. After computing $\hat{v}^n_t$, we update the value function using the standard equation

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

(17.10)

If we sample states at random (rather than following the trajectory) and repeat equations (17.9) and (17.10), we will eventually converge to the correct value of being in each state. 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, $W^n_{t+1}$ 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

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

(17.11)

Either way, using a lookup table representation we can update the value of being in state $S^n_t$ using

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

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
Step 0. Initialization:

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

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:

$$
\hat{v}_t^n = \max_{a_t} \left( C_t(S_t^{n-1}, x_t) + \gamma \sum_{\omega \in \hat{\Omega}_n} p^n(\omega) \overline{V}_{t+1}^{n-1}(S_t^{n-1}, x_t, W_{t+1}(\omega)) \right)
$$

and let $x_t^n$ be the value of $x_t$ that solves the maximization problem.

Step 2b: Compute:

$$
S_{t+1}^n = S^M(S_t^n, x_t^n, W_{t+1}(\omega^n)).
$$

Step 2c. Update the value function:

$$
\overline{V}_t^n \leftarrow U^{V}(\overline{V}_t^{n-1}, S_t^n, \hat{v}_t^n).
$$

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

Step 4. Return the value functions $(\overline{V}_t^n)_{t=1}^T$.

---

**Figure 17.1** Approximate dynamic programming using a pre-decision state variable.

sample realization (as we did with the post-decision formulation). An outline of the overall algorithm is given in figure 17.1.

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.

### 17.2.2 On-policy, off-policy and the exploration-exploitation problem

The algorithm in figure 17.1 uses a kind of default logic for determining the next state to visit. Specifically, we solve the optimization problem in equation (17.11) 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 a decision $x_t^n$. Then, in Step 2b of the algorithm, we use this decision to help determine the next state to visit using the transition function

$$
S_{t+1}^n = S^M(S_t^n, x_t^n, W_{t+1}(\omega^n)).
$$

Using the decision $x_t^n$, which is the decision determined by the policy we are trying to optimize, means that we are using a concept known as *trajectory following*. We refer to the policy that determines the decision we would like to take as the *implementation policy* (see section 9.11), although it is known in the reinforcement learning community as the *target policy*. When we are optimizing policies, what we are doing is trying to improve the implementation policy. The policy we follow to learn the value functions (to improve the implementation policy) is called the *learning policy*, a term we first introduced in section 9.11.

We can encounter problems if we use the implementation policy to determine the next state to visit. Consider the two-stage dynamic program illustrated in figure 17.2. 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, so 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
decisions at random. For example, we may flip a coin and choose a decision with probability
$\epsilon$, or choose the decision $x^n_t$ determined by the implementation policy with probability $1 - \epsilon$.

This policy is known in the literature as epsilon greedy.

If the implementation policy also determines the next state we visit, then we say that the
algorithm is on policy. If the learning policy is different than the implementation 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.

Exploration policies depend heavily on how we are approximating the value function. In
this simple example, we are using a lookup table for the value function (with independent
beliefs), which means observing the value of being in state 1 does not tell us anything
about the value of being in state 2. The situation changes dramatically if we are using a
parametric value function, since an observation for any state updates the approximation for
all states.

In the remainder of this chapter, we are going to make a distinction between on-policy
and off-policy algorithms.

17.2.3 Q-learning

One of the earliest and most widely studied algorithms in the reinforcement learning
literature is known as Q-learning. The name is derived simply from the notation used in
the algorithm, and appears to have initiated the tradition of naming algorithms after the
notation.

To motivate Q-learning, return for the moment to the classical way of making decisions
using dynamic programming. Normally we would want to solve

$$x^n_t = \arg \max_{x_t \in A^n_t} \left( C_t(S^n_t, x_t) + \gamma \mathbb{E} \left\{ V^n_{t+1}(S_{t+1}^{n+1}(S^n_t, x_t, W_{t+1})) \mid S^n_t, x_t \right\} \right) . \quad (17.12)$$

Solving equation (17.12) can be problematic for two different reasons. The first is that
we may not be able to compute the expectation because it is computationally too complex
(the second curse of dimensionality). The second is that we may simply not have the
information we need to compute the expectation. This might happen if a) we do not know
the probability distribution of the random information or b) we may not know the transition function. In either of these cases, we say that we do not “know the model” and we need to use a “model-free” formulation.

When we can compute the expectation, which means we have the transition function and we know the probability distribution, then we are using what is known as a “model-based” formulation. Many authors equate “model-based” with knowing the one-step transition function. In either of these cases, we say that we do not “know the model” and we need to use a “model-free” formulation.

Earlier, we circumvented this problem by approximating the expectation by using a subset of outcomes (see equation (17.11)), but this can be computationally clumsy for exogenous information is multidimensional.

The problem is that this means we are choosing \( x_t \) for a particular realization of the future information \( W_{t+1}(\omega^n) \). If we use the same sample realization of \( W_{t+1}(\omega^n) \) to make the decision that will actually happen (when we step forward in time), then this is what is known as cheating (peeking into the future), which can seriously distort the behavior of the system. If we use a single sample realization for \( W_{t+1}(\omega) \) that is different than the one we use when we simulate forward, then this is simply unlikely to produce good results (imagine computing averages based on a single observation).

What if we instead choose the decision \( x_t^n \) first, then observe \( W_{t+1}^n(\omega) \) (so we are not using this information when we choose our decision) and then compute the cost? Let the resulting cost be computed using

\[
\tilde{q}_t^n(S_t, x_t) = C(S_t, x_t) + \gamma \tilde{V}_{t+1}^{n-1}(S_{t+1}^n, x_{t+1}, W_{t+1}(\omega^n)).
\] (17.14)

We could now smooth these values to obtain

\[
\tilde{Q}_t^n(S_t, x_t) = (1 - \alpha_n - 1)\tilde{q}_t^n(S_t^n, x_t^n) + \alpha_n - 1 \tilde{q}_t^n(S_t, x_t).
\]

Not surprisingly, we can compute the value of being in a state from the \( Q \)-factors using

\[
\tilde{V}_t^n(S_t) = \max_x \tilde{Q}_t^n(S_t, x).
\] (17.15)

If we combine (17.15) and (17.14), we obtain

\[
\tilde{q}_t^n = C(S_t, x_t) + \gamma \max_{x_{t+1}} \tilde{Q}_t^{n-1}(S_{t+1}, x_{t+1}),
\]

where \( S_{t+1} = S_t(S_t, x_t, W_{t+1}(\omega^n)) \) is the next state resulting from the decision \( x_t \) and the sampled information \( W_{t+1}(\omega^n) \).

The functions \( Q_t(S_t, x_t) \) are known as \( Q \)-factors and they capture the value of being in a state and taking a particular decision. Recall from section 9.4.5 that a state-decision pair \( (S_t, x_t) \) is a form of post-decision state, although it is typically the least-compact form for representing a post-decision state.

We can now choose a decision by solving

\[
x_t^n = \arg \max_{x_t \in X_t^n} \tilde{Q}_t^{n-1}(S_t^n, x_t).
\] (17.16)
Step 0. Initialization:

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

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 decision using $\epsilon$-greedy. With probability $\epsilon$, choose a decision $x^n_t$ at random from $X_t$. With probability $1 - \epsilon$, choose $x^n_t$ using $x^n_t = \arg \max_{x_t \in X_t} \hat{Q}^{n-1}_t(S^n_t, x_t)$.

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

Step 2c. Compute $\hat{q}^n_t = C(S^n_t, x^n_t) + \gamma \max_{x_{t+1} \in X_{t+1}} Q^{n-1}_t(S^n_{t+1}, x_{t+1})$.

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

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

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

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

---

Figure 17.3 A Q-learning algorithm.

This strategy is known as Q-learning.

The complete algorithm is summarized in figure 17.3.

A variation of Q-learning is known as “Sarsa” which stands for “state, action, reward, state, action” (the computer science community has a culture of naming its algorithms around its notation). Imagine that we start in a state $s$ and make decision $x$. After this, we observe a reward $r$ and the next state $s'$. Finally, use some policy to choose the next decision $x'$. This sequence forms “sarsa.” A common strategy, given the initial state $s$, is to choose the decision $x$ and the decision $x'$ (given the state $s'$) using an epsilon-greedy strategy. When $x$ and $x'$ are chosen using the same policy, the algorithm is referred to as on-policy. Alternatively, we may choose the initial decision $x$ at random, but then choose $x'$ greedily, which is to say

$$x'_{t+1} = \arg \max_{x'} \hat{Q}^{n-1}_{t+1}(S', x').$$

If the policy used to choose $x$ is different than the policy used to choose $x'$, then we say that the algorithm is off-policy.

Many authors describe Q-learning as a technique for “model-free” dynamic programming, where we either do not know the transition function or the probability law of the exogenous information. The key idea is that we can choose a decision using (17.16) without needing to directly approximate the future in any way. After choosing a decision $x^n$ (or $x^n_t$), we can then simply observe the next state we transition to, without an explicit model of how we got there.
Step 0. Initialization:

Step 0a. Initialize an approximation for the value function $V^0_0(S^n_t)$ for all post-decision states $S^n_t$, $t = \{0, 1, \ldots, T\}$.

Step 0b. Set $n = 1$.

Step 0c. Initialize $S^n_0$.

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

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

Step 2a. Determine the decision using $\epsilon$-greedy. With probability $\epsilon$, choose a decision $x^n_t$ at random from $X$. With probability $1 - \epsilon$, choose $a^n_t$ using

$$
\hat{v}^n_t = \arg\max_{x_t \in X_t} \left(C(S^n_t, x_t) + \gamma V^{n-1}_{t-1}(S^{M,x}_{n-1}(S^n_t, x_t))\right).
$$

Let $x^n_t$ be the decision that solves the maximization problem.

Step 2b. Update $V^{n-1}_{t-1}$ using:

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

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

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

Step 4. Return the value functions $(V^n_t)_{t=0}^T$.

Figure 17.4 Approximate value iteration for finite horizon problems using the post-decision state variable.

17.2.4 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 decisions (and estimate the value of being in state $S^n_t$) using

$$
\hat{v}^n_t = \arg\max_{x_t \in X_t} \left(C(S^n_t, x_t) + \gamma V^{n-1}_{t}(S^{M,x}_{n-1}(S^n_t, x_t))\right).
$$

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 $V^{n-1}(S^n_t)$, we use $\hat{v}^n_t$ to update a post-decision value function approximation around the previous post-decision state $S^{x,n}_{t-1}$. This is done using

$$
\hat{v}^n_t = \arg\max_{x_t \in X_t} \left(C(S^n_t, x_t) + \gamma V^{n-1}_{t}(S^{M,x}_{n-1}(S^n_t, x_t))\right).
$$

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.

A complete summary of the algorithm is given in figure 17.4.

$Q$-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 a post-decision state given by the state/decision pair \((S, x)\) (we first introduced this form of post-decision state in section 9.4.5). We then have to learn the value of being in \((S, x)\), 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 \(V^n(S, x)\) around the post-decision state \(S^x = S^{M, x}(S, x)\), we can create Q-factors directly from the contribution function and the post-decision value function using

\[
\bar{Q}^n(S, x) = C(S, x) + \gamma V^n_t (S^{M, x}(S, x)).
\]

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 \(V^n(S^x)\) is much easier than estimating \(\bar{Q}(S, x)\).

### 17.2.5 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 17.5.

In this algorithm, we step forward through time creating a trajectory of states, decisions, 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 \(\bar{V}^{n-1}\). Unlike our forward-pass algorithm (where \(\hat{v}^n_t\) depends on the approximation \(V^{n-1}_t(S^x_t)\)), \(\hat{v}^n_t\) is a valid, unbiased estimate of the value of being in state \(S^n_t\) at time \(t\) and following the policy produced by \(\bar{V}^{n-1}\).

We introduce an inner loop so that rather than updating the value function approximation with a single \(\hat{v}^n_0\), we average across a set of samples to create a more stable estimate, \(\bar{v}^n_0\).

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 \(S^x_t = R^x_t = R_t - x_t\) which indicates whether we are holding the asset or not. Further, \(S_{t+1} = S^x_t\) since the resource transition function is deterministic.
**Step 0.** Initialization:

**Step 0a.** Initialize $V_0^{t}$, $t \in T$.

**Step 0b.** Initialize $S_0^n$.

**Step 0c.** Choose an initial policy $X_{\pi,0}$.

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

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

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

**Step 2a:** Find

$$x_t^n = X_{1,n-1}^{n-1}(S_t^n)$$

**Step 2b:** Update the state variable

$$S_{t+1}^n = S^M(S_t^n, x_t^n, W_t+1(\omega^n)).$$

**Step 3:** Set $\hat{v}_{t+1}^n = 0$ and do for $t = T, T-1, \ldots, 1$:

**Step 3a:** Update $\hat{v}_t^n$ using

$$\hat{v}_t^n = C(S_t^n, x_t^n) + \gamma \hat{v}_{t+1}^n.$$  

**Step 3b:** Update the value function approximation $\bar{V}_t^n$ using

$$\bar{V}_t^n \leftarrow U\bar{V}(\bar{V}^{n-1}_t, S_t^n, x_t^n, \hat{v}_t^n).$$

**Step 3c.** Update the policy

$$X_{1,n}^n(S) = \arg \max_{x \in X} \left( C(S_t^n, x) + \gamma \bar{V}_t^n(S^M(x, S_t^n)) \right).$$

**Step 4.** Increment $n$. If $n \leq N$ go to Step 1.

**Step 5.** Return the value functions $(\bar{V}^N_t)_{t=1}^T$.

---

**Figure 17.5** Double-pass version of the approximate dynamic programming algorithm for a finite horizon problem.

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

$$\hat{v}_t^n = \max_{x_t \in (0,1]} \left( \hat{p}_t^n x_t + (1-x_t)(-c_t + \hat{v}_{t-1}^{n-1}) \right). \quad (17.17)$$

Assume that the holding cost $c_t = 2$ for all time periods.

Table 17.1 illustrates three iterations of a single-pass algorithm for a three-period problem. We initialize $\hat{v}_0^t = 0$ for $t = 0, 1, 2, 3$. Our first decision is $x_1$ after we see $\hat{p}_1$. The first column shows the iteration counter, while the second shows the stepsizes $\alpha_{n-1} = 1/n$. For the first iteration, we always choose to sell because $\hat{v}_0^0 = 0$, which means that $\hat{v}_1^1 = \hat{p}_1^1$. Since our stepsizes are 1.0, this produces $\hat{v}_1^1 = \hat{p}_1^1$ for each time period.

In the second iteration, our first decision problem is

$$\hat{v}_1^2 = \max \{ \hat{p}_1^2, -c_1 + \hat{v}_1^1 \}$$

$$= \max \{ 24, -2 + 34 \}$$

$$= 32,$$
iteration and $Q$-learning using lookup tables

\[ t = 0 \quad t = 1 \quad t = 2 \quad t = 3 \]

<table>
<thead>
<tr>
<th>Iteration</th>
<th>$\alpha_{n-1}$</th>
<th>(0)</th>
<th>(0)</th>
<th>(0)</th>
<th>(0)</th>
<th>(0)</th>
<th>(0)</th>
<th>(0)</th>
<th>(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>1</td>
<td>34</td>
<td>34</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.50</td>
<td>31</td>
<td>32</td>
<td>24</td>
<td>0</td>
<td>31.5</td>
<td>29</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0.3</td>
<td>32.3</td>
<td>35</td>
<td>35</td>
<td>1</td>
<td>30.2</td>
<td>27.5</td>
<td>24</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 17.1 Illustration of a single-pass algorithm

which means $x_1^3 = 0$ (since we are holding). We then use $\hat{v}_1^2$ to update $\hat{v}_0^2$ using

\[
\hat{v}_0^2 = (1 - \alpha_1) \hat{v}_0^1 + \alpha_1 \hat{v}_1^1
\]

\[
= (0.5)30.0 + (0.5)32.0 = 31.0.
\]

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 17.2 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 $\hat{v}_2^2$ 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 $\hat{v}_2^2 = -2 + \hat{v}_3^2 = 27$. Similarly, holding the asset at time 1 means $\hat{v}_2^2 = -2 + \hat{v}_2^2 = 25$. The smoothing of $\hat{v}_n^m$ with $\bar{v}_{n-1}^{m-1}$ to produce $\bar{v}_{n-1}^m$ 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

\[
\text{Table 17.2 Illustration of a double-pass algorithm}
\]
Table 17.2  Illustration of a double-pass algorithm

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.

17.3 STYLES OF LEARNING

At this point it is useful to pause and discuss the different styles in which we can use the ideas from section 17.2 and chapter 16. In this section, we contrast three settings in which we might apply these ideas:

- The basic offline learning problem that we have been solving up to now.
- An online learning problem that would arise if we were optimizing a system while it operates in the field.
- An approximate lookahead policy where we apply these methods purely to make a decision \( x_t \) at time \( t \).

17.3.1 Offline learning

The algorithms presented in chapter 16 and section 17.2 are written in the context of running a simulator to approximate the expectation

\[
F^\pi = \mathbb{E}\sum_{t=0}^{T} C(S_t, X^\pi(S_t)),
\]

where, if we are simulating a sample path \( \omega^n \), we would write the results of a single simulation as

\[
\hat{F}^\pi(\omega^n) = \sum_{t=0}^{T} C(S_t(\omega^n), X_t^\pi(S_t(\omega^n))),
\]

where our transitions evolve according to

\[ S_{t+1}(\omega^n) = S^M(S_t(\omega^n), X_t^\pi(S_t(\omega^n)), W_{t+1}(\omega^n)) \]

for a sequence of exogenous inputs \( (W_1(\omega^n), \ldots, W_T(\omega)) \). We have been using this base model (where we use “base model” as it was introduced in chapter 9) with a policy

\[
X_t^\pi(S_t) = \arg\max_x (C(S_t, x) + V^x_{t-1}(S_t^x)),
\]
where \( S^x_t \) is our post-decision state, and \( \hat{V}^{x,n-1}_t(S^x_t) \) is our post-decision value function approximation learned after \( n - 1 \) updates. We may use TD(0), TD(1) or the general TD(\( \lambda \)) updates for using a sampled estimate \( \hat{v}^n_t \) to update \( \hat{V}^{x,n-1}_t(S^x_t) \) to obtain \( \hat{V}^{x,n}_t(S^x_t) \) using any approximation architecture. The ultimate goal is to solve the problem

\[
\max_{\pi} F^\pi
\]

using specific classes of value function approximations (assume we are restricting ourselves to VFA-based policies).

This whole approach assumes we are doing offline learning in a simulator, where we assume we have access to the transition function \( S_{t+1} = S^H(S_t, x_t, W_{t+1}) \) and a way of sampling \( (W_1, \ldots, W_T) \). We use this setting to do repeated training iterations, and it is particularly important when we use TD(\( \lambda \)) for \( \lambda > 0 \) since this requires the backward communication of updates described in section 16.1.4 (see in particular equation (16.13)).

### 17.3.2 From offline to online

Now imagine that we are trying to design our VFA-based policy without a simulator. Instead, we have an actual physical system we are trying to learn from and control. In this setting, we are no longer going to depend on knowing the transition function or observing the exogenous information \( W_t \); instead we are simply going to make a decision \( x_t \) and then observe the next state \( S_{t+1} \) (classic model-free dynamic programming). Although not critical for this discussion, we can assume that decisions are being made with our VFA-based policy that is being updated as we go, but how are these updates happening?

First, it does not make sense to be learning a time-dependent policy \( X^\pi_t(S_t) \) since once we pass time \( t \), we are no longer interested in \( X^\pi_t(S_t) \). So let’s start by assuming that we are going to estimate a stationary policy \( X^\pi(S_t) \) and a stationary value function approximation \( \hat{V}^{x,n}_t(S_t) \). Remember that in our offline setting, \( n \) counted how many times we had simulated our process \( W_1, \ldots, W_T \). We see that in our online setting, \( n = t \) because we update our value function approximation (which we label with \( n \)) once per time period (indexed by \( t \)).

Next, we can certainly apply classical TD(0) updates as we step forward, and this can work perfectly well for some problem classes. If this is the case, then we can step forward from state \( S_t \), execute action \( x_t = X^\pi(S_t) \) using \( \hat{V}^{x,n-1}_t(S_t) \). We then get our updated estimate of the value of being in state \( S_t \) given by \( \hat{v}^n_t \), which we use to update our value function approximation to obtain \( \hat{V}^{x,n}_t(S_t) \).

While TD(0) works very well in some problem classes, there are many problems where TD(\( \lambda \)), possibly using \( \lambda = 1 \), can work much better. If you need any convincing, flip back to table 16.1 and the discussion around those calculations to remind yourself how slow TD(0) can be. So we have to ask, if we transition to online learning, have we lost this powerful algorithmic strategy?

Fortunately, the answer is no, but we have to do some extra work. As we progress forward in time, we need to retain at least some history of states \( S_t \), decisions \( x_t \), states \( S_t \) (or, for our illustration, post-decision states \( S^x_t \)), and contributions \( c_t = C(S_t, x_t) \) for \( t = t - 1, t - 2, \ldots, t - H \). For convenience we compile this sequence into a history that allows us to trace backward in time.

Now recall how we did our TD(\( \lambda \)) updates for our discounted infinite-horizon problem in equation (16.12), but now we are going to first adapt it to an undiscounted, finite-horizon...
setting using
\[ V^n(s) = V^{n-1}(s) + \alpha_n \sum_{m=0}^{H} (\lambda)^m \delta^{n+m}, \]  
(17.20)

where \( \delta^n \) is our usual temporal-difference update
\[ \delta^n = C(s^n, x^n) + V^{n-1}(S^M, x^n) - V^{n-1}(s^n). \]

We are going to execute equation (17.20) adaptively, going backward in time. To make the logic as clear as possible, we are going to assume a lookup table value function, and we are going to start by indexing the value function by the time \( t' \) when we visit state \( S_{t'} \) just so we can keep track of the incremental updating. For this reason, we begin by defining
\[ V_{t', t}(s) = V_{t'}(s) = \text{the starting value of the estimate of } V_{t'}(S_{t'}) \text{ as of time } t', \]
\[ \delta_{t', t}(s) = \text{the partial update of } V_{t'}(s) \text{ that has occurred by time } t \geq t'. \]

Assume that \( V_{t'}(S_{t'}) \) is the approximate value of being in state \( S_{t'} \) when we visited it at time \( t' \). By time \( t > t' \), we would have a partially updated estimate \( V_{t', t}(S_{t'}) \) of the value of being in state \( S_{t'} \) given by
\[ V_{t', t}(S_{t'}) = V_{t'}(S_{t'}) + \alpha_{t'} \sum_{\tau = t'}^{t} \lambda^{t'-\tau} \delta_{\tau}. \]  
(17.21)

This means that our update by time \( t + 1 \) would be
\[ V_{t', t+1}(S_{t'}) = V_{t'}(S_{t'}) + \alpha_{t'} \sum_{\tau = t'}^{t+1} \lambda^{t'-\tau} \delta_{\tau}, \]
\[ = V_{t', t}(S_{t'}) + \lambda^{t+1-t'} \delta_{t+1}. \]  
(17.22)

This means that as we step forward to time \( t + 1 \), we have to run backward through history adding \( \lambda^{t+1-t'} \delta_{t+1} \) to each \( V_{t'}(S_{t'}) \) for \( t' = t, t-1, t-2, \ldots \), until \( \lambda^{t+1-t'} \) is small enough that we can stop.

As a final step, we drop the time index because we are updating a stationary policy.

### 17.3.3 Lookahead policies

Another perspective of approximate dynamic programming is in the context of a lookahead policy. This is an idea that we are going to revisit in more depth in chapter 19 which focuses on lookahead policies, but for completeness we are going to hint at what we would do right now for comparison.

Imagine that we feel that to make a good decision now, we have to plan into the future using our best estimates, say, of forecasts of various activities. We may have a situation such as planning inventories for a complex supply chain where a stationary policy would not work. Also, as we discuss in chapter 19, lookahead policies have the feature of embedding a lot of information in the form of latent variables, which is information that affects the modeling as we project into the future, but without adding to the complexity of the state variable as we project into the future.
This idea requires that we set up an approximate model that is then solved with approximate dynamic programming, using any of the algorithms discussed so far. We end up solving a problem at some time $t$ with the same structure as the one behind (17.18), but it starts at time $t$. Also, because it is in a lookahead model, it can be simpler, so we use modified states, decisions and exogenous information which we introduce in more detail in section 19.2:

$$X_t^\pi(S_t) = \arg\max_{x_t} \tilde{E} \left\{ \sum_{t'=t}^{t+H} C(S_{t''}, \tilde{X}_{t''}^\pi(S_{t''})) \right\}.$$ (17.23)

In other words, our policy will be to solve an approximate lookahead model, and use the decision $x_t = \tilde{X}_t^\pi(S_{tt})$ that looks best right now. We note that this has to be re-optimized (possibly from scratch, but not necessarily) each time period. Also, while this idea is computing value functions to obtain good policies, the primary interest is in the decision of what to do at time $t$.

### 17.4 APPROXIMATE VALUE ITERATION 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 17.6 depicts a basic adaptation of linear models updated using recursive least squares in an approximate value iteration. However, not only are there no convergence proofs for this algorithm, there are examples that show that it may not diverge, even for problems where the linear approximation has the potential for identifying the correct value function. This said, the method is popular because of its relative simplicity, and because

---

**Step 0.** Initialization:

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

**Step 1.** Solve

$$\hat{v}^{n} = \max_{x \in X^n} \left( C(S^n, x) + \gamma \sum_j \theta_j^{n-1} \phi_j(S^M, x(S^n, x)) \right)$$ (17.24)

and let $x^n$ be the value of $x$ that solves (17.24).

**Step 2.** Update the value function recursively using equations (3.41) - (3.45) from chapter 16 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 $S^n = S^M(S^n, x^n, W^{n+1})$.

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

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

---

**Figure 17.6** Approximate value iteration using a linear model.
it seems to work for many applications (recall that we used linear architectures for the benchmarking studies for backward approximate dynamic programming in section 15.5.1 with exceptional results).

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 over the widest range of states. 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 17.6 refers to the classic recursive least squares updating formulas in equations (3.41)-(3.45) in chapter 3. However, buried in these formulas is the implicit use of a stepsize rule of \( 1/n \). We show in chapter 6 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 \( M_n \) and \( \gamma_n \), which are given as

\[
M^n = M^{n-1} - \frac{1}{\gamma_n} (M^{n-1} \phi^n (\phi^n)^T M^{n-1}),
\]

\[
\gamma^n = 1 + (\phi^n)^T M^{n-1} \phi^n,
\]

in equations (3.44) and (3.45) with

\[
M^n = \frac{1}{\lambda} \left( M^{n-1} - \frac{1}{\gamma_n} (M^{n-1} \phi^n (\phi^n)^T M^{n-1}) \right),
\]

\[
\gamma^n = \lambda + (\phi^n)^T M^{n-1} \phi^n,
\]

in equations (3.47) and (3.48). 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 3.8.2, 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. In step 3, we use \( S^n = S^M(S^n, x^n, W(\omega^n)) \) which we are using what we think is our best decision to determine the next state. This strategy is known in the reinforcement learning community as an “on-policy” algorithm, which means that the decision that is used to update the value of being in a state is also used to determine the next state to visit. It is very easy to create examples where this strategy will not work, and it is further complicated if there are structural problems with our linear model.

The issue of choosing the next state to visit is known as the exploration vs. exploitation problem. We first encountered this in chapter 7. How it should be solved is problem dependent. If you have a small set of decisions, a standard strategy is to use \( \epsilon \)-greedy, which we first mentioned briefly in chapter 11. In this strategy, we choose a decision at random with probability \( \epsilon \), or use our optimal decision \( x^n \) from step 1 with probability
If the state variable is continuous, we might simply add a noise term to force an exploration to nearby states. See chapter 7 for a thorough discussion of the issues and other strategies that can be used.

Approximate value iteration using a linear architecture has to be used with care. Provable convergence results are rare, and there are examples of divergence. As with all policies (whether they use value function approximations or not), the performance of any particular policy is very problem dependent. It is particularly valuable to design some sort of benchmark. If you are using value functions, then your problem likely falls in a class that requires a policy that estimates the downstream impact of a decision made now. This means that some form of direct lookahead approximation (described in chapter 19) might be a natural benchmark.

17.5 APPLICATIONS

There are many problems where we can exploit structure in the state variable, allowing us to propose functions characterized by a small number of parameters which have to be estimated statistically. Section 3.6.3 represented one version where we had a parameter for each (possibly aggregated) state. The only structure we assumed was implicit in the ability to specify a series of one or more aggregation functions.

The remainder of this section illustrates the use of regression models in specific applications which include pricing an American option and playing lose tic-tac-toe, followed by a brief discussion of deterministic problems that arise in engineering control problems and games.

<table>
<thead>
<tr>
<th></th>
<th>Stock prices</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Outcome</td>
<td>Time period</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.21 1.08 1.17 1.15</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.09 1.12 1.17 1.13</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1.15 1.08 1.22 1.35</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.17 1.12 1.18 1.15</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.08 1.15 1.10 1.27</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1.12 1.22 1.23 1.17</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1.16 1.14 1.13 1.19</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.22 1.18 1.21 1.28</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1.08 1.11 1.09 1.10</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1.15 1.14 1.18 1.22</td>
<td></td>
</tr>
</tbody>
</table>

Table 17.3 Ten sample realizations of prices over four time periods
17.5.1 Pricing an American option

Consider the problem of determining the value of an American-style put option which gives us the right to sell an asset (or contract) at a specified price at any of a set of discrete time periods. For example, we might be able to exercise the option on the last day of the month over the next 12 months.

Assume we have an option that allows us to sell an asset at $1.20 at any of four time periods. We assume a discount factor of 0.95 to capture the time value of money. If we wait until time period 4, we must exercise the option, receiving zero if the price is over $1.20. At intermediate periods, however, we may choose to hold the option even if the price is below $1.20 (of course, exercising it if the price is above $1.20 does not make sense). Our problem is to determine whether to hold or exercise the option at the intermediate points.

From history, we have found 10 samples of price trajectories which are shown in table 17.3.

If we wait until time period 4, our payoff is shown in table 17.4, which is zero if the price is above 1.20, and $1.20 - p_4$ for prices below $1.20.

At time $t = 3$, we have access to the price history $(p_1, p_2, p_3)$. Since we may not be able to assume that the prices are independent or even Markovian (where $p_3$ depends only on $p_2$), the entire price history represents our state variable, along with an indicator that tells us if we are still holding the asset. We wish to predict the value of holding the option at time $t = 4$. Let $V_4(S_4)$ be the value of the option if we are holding it at time 4, given the state (which includes the price $p_4$) at time 4. Now let the conditional expectation at time 3 be

$$V_3(S_3) = \mathbb{E}\{V_4(S_4) | S_3\}.$$

<table>
<thead>
<tr>
<th>Option value at $t = 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Outcome</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

*Table 17.4*  The payout at time 4 if we are still holding the option
Our goal is to approximate $V_3(S_3)$ using information we know at time 3. We propose a linear regression of the form

$$Y = \theta_0 + \theta_1 X_1 + \theta_2 X_2 + \theta_3 X_3,$$

where

$$Y = V_4,$$
$$X_1 = p_2,$$
$$X_2 = p_3,$$
$$X_3 = (p_3)^2.$$

The variables $X_1$, $X_2$ and $X_3$ are our basis functions. Keep in mind that it is important that our explanatory variables $X_i$ must be a function of information we have at time $t = 3$, whereas we are trying to predict what will happen at time $t = 4$ (the payoff). We would then set up the data matrix given in table 17.5.

We may now run a regression on this data to determine the parameters $(\theta_i)_{i=0}^3$. It makes sense to consider only the paths which produce a positive value in the fourth time period, since these represent the sample paths where we are most likely to still be holding the asset at the end. The linear regression is only an approximation, and it is best to fit the approximation in the region of prices which are the most interesting (we could use the same reasoning to include some "near misses"). We only use the value function to estimate the value of holding the asset, so it is this part of the function we wish to estimate. For our illustration, however, we use all 10 observations, which produces the equation

$$V_3 \approx 0.0056 - 0.1234 p_2 + 0.6011 p_3 - 0.3903(p_3)^2.$$
Table 17.6 The payout if we exercise at time 3, and the expected value of holding based on our approximation. The best decision is indicated in bold.

<table>
<thead>
<tr>
<th>Outcome</th>
<th>Decision</th>
<th>Exercise</th>
<th>Hold</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.03</td>
<td>0.04155 × .95 = 0.03947</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.03</td>
<td>0.03662 × .95 = 0.03479</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.00</td>
<td>0.02397 × .95 = 0.02372</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.02</td>
<td>0.03346 × .95 = 0.03178</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.10</td>
<td>0.05285 × .95 = 0.05021</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.00</td>
<td>0.00414 × .95 = 0.00394</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.07</td>
<td>0.00899 × .95 = 0.00854</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.00</td>
<td>0.01610 × .95 = 0.01530</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0.11</td>
<td>0.06032 × .95 = 0.05731</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.02</td>
<td>0.03099 × .95 = 0.02944</td>
<td></td>
</tr>
</tbody>
</table>

\( \bar{V}_3 \) is an approximation of the expected value of the price we would receive if we hold the option until time period 4. We can now use this approximation to help us decide what to do at time \( t = 3 \). Table 17.6 compares the value of exercising the option at time 3 against holding the option until time 4, computed as \( \gamma \bar{V}_3(S_3) \). Taking the larger of the two payouts, we find, for example, that we would hold the option given samples 1-4, 6, 8, and 10, but would sell given samples 5, 7, and 9.

We can repeat the exercise to estimate \( \bar{V}_2(S_t) \). This time, our dependent variable “\( Y \)” can be calculated two different ways. The simplest is to take the larger of the two columns from table 17.6 (marked in bold). So, for sample path 1, we would have \( Y_1 = \max\{0.03, 0.03947\} = 0.03947 \). This means that our observed value is actually based on our approximate value function \( \bar{V}_3(S_3) \).

An alternative way of computing the observed value of holding the option in time 3 is to use the approximate value function to determine the decision, but then use the actual price we receive when we eventually exercise the option. Using this method, we receive 0.05 for the first sample path because we decide to hold the asset at time 3 (based on our approximate value function) after which the price of the option turns out to be worth 0.05. Discounted, this is worth 0.0475. For sample path 2, the option proves to be worth 0.07 which discounts back to 0.0665 (we decided to hold at time 3, and the option was worth 0.07 at time 4). For sample path 5 the option is worth 0.10 because we decided to exercise at time 3.

Regardless of which way we compute the value of the problem at time 3, the remainder of the procedure is the same. We have to construct the independent variables “\( Y \)” and regress them against our observations of the value of the option at time 3 using the price history \( (p_1, p_2) \). Our only change in methodology would occur at time 1 where we would have to use a different model (because we do not have a price at time 0).
17.5.2 Playing “lose tic-tac-toe”

The game of “lose tic-tac-toe” is the same as the familiar game of tic-tac-toe, with the exception that now you are trying to make the other person get three in a row. This nice twist on the popular children’s game provides the setting for our next use of regression methods in approximate dynamic programming.

Unlike our exercise in pricing options, representing a tic-tac-toe board requires capturing a discrete state. Assume the cells in the board are numbered left to right, top to bottom as shown in figure 17.7a. Now consider the board in figure 17.7b. We can represent the state of the board after the $t^{th}$ play using

\[
S_t = \begin{cases} 
1 & \text{if cell } i \text{ contains an “X,”} \\
0 & \text{if cell } i \text{ is blank,} \\
-1 & \text{if cell } i \text{ contains an “O,”}
\end{cases}
\]

\[
S_t = (S_{ti})_{i=1}^9.
\]

We see that this simple problem has up to $3^9 = 19,683$ states. While many of these states will never be visited, the number of possibilities is still quite large, and seems to overstate the complexity of the game.

We quickly realize that what is important about a game board is not the status of every cell as we have represented it. For example, rotating the board does not change a thing, but it does represent a different state. Also, we tend to focus on strategies (early in the game when it is more interesting) such as winning the center of the board or a corner. We might start defining variables (basis functions) such as

\[
\phi_1(S_t) = \begin{cases} 
1 & \text{if there is an “X” in the center of the board,} \\
0 & \text{otherwise,}
\end{cases}
\]

\[
\phi_2(S_t) = \text{The number of corner cells with an “X,”}
\]

\[
\phi_3(S_t) = \text{The number of instances of adjacent cells with an “X” (horizontally, vertically, or diagonally)}.
\]

There are, of course, numerous such functions we can devise, but it is unlikely that we could come up with more than a few dozen (if that) which appeared to be useful. It is important to realize that we do not need a value function to tell us to make obvious moves.

Once we form our basis functions, our value function approximation is given by

\[
\hat{V}_t(S_t) = \sum_{f \in \mathcal{F}} \theta_t f(S_t).
\]
We note that we have indexed the parameters by time (the number of plays) since this might play a role in determining the value of the feature being measured by a basis function, but it is reasonable to try fitting a model where $\theta_f = \theta_f$. We estimate the parameters $\theta$ by playing the game (and following some policy) after which we see if we won or lost. We let $Y^n = 1$ if we won the $n^{th}$ game, 0 otherwise. This also means that the value function is trying to approximate the probability of winning if we are in a particular state.

We may play the game by using our value functions to help determine a policy. Another strategy, however, is simply to allow two people (ideally, experts) to play the game and use this to collect observations of states and game outcomes. This is an example of supervised learning. If we lack a “supervisor” then we have to depend on simple strategies combined with the use of slowly learned value function approximations. In this case, we also have to recognize that in the early iterations, we are not going to have enough information to reliably estimate the coefficients for a large number of basis functions.

### 17.5.3 Approximate dynamic programming for deterministic problems

There has been considerable interest in applying ADP to two classes of deterministic problems:

- **Engineering control problems** - Imagine making decisions about how to control a drone or robot, where we have to apply a multidimensional force vector $u_t$ to the device (using the notation of control theory) to minimize some performance metric.

- **Playing games** - There has been considerable interest in using reinforcement learning for computer Go, chess, and an array of video games.

Neural networks have proven to be very popular in both settings, with reports of considerable success (although the techniques for computer games tends to require a hybrid). As we pointed out when we first introduced neural networks in section 3.9.3, the high-dimensionality of neural networks tends to make them sensitive to noise. However, for deterministic problems this is not an issue, and the ability of neural networks to represent complex functions without the struggle of identifying reasonable architectures can be particularly powerful.

It is beyond the scope of this volume to describe developments in these two rich fields in any depth. We encourage readers interested in either of these problem classes to look for more specialized presentations.

### 17.6 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 linear models.

c) Infinite horizon problems using linear models.
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 linear models.

17.6.1 Finite horizon problems using lookup tables

A fairly general purpose version of an approximate policy iteration algorithm is given in figure 17.8 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 $\overline{V}^{\pi,n}$. 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 14.8.

17.6.2 Finite horizon problems using linear models

The simplest demonstration of approximate policy iteration using linear models is in the setting of a finite horizon problem. Figure 17.9 provides an adaption of the algorithm using lookup tables when we are using linear models. There is an outer loop over $n$ where we fix
Step 0. Initialization:

Step 0a. Initialize $V^\pi_0$.

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 look-ahead horizon $T$).

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

Step 4a. Compute
\[ x_t^{n,m} = \arg \max_{x_t \in X_t} \left( C(S_t^{n,m}, x_t) + \gamma V^\pi_{n-1}(S^{M,x_t}(S_t^{n,m}, x_t)) \right). \]

Step 4b. Compute
\[ S_{t+1}^{n,m} = S^{M}(S_t^{n,m}, x_t^{n,m}, W_{t+1}(\omega^m)). \]

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

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

Step 6a: Accumulate $\hat{v}^{n,m}$:
\[ \hat{v}^{n,m}_t = C(S_t^{n,m}, x_t^{n,m}) + \gamma \hat{v}^{n,m}_{t+1}. \]

Step 6b: Update the approximate value of the policy:
\[ \bar{v}^{n,m} = (\frac{m-1}{m})\hat{v}^{n,m-1} + \frac{1}{m} \hat{v}^{n,m}_0. \]

Step 8. Update the value function at $S^n$:
\[ V^\pi_n = (1 - \alpha_{n-1})\bar{v}^{n-1} + \alpha_{n-1} \hat{v}^{n,M}_0. \]

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

Step 10. Return the value functions ($V^\pi_N$).

Figure 17.8 A policy iteration algorithm for infinite horizon problems

the policy using
\[ X^\pi_t(S_t) = \arg \max_x \left( C(S_t, x) + \gamma \sum_f \theta^x_{t,f} \phi_f(S_t, x) \right). \] (17.25)

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

We update the policy $X^\pi_t(s)$ 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 $\theta^x_{t,f}$. This step replaces step 6b in figure 17.8.

If we let $M \to \infty$, then the parameter vector $\theta^{n,M}$ approaches the best possible fit for the policy $X^\pi_t(s)$ determined by $\theta^{n-1}$. 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
Step 0. Initialization:

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

Step 0b. Initialize $\theta^{\pi,0}_t$ 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 $S^0_0$:

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

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

$$x^{n,m}_t = \arg \max_{x_t \in X_t^{m,n}} \left( C(S_t^{m,n}, x_t) + \gamma \sum_t \theta^{\pi,n-1}_t \phi_f(S_t^{m,n}, x_t) \right).$$

Step 5b. Compute

$$S^{n,m}_{t+1} = S_t^{m,n}(S_t^{m,n}, x^{n,m}_t, W_{t+1}(\omega^m)).$$

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

Step 7: Do for $t = T, T - 1, \ldots, 0$:

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

Step 8. Update $\theta^{n,m}_{t+1}$ using recursive least squares to obtain $\theta^{n,m}_t$ (see section 3.8).

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

Step 10. Return the value functions $(V^{n,N})$.

Figure 17.9 A policy iteration algorithm for finite horizon problems using linear models.

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 $\theta^{n,m}_t$ as $m$ goes to infinity, but we may still have a terrible approximation of the policy produced by $\theta^{\pi,n-1}_t$.

17.6.3 LSTD for infinite horizon problems using linear models

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 linear models, where we introduce the dimension of projecting contributions over an infinite horizon. There are several ways of accomplishing this (see section 16.1.2). 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 $S^m$ and decision $x^m$ be given by

$$\hat{C}^m = C(S^m, x^m).$$
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 decisions greedily based on a value function approximation given by $V^n(s) = \sum f \theta_f(s)$ (see equation (17.25)). 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 16.3, we can use standard linear regression to estimate $\theta_m$ using

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

(17.26)

As a reminder, the term $\phi_i - \gamma \phi_i^{i+1}$ can be viewed as a simulated, sample realization of $I - \gamma P^{\pi}$, projected into the feature space. Just as we would use $(I - \gamma P^{\pi})^{-1}$ in our basic policy iteration to project the infinite-horizon value of a policy $\pi$ (for a review, see section 14.7), we are using the term

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

to produce an infinite-horizon estimate of the feature-projected contribution

$$
\left[ \frac{1}{1+m} \sum_{i=1}^{m} \phi^i \hat{C}_i \phi^i \right].
$$

Equation (17.26) requires solving a matrix inverse for every observation. It is much more efficient to use recursive least squares, which is done by using

$$
\epsilon^m = \hat{C}^m - (\phi^m - \gamma \phi^{m+1})^T \theta^{m-1},
$$

(17.27)

$$
M^m = M^{m-1} - \frac{M^{m-1} \phi^m (\phi^m - \gamma \phi^{m+1})^T M^{m-1}}{1 + (\phi^m - \gamma \phi^{m+1})^T M^{m-1} \phi^m},
$$

(17.28)

$$
\theta^m = \phi^{m-1} + \frac{\epsilon^m M^{m-1} \phi^m}{1 + (\phi^m - \gamma \phi^{m+1})^T M^{m-1} \phi^m}.
$$

(17.29)

Figure 17.10 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 $V(s) = \sum f \phi_f(s)$ (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 $M \rightarrow \infty$, 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 decisions. There are several features of the algorithm that allow this. First, computing the policy $X^\pi(s)\theta^n$ requires solving a deterministic optimization problem. If we are using discrete decisions, it means simply enumerating the decisions and choosing the best one. If we have continuous decisions, 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. It can be very difficult implementing an exploration step for multidimensional decision vectors.
Step 0. Initialization:

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

Step 0b. Set the initial policy:

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

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 decision $x^{n,m} = X^\pi(S^n|\theta^{n-1})$.

Step 6b: Compute the post-decision state $S^{n,m} = S^M(x^{n,m}, x^{n,m})$.

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

Step 6d: Compute the input variable $\phi(S^{n,m}) - \gamma \phi(S^{n,m+1})$ for equation (17.26).

Step 7: Do the following:

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

Step 7b: Compute $\theta^{n,m}$ using equation (17.26).

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

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

$$X^{\pi,n+1}(s) = \arg\max_{x \in X} (C(s,x) + \gamma \phi(S^n(s,x))^T \theta^{n+1}).$$

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

Figure 17.10  Approximate policy iteration for infinite horizon problems using least squares temporal differencing.

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 (17.26). When we use lookup tables, we require exploration to guarantee that we eventually will visit every state infinitely often. When we 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.

17.7 THE ACTOR-CRITIC PARADIGM

It is very popular in some communities to view approximate dynamic programming in terms of an “actor” and a “critic.” Simply put, the actor is a policy that chooses the decision, and the critic is the value function that evaluates the policy. In engineering control applications, where states and controls are continuous, it is common to represent both the policy and the approximate value function using neural networks, and hence some authors refer to “actor-nets” and “critic nets.” Note that in this setting, the actor is a form of policy function approximation.
Step 0. Initialization:

Step 0a. Initialize $V_{\pi,0}^{n}$, $t \in T$.
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:

$$x_{t}^{n,m} = \arg \max_{x_{t} \in X^{n,m}} \left( C_{t}(S_{t}^{n,m}, x_{t}) + \gamma V_{\pi,n}^{n-1}(S_{t}^{M,x}(S_{t}^{n,m}, x_{t})) \right) \quad (17.30)$$

Step 5b. Compute:

$$S_{t}^{x,n,m} = S_{t}^{M,x}(S_{t}^{n,m}, x_{t})$$
$$S_{t+1}^{n,m} = S_{t+1}^{M,W}(S_{t}^{x,n,m}, W_{t+1}(\omega^{m})).$$

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

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

$$\hat{v}_{t}^{m} = C_{t}(S_{t}^{n,m}, x_{t}) + \gamma \hat{v}_{t+1}^{m}.$$

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

$$\nabla_{t-1}^{n,m} \leftarrow U^{\pi} \left( \nabla_{t-1}^{n,m-1}, S_{t-1}^{x,n,m}, \hat{v}_{t}^{m} \right) \quad (17.31)$$

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

Step 7. Update the policy value function

$$V_{\pi,n}^{n}(S_{t}^{n}) = \nabla_{t}^{n,M}(S_{t}^{n}) \quad \forall t = 0, 1, \ldots, T$$

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

Figure 17.11 Approximate policy iteration using value function-based policies.

The policy iteration algorithm in figure 17.11 provides one illustration of the actor-critic paradigm. The decision function is equation (17.30), where $V_{\pi,n}^{n-1}$ determines the policy (in this case). This is the actor. Equation (17.31), where we update our estimate of the value of the policy, is the critic. We fix the actor (that is, we fix the value function approximation used by 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 $\nabla_{t}$.

In other settings, the policy is a policy function approximation of some form that maps the state direction to a decision. 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 $j$,” which would be a form of lookup table policy. 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.

### 17.8 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 14.9 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
\[
\bar{Q}(s, x|\theta) = \sum_f \theta_f \phi_f(s, x),
\]
The policy is represented using a function such as
\[
X^\pi(s|\eta) = e^{\eta \phi(s, x)} \sum_{x'} \eta \phi(s, x').
\]
This choice of policy has the important feature that the probability that a decision is chosen is greater than zero. Also, \( X^\pi(s|\eta) \) is differentiable in the policy parameter vector \( \eta \).

In the language of actor-critic algorithms, \( \bar{Q}(s, x|\theta) \) is an approximation of the critic parameterized by \( \theta \), while \( X^\pi(s|\eta) \) is an approximate policy parameterized by \( \eta \). We can update \( \theta \) and \( \eta \) using standard stochastic gradient methods. We begin by defining
\[
\psi_\theta(s, x) = \nabla_\theta \ln \pi_\theta(x|s) = \nabla_\theta \pi_\theta(x|s) / \pi_\theta(x|s).
\]
Since we are maximizing the average reward per time period, we begin by estimating the average reward per time period using
\[
\bar{c}^{n+1} = (1 - \alpha_n)\bar{c}^n + \alpha_n C(S^{n+1}, x^{n+1}).
\]
We then compute the temporal difference in terms of the difference between the contribution of a state-decision pair, and the average contribution, using
\[
\delta^n = C(S^n, x^n) - \bar{c}^n + (\theta^n)^T \phi_{\theta^n}(S^{n+1}, x^{n+1}) - (\theta^n)^T \phi_{\theta^n}(S^n, x^n).
\]
Assume we are using a TD(\( \lambda \)) updating procedure where we assume \( 0 < \lambda < 1 \). We compute the eligibility trace using
\[
Z^{n+1} = \lambda Z^n + \phi_{\eta^n}(S^{n+1}, x^{n+1}).
\]
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
\[
\eta^{n+1} = \eta^n - \beta^n \Gamma(\theta^n)(\theta^n)^T \phi_{\eta^n}(S^{n+1}, x^{n+1}) \psi_{\theta^n}(S^{n+1}, x^{n+1}). \tag{17.32}
\]
The critic update is given by
\[ \theta^{n+1} = \theta^n + \alpha_n \delta^n Z^n. \] (17.33)

Equations (17.32) and (17.33) 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.

17.9 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 \( \bar{Q}^{n-1}(S,x) \). The problem is that the estimates \( \bar{Q}^{n-1}(S,x) \) are random variables. In the best of circumstances, assume that \( \bar{Q}^{n-1}(S,x) \) is an unbiased estimate of the true value \( V_t(S) \) of being in (post-decision) state \( S \). 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 decision 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 decision and use it to compute \( \hat{q}_n \).

To illustrate, assume we have to choose a decision \( x \in \mathcal{X} \), where \( C(S,x) \) is the contribution earned by using decision \( x \) (given that we are in state \( S \)) which then takes us to (post-decision) state \( S^{M,x}(S,x) \) where we receive an estimated value \( \bar{V}(S^{M,x}(S,x)) \). Normally, we would update the value of being in state \( S \) by computing
\[ \hat{v}^n = \max_{x \in \mathcal{X}} (C(S,x) + \bar{V}^{x,n-1}(S^{M,x}(S,x))). \]

We would then update the value of being in state \( S \) using our standard update formula
\[ \bar{V}^n(S) = (1 - \alpha_{n-1})\bar{V}^{n-1}(S) + \alpha_{n-1}\hat{v}^n. \]

Since \( \bar{V}^{n-1}(S^{M,x}(S,x)) \) is a random variable, sometimes it will overestimate the true value of being in state \( S^{M,x}(S,x) \) while other times it will underestimate the true value. Of course, we are more likely to choose a decision 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
\[ U_x = C(S,x) + \bar{V}(S^{M,x}(S,x)) \]
be the estimated value of using decision \( x \). The statistical error, which we represent as \( \beta \), is given by
\[ \beta = \mathbb{E}\{\max_{x \in \mathcal{X}} U_x\} - \max_{x \in \mathcal{X}} \mathbb{E}U_x. \] (17.34)

The first term on the right-hand side of (17.34) is the expected value of \( \bar{V}(S) \), 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.”
which means that we assume that the estimates \( \bar{V}(S^M,x(S,x)) \) are correct, and then try to estimate \( \mathbb{E}\{\max_{x \in X} U_x\} \). Thus, computing the second term in (17.34) is easy.

The challenge is computing \( \mathbb{E}\{\max_{x \in X} U_x\} \). We assume that while we have been computing \( \bar{V}(S^M,x(S,x)) \), we have also been computing \( \sigma^2(x) = \text{Var}(U_x) = \text{Var}(\bar{V}(S^M,x(S,x))) \). Using the plug-in principle, we are going to assume that the estimates \( \sigma^2(x) \) represent the true variances of the value function approximations. Computing \( \mathbb{E}\{\max_{x \in X} U_x\} \) 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 \( X = \{1,2,\ldots,|X|\} \). Now let

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

We can compute the mean and variance of \( \bar{U}_2 \) as follows. First, we temporarily define \( \alpha \) using

\[
\alpha^2 = \sigma_1^2 + \sigma_2^2 - 2\sigma_1\sigma_2\rho_{12}
\]

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

\[
z = \frac{\mu_1 - \mu_2}{\alpha},
\]

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

\[
\begin{align*}
\mathbb{E}\bar{U}_2 &= \mu_1\Phi(z) + \mu_2\Phi(-z) + \alpha\phi(z) \\
\text{Var}(\bar{U}_2) &= \left[(\mu_1^2 + \sigma_1^2)\Phi(z) + (\mu_2^2 + \sigma_2^2)\Phi(-z) + (\mu_1 + \mu_2)\alpha\phi(z)\right] - (\mathbb{E}\bar{U}_2)^2.
\end{align*}
\]

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 (17.35) and variance given by (17.36). 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

\[
\bar{U}_x = \mathbb{E}\max\{U_1,U_2,\ldots,U_x\} \approx \mathbb{E}\max\{U_x,\bar{U}_{x-1}\}.
\]

We can apply this repeatedly until we find the mean of \( \bar{U}_{|X|} \), which is an approximation of \( \mathbb{E}\{\max_{x \in X} \bar{U}_x\} \). This, in turn, allows us to compute an estimate of the statistical bias \( \beta \) given by equation (17.34).
Figure 17.12 plots $\beta = \mathbb{E} \max_{x} U_x - \max_{x} \mathbb{E} U_x$ as it is being computed for 100 decisions, averaged over 30 sample realizations. The standard deviation of each $U_x$ was fixed at $\sigma = 20$. The plot shows that the error increases steadily until the set $\mathcal{X}$ reaches about 20 or 25 decisions, after which it grows much more slowly. Of course, in an approximate dynamic programming application, each $U_x$ 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.
17.10 THE LINEAR PROGRAMMING METHOD USING LINEAR MODELS

In section 14.10, we showed that the determination of the value of being in each state can be found by solving the following linear program

$$\min_v \sum_{s \in S} \beta_s v(s)$$  \hspace{1cm} (17.37)

subject to

$$v(s) \geq C(s, x) + \gamma \sum_{s' \in S} p(s'|s, x) v(s') \quad \text{for all } s \text{ and } x.$$  \hspace{1cm} (17.38)

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-decision 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

$$\mathcal{V}(s|\theta) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(s).$$

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

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

subject to:

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

This is still a linear program, but now the decision variables are \((\theta_f)_{f \in \mathcal{F}}\) 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 (17.37)-(17.38) 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 decisions. Given a state space \(S\) and set of decisions \(\mathcal{X}\), we can randomly choose states and decisions 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 decisions. 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 17.13 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 \( K = \{1, 2, 3\} \) 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 17.13). Let \( D = \{1, 2, \ldots, 8\} \) represent a decision to serve a particular queue, and let \( D_a \) 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 \( d \in D_a \).

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 decisions. Let \( \mathcal{X} \) 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 decisions 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 17.7, showing that the ADP-based strategy significantly outperforms these other policies.

Figure 17.13 Queueing network with three servers serving a total of eight queues, two with exogenous arrivals (\( \lambda \)) and six with arrivals from other queues (from de Farias & Van Roy (2003)).
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 (17.38)), 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.

### 17.11 APPROXIMATE POLICY ITERATION USING KERNEL REGRESSION*

We build on the foundation provided in section 16.6 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

\[
\hat{C}_m = C(S^{n,m}, x^{n,m}).
\]

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

\[
k(S^{x,i}, S^{x,j}) = \frac{K_h(S^{x,i}, S^{x,j})}{\sum_{i=0}^{m-1} K_h(S^{x,i}, S^{x,j})},
\]

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

\[
P^{\pi,n}_{i,j} = k(S^{x,i-1}, S^{x,j}).
\]

By construction, \( P^{\pi,n} \) is a stochastic matrix (its rows sum to 1), which means that \( I - \gamma P^{\pi,n} \) 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 \( S^{x,0}, \ldots, S^{x,m+1} \) using

\[
\hat{M}^{\pi,m} V(s) = \sum_{i=0}^{m-1} k(S^{x,i}, s)(\hat{C}^i + \gamma V(S^{M,x}(S^i, x^i, W^{i+1})))
\]
FORWARD ADP II: POLICY OPTIMIZATION

Step 0. Initialization:

Step 0a. Initialize the policy $X^\pi(s)$.

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

Step 0c. Set $n = 1$.

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

Step 2. Choose an initial state $S_0^n$.

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

Step 4: Let $x^{n,m} = X^\pi_n(S^{n,m})$.

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

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

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

Step 8: Let $P^\pi_n$ be a $M \times M$ matrix where the $(i,j)$th entry is given by $K_h(S^{x,i}, S^{x,j})$ for $i, j \in \{1, \ldots, m\}$.

Step 9: Solve for $\hat{v}^n = (I - \gamma P^\pi_n)^{-1}$, where $\hat{v}^n$ is an $m$-dimensional vector with $i$th element $\hat{v}_i = \hat{C}_i + \gamma \hat{v}^n(S^{x,i})$ for $i = 1, \ldots, m$. This can be approximated using value iteration.

Step 10: Let $\sum_{i=0}^{m-1} K_h(S^{x,i}, s)(\hat{C}_i + \gamma \hat{v}^n(S^{x,i}))$ be our kernel-based value function approximation.

Step 11: Update the policy:

$$X^\pi_{n+1}(s) = \arg \max_x \left( C(s, x) + \gamma \sum_{i=0}^{m-1} K_h(S^{x,i}, s)(\hat{C}_i + \gamma \hat{v}^n(S^{x,i})) \right).$$

Step 12. Return the $X^\pi_{N}(s)$ and parameter $\theta^N$.

Figure 17.14 Approximate policy iteration using least squares temporal differencing and kernel regression.

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

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

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

$$\hat{V}^\pi_{k+1} = \sum_{i=0}^{m-1} K_h(S^{x,i}, s)(\hat{C}_i + \gamma \hat{V}^\pi(S^{x,i+1})).$$

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

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

This approximation forms the basis of our approximate policy iteration. The full algorithm is given in figure 17.14.
17.12 FINITE 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 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 17.15 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 $S_1$ 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 $S_1$ that we generate through Monte Carlo sampling. We might obtain the dotted curve labeled as $V_1(S_1)$, 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.
Figure 17.15  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 $S_0$.

17.13 BIBLIOGRAPHIC NOTES

Section 17.2 - 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.

Section 17.4 - 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.

Section 17.6 - 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 17.7 - 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 (2018)). 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 17.8 - 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 et al. (1983) compares several policy gradient algorithms. Szepesvári (2010b) provides a recent summary of policy gradient algorithms.

Section 17.10 - Schweitzer & Seidmann (1985) describes the use of basis functions in the context of the linear programming method. The idea is further developed in Farias & 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 17.11 - This material is based on Ma & Powell (2010).

EXERCISES

Review questions

17.1 Explain the difference between on-policy and off-policy learning.

17.2 Contrast, using only necessary notation (but you will need some) the essential differences between ADP using a pre-decision state, a post-decision state, and Q-learning.

17.3 Contrast ADP using a post-decision state versus Q-learning if \( x \) is a vector.

17.4 Explain in words the difference between the single-pass and double-pass versions of forward ADP.

17.5 Use notation to explain what is meant by the “actor” and the “critic” in the actor-critic paradigm.

Modeling questions

17.6 The most common strategy for using approximate dynamic programming is to train value function approximations offline using a simulator. Using the language introduced in section 9.11, where we classified problems based on a) whether they were state-independent or state-dependent, and b) whether we were optimizing the final reward or cumulative reward, training VFAs would fall in the class of state-dependent problems, where we are maximizing the final reward. In its most compact form, this objective can be written (see, for example, table 9.3)

\[
\max_{\pi, \theta} \mathbb{E} \left\{ C(S, X^{\pi \theta}(S|\theta), W)|S_0 \right\}.
\]
In equation (9.42), we expanded the expectations to make the underlying random variables explicit, which produced the equivalent expression

\[
\max_{\pi^{lrn}} \mathbb{E}\{C(S, X^{imp}(S|\theta^{imp}), \hat{W})|S_0\} = \\
\mathbb{E}_{S_0} \mathbb{E}_{W_{t=1}^{T}, \ldots, W_N|S_0} \mathbb{E}_{S_0}^{imp} \mathbb{E}_{\theta|S_0} C(S, X^{imp}(S|\theta^{imp}), \hat{W}). \tag{17.40}
\]

Using the context of the forward ADP algorithms presented in this chapter, answer the following:

(a) When optimizing over policies (such as the learning policies \(\pi^{lrn}\)), we have to search over classes of policies \(f \in F^{lrn}\), and any tunable parameters \(\theta \in \Theta_f\) within that class. Give two examples of “policy classes” and an example of a tunable parameter for each class.

(b) Throughout the book, we have used as our default objective for dynamic programming the function

\[
\max_{\pi} \mathbb{E} \left\{ \sum_{t=0}^{T} C(S_t, X^\pi(S_t)) | S_0 \right\}. \tag{17.41}
\]

This chapter (and we could include the backward ADP methods of chapter 15) presents different methods for training VFAs, after which we would run simulations to test the effectiveness by simulating the objective in (17.41). Explain what is meant by \(\pi^{lrn}\) and \(\pi^{imp}\) in equation (17.40).

(c) In section 9.11, we identified equation (17.39) as the objective for optimizing final reward and we showed (in equation (9.43)) that this could be simulated using

\[
\max_{\pi^{lrn}} \mathbb{E}_{S_0} \mathbb{E}_{(W^{imp}_{t})_{t=0}^{T}}^{imp} \mathbb{E}_{(W^{imp}_{t})_{t=0}^{T}} \frac{1}{T} \sum_{t=0}^{T-1} C(S_t, X^{imp}(S_t|\theta^{imp}), \hat{W}_{t+1}) \right\}. \tag{17.42}
\]

Make the case that when I am designing algorithms to solve the cumulative reward objective in (17.41) that I am actually solving the final-reward optimization problem given in (17.42).

**Computational exercises**

**17.7** We are going to revisit exercise 15.4 using forward ADP, which we repeat here. In this exercise you are going to solve a simple inventory problem using Bellman’s equations, to obtain an optimal policy. Then, the exercises that follow will have you implement various backward ADP policies that you can compare against the optimal policy you obtain in this exercise. Your inventory problem will span \(T\) time periods, with an inventory equation governed by

\[ R_{t+1} = \max\{0, R_t - \hat{D}_{t+1}\} + x_t. \]

Here we are assuming that product ordered at time \(t, x_t\), arrive at \(t + 1\). Assume that \(\hat{D}_{t+1}\) is described by a discrete uniform distribution between 1 and 20.

Next assume that our contribution function is given by

\[ C(S_t, x_t) = 50 \min\{R_t, \hat{D}_{t+1}\} - 10x_t. \]
a) Find an optimal policy by solving this dynamic program exactly using classical backward dynamic programming methods from chapter 14 (specifically equation (14.3)). Note that your biggest challenge will be computing the one-step transition matrix. Simulate the optimal policy 1,000 times starting with \( R_0 = 0 \) and report the performance.

b) Now solve the problem using forward ADP using a simple quadratic approximation for the value function approximation:

\[
\nabla_x^t(R^x_t) = \theta_{t0} + \theta_{t1}R^x_t + \theta_{t2}(R^x_t)^2.
\]

where \( R^x_t \) is the post-decision resource state which we might represent using

\[
R^x_t = \max\{0, R_t - \mathbb{E}\{\hat{D}_{t+1}\}\} + x_t.
\]

Use 100 forward passes to estimate \( \nabla_x(S_t) \) using the algorithm in figure 17.4.

c) Having found \( \nabla_x^t(R^x_t) \), simulate the resulting policy 1,000 times, and compare your results to your optimal policy.

d) Repeat (b) and (c) but this time use a value function approximation that is only linear in \( R^x_t \):

\[
\nabla_x^t(R^x_t) = \theta_{t0} + \theta_{t1}R^x_t.
\]

How does the resulting policy compare the your results from part (c)?

17.8 We are going to revisit exercise 15.3 using forward ADP, which we repeat here. We are going to solve the continuous budgeting problem presented in section 14.4.2 using backward approximate dynamic programming. The problem starts with \( R_0 \) resources which are then allocated over periods 0 to \( T \). Let \( x_t \) be the amount allocated in period \( t \) with contribution

\[
C_t(x_t) = \sqrt{x_t}.
\]

Assume that \( T = 20 \) time periods.

a) Use the results of section 14.4.2 to solve this problem optimally. Evaluate your simulation by simulating your optimal policy 1000 times.

b) Use the forward ADP algorithm described in figure 17.4 to obtain the value function approximations using

\[
\nabla_t(R_t) = \theta_{t0} + \theta_{t1}\sqrt{R_t}
\]

Use 100 forward passes to estimate \( \nabla_t(R_t) \). Use linear regression (either the methods in section 3.7.1, or a package) to fit \( \nabla_t(R_t) \). Then, simulate this policy 1000 times (ideally using the same sample paths as you used for part (a)). How do you think \( \theta_{t0} \) and \( \theta_{t1} \) should behave?

c) Use the forward ADP algorithm described in figure 15.5 to obtain the value function approximations using

\[
\nabla_t(R_t) = \theta_{t0} + \theta_{t1}R^x_t + \theta_{t2}(R^x_t)^2,
\]
where $R^x_t$ is the post-decision resource state $R^x_t = R_t - x_t$ (which is the same as $R_{t+1}$ since transitions are deterministic).

Use linear regression (either the methods in section 3.7.1, or a package) to fit $V_t(R_t)$.

Then, simulate this policy 1000 times (ideally using the same sample paths as you used for part (a)).

17.9 Repeat exercise, but this time use

$$C(x_t) = \ln(x_t).$$

For part (b), use

$$\nabla_t(R_t) = \theta_{t0} + \theta_{t1} \ln(x_t).$$

Theory questions

17.10 Prove that the newsvendor objective function

$$F(x) = \mathbb{E}\{p \min\{x, W\} - cx\}$$

is concave in $x$ as long as $p \geq c$.

Problem solving questions

17.11 We are going to try again to solve our asset selling problem. We assume we are holding a real asset and we are responding to a series of offers. Let $\hat{p}_t$ be the $t^{th}$ offer, which is uniformly distributed between 500 and 600 (all prices are in thousands of dollars). We also assume that each offer is independent of all prior offers. You are willing to consider up to 10 offers, and your goal is to get the highest possible price. If you have not accepted the first nine offers, you must accept the $10^{th}$ offer.

(a) Write out the decision function you would use in a dynamic programming algorithm in terms of a Monte Carlo sample of the latest price and a current estimate of the value function.

(b) Write out the updating equations (for the value function) you would use after solving the decision problem for the $t^{th}$ offer.

(c) Implement an approximate dynamic programming algorithm using synchronous state sampling. Using 1000 iterations, write out your estimates of the value of being in each state immediately after each offer. For this exercise, you will need to discretize prices for the purpose of approximating the value function. Discretize the value function in units of 5 dollars.

(d) From your value functions, infer a decision rule of the form “sell if the price is greater than $\bar{p}_t$.”

17.12 We wish to use Q-learning to solve the problem of deciding whether to continue playing a game where you win $1 if you flip a coin and see heads, and lose $1 if you see
tails. Using a stepsize $\alpha = \frac{\theta}{\pi^2m}$, implement the $Q$-learning algorithm in equations (11.18) and (11.19). Initialize your estimates $\bar{Q}(s, a) = 0$, and run 1000 of the algorithm using $\theta = 1, 10, 100$ and 1000. Plot $Q^n$ for each of the three values of $\theta$, and discuss the choice you would make if your budget was $N = 50, 100$ or 1000.

**Diary problem**

The diary problem is a single problem you chose (see chapter 1 for guidelines). Answer the following for your diary problem.

17.13 For your diary problem, compare the pure forward pass algorithm in figure 17.4 to the two-pass algorithm in figure 17.5 in terms of both computational complexity and likely performance.