APPROXIMATE DYNAMIC PROGRAMMING—II: ALGORITHMS

WARREN B. POWELL
Department of Operations Research and Financial Engineering, Princeton University, Princeton, New Jersey

INTRODUCTION

Approximate dynamic programming (ADP) represents a powerful modeling and algorithmic strategy that can address a wide range of optimization problems that involve making decisions sequentially in the presence of different types of uncertainty. A short list of applications, which illustrate different problem classes, include the following:

- *Option Pricing.* An American option allows us to buy or sell an asset at any time up to a specified time, where we make money when the price goes under or over (respectively) a set strike price. Valuing the option requires finding an optimal policy for determining when to exercise the option.
- *Playing Games.* Computer algorithms have been designed to play backgammon, bridge, chess, and recently, the Chinese game of Go.
- *Controlling a Device.* This might be a robot or unmanned aerial vehicle, but there is a need for autonomous devices to manage themselves for tasks ranging from vacuuming the floor to collecting information about terrorists.
- *Storage of Continuous Resources.* Managing the cash balance for a mutual fund or the amount of water in a reservoir used for a hydroelectric dam requires managing a continuous resource over time in the presence of stochastic information on parameters such as prices and rainfall.
- *Asset Acquisition.* We often have to acquire physical and financial assets over time under different sources of uncertainty about availability, demands, and prices.
- *Resource Allocation.* Whether we are managing blood inventories, financial portfolios, or fleets of vehicles, we often have to move, transform, clean, and repair resources to meet various needs under uncertainty about demands and prices.
- *R&D Portfolio Optimization.* The Department of Energy has to determine how to allocate government funds to advance the science of energy generation, transmission, and storage. These decisions have to be made over time in the presence of uncertain changes in technology and commodity prices.

These problems range from relatively low-dimensional applications to very high-dimensional industrial problems, but all share the property of making decisions over time under different types of uncertainty. In Ref. 1, a modeling framework is described, which breaks a problem into five components.

- *State.* S_t (or x_t in the control theory community), capturing all the information we need at time t to make a decision and model the evolution of the system in future.
- *Action / Decision / Control.* Depending on the community, these will be modeled as a, x, or u. Decisions are made using a decision function or policy. If we are using action a, we represent the decision function using $A^\pi(S_t)$, where $\pi \in \Pi$ is a family of policies (or functions). If our action is x, we use $X^\pi(S_t)$. We use a if our problem has a small number of discrete actions. We use x when the decision might be a vector (of discrete or continuous elements).
- *Exogenous Information.* Lacking standard notation, we let W_t be the family

Wiley Encyclopedia of Operations Research and Management Science, edited by James J. Cochran

of random variables that represent new information that first becomes known by time t.

- *Transition Function.* Also known as the *system model* (or just model), this function is denoted by $S^M(\cdot)$, and is used to express the evolution of the state variable, as in $S_{t+1} = S^M(S_t, x_t, W_{t+1})$.
- *Objective Function.* Let $C(S_t, x_t)$ be the contribution (if we are maximizing) received when in state S_t if we take action x_t. Our objective is to find a decision function (policy) that solves

$$\max_{\pi \in \Pi} \mathbb{E}\left\{\sum_{t=0}^{T} \gamma^t C(S_t, X^\pi(S_t))\right\}. \quad (1)$$

We encourage readers to review Ref. 1 (in this volume) or Chapter 5 in Ref. 2, available at http://www.castlelab.princeton.edu/adp.htm, before attempting to design an algorithmic strategy. For the remainder of this article, we assume that the reader is familiar with the modeling behind the objective function in Equation (1), and in particular the range of policies that can be used to provide solutions. In this article, we assume that we would like to find a good policy by using Bellman's equation as a starting point, which we may write in either of two ways:

$$V_t(S_t) = \max_a \big(C(S_t, a) + \gamma \sum_{s'} p(s'|S_t, a) V_{t+1}(s')\big), \quad (2)$$

$$= \max_a \big(C(S_t, a) + \gamma \mathbb{E}\{V_{t+1}(S_{t+1})|S_t\}\big), \quad (3)$$

where $V_t(S_t)$ is the value of being in state S_t and following an optimal policy from t until the end of the planning horizon (which may be infinite). The control theory community replaces $V_t(S_t)$ with $J_t(S_t)$, which is referred to as the cost-to-go function. If we are solving a problem in steady state, $V_t(S_t)$ would be replaced with $V(S)$.

If we have a small number of discrete states and actions, we can find the value function $V_t(S_t)$ using the classical techniques of value iteration and policy iteration [3]. Many practical problems, however, suffer from one or more of the three curses of dimensionality: (i) vector-valued (and possibly continuous) state variables, (ii) random variables W_t for which we may not be able to compute the expectation, and (iii) decisions (typically denoted by x_t or u_t), which may be discrete or continuous vectors, requiring some sort of solver (linear, nonlinear, or integer programming) or specialized algorithmic strategy.

The field of ADP has historically focused on the problem of multidimensional state variables, which prevent us from calculating $V_t(s)$ for each discrete state s. In our presentation, we make an effort to cover this literature, but we also show how we can overcome the other two curses using a device known as the *postdecision state variable*. However, a short article such as this is simply unable to present the vast range of algorithmic strategies in any detail. For this, we recommend for additional reading, the following: Ref. 4, especially for students interested in obtaining thorough theoretical foundations; Ref. 5 for a presentation from the perspective of the reinforcement learning (RL) community; Ref. 2 for a presentation that puts more emphasis on modeling, and more from the perspective of the operations research community; and Chapter 6 of Ref. 6, which can be downloaded from http://web.mit.edu/dimitrib/www/dpchapter.html.

A GENERIC ADP ALGORITHM

Equation (2) (or Eq. 3) is typically solved by stepping backward in time, where it is necessary to loop over all the potential states to compute $V_t(S_t)$ for each state S_t. For this reason, classical dynamic programming is often referred to as *backward dynamic programming*. The requirement of looping over all the states is the first computational step that cannot be performed when the state variable is a vector, or even a scalar continuous variable.

ADP takes a very different approach. In most ADP algorithms, we step forward in time following a single sample path. Assume we are modeling a finite horizon problem. We are going to iteratively simulate this problem.

At iteration n, assume that at time t we are in state S_t^n. Now assume that we have a policy of some sort that produces a decision x_t^n. In ADP, we are typically solving a problem that can be written as

$$\hat{v}_t^n = \max_{x_t} \big(C(S_t^n, x_t) + \gamma \mathbb{E}\{\overline{V}_{t+1}^{n-1}(S^M(S_t^n, x_t, W_{t+1}))|S_t\}\big). \quad (4)$$

Here, $\overline{V}_{t+1}^{n-1}(S_{t+1})$ is an approximation of the value of being in state $S_{t+1} = S^M(S_t^n, x_t, W_{t+1})$ at time $t+1$. If we are modeling a problem in steady state, we drop the subscript t everywhere, recognizing that W is a random variable. We note that x_t here can be a vector, where the maximization problem might require the use of a linear, nonlinear, or integer programming package. Let x_t^n be the value of x_t that solves this optimization problem. We can use $\hat{v}_t^n$ to update our approximation of the value function. For example, if we are using a lookup table representation, we might use

$$\overline{V}_t^n(S_t^n) = (1-\alpha_{n-1})\overline{V}_t^{n-1}(S_t^n) + \alpha_{n-1}\hat{v}_t^n, \quad (5)$$

where α_{n-1} is a step size between 0 and 1 (more on this later).

Now we are going to use Monte Carlo methods to sample our vector of random variables, which we denote by W_{t+1}^n, representing the new information that would have first been learned between time t and $t+1$. The next state would be given by

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

The overall algorithm is given in Fig. 1. This is a very basic version of an ADP algorithm, one which would generally not work in practice. But it illustrates some of the basic elements of an ADP algorithm. First, it steps forward in time, using a randomly sampled set of outcomes, visiting one state at a time. Secondly, it makes decisions using some sort of statistical approximation of a value function, although it is unlikely that we would use a lookup table representation. Thirdly, we use information gathered as we step forward in time to update our value function approximation, almost always using some sort of recursive statistics.

Step 0. Initialization:

Step 0a. Initialize $\bar{V}_t^0(S_t)$ for all states S_t.

Step 0b. Choose an initial state S_0^1.

Step 0c. Set $n = 1$.

Step 1. Choose a sample path ω^n.

Step 2. For $t = 0, 1, 2, \ldots, T$ do:

Step 2a. Solve

$$\hat{v}_t^n = \max_{x_t \in \mathcal{X}_t^n} \left(C_t(S_t^n, x_t) + \gamma \mathbb{E}\bar{V}_{t+1}^{n-1}(S^M(S_t^n, x_t, W_{t+1}))\right)$$

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

Step 2b. Update $\bar{V}_t^{n-1}(S_t)$ using

$$\bar{V}_t^n(S_t) = \begin{cases} (1-\alpha_{n-1})\bar{V}_t^{n-1}(S_t^n) + \alpha_{n-1}\hat{v}_t^n & S_t = S_t^n \\ \bar{V}_t^{n-1}(S_t) & \text{otherwise.} \end{cases}$$

Step 2c. Compute the next state to visit. This may be chosen at random (according to some distribution), or from

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

Step 3. Let $n = n+1$. If $n < N$, go to step 1.

Figure 1. An approximate dynamic programming algorithm using expectations.

The algorithm in Fig. 1 goes under different names. It is a basic "forward pass" algorithm, where we step forward in time, updating value functions as we progress. Another variation involves simulating forward through the horizon without updating the value function. Then, after the simulation is done, we step backward through time, using information about the entire future trajectory.

We have written the algorithm in the context of a finite horizon problem. For infinite horizon problems, simply drop the subscript t everywhere, remembering that you have to keep track of the "current" and "future" state.

There are different ways of presenting this basic algorithm. To illustrate, we can rewrite Equation (5) as

$$\overline{V}^n(S_t^n) = \overline{V}^{n-1}(S_t^n) + \alpha_{n-1}(\hat{v}_t^n - \overline{V}^{n-1}(S_t^n)). \tag{6}$$

The quantity

$$\hat{v}_t^n - \overline{V}^{n-1}(S_t^n) = C_t(S_t^n, x_t) + \gamma \overline{V}_{t+1}^{n-1} \times (S_{t+1}) - \overline{V}^{n-1}(S_t^n)$$

where

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

is known as the *Bellman error*, since it is a sampled observation of the difference between what we think is the value of being in state S_t^n and an estimate of the value of being in state S_t^n. In the RL community, it has long been called the *temporal difference (TD)*, since it is the difference between estimates at two different iterations, which can have the interpretation of time, especially for steady-state problems, where n indexes transitions forward in time.

The RL community would refer to this algorithm as TD learning (first introduced in Ref. 7). If we use a backward pass, we can let $\hat{v}_t^n$ be the value of the entire future trajectory, but it is often useful to introduce a discount factor (usually represented as λ) to discount rewards received in the future. This discount factor is introduced purely for algorithmic reasons, and should not be confused with the discount γ, which is intended to capture the time value of money. If we let $\lambda = 1$, then we are adding up the entire future trajectory. But if we use $\lambda = 0$, then we obtain the same updating as our forward pass algorithm in Fig. 1. For this reason, the RL community refers to this family of updating strategies as TD(λ).

TD(0) (TD learning with $\lambda = 0$) can be viewed as an approximate version of classical value iteration. It is important to recognize that after each update, we are not just changing the value function approximation, we are also changing our behavior (i.e., our policy for making decisions), which in turn changes the distribution of $\hat{v}_t^n$ given a particular state S_t^n. TD(1), on the other hand, computes $\hat{v}_t^n$ by simulating a policy into the future. This policy depends on the value function approximation, but otherwise $\hat{v}_t^n$ does not directly use value function approximations.

An important dimension of most ADP algorithms is Step 2c, where we have to choose which state to visit next. For many complex problems, it is most natural to choose the state determined by the action x_t^n, and then simulate our way to the next state using $S_{t+1}^n = S^M(S_t^n, x_t^n, W_{t+1}(\omega^n))$. Many authors refer to this as a form of "real-time dynamic programming" (RTDP) [8], although a better term is trajectory following [9], since RTDP involves other algorithmic assumptions. The alternative to trajectory following is to randomly generate a state to visit next. Trajectory following often seems more natural since it means you are simulating a system, and it visits the states that arise naturally, rather than from what might be an unrealistic probability model. But readers need to understand that there are few convergence results for trajectory following models, due to the complex interaction between random observations of the value of being in a state and the policy that results from a specific value function approximation, which impacts the probability of visiting a state.

The power and flexibility of ADP has to be tempered with the reality that simple algorithms generally do not work, even on simple problems. There are several issues we have to address if we want to develop an effective ADP strategy:

- We avoid the problem of looping over all the states, but replace it with the problem of developing a good statistical approximation of the value of being in each state that we might visit. The challenge of designing a good value function approximation is at the heart of most ADP algorithms.
- We have to determine how to update our approximation using new information.
- We assume that we can compute the expectation, which is often not the case, and which especially causes problems when our decision/action variable is a vector.
- We can easily get caught in a circle of choosing actions that take us to states that we have already visited, simply because we may have pessimistic estimates of states that we have not yet visited. We need some mechanism to force us to visit states just to learn the value of these states.

The remainder of this article is a brief tour through a rich set of algorithmic strategies for solving these problems.

Q-LEARNING AND THE POSTDECISION STATE VARIABLE

In our classical ADP strategy, we are solving optimization problems of the form

$$\hat{v}_t^n = \max_{x_t \in \mathcal{X}_t^n} \big(C(S_t^n, x_t) + \gamma \mathbb{E}\{\overline{V}_{t+1}(S^M(S_t^n, x_t, W_{t+1}))|S_t\}\big). \quad (7)$$

There are many applications of ADP, which introduce additional complications: (i) we may not be able to compute the expectation, and (ii) the decision x may be a vector, and possibly a very big vector (thousands of dimensions). There are two related strategies that have evolved to address these issues: *Q*-learning, a technique widely used in the RL community, and ADP using the postdecision state variable, a method that has evolved primarily in the operations research community. These methods are actually closely related, a topic we revisit at the end.

Q-Factors

To introduce the idea of *Q*-factors, we need to return to the notation where a is a discrete action (with a small action space). The *Q* factor (so named because this was the original notation used in Ref. 10) is the value of being in a state *and* taking a specific action,

$$Q(s,a) = C(s,a) + \gamma \mathbb{E}\overline{V}(S^M(S^n, a, W)).$$

Value functions and *Q*-factors are related using

$$V(s) = \max_a Q(s,a),$$

so, at first glance it might seem as if we are not actually gaining anything. In fact, estimating a *Q*-factor actually seems harder than estimating the value of being in a state, since there are more state-action pairs than there are states.

The power of *Q*-factors arises when we have problems where we either do not have an explicit transition function $S^M(\cdot)$, or do not know the probability distribution of W. This means we cannot compute the expectation, but not because it is computationally difficult. When we do not know the transition function, and/or cannot compute the expectation (because we do not know the probability distribution of the random variable), we would like to draw on a subfield of ADP known as *model-free* dynamic programming. In this setting, we are in state S^n, we choose an action a^n, and then observe the next state, which we will call s'. We can then record a value of being in state s and taking action a using

$$\hat{q}^n = C(S^n, a^n) + \overline{V}^{n-1}(s')$$

where $\overline{V}(s') = \max_{a'} Q^{n-1}(s', a')$. We then use this value to update our estimate of the *Q*-factor using

$$Q^n(S^n, a^n) = (1-\alpha)Q^{n-1}(S^n, a^n) + \alpha \hat{q}^n.$$

Whenever we are in state S^n, we choose our action using

$$a^n = \arg\max_a Q^{n-1}(S^n, a). \quad (8)$$

Note that we now have a method that does not require a transition function or the need to compute an expectation, we only need access to some exogenous process that tells us what state we transition to, given a starting state and an action.

The Postdecision State Variable

A closely related strategy is to break the transition function into two steps: the pure effect of the decision x_t, and the effect of the new, random information W_{t+1}. In this section, we use the notation of operations research, because what we are going to do is enable the solution of problems where x_t may be a very large vector. We also return to time-dependent notation since it clarifies when we are measuring a particular variable.

We can illustrate this idea using a simple inventory example. Let S_t be the amount of inventory on hand at time t. Let x_t be an order for a new product, which we assume arrives immediately. The typical inventory equation would be written as

$$S_{t+1} = \max\{0, S_t + x_t - \hat{D}_{t+1}\}$$

where $\hat{D}_{t+1}$ is the random demand that arises between t and $t+1$. We denote the pure effect of this order using

$$S_t^x = S_t + x_t.$$

The state S_t^x is referred to as the *postdecision state* [11], which is the state at time t, immediately after a decision has been made, but before any new information has arrived. The transition from S_t^x to S_{t+1} in this example would then be given by

$$S_{t+1} = \max\{0, S_t^x - \hat{D}_{t+1}\}.$$

Another example of a postdecision state is a tic-tac-toe board after you have made your move, but before your opponent has moved. Reference 5 refers to this as the after-state variable, but we prefer the notation that emphasizes the information content (which is why S_t^x is indexed by t).

A different way of representing a postdecision state is to assume that we have a forecast $\overline{W}_{t,t+1} = \mathbb{E}\{W_{+1}|S_t\}$, which is a point estimate of what we think W_{t+1} will be given what we know at time t. A postdecision state can be thought of as a forecast of the state S_{t+1} given what we know at time t, which is to say

$$S_t^x = S^M(S_t, x_t, \overline{W}_{t,t+1}).$$

This approach will seem more natural in certain applications, but it would not be useful when random variables are discrete. Thus, it makes sense to use the expected demand, but not the expected action of your tic-tac-toe opponent.

In a typical decision tree, S_t would represent the information available at a decision node, while S_t^x would be the information available at an outcome node. Using these two steps, Bellman's equations become

$$V_t(S_t) = \max_{x_t} \big(C(S_t, x_t) + \gamma V_t^x(S_t^x)\big), \quad (9)$$

$$V_t^x(S_t^x) = \mathbb{E}\{V_{t+1}(S_{t+1})|S_t\}. \quad (10)$$

Note that Equation (9) is now a deterministic optimization problem, while Equation (10) involves only an expectation. Of course, we still do not know $V_t^x(S_t^x)$, but we can approximate it just as we would approximate $V_t(S_t)$. However, now we have to pay attention to the nature of the optimization problem. If we are just searching over a small number of discrete actions, we do not really have to worry about the structure of our approximate value function around S_t^x. But if x is a vector, which might have hundreds or thousands of discrete or continuous variables, then we have to recognize that we are going to need some sort of solver to handle the maximization problem.

To illustrate the basic idea, assume that we have discrete states and are using a lookup table representation for a value function. We would solve Equation (9) to obtain a decision x_t^n, and we would then simulate our way to S_{t+1}^n using our transition function. Let $\hat{v}_t^n$ be the objective function returned by solving Equation (9) at time t, when we are in state S_t^n. We would then use this value to update the value at the *previous postdecision state*. That is,

$$\overline{V}_{t-1}^n(S_{t-1}^n) = (1-\alpha_{n-1})\overline{V}_{t-1}^{n-1}(S_{t-1}^n) + \alpha_{n-1}\hat{v}_t^n. \quad (11)$$

Thus, the updating is virtually the same as we did before; it is just that we are now using our estimate of the value of being in a state to update the value of the previous postdecision state. This is a small change, but it has a huge impact by eliminating the expectation in the decision problem.

Now imagine that we are managing different types of products, where S_{ti} is the inventory of product of type i, and S^x_{ti} is the inventory after placing a new order. We might propose a value function approximation that looks like

$$\overline{V}^x_t(S^x_t) = \sum_i \left(\theta_{1i} S^x_{ti} - \theta_{2i}(S^x_{ti})^2\right).$$

This value function is separable and concave in the inventory (assuming $\theta_{2i} > 0$). Using this approximation, we can probably solve our maximization problem using a nonlinear programming algorithm, which means we can handle problems with hundreds or thousands of products. Critical to this step is that the objective function in Equation (9) is deterministic.

Comments

On first glance, Q-factors and ADP using the postdecision state variable seem to be very different methods to address very different problems. However, there is a mathematical commonality to these two algorithmic strategies. First, we note that if S is our current state, (S, a) is a kind of postdecision state, in that it is a deterministic function of the state and action. Second, if we take a close look at Equation (9), we can write

$$Q(S_t, x_t) = C(S_t, x_t) + \gamma V^x_t(S^x_t).$$

We assume that we have a known function S^x_t that depends only on S_t and x_t, but we do not require the full transition function (which takes us to time $t + 1$), and we do not have to take an expectation. However, rather than develop an approximation of the value of a state and an action, we only require the value of a (postdecision) state. The task of finding the best action from a set of Q-factors (Eq. 8) is identical to the maximization problem in Equation (9). What has been overlooked in the ADP/RL community is that this step makes it possible to search over large, multidimensional (and potentially continuous) action spaces. Multidimensional action spaces are a relatively overlooked problem class in the ADP/RL communities.

APPROXIMATE POLICY ITERATION

An important variant of ADP is approximate policy iteration, which is illustrated in Fig. 2 using the postdecision state variables. In this strategy, we simulate a policy, say, M times over the planning horizon (or M steps into the future). During these inner iterations, we fix the policy (i.e., we fix the value function approximation used to make a decision) to obtain a better estimate of the value of being in a state. The general process of updating the value function approximation is handled using the update function $U^V(\cdot)$. As $M \to \infty$, we obtain an exact estimate of the value of being in a particular state, while following a fixed policy (see Ref. 12 for the convergence theory of this technique).

If we randomize on the starting state and repeat this simulation for an infinite number of times, we can obtain the best possible value function given a particular approximation. It is possible to prove convergence of classical stochastic approximation methods when the policy is fixed. Unfortunately, there are very few convergence results when we combine sampled observations of the value of being in a state with changes in the policy.

If we are using low-dimensional representations of the value function (as occurs with basis functions), it is important that our update be of sufficient accuracy. If we use $M = 1$, which means we are updating after a single forward traversal, the result may mean that the value function approximation (and therefore the policy) is changing too rapidly from one iteration to another. The resulting algorithm may actually diverge [13,14].

TYPES OF VALUE FUNCTION APPROXIMATIONS

At the heart of approxmiate dynamic programming is approximating the value of

Step 0. Initialization:

Step 0a. Initialize $V_t^{\pi,0}, \quad t \in \mathcal{T}$.

Step 0b. Set $n = 1$.

Step 0c. Initialize S_0^1.

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

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

Step 3. Choose a sample path ω^m.

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

Step 4a. Solve:

$$x_t^{n,m} = \arg \max_{x_t \in \mathcal{X}_t^{n,m}} \left(C_t(S_t^{n,m}, x_t) + \gamma V_t^{\pi,n-1}(S^{M,x}(S_t^{n,m}, x_t))\right) \tag{12}$$

Step 4b. Simulate:

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

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

Step 5a. Accumulate the path cost (with $\hat{v}_T^m = 0$)

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

Step 5b. For $t > 1$, update approximate value of the policy starting at time t:

$$\bar{V}_{t-1}^{n,m} \leftarrow U^V(\bar{V}_{t-1}^{n,m-1}, S_{t-1}^{x,n,m}, \hat{v}_t^m). \tag{13}$$

Step 6. Update the policy value function

$$V_t^{\pi,n}(S_t^x) = \bar{V}_t^{n,M}(S_t^x) \quad \forall t = 0, 1, \ldots, T$$

Step 7. Return the value functions $(V_t^{\pi,N})_{t=1}^T$.

Figure 2. Approximate policy iteration using value function-based policies.

being in a state. It is important to emphasize that the problem of designing and estimating a value function approximation draws on the entire field of statistics and machine learning. Below, we discuss two very general approximation methods. You will generally find that ADP is opening a doorway into the field of machine learning, and that this is the place where you will spend most of your time. You should pick up a good reference book such as Ref. 15 on statistical learning methods, and keep an open mind regarding the best method for approximating a value function (or the policy directly).

Lookup Tables with Aggregation

The simplest form of statistical learning uses the lookup table representation that we first illustrated in Equation (5). The problem with this method is that it does not apply to continuous states, and requires an exponentially large number of parameters (one per state) when we encounter vector-valued states.

A popular way to overcome large state spaces is through aggregation. Let $\mathcal{S}$ be the set of states. We can represent aggregation using

$$G^g : \mathcal{S} \rightarrow \mathcal{S}^{(g)}.$$

$\mathcal{S}^{(g)}$ represents the g^{th} level of aggregation of the state space $\mathcal{S}$, where we assume that $\mathcal{S}^{(0)} = \mathcal{S}$. Let

$s^{(g)}$ = $G^g(s)$, the g^{th} level aggregation of state s.

$\mathcal{G}$ = The set of indices corresponding to the levels of aggregation.

We then define $\overline{V}^g(s)$ to be the value function approximation for state $s \in \mathcal{S}^{(g)}$.

We assume that the family of aggregation functions $G^g, g \in \mathcal{G}$ is given. A common strategy is to pick a single level of aggregation that seems to work well, but this introduces several complications. First, the best level of aggregation changes as we acquire more data, creating the challenge of working out how to transition from approximating the value function in the early iterations versus later iterations. Second, the right level of

aggregation depends on how often we visit a region of the state space.

A more natural way is to use a weighted average. At iteration n, we can approximate the value of being in state s using

$$\overline{v}^n(s) = \sum_{g \in \mathcal{G}} w^{(g,n)}(s) \overline{v}^{(g,n)}(s).$$

where $w^{(g,n)}(s)$ is the weight given to the g^{th} level of aggregation when measuring state s. This weight is set to zero, if there are no observations of states $s \in \mathcal{S}^{(g)}$. Thus, while there may be an extremely large number of states, the number of positive weights at the most disaggregate level $w^{(0,n)}$ is never greater than the number of observations. Typically, the number of positive weights at more aggregate levels will be substantially smaller than this.

There are different ways to determine these weights, but an effective method is to use weights that are inversely proportional to the total squared variation of each estimator. Let

$\overline{\sigma}^2(s)^{(g,n)}$	the variance of the estimate of the value of state s at the g^{th} level of aggregation after collecting n observations,
$\overline{\mu}^{(g,n)}(s)$	an estimate of the bias due to aggregation.

We can then use weights that satisfy

$$w^{(g)}(s) \propto \left((\overline{\sigma}^2(s))^{(g,n)} + \left(\overline{\mu}^{(g,n)}(s) \right)^2 \right)^{-1}. \quad (14)$$

The weights are then normalized so that they sum to one. The quantities $\overline{\sigma}^2(s)^{(g,n)}$ and $\overline{\mu}^{(g,n)}(s)$ are computed using the following:

$$\begin{aligned} (\overline{\sigma}^2(s))^{(g,n)} &= Var[\overline{v}^{(g,n)}(s)] \\ &= \lambda^{(g,n)}(s)(s^2(s))^{(g,n)}, \end{aligned} \quad (15)$$

$$(s^2(s))^{(g,n)} = \frac{\overline{v}^{(g,n)}(s) - (\overline{\beta}^{(g,n)}(s))^2}{1 + \lambda^{n-1}}, \quad (16)$$

$$\lambda^{(g,n)}(s) = \begin{cases} (\alpha^{(g,n-1)}(s))^2, & n = 1, \\ (1 - \alpha^{(g,n-1)}(s))^2 \\ \quad \times \lambda^{(g,n-1)}(s) \\ \quad + (\alpha^{(g,n-1)}(s))^2, & n > 1, \end{cases} \quad (17)$$

$$\begin{aligned} \overline{v}^{(g,n)}(s) &= (1 - \alpha_{n-1}) \overline{v}^{(g,n-1)}(s) \\ &\quad + \alpha_{n-1} (\overline{v}^{(g,n-1)}(s) - \hat{v}^n(s))^2, \end{aligned} \quad (18)$$

$$\begin{aligned} \overline{\beta}^{(g,n)}(s) &= (1 - \alpha_{n-1}) \overline{\beta}^{(g,n-1)}(s) \\ &\quad + \alpha_{n-1} (\hat{v}^n - \overline{v}^{(g,n-1)}(s)), \end{aligned} \quad (19)$$

$$\overline{\mu}^{(g,n)}(s) = \overline{v}^{(g,n)}(s) - \overline{v}^{(0,n)}(s). \quad (20)$$

The term $\overline{v}^{(g,n)}(s)$ is the total squared variation in the estimate, which includes variation due to pure noise, $(s^2(s))^{(g,n)}$, and two sources of bias. The term $\overline{\beta}^{(g,n)}(s)$ is the bias introduced when smoothing a nonstationary data series (if the series is steadily rising, our estimates are biased downward), and $\overline{\mu}^{(g,n)}(s)$ is the bias due to aggregation error.

This system for computing weights scales easily to large state spaces. As we collect more observations of states in a particular region, the weights gravitate toward putting more weight on disaggregate estimates, while keeping higher weights on more aggregate estimates for regions that are only lightly sampled. For more on this strategy, see Ref. 16 or Chapter 7 in Ref. 2. For other important references on aggregation see Refs 4, 17, 18 and 19.

Basis Functions

Perhaps the most widely cited method for approximating value functions uses the concept of basis functions. Let

$\mathcal{F}$	A set of features drawn from a state vector,
$\phi_f(S)$	A scalar function that draws what is felt to be a useful piece of information from the state variable, for $f \in \mathcal{F}$.

The function $\phi_f(S)$ is referred to as a *basis function*, since in an ideal setting we would have a family of functions that spans the state space, allowing us to write

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

for an appropriately chosen vector of parameters θ. In practice, we cannot guarantee that we can find such a set of basis functions, so we write

$$\overline{V}(S|\theta) \approx \sum_{f\in\mathcal{F}} \theta_f \phi_f(S).$$

Basis functions are typically referred to as *independent variables* or *covariates* in the statistics community. If S is a scalar variable, we might write

$$\overline{V}(S|\theta) = \theta_0 + \theta_1 S + \theta_2 S^2 + \theta_3 \sin(S) + \theta_4 \ln(S).$$

This type of approximation is referred to as *linear*, since it is linear in the parameters. However, it is clear that we are capturing nonlinear relationships between the value function and the state variable.

There are several ways to estimate the parameter vector. The easiest, which is often referred to as *least squares temporal differencing* (LSTD), uses $\hat{v}_t^n$ as it is calculated in Equation (4), which means that it depends on the value function approximation. An alternative is least squares policy estimation (LSPE), where we use repeated samples of rewards from simulating a fixed policy. The basic LSPE algorithm estimates the parameter vector θ by solving

$$\theta^n = \arg\max_{\theta} \sum_{m=1}^{n} \left(\hat{v}_t^m - \sum_{f\in\mathcal{F}} \theta_f \phi_f(S_t^m) \right)^2.$$

This algorithm provides nice convergence guarantees, but the updating step becomes more and more expensive as the algorithm progresses. Also, it is putting equal weight on all iterations, a property that is not desirable in practice. We refer the reader to Ref. 6 for a more thorough discussion of LSTD and LSPE.

A simple alternative uses a stochastic gradient algorithm. Assuming we have an initial estimate θ^0, we can update the estimate of theta at iteration n using

$$\begin{aligned}\theta^n &= \theta^{n-1} - \alpha_{n-1}\left(\overline{V}_t(S_t^n|\theta^{n-1}) - \hat{v}_t^n\right)\nabla_\theta \overline{V}_t \\ &\quad \times (S_t^n|\theta^{n-1}) \\ &= \theta^{n-1} - \alpha_{n-1}\left(\overline{V}(S_t^n|\theta^{n-1}) - \hat{v}^n(S_t^n)\right) \\ &\quad \times \begin{pmatrix} \phi_1(S_t^n) \\ \phi_2(S_t^n) \\ \vdots \\ \phi_F(S_t^n) \end{pmatrix}. \qquad (21)\end{aligned}$$

Here, α_{n-1} is a step size, but it has to perform a scaling function, so it is not necessarily less than 1. Depending on the problem, we might have a step size that starts around 10^6 or 0.00001.

A more powerful strategy uses recursive least squares. Let ϕ^n be the vector created by computing $\phi(S^n)$ for each feature $f \in \mathcal{F}$. The parameter vector θ can be updated using

$$\theta^n = \theta^{n-1} - H^n \phi^n \hat{\varepsilon}^n, \qquad (22)$$

where ϕ^n is the vector of basis functions evaluated at S^n, and $\hat{\varepsilon}^n = \overline{V}(S_t^n|\theta^{n-1}) - \hat{v}^n(S_t^n)$ is the error in our prediction of the value function. The matrix H^n is computed using

$$H^n = \frac{1}{\gamma^n} B^{n-1}. \qquad (23)$$

B^{n-1} is an $F+1$ by $F+1$ matrix (where $F = |\mathcal{F}|$), which is updated recursively using

$$B^n = B^{n-1} - \frac{1}{\gamma^n}(B^{n-1}\phi^n(\phi^n)^T B^{n-1}). \qquad (24)$$

γ^n is a scalar computed using

$$\gamma^n = 1 + (\phi^n)^T B^{n-1} \phi^n. \qquad (25)$$

Note that there is no step size in these equations. These equations avoid the scaling issues of the stochastic gradient update in Equation (21), but hide the fact that they are implicitly using a step size of $1/n$. This can be very effective, but can work very poorly. We can overcome this by introducing a factor λ and revising the updating of γ^n and B^n using

$$\gamma^n = \lambda + (x^n)^T B^{n-1} x^n, \qquad (26)$$

and the updating formula for B^n, which is now given by

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

This method puts a weight of λ^{n-m} on an observation from iteration $n-m$. If $\lambda = 1$, then we are weighting all observations equally. Smaller values of λ put a lower weight on earlier observations. See Chapter 7 of Ref. 2 for a more thorough discussion of this strategy. For a thorough presentation of recursive estimation methods for basis functions in ADP, we encourage readers to see Ref. 19.

Values versus Marginal Values

There are many problems, where it is more effective to use the derivative of the value function with respect to a state, rather than the value of being in the state. This is particularly powerful in the context of resource allocation problems, where we are solving a vector-valued decision problem using linear, nonlinear, or integer programming. In these problem classes, the derivative of the value function is much more important than the value function itself. This idea has long been recognized in the control theory community, which has used the term heuristic dynamic programming to refer to ADP using the value of being in a state, and "dual heuristic" dynamic programming when using the derivative (see Ref. 20 for a nice discussion of these topics from a control-theoretic perspective). Chapters 11 and 12 in Ref. 2 discuss the use of derivatives for applications that arise in operations research.

Discussion

Approximating value functions can be viewed as just an application of a vast array of statistical methods to estimate the value of being in a state. Perhaps the biggest challenge is model specification, which can include designing the basis functions. See Refs 14 and 21 for a thorough discussion of feature selection. There has also been considerable interest in the use of nonparametric methods [22]. An excellent reference for statistical learning techniques that can be used in this setting is Ref. 15.

The estimation of value function approximations, however, involves much more than just a simple application of statistical methods. There are several issues that have to be addressed that are unique to the ADP setting:

1. Value function approximations have to be estimated recursively. There are many statistical methods that depend on estimating a model from a fixed batch of observations. In ADP, we have to update value function approximations after each observation.
2. We have to get through the early iterations. In the first iteration, we have no observations. After 10 iterations, we have to work with value functions that have been approximated using 10 data points (even though we may have hundreds or thousands of parameters to estimate). We depend on the value function approximations in these early iterations to guide decisions. Poor decisions in the early iterations can lead to poor value function approximations.
3. The value $\hat{v}_t^n$ depends on the value function approximation $\overline{V}_{t+1}(S)$. Errors in this approximation produce biases in $\hat{v}_t^n$, which then distorts our ability to estimate $\overline{V}_t(S)$. As a result, we are recursively estimating value functions using nonstationary data.
4. We can choose which state to visit next. We do not have to use $S_{t+1}^n = S^M(S_t^n, x_t^n, W_{t+1}^n)$, which is a strategy known as *trajectory following*. This is known in the ADP community as the "exploration vs. exploitation" Problem—do we explore a state just to learn about it, or do we visit a state that appears to be the best? Although considerable attention has been given to this problem, it is a largely unresolved issue, especially for high-dimensional problems.
5. Everyone wants to avoid the art of specifying value function approximations. Here, the ADP community shares the broader goal of the statistical learning community, which is a method that will approximate a value function with no human intervention.

THE LINEAR PROGRAMMING METHOD

The best-known methods for solving dynamic programs (in steady state) are value iteration and policy iteration. Less well known is that we can solve for the value of being in each (discrete) state by solving the linear program

$$\min_{v} \sum_{s\in\mathcal{S}} \beta_s v(s) \tag{27}$$

subject to

$$v(s) \geq C(s,a) + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a)v(s') \text{ for all } s \text{ and } a, \tag{28}$$

where β is simply a vector of positive coefficients. In this problem, the decision variable is the value v(s) of being in state s, which has to satisfy the constraints (Eq. 26). The problem with this method is that there is a decision variable for each state, and a constraint for each state-action pair. Not surprisingly, this can produce very large linear programs very quickly.

The linear programming method received a new lease on life with the work of de Farias and Van Roy [23], who introduced ADP concepts to this method. The first step to reduce complexity involves introducing basis functions to simplify the representation of the value function, giving us

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

subject to

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

Now, we have reduced a problem with $|\mathcal{S}|$ variables (one value per state) to a vector $\theta_f, f \in \mathcal{F}$. Much easier, but we still have the problem of too many constraints. de Farias and Van Roy [23] then introduced the novel idea of using a Monte Carlo sample of the constraints. As of this writing, this method is receiving considerable attention in the research community, but as with all ADP algorithms, considerable work is needed to produce robust algorithms that work in an industrial setting.

STEP SIZES

While designing a good value function, approximations are the most central challenge in the design of an ADP algorithm; a critical and often overlooked choice is the design of a step size rule. This is not to say that step sizes have been ignored in the research literature, but it is frequently the case that developers do not realize the dangers arising from an incorrect choice of step size rule.

We can illustrate the basic challenge in the design of a step size rule by considering a very simple dynamic program. In fact, our dynamic program has only one state and no actions. We can write it as computing the following expectation

$$F = \mathbb{E} \sum_{t=0}^{\infty} \gamma^t \hat{C}_t.$$

Assume that the contributions $\hat{C}_t$ are identically distributed as a random variable $\hat{C}$ and that $\mathbb{E}\hat{C} = c$. We know that the answer to this problem is $c/(1-\gamma)$, but assume that we do not know c, and we have to depend on random observations of $\hat{C}$. We can solve this problem using the following two-step algorithm:

$$\hat{v}^n = \hat{C}^n + \gamma \overline{v}^{n-1}, \tag{29}$$

$$\overline{v}^n = (1-\alpha_{n-1})\overline{v}^{n-1} + \alpha_{n-1}\hat{v}^n. \tag{30}$$

We are using an iterative algorithm, where $\overline{v}^n$ is our estimate of F after n iterations. We assume that $\hat{C}^n$ is the n^{th} sample realization of $\hat{C}$.

We note that Equation (29) handles the problem of summing over multiple contributions. Since we depend on Monte Carlo observations of $\hat{C}$, we use Equation (30) tothen perform smoothing. If $\hat{C}$ were deterministic, we would use a step size $\alpha_{n-1} = 1$ to achieve the fastest convergence. On the other hand,

if $\hat{C}$ is random, but if $\gamma = 0$ (which means we do not really have to deal with the infinite sum), then the best possible step size is $1/n$ (we are just trying to estimate the mean of $\hat{C}$). In fact, $1/n$ satisfies a common set of theoretical conditions for convergence which are typically written

$$\sum_{n=1}^{\infty} \alpha_n = \infty,$$

$$\sum_{n=1}^{\infty} (\alpha_n)^2 < \infty.$$

The first of these conditions prevents stalling (a step size of 0.8^n would fail this condition), while the second ensures that the variance of the resulting estimate goes to zero (statistical convergence).

Not surprisingly, many step size formulas have been proposed in the literature. A review is provided in Ref. 24 (see also Chapter 6 in Ref. 2). Two popular step size rules that satisfy the conditions above are

$$\alpha_n = \frac{a}{a + n - 1}, \quad \text{and}$$

$$\alpha_n = \frac{1}{n^{\beta}}.$$

In the first formula, $a = 1$ gives you $1/n$, whereas larger values of a produce a step size that declines more slowly. In the second rule, β has to satisfy $0.5 < \beta \leq 1$. Of course, these rules can be combined, but both a and β have to be tuned, which can be particularly annoying.

Reference 24 introduces the following step size rule (called the *bias adjusted Kalman filter* (BAKF) step size rule):

$$\alpha_n = 1 - \frac{\sigma^2}{(1 + \lambda^n)\sigma^2 + (\beta^n)}, \tag{31}$$

where

$$\lambda^n = \begin{cases} (\alpha_{n-1})^2, & n = 1 \\ (1 - \alpha_{n-1})^2 \lambda^{n-1} + (\alpha_{n-1})^2, & n > 1. \end{cases}$$

Here, σ^2 is the variance of the observation noise, and β^n is the bias measuring the difference between the current estimate of the value function $\overline{V}^n(S^n)$ and the true value function $V(S^n)$. Since these quantities are not generally known they have to be estimated from data (see Refs 2 and 24, Chapter 6 for details).

This step size enjoys some useful properties. If there is no noise, then $\alpha_{n-1} = 1$. If $\beta^n = 0$, then $\alpha_{n-1} = 1/n$. It can also be shown that $\alpha_{n-1} \geq 1/n$ at all times. The primary difficulty is that the bias term has to be estimated from noisy data, which can cause problems. If the level of observation noise is very high, we recommend a deterministic step size formula. But if the observation noise is not too high, the BAKF rule can work quite well, because it adapts to the data.

PRACTICAL ISSUES

ADP is a flexible modeling and algorithmic framework that requires that a developer make a number of choices in the design of an effective algorithm. Once you have developed a model, designed a value function approximation, and chosen a learning strategy, you still face additional steps before you can conclude that you have an effective strategy.

Usually the first challenge you will face is debugging a model that does not seem to be working. For example, as the algorithm learns, the solution will tend to get better, although it is hardly guaranteed to improve from one iteration to the next. But what if the algorithm does not seem to produce any improvement at all?

Start by making sure that you are solving the decision problem correctly, given the value function approximation (even if it may be incorrect). This is really only an issue for more complex problems, such as where x is an integer-valued vector, requiring the use of more advanced algorithmic tools.

Then, most importantly, verify the accuracy of the information (such as $\hat{v}$) that you are deriving to update the value function. Are you using an estimate of the value of being in a state, or perhaps the marginal value of an additional resource? Is this value correct? If you are using a forward pass algorithm (where $\hat{v}$ depends on a value function approximation of the future), assume the

approximation is correct and validate $\hat{v}$. This is more difficult if you are using a backward pass (as with TD(1)), since $\hat{v}$ has to track the value (or marginal value) of an entire trajectory.

When you have confidence that $\hat{v}$ is correct, are you updating the value function approximation correctly? For example, if you compare $\hat{v}_t^n$ with $\overline{V}^{n-1}(S_t^n)$, the observations of $\hat{v}^n$ (over iterations) should look like the scattered noise of observations around an estimated mean. Do you see a persistent bias?

Finally, think about whether your value function approximations are really contributing better decisions. What if you set the value function to zero? What intelligence do you feel your value function approximations are really adding? Keep in mind that there are some problems where a myopic policy can work reasonably well.

Now, assume that you are convinced that your value function approximations are improving your solution. One of the most difficult challenges is evaluating the quality of an ADP solution. Perhaps you feel you are getting a good solution, but how far from optimal are you?

There is no single answer to the question of how to evaluate an ADP algorithm. Typically, you want to ask the question: If I simplify my problem in some way (without making it trivial), do I obtain an interesting problem that I can solve optimally? If so, you should be able to apply your ADP algorithm to this simplified problem, producing a benchmark against which you can compare.

How a problem might be simplified is situation specific. One strategy is to eliminate uncertainty. If the deterministic problem is still interesting (for example, this might be the case in a transportation application, but not a financial application), then this can be a powerful benchmark. Alternatively, it might be possible to reduce the problem to one that can be solved exactly as a discrete Markov decision process.

If these approaches do not produce anything, the only remaining alternative is to compare an ADP-based strategy against a policy that is being used (or most likely to be used) in practice. If it is not the best policy, is it at least an improvement? Of course, there are many settings, where ADP is being used to develop a model to perform policy studies. In such situations, it is typically more important that the model behave reasonably. For example, do the results change in response to changes in inputs as expected?

ADP is more like a toolbox than a recipe. A successful model requires some creativity and perseverance. There are many instances of people who try ADP, only to conclude that it "does not work." It is very easy to miss one ingredient to produce a model that does not work. But a successful model not only solves a problem, but it also enhances your ability to understand and solve complex decision problems.

REFERENCES

1. Powell WB. Approximate dynamic programming — I: Modeling. Encyclopedia for operations research and management science. New York: John Wiley & Sons, Inc.; 2010.
2. Powell WB. Approximate dynamic programming: solving the curses of dimensionality. New York: John Wiley & Sons, Inc.; 2007.
3. Puterman ML. Markov decision processes. New York: John Wiley & Sons, Inc.; 1994.
4. Bertsekas D, Tsitsiklis J. Neuro-dynamic programming. Belmont (MA): Athena Scientific; 1996.
5. Sutton R, Barto A. Reinforcement learning. Cambridge (MA): The MIT Press; 1998.
6. Bertsekas D. Volume II, Dynamic programming and optimal control. 3rd ed. Belmont (MA): Athena Scientific; 2007.
7. Sutton R. Learning to predict by the methods of temporal differences. Mach Learn 1988;3(1):9–44.
8. Barto AG, Bradtke SJ, Singh SP. Learning to act using real-time dynamic programming. Artif Intell Spec Vol Comput Res Interact Agency 1995;72:81–138.
9. Sutton R. On the virtues of linear learning and trajectory distributions. In: Proceedings of the Workshop on Value Function Approximation, Machine Learning Conference; 1995. pp. 95–206.
10. Watkins C. Learning from delayed rewards [PhD thesis]. Cambridge: Cambridge University; 1989.
11. Van Roy B, Bertsekas DP, Lee Y, *et al*. A neuro-dynamic programming approach to

retailer inventory management. In: Proceedings of the IEEE Conference on Decision and Control, Volume 4, 1997. pp. 4052–4057.

12. Tsitsiklis J, Van Roy B. An analysis of temporal-difference learning with function approximation. IEEE Trans Autom Control 1997;42:674–690.
13. Bell DE. Risk, return, and utility. Manage Sci 1995;41:23–30.
14. Tsitsiklis JN, Van Roy B. Feature-based methods for large-scale dynamic programming. Mach Learn 1996;22:59–94.
15. Hastie T, Tibshirani R, Friedman J. The elements of statistical learning, Springer series in Statistics. New York: Springer; 2001.
16. George A, Powell WB, Kulkarni S. Value function approximation using multiple aggregation for multiattribute resource management. J Mach Learn Res 2008;2079–2111.
17. Bertsekas D, Castanon D. Adaptive aggregation methods for infinite horizon dynamic programming. IEEE Trans Automat Control 1989;34(6):589–598.
18. Luus R. Iterative dynamic programming. New York: Chapman & Hall/CRC; 2000.
19. Choi DP, Van Roy B. A generalized Kalman filter for fixed point approximation and efficient temporal-difference learning. Discrete Event Dyn Syst 2006;16:207–239.
20. Ferrari S, Stengel RF. Model-based adaptive critic designs. In: Si J, Barto AG, Powell WB, *et al*., editors. Handbook of learning and approximate dynamic programming. New York: IEEE Press; 2004. pp. 64–94.
21. Fan J, Li R. Statistical challenges with high dimensionality: feature selection in knowledge discovery. In: Sanz-Solé M, Soria J Varona JL, *et al*. editors. Volume III, Proceedings of international congress of mathematicians. Zurich: European Mathematical Society Publishing House; 2006. pp. 595–622.
22. Fan J, Gijbels I. Local polynomial modeling and its applications. London: Chapman and Hall; 1996.
23. de Farias D, Van Roy B. On constraint sampling in the linear programming approach to approximate dynamic programming. Math Oper Res 2004;29(3):462–478.
24. George A, Powell WB. Adaptive step sizes for recursive estimation with applications in approximate dynamic programming. Mach Learn 2006;65(1):167–198.