What you should know about approximate dynamic programming

Warren B. Powell
Department of Operations Research and Financial Engineering
Princeton University, Princeton, NJ 08544

December 16, 2008

Abstract

Approximate dynamic programming (ADP) is a broad umbrella for a modeling and algorithmic strategy for solving problems that are sometimes large and complex, and are usually (but not always) stochastic. It is most often presented as a method for overcoming the classic curse of dimensionality that is well-known to plague the use of Bellman's equation. For many problems, there are actually up to three curses of dimensionality. But the richer message of approximate dynamic programming is learning what to learn, and how to learn it, to make better decisions over time. This article provides a brief review of approximate dynamic programming, without intending to be a complete tutorial. Instead, our goal is to provide a broader perspective of ADP and how it should be approached from the perspective on different problem classes.

There is a wide range of problems that involve making decisions over time, usually in the presence of different forms of uncertainty. Using the vocabulary of operations research (which hardly owns this broad problem class), we might describe our system as being in a state S_t, from which we take an action x_t and then observe new information W_{t+1} which takes us to a new state S_{t+1}. We can represent our rule (or policy) for making a decision using the function $X^\pi(S_t)$. Most of the time, we can assume that we have a transition function $S^M(\cdot)$ (also known as the "system model" or sometimes just "model") which describes how a system evolves from S_t to S_{t+1}. The dynamics of our problem can then be described using

$$x_t = X^\pi(S_t), \tag{1}$$

$$S_{t+1} = S^M(S_t, x_t, W_{t+1}). \tag{2}$$

We assume that there is a choice of decision functions X^π where $\pi \in \Pi$ designates a particular function or policy (we use decision function and policy interchangeably). After we make a decision x_t, we assume we earn a contribution (cost if we are minimizing) given by $C(S_t, x_t)$ which usually depends on the state. In some settings, the contribution depends on W_{t+1}, in which case it would be written $C(S_t, x_t, W_{t+1})$. Our goal is to find the best policy $\pi \in \Pi$ that solves

$$\sup_{\pi \in \Pi} \mathbb{E} \sum_{t=0}^{T} \gamma^t C(S_t, x_t), \tag{3}$$

where γ is a discount factor. We need an expectation because the information variable W_t is random (at times before time t).

This simple framework covers a wide variety of problems. There has been a desire among some in the research community to design a general solution framework that covers all problems, but we are quite sure that this is going to prove fruitless. What has happened is that there have emerged fairly distinct subcommunities that work on problems that tend to share certain empirical characteristics. Somewhat frustratingly, these communities have also developed different vocabularies and different notational systems. The three major communities are control theory, which includes mainstream engineering (e.g., electrical, mechanical and chemical) as well as economics, artificial intelligence (primarily computer science) and operations research. Operations research spans different subcommunities that work on stochastic optimization, including the Markov decision process community (whose notation has been adopted by the artificial intelligence community), stochastic programming (a subcommunity of math programming that considers uncertainty), and the simulation community, which approaches stochastic optimization from a somewhat different perspective.

These communities have evolved a rich vocabulary reflecting the different domains in which the work takes place. In operations research, the state variable is denoted S_t and is most often understood to represent the physical state of the system. In control theory, a state is x_t which might refer to the physical state of a production plant, the location, velocity and acceleration of an aircraft, or a vector of parameters characterizing a statistical model. We use S_t as our state, x_t as our action, and W_{t+1} as the information arriving between t and $t+1$. In the dynamic programming community, it is customary to use the one-step transition matrix $p(s'|s,x)$ which is the probability that we transition to state s' given that we are currently in state s and take action x. In the controls community, it is more natural to use the transition function $S^M(\cdot)$, which describes the physics of how the state evolves over time.

Given the diversity of applications, it should not be surprising that there is more than one way to approach solving the optimization problem in Equation (3). The most important of these are

Simulation-optimization - Here we assume that our decision function $X^\pi(S_t)$ depends only on what we know now, and makes no attempt to use any sort of forecast of how decisions now might impact the future. These are generally known as myopic policies, and often these depend on a set of parameters. We want to find the set of parameters that perform the best over many realizations of the future. Choosing the best set of parameters is typically done using the tools of stochastic search (Spall (2003)) and the closely related field of simulation optimization (see Fu (2002) and Chang et al. (2007)).

Rolling-horizon procedures - A RHP (sometimes called a receding-horizon procedure) uses either a deterministic or stochastic forecast of future events based on what we know at time t. We then use this forecast to solve a problem that extends over a planning horizon, but only implement the decision for the immediate time period.

Dynamic programming - Dynamic programming makes decisions which use an estimate of the value of states to which an action might take us. The foundation of dynamic programming is Bellman's equation (also known as the Hamilton-Jacobi equations in control theory) which is most typically written (Puterman (1994))

$$V_t(S_t) = \max_{x_t} \big(C(S_t, x_t) + \gamma \sum_{s' \in \mathcal{S}} p(s'|S_t, x_t) V_{t+1}(s') \big). \tag{4}$$

Recognizing that all three approaches can be valid ways of solving a stochastic optimization problem, approximate dynamic programming specifically focuses on using Bellman's equation.

The remainder of this article provides a brief introduction to the very rich field known as approximate dynamic programming (ADP). As of this writing, there are three books dedicated to this topic, each representing different communities. *Neuro-Dynamic Programming* (Bertsekas and Tsitsiklis (1996)) is a primarily theoretical treatment of the field using the language of control theory; *Reinforcement Learning* (Sutton and Barto (1998)) describes the field from the perspective of artificial intelligence/computer science; and *Approximate Dynamic Programming* (Powell (2007)) uses the language of operations research, with more emphasis on the high-dimensional problems that typically characterize the problems in this community. Judd (1998) provides a nice discussion of approximations for continuous dynamic programming problems that arise in economics, and Haykin (1999) is an in-depth treatment of neural networks, with a chapter devoted to their use in dynamic programming. We do not have the space to cover all these perspectives, and focus instead on issues that tend to arise in applications from operations research.

1 A brief introduction to ADP

From the seminal work of Bellman (1957) to Puterman (1994) and including numerous authors in between, the field that is typically referred to as Markov decision processes writes Bellman's equation in the form given in Equation (4), which we refer to as the *standard form* of Bellman's equation. For our purposes, it is more convenient (but mathematically equivalent) to write Bellman's equation using the *expectation form* given by

$$V_t(s) = \max_{x_t} \big(C(S_t, x_t) + \gamma \mathbb{E} \{ V_{t+1}(S_{t+1}) | S_t = s \} \big). \tag{5}$$

where $S_{t+1} = S^M(S_t, x_t, W_{t+1})$, and the expectation is over the random variable W_{t+1}. $V_t(S_t)$ is the value function (in control theory this is represented by J and is called the cost-to-go function) which gives the expected value of being in state S_t at time t and following an optimal policy forward. We index the value functions by time which is appropriate for finite-horizon models. We use a finite-horizon model because it makes the modeling of information (and randomness) more explicit, especially when we use the expectation form.

The standard presentation in dynamic programming texts (e.g., Puterman (1994)) is that the state space is discrete and can be represented as $\mathcal{S} = (1, 2, \ldots, |\mathcal{S}|)$. This is known in the artificial intelligence community as a *flat representation* (Boutilier et al. (1999)). The "textbook" solution to dynamic programming assumes $V_{t+1}(s)$ is known and computes $V_t(s)$ for each $s \in \mathcal{S}$. Since this step requires stepping backward through time, this is often referred to as backward dynamic programming.

If S_t is a discrete, scalar variable, enumerating the states is typically not too difficult. But if it is a vector, then the number of states grows exponentially with the number of dimensions. If S_t is continuous (even if it is scalar), then we cannot use this strategy at all. The essence of approximate dynamic programming is to replace the true value function $V_t(S_t)$ with some sort of statistical approximation that we refer to as $\bar{V}_t(S_t)$, an idea that was suggested in Bellman and Dreyfus (1959).

The second step in approximate dynamic programming is that instead of working backward through time (computing the value of being in each state), ADP steps forward in time, although there are different variations which combine stepping forward in time with backward sweeps to update the value of being in a state. Continuing to use our finite-horizon model, we are going to start with a given state S_0 and follow a particular sample path $\omega \in \Omega$. We are going to do this iteratively, so assume we are about to start iteration n. After iteration $n-1$, we have an approximation $\bar{V}_t^{n-1}(S_t)$. We use this approximation to make decisions while we are following the n^{th} sample path ω^n. This means that our decision function is given by

$$X^\pi(S_t^n) = \max_{x_t} \left(C(S_t^n, x_t) + \gamma \mathbb{E} \left\{ \bar{V}_{t+1}^{n-1}(S_{t+1}) | S_t^n \right\} \right). \tag{6}$$

Note that this provides us with a specific definition of what is meant by a policy π. In this setting, the policy is determined by the value function approximation $\bar{V}_{t+1}(S_{t+1})$, which means the policy space Π is the set of all possible value function approximations.

Let x_t^n be the value of x_t that solves this problem, and let

$$\hat{v}_t^n = C(S_t^n, x_t^n) + \gamma \mathbb{E} \left\{ \bar{V}_{t+1}^{n-1}(S_{t+1}) | S_t \right\} \tag{7}$$

be a sample estimate of the value of being in state S_t^n. There are numerous ways to approximate a

```
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 $\omega^n$.
Step 2. For $t = 0, 1, 2, \ldots, T$ do:
        Step 2a. Solve
            $\hat{v}_t^n = \max_{x_t} \left( C_t(S_t^n, x_t) + \gamma \mathbb{E}\{\bar{V}_{t+1}^{n-1}(S_{t+1}) | S_t\} \right)$
            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 $S_{t+1}^n = S^M(S_t^n, x_t^n, W_{t+1}(\omega^n))$.
Step 3. Let $n = n + 1$. If $n < N$, go to step 1.
```

Figure 1: A generic approximate dynamic programming algorithm using a lookup-table representation.

value function, and as a result there are many ways to estimate a value function approximation. But the simplest approximation is known as a lookup table, which means that for each discrete state s, we have an estimate $\bar{V}_t(s)$ which gives the value of being in state s. If we are using a lookup-table representation, we would update our estimate using

$$\bar{V}_t^n(S_t^n) = (1-\alpha_{n-1})\bar{V}_t^{n-1}(S_t^n) + \alpha_{n-1}\hat{v}_t^n. \tag{8}$$

Here, α_{n-1} is known as a stepsize, since Equation (8) can be derived from a type of stochastic optimization algorithm (see Powell (2007), Chapter 6 for a more detailed discussion). More on this later.

Figure 1 summarizes a generic ADP algorithm. Note that it steps forward in time, and at no point does it require that we enumerate all the states in the state space, as is required in classical backward dynamic programming. On the surface, it seems as if we have eliminated the curse of dimensionality.

The algorithm in Figure 1 serves as a reference point for all the research that takes place in the approximate dynamic programming literature. While it is nice to eliminate the need to enumerate states, this algorithm is unlikely to work for any problem that cannot already be solved exactly.

First, while we do not explicitly enumerate all the states, we need an approximation of the value of being in any state that we *might* visit. While this limits us to states that are visited, and any state that might be visited from any state that is actually visited, this can still be a very large number.

Second, this algorithm only updates the values of states that we actually visit. Assume these values are positive, and that our initial approximation uses zero. This means that any state that we do not actually visit will keep a value of zero. Once we visit a state (presumably raising its value), we are more inclined to take actions that might take us back to this state. This means we have to find a way to visit states just to learn their value.

There are other problems. While we may have created the appearance of solving the curse of multidimensional state variables, we still may have a problem with multidimensional information variables. Consider a problem of managing inventories for P product types, where S_{tp} is the number of units of product p. Now let $\hat{D}_{tp}$ be the random demand for product p at time t, where demands may be correlated. Finding the expectation over the multivariate distribution of the vector D_t now becomes computationally intractable. This is the second curse of dimensionality.

Finally, consider what happens when x_t is a vector. Let x_{tp} be the number of products of type p that we are ordering, so x_t is now a vector. If we actually were using a lookup-table representation for a value function, the only way to solve the optimization problem in Equation (6) is to enumerate the action space. When x_t is a vector, the number of potential actions grows just as it does for the state space. This is the third curse of dimensionality. A separate but related issue arises when states, information and actions are continuous. This introduces issues even when these variables are scalar. Interestingly, approximate dynamic programming handles multidimensional discrete variables as if they are continuous.

2 Overcoming the curses of dimensionality

Before we progress too far, it is important to address the three curses of dimensionality, beginning with state variables. We begin by noting that the so-called curse of dimensionality is really an artifact of flat representations. If the state S_t is a multidimensional vector of discrete variables, it is almost never going to be practical to list out all possible combinations of the state variable in a single list (the flat representation). Instead, it is much better to retain the original structure of the state variable, something that the artificial intelligence community refers to as a *factored representation*

(Boutilier et al. (1999)).

State variables can be a mixture of discrete, continuous and categorical values. Assume for the moment that all the elements of the state variable are numerical (discrete or continuous). The first trick that is widely used to overcome multidimensional variables is to simply treat the vector S_t as continuous. For example, imagine that S_{ti} is the number of units of product of type i. We might then approximate the value function using

$$\bar{V}_t(S_t|\theta) = \sum_{i \in \mathcal{I}} \theta_i S_{ti}. \tag{9}$$

This is a very simple approximation that assumes that the behavior of the value function is linear in the number of units of product. The parameter θ_i captures the marginal value of products of type i. Now, with just $|\mathcal{I}|$ parameters, we have a value function that covers the entire state space. Of course, this particular approximation architecture (linear in S_t) may not provide a very good approximation, but it hints at the basic strategy for overcoming the curse of dimensionality. Section 3 deals with value function approximations in more detail.

The second problem is the expectation. There are many real-world problems where the random information is a large vector of prices and demands, making it computationally impossible to compute the expectation exactly. It is possible to approximate the expectation by using a sample, but this can complicate the optimization problem when x_t is a vector.

There are many problems where the decision is fairly simple (what price to charge, what quantity should be ordered), but there are also problems where x_t is a potentially high dimensional vector. How do I allocate my workforce to different assignments around the world? Which drug treatments should I test? When x_t is a vector, then we have to draw on the field of mathematical programming, whether it be linear or nonlinear programming, integer programming, or a messy combinatoric problem that requires your favorite metaheuristic.

An elegant way to circumvent the imbedded expectation is to use a concept called the post-decision state variable. The post-decision state captures the state of the system immediately after we make a decision but before any new information has arrived. This means that the post-decision state is a deterministic function of S_t (also known as the pre-decision state) and the decision x_t.

To our knowledge, the term "post-decision state" was first introduced in Van Roy et al. (1997), but the concept has been widely used, although without much attention given to its importance to handling problems that exhibit vector-valued decision, information and state variables. Judd (1998) refers to the post-decision state as the end-of-period state, while in the reinforcement learning community, it is called the after-state variable (Sutton and Barto (1998)). The decision tree literature has used it extensively when it breaks down problems into decision nodes (which represent pre-decision states) and outcome nodes (which represent post-decision states). We have found that when dealing vector-valued decision, information and state variables (the three curses of dimensionality), the post-decision state variable takes on critical importance.

Since we are using x_t as our decision variable, we let S_t^x represent our post-decision state variable. We assume that we are given a deterministic function $S^{M,x}(S_t, x_t)$ that returns S_t^x. Examples of post-decision state variables include

Tic-tac-toe - The pre-decision state is the board just before we make our move. The post-decision state is the board immediately after we make our move, but before our opponent makes his move.

Inventory problems - Classical inventory problems are described using the equation $S_{t+1} = \max\{0, S_t + x_t - \hat{D}_{t+1}\}$ where S_t is a scalar giving the amount of inventory, x_t is the new product we have just ordered and $\hat{D}_{t+1}$ is the random demand (unknown at time t). S_t is the pre-decision state. We would write $S_t^x = S_t + x_t$ as the post-decision state.

Blood management - Let R_{ti} be the number of units of blood of type i, and let D_{tj} be the demand for blood of type j. For our illustration, assume that unsatisfied demands are lost to our system (they obtain blood supplies elsewhere). Our state variable would be $S_t = (R_t, D_t)$. Let x_{tij} be the number of units of blood supply of type i that are assigned to demands of type j. The leftover blood would be

$$R_{ti}^x = R_{ti} - \sum_j x_{tij}.$$

Since unsatisfied demands are lost, our post-decision state variable is $S_t^x = R_t^x$.

It takes some time to get used to defining the post-decision state. The most important feature is that it has to be a deterministic function of S_t and x_t. This does not prevent us from using a forecast of future information. For example, let $\hat{R}_{t+1,i}$ be a random variable giving us the donations of blood that arrive between t and $t+1$. Now let $\bar{R}_{t,t+1,i}$ be a forecast of what will be donated between t and

$t+1$. Of course, this forecast is known at time t, and is based on information that arrived before time t. We can write the post-decision state variable as

$$R^x_{ti} = R_{ti} - \sum_j x_{tij} + \bar{R}_{t,t+1,i}.$$

Another form of post-decision state variable is $S^x_t = (S_t, x_t)$, which is a concatenation of the pre-decision state and the decision vector. For problems in operations research, this looks hopelessly clumsy. However, this is precisely what is done in a branch of the reinforcement learning community that uses a concept known as Q-learning which focuses on learning the value of being in a state S_t and taking an action x_t.

Once we have defined our post-decision state, we then have to fit our value function approximation around the post-decision state instead of the pre-decision state. Let $\bar{V}^{x,n}_t(S^x_t)$ be our estimate of the value of being in post-decision state after n observations. If we are using a lookup-table representation (one value per state), then instead of using the update given in Equation (8), we would use

$$\bar{V}^{x,n}_{t-1}(S^{x,n}_{t-1}) = (1-\alpha_{n-1})\bar{V}^{x,n-1}_{t-1}(S^{x,n}_{t-1}) + \alpha_{n-1}\hat{v}^n_t. \tag{10}$$

Comparing (8) and (10), we see that the update is almost the same, except that we use $\hat{v}^n_t$ (measured at time t) to update the value function approximation $\bar{V}^{x,n}_{t-1}$ around the *previous* post-decision state variable $S^{x,n}_{t-1}$. We note that the smoothing of $\hat{v}^n_t$ into $\bar{V}^{x,n-1}_{t-1}(\cdot)$ represents the step where we approximate the expectation. This means that we make decisions using

$$X^\pi(S^n_t) = \max_{x_t}\big(C(S^n_t, x_t) + \gamma \bar{V}^{x,n-1}_t(S^x_t)\big). \tag{11}$$

Note that the right hand side of (11) is deterministic (everything is indexed by t). This makes it much easier to use the large library of solvers for deterministic problems. We just have to use some care when designing the value function approximation. For example, if $\max_{x_t} C(S^n_t, x_t)$ is a linear program, then we would ideally like to choose an approximation architecture for $\bar{V}^{x,n-1}_t(S^x_t)$ that does not destroy this structure. If it is a concave nonlinear program (note that we are maximizing), then the value function approximation should retain concavity (and perhaps differentiability). Lookup-table representations can only be used if our search procedure is something like a local search heuristic.

The post-decision state variable provides at least a path for solving problems which can handle vector-valued states, information and decisions. This returns us to the central challenge of approximation dynamic programming which is designing and estimating the value function approximation.

3 Fitting a value function approximation

Central to approximate dynamic programming is the use of an approximation of the value function for making decisions. The holy grail of ADP is to define an approximation strategy that works for any problem, without tuning. Since we have not reached this goal, we tend to seek approximation strategies that work for the broadest possible classes. Below, we describe three very general strategies, but recognize that ADP remains an art form which requires taking advantage of problem structure.

3.1 Multilevel aggregation

Aggregation is a powerful technique that requires perhaps the least amount of problem structure among the family of statistical tools available. We assume that we have a state variable S_t that is usually multidimensional, and may include discrete, continuous and categorial elements. We are going to make no assumptions about the behavior of the value function itself (e.g., concavity, monotonicity or even continuity). But we are going to assume that there is a natural way to aggregate the state space into successively coarser representations.

Let $\mathcal{S}$ be the original state space, and let $\mathcal{S}^{(g)}$ be the state space at the g^{th} level of aggregation. Let G^g be a function mapping $\mathcal{S}$ to $\mathcal{S}^{(g)}$, and let $\mathcal{G}$ be the set of indices corresponding to the levels of aggregation. We assume that $g = 0$ is the most disaggregate level, but even at this level we assume the state space has been aggregated into a set of discrete values. But the number of possible aggregated values may be too large to enumerate (even at higher levels of aggregation).

Our value function approximation estimates a single value for a discrete state s at each level of aggregation, but these values are only updated for states that we actually visit. Assume that we are at time t, iteration n and we are visiting state S_t^n where the value of being in this state is given by $\hat{v}_t^n$ (see Equation (7)). We let $\bar{V}_t^{(g,n)}(s)$ be an estimate of the value of being in state s at the g^{th} level of aggregation. This is updated using

$$\bar{V}_t^{(g,n)}(s) \quad = \quad \begin{cases} (1-\alpha_{n-1})\bar{V}_t^{(g,n-1)}(s) + \alpha_{n-1}\hat{v}_t^n & \text{if } G^g(S_t^n) = s, \\ \bar{V}_t^{(g,n)}(s^{(g)}) & \text{otherwise.} \end{cases} \tag{12}$$

We note that the stepsize α_{n-1} is not just a function of the iteration counter, but is typically a function of the number of times that we have visited a particular state at a particular level of aggregation.

Aggregation has been widely used to overcome the problem of large state spaces (see Rogers et al. (1991) for a review). The most common strategy is to choose a single level of aggregation, which raises the issue of determining the right level of aggregation. The problem with aggregation is that the right level changes with the number of times you observe a set of states. Some authors have suggested changing the level of aggregation with the number of iterations (Bertsekas and Castanon (1989), Luus (2000)).

A more flexible strategy is to use a weighted sum of estimates from different levels of aggregation, given by

$$\bar{V}^n(s) = \sum_{g\in\mathcal{G}} w^{(g,n)}(s)\bar{V}^{(g,n)}(s). \tag{13}$$

Note that the weight we put on the g^{th} level of aggregation depends on both the iteration counter and the state. Since there are many states, this means that we have to estimate a large number of weights. A simple strategy is to weight estimates in inverse proportion to the total variation of the error, given by

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

where $\hat{\sigma}^2(s))^{(g,n)}$ is an estimate of the variance of the observations of state s after n iterations, at the g^{th} level of aggregation. $N^{g,n}(s)$ is the number of times we have observed state s at the g^{th} level of aggregation, and $\bar{\mu}^{(g,n)}(s)$ is an estimate of the bias between the value at the g^{th} level of aggregation, and the more disaggregate level $(g = 0)$. $\hat{\sigma}^2(s)^{(g,n)}/N^{g,n}(s)$ is an estimate of the variance of $\bar{V}^{(g,n)}(s)$ which declines as we have more observations of state s. If there are states that we do not visit very often, the variance tends to remain high because of this term. If we visit a state many times, the total variation may decline considerably, or it may remain high if our estimate of the bias remains high. The weights are then normalized so they sum to one. The computations are quite simple, and requires only that we accumulate some statistics for every state we visit, at each level of aggregation (see Powell (2007), Section 7.1 for a complete presentation).

This strategy is very easy to implement, and scales to very large problems. If a state has never

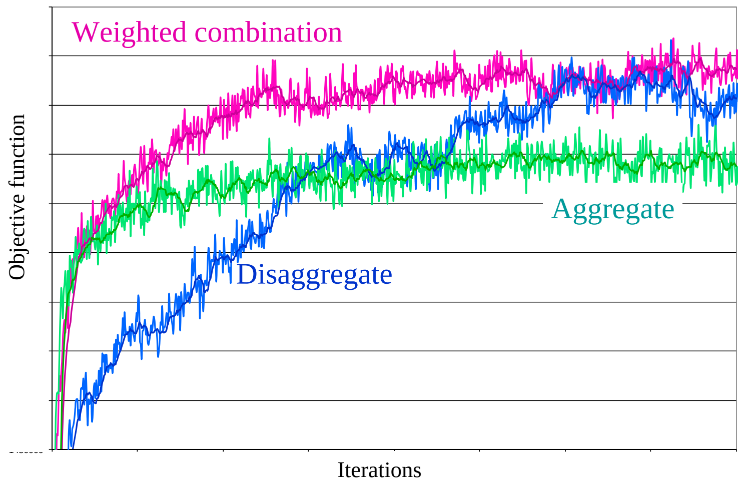

Figure 2: The weights given to two levels of aggregation, showing how they put increasingly more weight on disaggregate estimates as more information becomes available.

been observed at a particular level of aggregation, then it is simply assigned a weight of zero. This logic tends to put higher weights on more aggregate estimates early in the process. As there are more observations for certain states, the weight put on the disaggregate levels increases, although generally not for all states. The effect of this logic is shown in Figure 2, which displays the objective function if the value function is approximated at a single aggregate level, a single disaggregate level, or using a weighted combination of both levels of aggregation with weights given by (14). The aggregate value function gives faster initial convergence, while the disaggregate estimates give a better overall solution, but the weighted combination gives the best results overall.

3.2 Basis functions

Perhaps the most widely publicized strategy for approximating value functions is to capture important quantities from the state variable, and build an approximation around these quantities. These quantities are captured using *basis functions*, $\phi_f(s)$, $f \in \mathcal{F}$ where f is referred to as a feature, and $\phi_f(s)$ is a function that captures some particular quality from the state variable that is felt to provide explanatory power. We might then write a value function using

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

This is referred to as a linear model because it is linear in the parameters. The basis functions may capture different types of nonlinear behaviors. Returning to our product management application where S_{ti} is the number of units of product type i, we might start with basis functions $\phi_f(S_t) = (S_{ti})^2$ (one feature per product type), and $\phi_f(S_t) = S_{ti}S_{tj}$ (one feature per pair of product types).

The challenge now is estimating the parameter vector θ. A simple strategy is to use a stochastic gradient algorithm, given by

$$\begin{aligned} \theta^n &= \theta^{n-1} - \alpha_{n-1}\big(\bar{V}_t(S_t^n|\theta^{n-1}) - \hat{v}_t^n\big)\nabla_\theta \bar{V}_t(S_t^n|\theta^{n-1}) \\ &= \theta^{n-1} - \alpha_{n-1}\big(\bar{V}(S_t^n|\theta^{n-1}) - \hat{v}^n(S_t^n)\big)\begin{pmatrix} \phi_1(S_t^n) \\ \phi_2(S_t^n) \\ \vdots \\ \phi_F(S_t^n) \end{pmatrix}. \end{aligned} \tag{15}$$

This method requires that we start with an initial estimate θ^0, after which θ is updated following each observation $\hat{v}_t^n$, typically calculated using (7). This method of updating θ is popular because it is simple, but it can be unstable. One challenge is that the stepsize α has to be scaled to handle the difference in units between θ and the gradient, but it may be necessary to use different scaling factors for each feature. More effective algorithms use recursive least squares (described in Bertsekas and Tsitsiklis (1996) and Powell (2007), Chapter 7). The Kalman filter is a particularly powerful algorithmic framework, although it is more difficult to implement (see Choi and Van Roy (2006) for a nice discussion).

Basis functions are appealing because of their relative simplicity, but care has to be put into the design of the features. It is tempting to make up a large number of functions $\phi_f(S)$ and throw them into the model, but putting some care into the choice of functions is generally time well spent. The challenge with any choice of basis functions is demonstrating that they actually contribute to the quality of the solution.

3.3 Other statistical methods

It is important to draw on the widest possible range of techniques from statistics and machine learning. An excellent reference is Hastie et al. (2001), which covers a broad range of methods for statistical learning. Bertsekas and Tsitsiklis (1996) and Haykin (1999) provide in-depth discussions on the use of neural networks, and Chapters 6 and 12 in Judd (1998) provide a very thorough overview of approximation methods for continuous functions.

A major issue that arises when adapting these methods to approximate dynamic programming is the importance of using recursive methods, and developing methods that produce "good" approximations as quickly as possible. The value of being in a state depends on the quality of decisions that are being made which in turn depend on the quality of the value function approximation. A poor initial approximation can yield poor decisions which bias the estimates $\hat{v}_t^n$ being used to estimate the value function. Not surprisingly, it is easy to develop ADP algorithms that either do not converge at all, or converge to very poor solutions.

3.4 Approximations for resource allocation

Aggregation and basis functions are two methods that have the advantage of being quite general. But with ADP, it is especially important to take advantage of problem structure. One very broad problem class can be referred to as "resource allocation" where decisions involve the management of people, equipment, energy and agricultural commodities, consumer goods and money. Examples of these problems arise in transportation (allocation of vehicles), supply chain management, financial portfolios, vaccine distribution, emergency response management and sensor management (to name just a handful).

A simple but flexible mathematical model consists of

$$\begin{aligned} a &= \text{vector of attributes describing a resource, where } a \in \mathcal{A}, \\ R_{ta} &= \text{number of resources available with attribute } a \text{ at time } t, \\ R_t &= (R_{ta})_{a \in \mathcal{A}}, \end{aligned}$$

$$\begin{aligned} d &= \text{a type of decision that acts on a single type of resource, where } d \in \mathcal{D}, \\ x_{tad} &= \text{the number of times we act on resources with attribute } a \text{ using a decision of type } d \text{ at time } t, \\ x_t &= (x_{tad})_{a \in \mathcal{A}, d \in \mathcal{D}}, \\ C(R_t, x_t) &= \text{contribution earned by implementing decision vector } x_t \text{ given the resource state vector } R_t. \end{aligned}$$

Resources are often reusable, which means if we act on a resource with attribute a using decision d, we obtain a resource with attribute a' (this can be random, but this is not central to our discussion). Let $R^M(R_t, x_t, W_{t+1})$ be the *resource transition function*, comparable to our state transition function we defined earlier. We can also assume that we have a post-decision resource transition function that returns $R_t^x = R^{M,x}(R_t, x_t)$. Further assume for illustrative purposes that we are using a linear value function approximation of the form

$$\bar{V}_t(R_t^x) = \sum_{a\in\mathcal{A}} \bar{v}_{ta} R_{ta}.$$

If we are in state R_t^n, we would make a decision using

$$x_t^n = \arg\max_{x_t} \big(C(R_t^n, x_t) + \gamma \bar{V}_t^{n-1}(R_t^x) \big). \tag{16}$$

This problem generally has to be solved subject to constraints that include

$$\begin{aligned} \sum_{d\in\mathcal{D}} x_{tad} &= R_{ta} \qquad (17) \\ x_{tad} &\geq 0. \end{aligned}$$

Earlier (Equation (7)) we used the objective function value $\hat{v}_t^n$ to update our estimate of the value of being in a state. When we are solving resource allocation problems, a much more effective technique recognizes that we are not so much interested in the value of being in a state as we are interested in the marginal value of a resource. Furthermore, we can obtain estimates of the marginal value of each type of resource from the dual variables of the flow conservation constraints (17). Let $\hat{v}_{ta}^n$ be the dual variable associated with Equation (17). Note that this returns a vector $(\hat{v}_{ta}^n)_{a\in\mathcal{A}}$ rather than a single scalar, as was the case before. If we are in fact using a linear value function approximation, we can update the slopes using

$$\bar{v}_{t-1,a}^n = (1-\alpha_{n-1})\bar{v}_{t-1,a}^{n-1} + \alpha_{n-1}\hat{v}_{ta}^n.$$

Updating these marginal values using dual variables is extremely powerful. Not only do we get an entire vector of marginal values, but we also get information on slopes (which is what we really need) rather than an estimate of the value of being in a state. This logic can be extended to a range of value function approximation strategies, including piecewise-linear separable, continuously differentiable concave functions, and multidimensional cuts (see Powell (2007), Chapter 11 for a more complete discussion). We note that this general strategy includes techniques such as stochastic decomposition (see Higle and Sen (1991), Birge and Louveaux (1997)) which are traditionally thought of as belonging to the field of stochastic programming.

Resource allocation is one problem class where the dimensionality of x_t can be particularly large, and there may also be integrality requirements. For these problems, it is particularly important to design value function approximations which work well with commercial solvers or particular algorithmic strategy suited to the problem.

4 Other algorithmic issues

Assuming that you have found a way to approximate the value function that captures the important characteristics of your problem, you now know enough to get yourself into some serious trouble. Even applied to simple problems, the techniques above can work well, but may work very poorly. You would not be the first person to run some experiments and conclude that "approximate dynamic programming does not work." Below, we address two issues that can be described as how fast we learn, and what we learn.

4.1 The challenge of stepsizes

Up to now, we have routinely smoothed old estimates with new observations to produce updated estimates (see Equation (8)). This smoothing is done with a quantity α_{n-1} that we have referred to as a stepsize. For the purposes of our discussion, we assume that we are using the stepsize to smooth between old and new estimates (as in (8)), where we can assume that the stepsize is between 0 and 1, rather than in an equation such as (15), where the stepsize has to perform a scaling function.

The smoothing step in Equation (8) is needed only because we have to resort to Monte Carlo sampling to compute $\hat{v}_t^n$. This would not be necessary if we could compute the expectation in Bellman's Equation (4) (or (5)). If we could compute the expectation, we would use $\alpha_{n-1} = 1$, giving us traditional value iteration (Puterman (1994)). In such a setting, the value of a state typically increases over time, representing the approximation of the summation in Equation (3). However, we generally depend on Monte Carlo sampling when we are not able to compute the expectation, as is typically the case.

Imagine now that we can only measure contributions with uncertainty. The uncertainty might be in the contribution function itself, or in the constraints that govern the choice of a decision. Further imagine that the discount factor $\gamma = 0$, which means we do not even care about the downstream impact of decisions made now. If this were the case, the best possible stepsize would be $\alpha_{n-1} = 1/n$, which is equivalent to simply averaging different sample observations. The so-called "one over n" stepsize rule is well-known to satisfy important conditions for convergence, but it can also produce a rate of convergence so slow that it should never be used (see Section 9.3 of Powell (2007)). Care has to be used when applying recursive least squares to estimate the parameter vector θ, since this uses an implicit $1/n$ stepsize (see Section 7.3.3 of Powell (2007) for a way of overcoming this).

Choosing the best stepsize, then, requires performing a difficult balancing act between the need to add up contributions over an extended horizon against the need to smooth out the noise from sampling error. There is a wide range of stepsize formulas that have been proposed, coming from different communities (see George and Powell (2006) for a review), but our work has suggested that it is useful to consider three classes of formulas:

Constant stepsize - Start by simply choosing a constant stepsize $\alpha_n = \alpha_0$, and then experiment with α_0.

Harmonic stepsize sequence - The problem with a constant stepsize is that it does not decline, which means you will never get strong convergence results. A good alternative is

$$\alpha_{n-1} = \frac{a}{a + n^{\beta} - 1}$$

Start with $\beta = 1$, but try $\beta = 0.7$. Choose a after performing experiments with a constant stepsize. There are problems where 100 iterations produces very good results, while others need 100,000 iterations (or millions). Choose a so that you get a "small" stepsize as the algorithm appears to be converging, based on your experiments with a constant stepsize. There are problems where $a = 5$ works well, while other problems need $a = 5000$.

Stochastic stepsizes - These are stepsizes that adjust themselves to the actual behavior of the algorithm. A strategy called the bias-adjusted Kalman filter (BAKF) stepsize rule minimizes the total variation between the actual and predicted value functions, given by

$$\alpha_{n-1} = 1 - \frac{\sigma^2}{(1 + \lambda^{n-1})\sigma^2 + (\beta^n)^2} \tag{18}$$

where λ^n, σ^2 and β^n are computed using simple recursions (see Powell (2007), section 7.1). Because σ^2 and β^n have to be estimated from data, the stepsize is stochastic. If the noise $\sigma^2 = 0$, we get $\alpha = 1$. As σ^2 grows (relative to the bias β^n, the stepsizes tends to $1/n$.

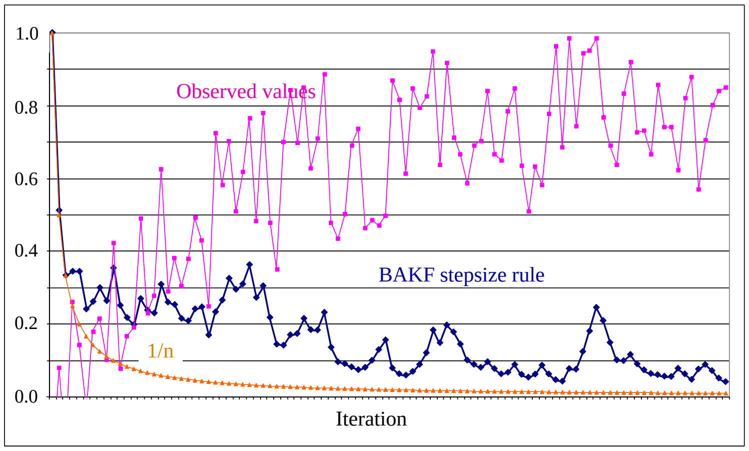

Figure 3: The stepsizes produced by the BAKF stepsize rule and the $1/n$ rule for a typical nonstationary series.

Stochastic stepsize formulas are appealing because they adapt to the data, avoiding the need to tune parameters such as a and β. But they do not work well when there is too much noise. Figure 3 illustrates the BAKF stepsize rule (18) for a nonstationary series typical of approximate dynamic programming. The stepsize is initially larger, declining as the data stabilizes. Note that the stepsizes never fall below $1/n$, which is critical to avoid stalling. But if the data is very noisy, what can happen is that the stepsize can be fooled into thinking that the underlying signal is changing, producing a larger stepsize when we really need a smaller stepsize.

4.2 Exploration vs. exploitation

One of the best-known challenges in approximate dynamic programming is famously referred to as the exploration vs. exploitation problem. Throughout our presentation (as in the algorithm in Figure 1), we have presented ADP as a procedure where we find an action x_t^n, and then use this to determine the next state. For most problems, this is not going to work, although the issues depend very much on the nature of how you are approximating the value function.

Imagine that you are using a lookup table, and that you only update the value of a state if you visit the state. Further assume that you initialize the value of being in a state to a number that is lower than what you would expect in the limit. Each time you visit a state, the value of being in that state tends to rise, producing a situation where you are biased toward revisiting states that you have already visited. Quickly, the procedure can become stuck cycling between a relatively small number of states. If values are initialized too high, the algorithm tends to endlessly explore.

Striking a balance between exploration and exploitation remains one of the fundamentally unsolved problems in approximate dynamic programming (and since ADP mimics life, we would argue that this stands alongside one of the unsolved problems of life). There are simple heuristics that are often used. For example, before choosing the state to visit next, you might flip a coin. With probability ρ, you choose the next state at random. Otherwise, you visit the state determined by the action x_t^n. This works well on small problems, but again, if the state space is large, an exploration step has almost no meaning.

We refer the reader to Chapter 10 of Powell (2007) for a more in-depth discussion of strategies to balance exploration and exploitation, as well as the strategies for state sampling that are presented in Sutton and Barto (1998). However, we feel that this general problem area represents many opportunities for research.

4.3 Evaluating an ADP strategy

The last challenge in the designing of an ADP algorithm is deciding how to evaluate it. While there is no single answer to the question of how to do this, it is very important that some sort of benchmark be prepared, since it is possible for an improperly ADP algorithm to work very poorly. Some strategies that might be considered include:

Compare to an optimal MDP - In some cases, it is possible to simplify a problem (but not to the point where it becomes meaningless) so that it can be solved optimally as a standard backward Markov decision process (Puterman (1994)). If this is possible, then apply the ADP algorithm, designed for the more complicated problem, to this simpler problem. If the ADP reasonably approximates the optimal solution, then we have more confidence that it is doing a good job on the real problem.

Compare to an optimal deterministic problem - In some cases, a deterministic version of the problem is meaningful and can be solved optimally using a commercial solver. ADP can handle uncertainty, but it should also be possible to apply to a deterministic version of the same problem (Topaloglu and Powell (2006) and Godfrey and Powell (2002) provide examples of this type of comparison).

Comparison to a myopic or rolling horizon policy - We opened this article with a description of three broad strategies for solving stochastic optimization problems. Sometimes, the best

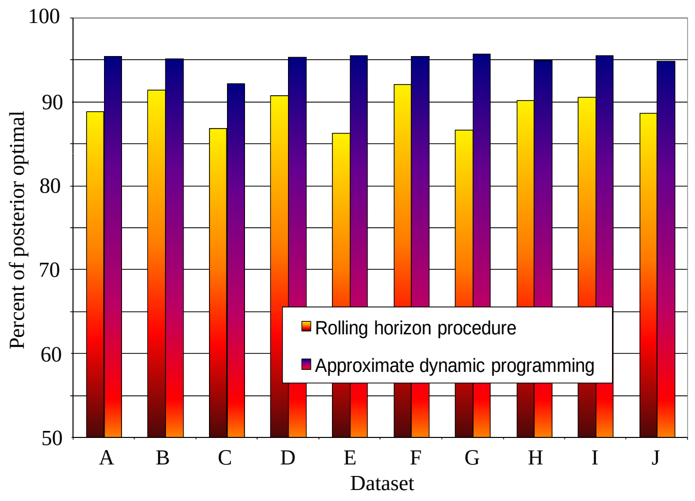

Figure 4: The objective function produced using approximate dynamic programming and a rolling horizon procedure compared against a posterior bound for 10 different datasets, from Topaloglu and Powell (2006).

competition will be one of these other strategies.

Figure 4 illustrates the evaluation of policy based on approximate dynamic programming where it is compared to a rolling horizon procedure. Both methods are reported relative to a posterior bound where an optimal solution is computed after all random quantities become known. Of course, this is not possible with all problems, but it is a useful benchmark when available.

5 Applications and implementations

Approximate dynamic programming is an exceptionally powerful modeling and algorithmic strategy that makes it possible to design practical algorithms for a wide range of complex, industrial-strength problems. This probably explains its broad appeal with the computer science and control theory communities under names that include reinforcement learning and neuro-dynamic programming. Our own personal experience began with projects in freight transportation which has produced practical tools that have been implemented and adopted in industry. These applications include managing freight cars for Norfolk Southern Railroad (Powell and Topaloglu (2005)), planning drivers for Schneider National (Simao et al. (2008)), managing high-value spare parts for Embraer (Simao and Powell (2009)), and optimizing locomotives for Norfolk Southern. These are production applications, adopted by industry.

However, approximate dynamic programming is also famous for its laboratory successes but field failures, not unlike other areas of science and technology. It is not hard to find people who have tried ADP, only to report that it did not work. Others complain that it requires substantial tuning. These complaints are valid, but can be avoided if care is given to the following issues:

a) ADP is a powerful modeling and algorithmic technology. Not all problems need this. If you can solve your problem using policy optimization (tuning a myopic policy), then this is usually going to work better, but this typically only works well when you can obtain good solutions using a relatively simple policy.

b) It is useful to understand what behavior you are looking for from a value function. For example, if you set the value function to zero, what is wrong with the solution? As you design a value function approximation, be sure that it directly addresses whatever issue you raise.

c) The value function approximation should capture the structure of your problem. Blindly making up basis functions is unlikely to work. You have to have a good understanding of the important elements of the state variable that impact the future in a way that you feel is important.

d) If you have a resource allocation problem (which describes a large number of problems in operations research), use approximations that reflect the unique behaviors of resource allocation problems. In particular, make sure you take advantage of derivative information where available.

e) The wrong stepsize formula can ruin an ADP algorithm. If the stepsizes decrease too quickly, the algorithm can appear to converge when in fact it is far from the correct solution. A stepsize that is too large can produce unstable behavior. This is particularly problematic when using basis functions where a small change in the regression vector θ can change the entire value function approximation. These updates have to be carefully designed.

f) A topic that we only touched on is the exploration vs. exploitation problem (Chapter 10 of Powell (2007)). The importance of this issue is problem-dependent, and for many problems remains unresolved.

ADP seems to work best when there is natural problem structure that guides the choice of the value function approximation, but this alone is not enough. Our work in resource allocation problems work well because the value function is naturally concave (when posed as a maximization problem), which also avoids issues associated with exploration (the algorithm naturally seeks out the maximum of the function).

The good news is that ADP solves problems the way people do, which is a reason why it has been so popular in the artificial intelligence community. Getting ADP to work well teaches you how to think about a problem. The most important dimension of ADP is "learning how to learn," and as a result the process of getting approximate dynamic programming to work can be a rewarding educational experience.

Acknowledgements

This research was supported in part by grant AFOSR-F49620-93-1-0098 from the Air Force Office of Scientific Research.

References

Bellman, R. (1957), *Dynamic Programming*, Princeton University Press, Princeton.

Bellman, R. and Dreyfus, S. (1959), 'Functional Approximations and Dynamic Programming', *Mathematical Tables and Other Aids to Computation* **13**(68), 247–251.

Bertsekas, D. and Castanon, D. (1989), 'Adaptive aggregation methods for infinite horizon dynamic programming', *IEEE Transactions on Automatic Control* **34**(6), 589–598.

Bertsekas, D. and Tsitsiklis, J. (1996), *Neuro-Dynamic Programming*, Athena Scientific, Belmont, MA.

Birge, J. and Louveaux, F. (1997), *Introduction to Stochastic Programming*, Springer-Verlag, New York.

Boutilier, C., Dean, T. and Hanks, S. (1999), 'Decision-theoretic planning: Structural assumptions and computational leverage', *Journal of Artificial Intelligence Research* **11**, 1–94.

Chang, H., Fu, M., Hu, J. and Marcus, S. (2007), *Simulation-Based Algorithms for Markov Decision Processes*, Springer, Berlin.

Choi, D. P. and Van Roy, B. (2006), 'A generalized Kalman filter for fixed point approximation and efficient temporal-difference learning', *Discrete Event Dynamic Systems* **16**, 207–239.

Fu, M. (2002), 'Optimization for simulation: Theory vs. practice', *INFORMS Journal on Computing* **14**(3), 192–215.

George, A. and Powell, W. B. (2006), 'Adaptive stepsizes for recursive estimation with applications in approximate dynamic programming', *Machine Learning* **65**(1), 167–198.

Godfrey, G. and Powell, W. B. (2002), 'An adaptive, dynamic programming algorithm for stochastic resource allocation problems I: Single period travel times', *Transportation Science* **36**(1), 21–39.

Hastie, T., Tibshirani, R. and Friedman, J. (2001), *The Elements of Statistical Learning*, Springer series in Statistics, New York, NY.

Haykin, S. (1999), *Neural Networks: A Comprehensive Foundation*, Prentice Hall.

Higle, J. and Sen, S. (1991), 'Stochastic decomposition: An algorithm for two stage linear programs with recourse', *Mathematics of Operations Research* **16**(3), 650–669.

Judd, K. (1998), *Numerical Methods in Economics*, MIT Press.

Luus, R. (2000), *Iterative Dynamic Programming*, Chapman and Hall/CRC, New York.

Powell, W. B. (2007), *Approximate Dynamic Programming: Solving the curses of dimensionality*, John Wiley and Sons, New York.

Powell, W. B. and Topaloglu, H. (2005), Fleet management, *in* S. Wallace and W. Ziemba, eds, 'Applications of Stochastic Programming', Math Programming Society - SIAM Series in Optimization, Philadelphia.

Puterman, M. L. (1994), *Markov Decision Processes*, John Wiley and Sons, New York.

Rogers, D., Plante, R., Wong, R. and Evans, J. (1991), 'Aggregation and disaggregation techniques and methodology in optimization', *Operations Research* **39**(4), 553–582.

Simao, H. P. and Powell, W. B. (2009), 'Approximate dynamic programming for management of high value spare parts', *J. of Manufacturing Technology Management* **20**(9), 00–00.

Simao, H. P., Day, J., George, A. P., Gifford, T., Nienow, J. and Powell, W. B. (2008), 'An approximate dynamic programming algorithm for large-scale fleet management: A case application', *Transportation Science.*

Spall, J. C. (2003), *Introduction to Stochastic Search and Optimization: Estimation, Simulation and Control*, John Wiley and Sons, Hoboken, NJ.

Sutton, R. and Barto, A. (1998), *Reinforcement Learning*, The MIT Press, Cambridge, Massachusetts.

Topaloglu, H. and Powell, W. B. (2006), 'Dynamic programming approximations for stochastic, time-staged integer multicommodity flow problems', *Informs Journal on Computing* **18**(1), 31–42.

Van Roy, B., Bertsekas, D. P., Lee, Y. and Tsitsiklis, J. N. (1997), A neuro-dynamic programming approach to retailer inventory management, *in* 'Proceedings of the IEEE Conference on Decision and Control', Vol. 4, pp. 4052–4057.