APPROXIMATE DYNAMIC PROGRAMMING: LESSONS FROM THE FIELD

Warren B. Powell

Department of Operations Research and Financial Engineering
Princeton University
Princeton, NJ 08544, U.S.A.

ABSTRACT

Approximate dynamic programming is emerging as a powerful tool for certain classes of multistage stochastic, dynamic problems that arise in operations research. It has been applied to a wide range of problems spanning complex financial management problems, dynamic routing and scheduling, machine scheduling, energy management, health resource management, and very large-scale fleet management problems. It offers a modeling framework that is extremely flexible, making it possible to combine the strengths of simulation with the intelligence of optimization. Yet it remains a sometimes frustrating algorithmic strategy which requires considerable intuition into the structure of a problem. There are a number of algorithmic choices that have to be made in the design of a complete ADP algorithm. This tutorial describes the author's experiences with many of these choices in the course of solving a wide range of problems.

1 INTRODUCTION

This is my third in a series of tutorials on approximate dynamic programming that I have given at the Winter Simulation Conference. In (Powell 2005), I described approximate dynamic programming as a method of making intelligent decisions within a simulation. For simulation problems where there is a need to optimize *over time* (that is, decisions now have to consider their impact on the future), ADP offers a powerful framework for calculating the impact of a decision on the future, and using this measurement to make better decisions. In this view, ADP is a form of "optimizing simulator." It is important not to confuse this view with the more familiar simulation-optimization community which uses a decision rule governed by one or more parameters which then need to be chosen optimally.

In my second tutorial (Powell 2006), I focused on ADP as a method for solving high-dimensional dynamic programming problems that suffer from the three curses of

dimensionality: the state variable, exogenous information and the decision variable. A major algorithmic strategy for these problems involves fitting the value function around the post-decision state variable, which measures the state of the system after a decision is made but before new information arrives. This means that the value function is a deterministic function of the state and action, a feature that is very important in the use of scalable optimization algorithms. In addition to this tutorial, my book on approximate dynamic programming (Powell 2007) appeared in 2007, which is kind of ultimate tutorial, covering all these issues in far greater depth than is possible in a short tutorial article.

In this tutorial, I am going to focus on the behind-the-scenes issues that are often not reported in the research literature. In some cases, I will reinforce ideas that have been presented in my book, but some of the topics are not even covered there (in part because of experiences that have occurred in the last year).

2 Three perspectives of ADP

As a result of its name, approximate dynamic programming is typically viewed as a method for solving complex dynamic programs. While this is true, it hides the breadth of its range of applications. We have found that ADP can be of use in three very distinct methodological communities:

a)   Large-scale (deterministic) optimization. Our own work in ADP got its start solving very large optimization problems arising in the context of freight transportation. We have found that commercial packages such as Cplex can handle very large static problems, but struggle when a time dimension is introduced. We have used ADP successfully to break problems arising in the largest freight transportation companies into components that Cplex handles very easily. In this setting, ADP is simply a decomposition strategy that breaks problems with long horizons into a series of shorter problems. The
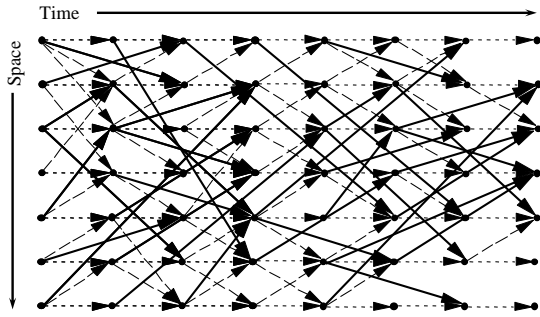
Figure 1: Illustration of time-space network

Figure 2: Illustration of a single subproblem

fact that ADP can also handle uncertainty in this context is a nice byproduct that we often exploit.

b) Making simulations intelligent. There are many stochastic, dynamic problems that are solved using myopic policies where decisions at time t ignore their impact on the future. ADP provides a method for capturing the impact of these decisions on the future, and then communicating this impact backward so that decisions can be made more intelligently. The result is not just higher quality decisions, but also more realistic decisions, since humans are actually quite good at anticipating downstream impacts.

c) Solving complex dynamic programs. It is sometimes natural to formulate a problem as a dynamic program, only to find that the resulting problem is computationally impossible. ADP offers a rich set of algorithmic strategies for providing good solutions to otherwise intractable stochastic, dynamic programs.

Figure 1 is an illustration of a typical time-space network that is often used in dynamic resource allocation problems. In large-scale applications such as those that arise in freight transportation, these graphs become exceptionally large (millions of "space" nodes, billions of links). Instead of solving the entire problem, ADP solves sequences of smaller subproblems such as that shown in figure 2. We have found that over short time horizons, Cplex can handle what would normally be described as very difficult integer programs, even for very large-scale problems. But we have to estimate some sort of approximation to capture the impact of decisions now on the future. This is where approximate dynamic programming comes in.

There are many dynamic applications where standard practice is to simulate a myopic policy. In dynamic programming, a policy is any rule for making decisions. A myopic policy is a rule that ignores the impact of a decision now on the future. Let x_t be a decision (how much to order,
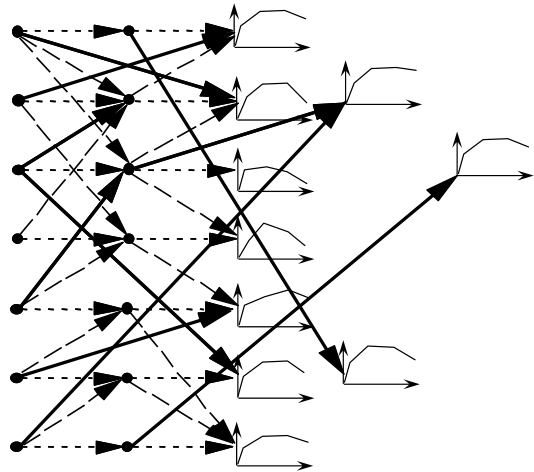
what machine to assign a job to) or vector of decisions (which drivers should handle which loads, what types of energy resources should we use) at time t. Now let $X^\pi(S_t)$ be a function that determines a decision given the information in the state variable S_t. $X^\pi(S_t)$ is sometimes called a decision function, a decision rule or simply a policy. We assume there may be a family of policies, so we let $\pi \in \Pi$ index a set of policies from the set Π.

X^π might be the optimization problem in figure 2 without the nonlinear functional approximations. We might write this policy as

$$X^\pi(S_t) = \arg\max_{x \in \mathcal{X}_t} c_t x_t$$

where $\mathcal{X}_t$ is a feasible region. After we make our decision (either by solving a math programming, or any other method), we update our system state variable S_t using

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

where $S^M(\cdot)$ is the transition function (also known as the system model), and $W_{t+1}(\omega)$ is the information that becomes available between t and $t+1$ when we are following sample path ω.

Approximate dynamic programming would try to improve this myopic policy by replacing it with

$$X^\pi(S_t) = \arg\max_{x \in \mathcal{X}_t} c_t x_t + \bar{V}_t(x)$$

where $\bar{V}_t(x)$ is some sort of approximation that captures the impact of decisions now, x_t, on the future.

Finally, if we are trying to solve a dynamic program, we have probably written out Bellman's equation as

$$V_t(S_t) = \max_{x \in \mathcal{X}_t} \left(C(S_t, x_t) + \gamma \mathbb{E}\left\{ V_{t+1}(S_{t+1}) | S_t \right\} \right). \qquad (2)$$

Often, the first problem people encounter in realistic problems is that we cannot compute $V_t(S_t)$. Standard textbooks in Markov decision processes ((Puterman 1994) is a good reference) assume that the state variable is discrete with states $s = (1, 2, \ldots, |\mathcal{S}|)$. The problem is that S_t might be a vector, and it might be continuous. Even if it is discrete, if S_t is a vector the number of possible states grows extremely quickly. Problems with as few as four or five dimensions can be computationally intractable (this is the famous "curse of dimensionality"). Approximate dynamic programming proceeds by replacing $V_{t+1}(S_{t+1})$ with an approximation $\bar{V}_{t+1}(S_{t+1})$, which might be discrete or continuous (even if S_t is discrete). The body of literature for approximating the value function effectively draws on the entire field of statistics and machine learning (see chapters 6, 7 and 11 of (Powell 2007) for an introduction).

3   A basic ADP algorithm

There are many variations of approximate dynamic programming algorithms. Figure 3 describes a basic ADP procedure which illustrates several key elements of most ADP procedures. First, the algorithm steps forward in time, simulating a sample path. In classical dynamic programming, we proceed by stepping backward in time, where we have to solve equation (2) for each state S_t. Most ADP algorithms proceed by stepping forward in time, following a particular sample path which we index by ω^n, where n indexes our iteration counter. We let $W_t(\omega^n)$ be the information that arrives between the decision made at time $t-1$ and the decision made at time t. If we make decision x_t^n at time t during iteration n, then our next state is given by equation (1).

The second element of ADP is how we make decisions. Most ADP algorithms solve an equation of the form

$$x_t^n = \arg\max_{x_t \in \mathcal{X}_t} \left(C(S_t^n, x_t) + \gamma \mathbb{E}\{\bar{V}_{t+1}^{n-1}(S_{t+1}) | S_t^n\} \right) \qquad (3)$$

where $S_{t+1} = S^M(S_t, x_t, W_{t+1})$ where W_{t+1} is a random variable at time t. Here, $\bar{V}_{t+1}^{n-1}(S_{t+1})$ is some sort of approximation of the value of being in state S_{t+1} (more on this in section 4).

Even if we have some sort of approximation for the value function, equation (3) may still be very hard to solve. First, we assume we can compute the expectation. If the random variable is simple (for example, a binomial random variable indicating the arrival of a customer, or the random change of a scalar stock price), then this may be easy.

Step 0.   Initialization:

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

    Step 0b.   Set $n = 1$.

    Step 0c.   Initialize S_0^1.

Step 1.   Choose a sample path ω^n.

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

    Step 2a.   Solve:

$$\hat{v}_t^n = \max_{x_t \in \mathcal{X}_t^n} \left(C(S_t^n, x_t) + \gamma \bar{V}_t^{n-1}(S^{M,x}(S_t^n, x_t)) \right) \qquad (5)$$

    and let x_t^n be the value of x_t that solves (5).

    Step 2b.   If $t > 0$, update the value function:

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

    Step 2c.   Update the states:

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

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

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

Figure 3: A basic ADP algorithm.

But this problem may have a complex vector of random variables, making the expectation intractable.

We can overcome this problem using the concept of a post-decision state, denoted S_t^x. The post-decision state is the state immediately after we make a decision, but before any new information has arrived. We assume we have a function $S_t^x = S^{M,x}(S_t, x_t)$ that gives us the post-decision state as a function of S_t and x_t. Now assume that we have estimated $\bar{V}_t(S_t^x)$ around the post-decision state. In this case, equation (3) becomes

$$x_t^n = \arg\max_{x_t \in \mathcal{X}_t} \left(C(S_t^n, x_t) + \gamma \bar{V}_{t+1}^{n-1}(S_{t+1}) \right). \qquad (4)$$

Now, the decision problem no longer has to deal with an expectation. We note, however, that the structure of a post-decision state is highly problem-dependent. There are problems where the post-decision state is much simpler than the pre-decision state, and others where it does not offer any advantage (but it never makes the problem more complicated).

The next challenge is making a decision. There are communities which assume that there is a finite (and small) set of actions which can be easily enumerated and evaluated. In operations research, there are many problems where x_t is a vector, of possibly very high-dimensionality. For these problems, we need to draw from a vast array of optimization algorithms, ranging from linear, nonlinear and integer programming through the entire family of metaheuristics.

After we make a decision, we often update the value function approximation using information derived from the

optimization problem where we made a decision (in other variations of ADP, the value functions are only updated after completing a forward trajectory). If $\hat{v}_t^n$ is the value of being in state S_t^n, we can update the value of being in this state using

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

Equation (6) is updating the value function approximation around the pre-decision state S_t^n. This updating scheme is using a standard lookup-table representation, where we have a value of being in each discrete state s. Alternatively, we can update the value around the post-decision state using

$$\bar{V}_{t-1}^n(S_{t-1}^{x,n}) = (1 - \alpha_{n-1})\bar{V}_{t-1}^{n-1}(S_{t-1}^{x,n}) + \alpha_{n-1}\hat{v}_t^n. \qquad (7)$$

Note that we are using $\hat{v}_t^n$ to update the value function around the previous post-decision state $S_{t-1}^{x,n}$.

The final element of an ADP algorithm is the simulation from S_t^n to the next state S_{t+1}^n using the transition function.

4 Designing value function approximations

Typically the first introduction to approximate dynamic programming uses simple lookup-table representations for value functions. To use a lookup-table representation, we assume that the state space $\mathscr{S}$ has been discretized into a series of elements which we number $(1, 2, \ldots, |\mathscr{S}|)$. We then assume we have an estimate $\bar{V}(s)$ for each state $s \in \mathscr{S}$. We estimate the value of being in each state using (6). This strategy appears to avoid the need to loop over all the states (as is required when we solve Bellman's equation backward in time as in equation (2)). However, it replaces the need to enumerate the states with the need to estimate the value of being in any state that *might* be visited.

The central challenge with any ADP algorithm is finding a value function approximation which can be represented using the fewest possible number of parameters. With a lookup-table representation, there is one parameter for each state (that is, we have to estimate the value of each state). A common way of reducing the number of parameters is to aggregate the state space. This is particularly useful for problems with a small number of continuous dimensions. For example, (Nascimento and Powell 2008) describe an application for managing the cash balance for mutual funds. The state variable has three continuous dimensions: the amount of cash on hand, the return on investments and interest rates. Using a fine discretization, an exact solution of Bellman's equation takes about three days. Coarser discretization quickly reduce this, but introduce discretization errors.

From the origins of dynamic programming, it has been recognized that the most promising way to overcome the challenge of approximating value functions is through the use of functional approximations (Bellman and Dreyfus 1959). Perhaps the modern era for studying value function approximations can be traced to (Schweitzer and Seidmann 1985), but major references include (Tsitsiklis and Van Roy 1996) and (Bertsekas and Tsitsiklis 1996). This line of research has used the vocabulary of approximation theory which assumes that we are given a family of *basis functions* $(\phi_f(S))_{f \in \mathscr{F}}$. A basis function is also referred to as *feature*. The function $\phi_f(S)$ is assumed to extract information from the state variable S that helps explain the behavior of the value function. The simplest class of function approximations are linear in the basis functions, which is to say that

$$V(S) \approx \bar{V}(S) = \sum_{f \in \mathscr{F}} \theta_f \phi_f(S). \qquad (8)$$

If there exists θ such that $V(S) = \sum_{f \in \mathscr{F}} \theta_f \phi_f(S)$, then we say that the functions $\phi_f(S)$ form a basis. In practice, this is generally not the case (or is unverifiable). In the language of statistical regression, we would refer to the functions $\phi_f(S)$ as explanatory variables which are often represented as $X_f = \phi_f(S)$. We would write a linear regression model as

$$Y = \sum_{f \in \mathscr{F}} \theta_f X_f + \varepsilon$$

where Y would be our observation of the value of being in a state, and ε explains any discrepancy between the observed values and the regression estimate.

Functions with the form given in equation (8) are referred to as linear approximations because they are linear in the parameters. However, it is also important to understand the structure of the basis functions themselves. For example, consider a resource allocation problem where R_{ti} is the number of resources of type i at time t. We could construct a value function approximation that is linear in the resource variable, giving us

$$\bar{V}(R) = \sum_i \theta_i R_{ti}.$$

Such an approximation assumes that the value of resources of type i is a constant given by θ_i. For many resource allocation problems, using an approximation that is linear in the number of resources offers particularly nice structure, often allowing problems to be decomposed for the purpose of parallel or distributed computation. However, many problems exhibit nonlinear behavior, and a common strategy is to use a quadratic polynomial such as

$$\bar{V}(R) = \sum_i \left(\theta_{1i} R_{ti} + \theta_{2i} R_{ti}^2 \right).$$

Some researchers assume almost automatically that nonlinear functions are always better than linear ones. The reality is that it depends on what you want to achieve with a value function approximation.

It is very important to understand what you want a value function to do for you. Start by asking - what would go wrong if you use $\bar{V}_t(S_t) = 0$? In our work, we have worked with two classes of resource allocation problems that arise in transportation. One class involves the management of freight cars where it is important to determine *how many* freight cars to move to a location (see (Powell and Topaloglu 2005) for an illustration). For such a setting, a value function that is linear in the resource state would not be able to help with the decision of how many to move.

By contrast, (Simao, Day, George, Gifford, Nienow, and Powell 2008) describes a fleet management problem arising in truckload trucking, where the challenge is to determine *what type* of driver to assign to a load. For this setting, a value function that is linear in the resource state worked perfectly well.

Another common tendency in the literature is the assumption that if you need a nonlinear function, then the function should be some sort of low-order polynomial. If an application involves managing small numbers of discrete resources (locomotives, aircraft, small numbers of expensive equipment), then a polynomial is unlikely to work well. Such problems are better suited to piecewise linear functional approximations ((Godfrey and Powell 2001), (Topaloglu and Powell 2006)).

There are many resource allocation problems which require a nonlinear, nonseparable function to capture the interaction between different resources. The most popular technique emerged from within the stochastic programming community as Benders decomposition. This technique approximates the value of the future using a series of cuts. The optimization problem is typically written

$$\max_{x_t \in \mathcal{X}_t} \left(c_t x_t + z \right) \tag{9}$$

subject to

$$z \leq \alpha_{t+1}(\hat{v}_{t+1}) + (\beta_{t+1}(\hat{v}_{t+1}))^T x_t \quad \forall \hat{v}_{t+1} \in \mathcal{V}_{t+1}. \tag{10}$$

Here, the set $\mathcal{V}_{t+1}$ is a set of vertices that have been generated as the algorithm progresses by using information from the dual of the optimization problem at time $t+1$. For detailed descriptions of this technique, see (Higle and Sen 1991) and (Sen and Higle 1999).

There are many applications where the state variable is a mixture of different measurements. For example, an application involving the testing of cardiovascular patients might involve biometric measures such as weight, blood pressure and cholesterol, behavioral indicators such as smoking and exercise, and family history. The question is, are all these variables important? (Fan and Li 2006) describe the challenges of feature selection for high-dimensional applications, and (Tsitsiklis and Van Roy 1996) describe the use of feature-based methods in approximation dynamic programming (without addressing the problem of choosing the features). The special challenge of approximate dynamic programming is the challenge of choosing features as the algorithm progresses.

The key insight here is very simple. Before designing a value function approximation, it is extremely important that you understand the properties of your problem, and the behavior you would like to achieve with your approximation. You then need to design a value function which reasonably captures the shape of the true value function, and which will give you the behavior that you are looking for.

5   Fitting a value function approximation

One method, albeit a clumsy one, for fitting a value function approximation is to take observations of a series of states $(S^m)_{m=1}^n$ and the observed value of being in these states $(\hat{v}^m)_{m=1}^n$, and use this data to fit the parameters of a regression model. This is typically referred to as batch regression, and becomes very clumsy as the number of iterations grow.

Most applications of ADP use some form of recursive estimation. The simplest arises with lookup-table representations which use the updating formula given in equation (3). Here, the only decision is the choice of stepsize, which we address in greater depth in section 7. When we use basis functions, a popular method for updating the parameter vector θ is the stochastic gradient equation given by

$$\begin{aligned}
\theta^n &= \theta^{n-1} - \alpha_{n-1}(\bar{V}(\theta^{n-1}) - \hat{v}^n)\nabla_\theta \bar{V}(\theta^n) \\
&= \theta^{n-1} - \alpha_{n-1}(\bar{V}(\theta^{n-1}) - \hat{v}^n)\begin{pmatrix} \phi_1(S^n) \\ \phi_2(S^n) \\ \vdots \\ \phi_F(S^n) \end{pmatrix}
\end{aligned} \tag{11}$$

There are many different methods for choosing the stepsize for the lookup table in equation (3), but they all share the property that $0 < \alpha_n \leq 1$. In equation (11), we introduce a scaling problem, since the units of θ and the units of the error $(\bar{V}(\theta^{n-1}) - \hat{v}^n)$, as well as the basis functions themselves, are different. As a result, we have to decide how to scale the stepsize. This issue is fairly significant. Depending on the units of the error and the basis functions, you might need to limit the stepsize to a number under 10^{-3} or 10^3. If you scale the stepsize improperly, convergence will either be impractically slow, or completely unstable.

One way to overcome the scaling problem is to make multiple observations of the value of being in a state and then use standard batch estimation techniques (see (Bertazzi, Bertsekas, and Speranza 2000) for an illustration). But a

more elegant strategy is to use recursive least squares. This is accomplished by updating the parameter vector θ using

$$\theta^n = \theta^{n-1} - H^n \phi^n \hat{\varepsilon}^n, \tag{12}$$

where ϕ^n is a column vector of basis functions evaluated at S^n and H^n is a matrix computed using

$$H^n = \frac{1}{\gamma^n} B^{n-1}.$$

B^{n-1} is an $F+1$ by $F+1$ matrix (where F is the number of features) which is updated recursively using

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

γ^n is a scalar computed using

$$\gamma^n \quad = \quad 1 + (x^n)^T B^{n-1} x^n.$$

One problem with recursive least squares is that it has the effect of equally weighting all prior observations. In approximate dynamic programming, we have to deal with the fact that the observations of the value of being in a particular state evolve higher (or possibly lower) as the value of being in a state reflects the future trajectory. We handle this issue by modifying the RLS equations slightly using (see (Powell 2007), section 7.3.3)

$$\bar{\theta}^n = \bar{\theta}^{n-1} - (G^n)^{-1}\phi^n\hat{\varepsilon}^n.$$

The matrix G^n is computed using

$$G^n = \lambda_n G^{n-1} + \phi^n(\phi^n)^T,$$

with $G^0 = 0$. We determine the parameter λ below. B^n is now given by

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

γ is computed using

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

These equations are the same as the original recursive least squares if we set $\lambda = 1$. λ plays the role of a discount factor which determines how much weight we want to put on past data. One way to determine the best value of λ is to simply try different values starting at 1 and then decreasing. Another way is to use the extensive literature on this topic for setting step sizes. If you have a stepsize formula that you are comfortable with, a rough relationship between stepsize

and λ is given by

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

It is important to note that the ADP algorithm presented in figure 3, where the function $U^V(\cdot)$ refers to the updating of a set of basis functions, has not been shown to converge. In fact, (Bertsekas 1995) provides counter-examples where the algorithm actually diverges. The problem is that recursive least squares is designed to deal with stationary forms of noise, but if we change the value function and allow this to change the policy, then it is like fitting a function to a moving target. (Tsitsiklis and Van Roy 1997) proves convergence if we hold the policy constant. This means locking in the value function approximation, which eliminates the problem of fitting to a moving target.

6 Values vs. slopes

The standard way of describing an ADP algorithm involves computing a value $\hat{v}_t^n$ which is an estimate of the value of being in a state S_t^n. This is the style used in the algorithm in figure 3. There are many problems, however, where it is more natural to use the derivative of the value function rather than the value function itself. This is particularly true of problems that involve managing resources, such as those that arise in inventory planning, supply chain management, demand management and the management of natural resources. For these problems, it is natural to let R_{ta} be the amount of resources with attribute $a \in \mathscr{A}$ (a may be a vector) at time t, and let $R_t = (R_{ta})_{a \in \mathscr{A}}$ be the resource state vector. Now let x_{tad} be a decision to act on resources with attribute a with a decision of type $d \in \mathscr{D}$ (d may represent buying, selling, moving from one location to another, repairing, modifying). We would then solve the optimization problem

$$x_t^n = \arg\max_{x_t \in \mathscr{X}_t^n}\left(C(R_t^n, x_t) + \gamma\bar{V}_t^{n-1}(R_t^x)\right)$$

where the constraint set $\mathscr{X}_t^n$ would include the constraint

$$\sum_{d \in \mathscr{D}} x_{tad} = R_{ta}^n. \tag{13}$$

Of course, there may be other constraints. Many resource allocation problems are naturally solved as mathematical programs, where it is important to capture the slope of the value function (adding a constant does not change the optimal solution). If the optimization problem is a linear or nonlinear program, we obtain a dual variable $\hat{v}_{ta}^n$ for each constraint (13). In some cases, we might even estimate $\hat{v}_{ta}^n$ using numerical derivatives. In this case, instead of obtaining a single scalar value $\hat{v}_t^n$ giving the value of being in state S_t^n, we obtain a vector $(\hat{v}_{ta}^n)_{a \in \mathscr{A}}$ which can be used

to update $\bar{V}_t^{n-1}(R_t^x)$. Since we are primarily interested in the slopes of $\bar{V}_t^{n-1}(R_t^x)$, an updated based on the vector of dual variables can be far more effective.

7   The problem of stepsizes

Arguably the most vexing and poorly understood issue in approximate dynamic programming is the choice of stepsize. I have met people who insist that you just set the stepsize to a constant (such as 0.10), while I have seen others casually assume $\alpha_n = 1/n$ without debate. For our discussion, we are going to assume that we are using a simple lookup-table representation for our value function, so that we can assume that our stepsizes should be between 0 and 1.

A simple example illustrates the challenges in choosing an appropriate stepsize. Consider a stochastic dynamic program with no decisions and only one state. Further assume that the contribution we earn at each time period can only be determined using sampling. Let $\hat{C}$ be a sample observation of the contribution. After receiving the reward, we return to the same state, where the estimated value of being in this state after n observations is $\bar{v}^n$. At iteration n, the sampled value of being in the state is given by

$$\hat{v}^n = \hat{C}^n + \gamma\bar{v}^{n-1}.$$

Because of the noise in the observation of $\hat{C}$, we have to perform smoothing using

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

If $\hat{C}$ were not random, the optimal stepsize would be $\alpha_{n-1} = 1$, which gives us $\bar{v}^n = \hat{C} + \gamma\bar{v}^{n-1}$, which is classical value iteration from Markov decision processes. Now assume $\gamma = 0$. In this case, we are just trying to estimate $\mathbb{E}\hat{C}$ using sample observations. It is well known that $\alpha_{n-1} = 1/n$ is the best possible stepsize in the sense of producing the lowest possible variance for the estimator. For $0 < \gamma < 1$, it is also known that $\alpha_{n-1} = 1/n$ also produces a sequence $\bar{v}^n \to v^*$ which is guaranteed to reach the optimal solution. (Frazier and Powell 2007) shows that after n iterations, $\bar{v}(n)$ is bounded by

$$\bar{v}(n) \quad \geq \quad \frac{c}{1-\gamma}\left(1 - (n+1)^{-(1-\gamma)}\right) \tag{14}$$

$$\bar{v}(n) \quad \leq \quad \frac{c}{1-\gamma}\left(1 - bn^{-(1-\gamma)} - \frac{1-\gamma}{\gamma}\frac{1}{n}\right) \tag{15}$$

where for any n_0,

$$b = n_0^{1-\gamma}\left(1 - \frac{1-\gamma}{n_0\gamma} - \frac{1-\gamma}{c}\bar{v}(n_0)\right).$$

Using these bounds, it is possible to show that if $\gamma = 0.95$ that it will require over 10^{10} iterations to get a solution within one percent of optimal. In effect, these bounds show that $1/n$ produces a convergent sequence that can be so slow that it should never be used for larger values of γ. Thus, the classic $1/n$ stepsize rule can be the best possible, or so slow it would never converge in a reasonable amount of time.

The balancing of noise and the need to sum future values is one reason why there is such a debate over stepsizes. The right stepsize really depends on the nature of your problem. A popular stepsize rule is the harmonic series

$$\alpha_{n-1} = \frac{a}{a+n-1}. \tag{16}$$

If $a = 1$, we get the $1/n$ stepsize rule. For larger values of a, we get stepsizes that decline arithmetically, but not as quickly as $1/n$. Of course, this means that you have to decide the best value of a.

One issue is that each state (or parameter) might need its own stepsize. The behavior of being in each state may be unique. As a result, there has been considerable interest in so-called stochastic stepsizes which adapt to the data as it arrives. We have had considerable success with a rule that we developed (George and Powell 2006) given by

$$\alpha_{n-1} = 1 - \frac{\sigma^2}{(1+\lambda^n)\sigma^2 + (\beta^n)} \tag{17}$$

where

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

and where 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 $\bar{V}^n(S^n)$ and the true value function $V(S^n)$. In practice, of course, these quantities are not known, but they can be estimated from data (see (George and Powell 2006) or (Powell 2007), chapter 6 for details). This stepsize formula has some nice features. First, if there is no noise, we get $\alpha_{n-1} = 1$, which we earlier argued was optimal when there is no noise. Second, if $\beta^n = 0$ (which is to say that we do not have any bias due to the growth in the value function), then it is possible to show that we get $\alpha_{n-1} = 1/n$. It is also possible to show that $\alpha_{n-1} \geq 1/n$ at all times.

(George and Powell 2006) calls this the "optimal stepsize algorithm" (or OSA), and (Powell 2007) refers to it as the "bias-adjusted Kalman filter" (BAKF) stepsize rule, since this is the behavior that can be derived from the Kalman filter. In our experimental work, we have found projects where this stepsize works superbly (see, for ex-

ample, (Simao, Day, George, Gifford, Nienow, and Powell 2008), where it nicely adapted to the rate of convergence of each of many thousands of parameters). Sometimes we work on deterministic applications, and have found that the BAKF stepsize rule gives us much higher stepsizes (as we would like) than other rules would have.

But all of these rules have limitations. A constant stepsize such as 0.1 can be far too small in some applications, and since it does not decline to zero, the estimates never converge (a problem if we want the variance of the estimate under some number). The harmonic stepsize rule requires that you tune a, and it also requires (for reasons of practicality) that we use the same value of a for all parameters. Finally, stochastic stepsize rules can work poorly in the presence of highly random data. We have found in particular that BAKF does not work well in the presence of rare events (e.g. a failure that occurs one time out of 100).

(George and Powell 2006) reviews a long list of stepsize rules (see also chapter 6 in (Powell 2007)). Based on our experiences with many projects, we suggest the following strategy:

a) Start with a constant stepsize. For problems with noise, 0.10 is a good starting point, but do not be afraid to try both larger and smaller values. Get a sense of the best constant stepsize for your problem, and the number of iterations required before the algorithm appears to converge.

b) Next try a harmonic stepsize rule. Choose a so that it roughly produces a stepsize comparable to the best constant stepsize by the number of iterations where the constant stepsize appeared to convergence. Depending on the problem, this could be 100 iterations, or 100,000 iterations. Now try varying a from this original value.

c) Try the BAKF stepsize rule. There is one tunable parameter (a target stepsize) described in (George and Powell 2006), but we have found that you should try both 0.10 and 0 for this target stepsize. Be sure to plot the average stepsize (over all your parameters) at each iteration for the stochastic stepsize. Be aware of the problems with very high levels of noise.

It is important to keep in mind that the best stepsize may be very large, possibly close to 1, if there is very little noise. As the noise increases, you need a stepsize closer to $1/n$, but the ideal stepsize rises again if the value of being in a state at time t requires summing the rewards over a number of time periods into the future (10 is a fairly large number). For example, if you are solving an inventory problem where you would like to cover 90 percent of the demand, the last unit of inventory will be used with probability less than 10 percent, which means one time period out of 10. In order

for the model to justify ordering this unit, the value has to reflect the expected value over the next 10 time periods. A small stepsize will have significant difficulty estimating this value.

Stochastic stepsizes are the most appealing in theory. We have found that the BAKF stepsize in equation (17) can work quite well. But this rule, and we believe all stochastic stepsize rules, struggle when there is a very high degree of noise. In particular, imagine again an inventory problem where you have a single high-value part, with very low frequency demands (the same would be true in a traditional inventory problem where the goal is to satisfy demand a high percentage of the time). When there is a demand, the value of a part jumps up, and the BAKF rule quickly increases to adapt to what it perceives as a change in the signal. Then, it quickly decreases when there is a long stretch with no demand.

8 Debugging an ADP model

So now you have an ADP algorithm up and running, but it just does not seem to work. Perhaps the behavior does not seem reasonable, or it underperforms a simple rule. A common symptom is that the solution does not seem to improve (it might even get worse). The following steps have proved useful in our work:

- $\hat{v}_t^n$ gives an estimate of the value of being in state S_t^n, so the first debugging tool is to make sure that this appears to be accurate. The complication here is that $\hat{v}_t^n$ depends on the contribution function, the decision and the value function $\bar{V}_t^{n-1}(S_t^x)$. The value function plays a critical role in determining a policy. For some problems, you get a reasonable (if suboptimal) decision if you simply set $\bar{V}_t^{n-1}(S_t^x) = 0$. But this step may be an effective debugging tool. For other problems, setting the value function to 0 gives a trivial behavior which does not provide any information. If this is the case, you have to assume that the value function is exact. Given this, does $\hat{v}_t^n$ seem reasonable?

- Verify that your value function is reasonable. This means that the structure of the function is reasonable, and that the parameters that you are fitting are reasonable. If you can use a simpler approximation (e.g. linear), then do this (as a debugging tool). In the early iterations of an algorithm, it is possible to get extremely poor estimates of the value of being in a state, and these estimates can distort the value function in later iterations.

- Try switching to a deterministic problem (or one where you always use the same sample path).

- Watch out for stepsizes. It is possible that a decision at time t depends on costs and rewards earned

over many time periods. If your stepsize is too small, backward communication can be so slow that the algorithm will never converge. Try solving a deterministic problem, and switch to a large stepsize, possibly as large as 1.0. If you are worried about backward communication, try switching to a small discount factor (e.g. $\gamma = 0.2$). If this improves performance, then it is possible that your stepsizes are too small, or you may need to switch to a backward pass.

- Perhaps your algorithm is working well, but not as well as you hoped. Again, stepsizes may be a problem. Some people like to use a fixed stepsize such as 0.10. This may provide a good solution but prevents the algorithm from ever converging. For this, you need a declining stepsize, but you have to be sure the stepsize does not decline too quickly. Try using a harmonic stepsize, but be sure to fix a so that it hits what appears to be a reasonable stepsize by a certain number of iterations. We have problems where 100 iterations produces superb results, and other problems where we run a million iterations. It does not make sense to use the same value of a for both problems.

- If possible, try to solve a simplified version of your problem optimally. You might be able to reduce the state space and solve it exactly as a discrete Markov decision process. Or it may be a problem that can be solved deterministically as a linear program.

While we have used all of these methods, the most valuable debugging tool is making sure that $\hat{v}_t^n$ is correct. This step also helps verify that the software is coded correctly.

9 Evaluating an ADP policy

Now imagine that your algorithm is up and running, and seems to be working well. For example, your final solution is better than your initial solution (equivalent to a myopic policy) or your favorite heuristic. But the question remains: how good is your policy? The first issue that you need to resolve is whether there is a good alternative to your ADP policy. Consider:

- Solve a deterministic version of your problem, ideally obtaining an optimal solution. This does not work for all problems. In some problems, removing uncertainty makes the problem trivial. In other cases, a deterministic problem is extremely hard to solve. If you can solve a deterministic version of the problem optimally, try applying ADP to the same deterministic problem. Also, you can take a sample path, assume you know the sample

path and solve the resulting deterministic problem, producing a posterior bound.
- Simulate a good policy. Perhaps your problem lends itself to a "good" decision rule (assign a job to the shortest queue, order inventory up to a certain level, sell when the price goes under or over some amount) which either reflects standard practice (or a rule used in the literature). Many policies have a parameter (the order-up-to rule, the trigger price) which can be optimized. Such problems lend themselves to techniques that fall under the umbrella of "simulation optimization."
- Try a competing technique. There are different communities proposing competing techniques that go under names such as robust optimization, stochastic programming, the linear-programming method, and simulation optimization. These techniques may be completely inappropriate for your problem, but in some cases they may even outperform your method. For example, simulation optimization is a powerful method for finding a good myopic policy. If this works well for your problem, there is a good chance that it may outperform a good ADP algorithm, which is best suited for harder problems.

ACKNOWLEDGMENTS

REFERENCES

Bellman, R., and S. Dreyfus. 1959. Functional approximations and dynamic programming. *Mathematical Tables and Other Aids to Computation* 13:247–251.

Bertazzi, L., D. Bertsekas, and M. G. Speranza. 2000. Optimal and neuro-dynamic programming solutions for a stochastic inventory trasportation problem. Unpublished technical report, Universita Degli Studi Di Brescia.

Bertsekas, D., and J. Tsitsiklis. 1996. *Neuro-dynamic programming*. Belmont, MA: Athena Scientific.

Bertsekas, D. P. 1995. A counterexample to temporal difference learning. *Neural computation* 7:270–279.

Fan, J., and R. Li. 2006. Statistical challenges with high dimensionality: Feature selection in knowledge discovery. In *Proceedings of International Congress of Mathematicians*, ed. J. V. J. V. e. M. Sanz-Solé, J. Soria, Volume III, 595–622.

Frazier, P., and W. B. Powell. 2007. Approximate value iteration converges slowly when smoothed with a $1/n$ stepsize. Technical report, Princeton University.

George, A., and W. B. Powell. 2006. Adaptive stepsizes for recursive estimation with applications in approximate

dynamic programming. *Machine Learning* 65 (1): 167–198.

Godfrey, G. A., and W. B. Powell. 2001. An adaptive, distribution-free approximation for the newsvendor problem with censored demands, with applications to inventory and distribution problems. *Management Science* 47 (8): 1101–1112.

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

Nascimento, J., and W. B. Powell. 2008. An optimal approximate dynamic programming algorithm for the lagged asset acquisition problem. *Mathematics of Operations Research*.

Powell, W. B. 2005. The optimizing-simulator: Merging simulation and optimization using approximate dynamic programming. In *Proceedings of the Winter Simulation Conference*. New York: OMNIPress.

Powell, W. B. 2006. Approximate dynamic programming for high-dimensional applications. In *Proceedings of the Winter Simulation Conference*. New York: OMNIPress.

Powell, W. B. 2007. *Approximate dynamic programming: Solving the curses of dimensionality*. New York: John Wiley and Sons.

Powell, W. B., and H. Topaloglu. 2005. Fleet management. In *Applications of Stochastic Programming*, ed. S. Wallace and W. Ziemba. Philadelphia: Math Programming Society - SIAM Series in Optimization.

Puterman, M. L. 1994. *Markov decision processes*. New York: John Wiley & Sons.

Schweitzer, P., and A. Seidmann. 1985. Generalized polynomial approximations in Markovian decision processes. *Journal of Mathematical Analysis and Applications* 110:568–582.

Sen, S., and J. Higle. 1999. An introductory tutorial on stochastic linear programming models. *Interfaces* 29 (2): 33–61.

Simao, H. P., J. Day, A. P. George, T. Gifford, J. Nienow, and W. B. Powell. 2008. An approximate dynamic programming algorithm for large-scale fleet management: A case application. *Transportation Science* (to appear).

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

Tsitsiklis, J., and B. Van Roy. 1997. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control* 42:674–690.

Tsitsiklis, J. N., and B. Van Roy. 1996. Feature-based methods for large scale dynamic programming. *Machine Learning* 22:59–94.

AUTHOR BIOGRAPHY

WARREN B. POWELL is a professor in the Department of Operations Research and Financial Engineering at Princeton University. He is director of CASTLE Laboratory and has implemented optimizing-simulator models in both military and civilian settings, including a number of the largest freight transportation companies in the U.S. The coauthor of over 100 refereed publications, he is an Informs Fellow, and has served in numerous leadership positions within Informs. He recently authored *Approximate Dynamic Programming: Solving the curses of dimensionality*.