Approximate Dynamic Programming: A Melting Pot of Methods

Warren B. Powell ▷
Princeton University (▷BibTeX entry)

Warren Powell is Professor of Operations Research and Financial Engineering at Princeton University, where he has taught since 1981. He is director of CASTLE Laboratory which specializes in the solution of large-scale stochastic optimization, with considerable experience in freight transportation. This work led to the development of methods to integrate mathematical programming and simulation within the framework of approximate dynamic programming, summarized in his book *Approximate Dynamic Programming: Solving the Curses of Dimensionality* [Wiley, 2007].

Stochastic optimization addresses the problem of making decisions over time as new information becomes available. This challenge arises in a number of disciplines, including operations research, economics, artificial intelligence and engineering. These fields each introduce different modeling issues and computational challenges. These issues, combined with the fundamental complexity of making decisions under uncertainty, has made stochastic optimization one of the richest and most challenging fields in applied mathematics.

Each of these disciplines has worked to solve the problem of making decisions over time, in the presence of different forms of uncertainty, within the context of their problem domain. It is perhaps not surprising, then, that there has been a certain amount of rediscovery of similar concepts under different names. Approximate dynamic programming, reinforcement learning, neuro-dynamic programming, optimal control and stochastic programming are all variations on a theme representing similar ideas discovered from the perspectives of different fields, often with different languages and notation.

As so often happens with languages, there is more to the differences than just words and notation. The more significant differences are the nature of the problems that each field faces. As a result, each field has discovered tricks and techniques that address the issues that arise within their problem domain. It is here that the fields have something to learn from each other.

In this article, I am going to show that the fundamentals of Markov decision processes, stochastic programming, control theory, and yes, even decision trees, can be combined within a general framework that integrates simulation and machine learning. The result is a scalable set of methods that are providing practical solutions to industrial-strength problems.

Notation

To describe our ideas, we need a notational system, which inevitably means making choices among the communities that are contributing to these problems. Choosing notation typically involves making a compromise between choosing variables that seem the most intuitive, while recognizing that notation is a language, and there is significant value in using notation that is familiar to the largest possible community. Somewhat more problematic is that notation tends to be associated with the characteristics of the problems that a community works on.

There is not enough space to address notational issues with any care (for a detailed discussion, see [5, Chapter 5]). I use S_t as the traditional variable for state, and x_t for the usual (in operations research) vector for decision variables. Decisions are made in discrete time, but activities (the arrival of information and the movement of resources) are made in continuous time. Time starts at $t = 0$, and decisions are made at $t = 0, 1, 2, \ldots$, which also corresponds to when we measure the state of the system (we only measure the state to make a decision). We let W_t be the information arriving during the time interval from $t-1$ to t, which allows us to claim that any variable indexed by t is known (deterministically) at time t. We note that this contrasts with the conventional style of the control theory community, where W_t would refer to the information arriving between t and $t + dt$ (in continuous time).

Central to dynamic systems is describing how it evolves from t to $t + 1$. I use the classical concept of a transition function, which is represented using

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

The function $S^M(\cdot)$ goes under various names: "plant model," "plant equation," "system model" or just "model." It is more common to use $f(\cdot)$ for the transition function, but this uses up another valuable letter of the alphabet. The MDP community most commonly uses the one-step transition matrix $p(s'|s, x)$, which gives the probability of moving to state s' given that we are in state s and take action x. The OR community prefers to use systems of equations such as

$$A_t x_t - B_{t-1} x_{t-1} = b_t.$$

In many applications, these matrices are extremely large, and may be very difficult to write out explicitly. As a result, it is common in the control community to simply assume there is a function such as $S^M(\cdot)$, unless there is some specific need to exploit its properties.

Our challenge is to make decisions. If we were solving a traditional math programming-based model, we might formulate the problem as

$$\min_{x_0,\ldots,x_T} \sum_{t=0}^{T} \gamma^t c_t x_t$$

subject to various constraints on x_t including the linking constraints above. Here, γ is a discount factor. The difficulty is that this formulation cannot handle uncertainty in the form of an evolving information process, with decisions that are allowed to adapt to the process.

What we really want are decisions that adapt to the information as it arrives, but without peeking into the future.

The stochastic programming community has learned how to formulate these problems using *nonanticipativity* constraints (see [2] for a complete presentation). But the resulting models are typically too large to be solved, and authors generally simplify the problem by limiting the number of outcomes in the future. So this leaves us with the question — how do we make decisions?

Making Decisions

The dynamic programming community addresses this problem by proposing to design a decision function $X^\pi(S_t)$ which returns a decision x_t which we assume is in a feasible region $\mathcal{X}_t$ (which might depend on S_t). There is typically a family of functions which we index using $\pi \in \Pi$, where π refers to a policy (to use the language of dynamic programming). Policies come in many flavors.

It is more conventional in this community to maximize a reward (we use a contribution) $C(S_t, x_t)$ that depends on the state and action (it might also depend on random information W_{t+1}). In this setting, our problem is to find the best policy π (or decision function X^π) that solves

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

$\mathbb{E}$ symbol ok?

where $\mathbb{E}$ denotes the expected value, and we assume that we have some guarantee that a solution exists. If we use a properly designed decision function that depends on S_t, and if S_t is purely a function of $W_1, \ldots, W_t$, then the resulting decisions are automatically nonanticipative, which is why these terms typically do not arise in the dynamic programming or control theory communities.

changed 'which' to 'that' and added 's'

There are numerous ways to create a decision function that represents a policy.

Myopic Policies

A myopic policy is any decision rule that does not attempt to project into the future. For example, we might simply solve problems that maximize the contribution in each period,

$$X^\pi(S_t) = \underset{x_t \in \mathcal{X}_t}{\text{argmax}}\, C(S_t, x_t).$$

As stated, this is just a single policy with a solution that might be good enough in some applications, but may be quite poor. There are numerous examples in engineering practice where people will play with the contribution function, adding bonuses and penalties to the contributions to try to get the model to produce good long-term results. This strategy might be written

$$X^\pi(S_t) = \underset{x_t \in \mathcal{X}_t}{\text{argmax}}\, C^\pi(S_t, x_t).$$

Here, $C^\pi(S_t, x_t)$ is parameterized by the various bonus and penalties, and now the challenge is to find the best values for these parameters.

Dynamic Programming

The most classical way of solving the objective function in equation (WP.1) is through dynamic programming, where we write

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

This is generally known as Bellman's equation, the Hamilton-Jacobi equation, the HJB equation (to cover all our bases), or just the optimality equation.

The textbook approach to solving (WP.2) (see [7] and its predecessors) is to solve this equation for each state S_t, and step backward through time (value iteration for infinite horizon problems does basically the same thing, although it is presented as an iteration counter rather than stepping backward through time). The problem here is that for many applications, S_t is a vector (for example, the number of different types of products in inventory), in which case the size of the state space grows exponentially in the number of dimensions. This is the "curse of dimensionality" that is so widely cited as a reason why "dynamic programming does not work."

Of course, this technique does work for some applications. But for the vast majority of real-world problems (not sure how to verify this claim), there are actually three curses of dimensionality: the state space, the outcome space (which determines the complexity of the expectation), and the action space (that is, the feasible region $\mathcal{X}_t$). The problem with so-called classical dynamic programming is that it assumes that you are using a lookup-table representation for the value function $V_t(S_t)$ (that is, there is a distinct value for each discrete state S_t). The method also assumes the expectation can be solved exactly, and it assumes that you can evaluate each action separately.

For many applications, the expectation cannot be computed exactly, and the vector x has to be chosen with one of a wide range of algorithms that have evolved from the math programming community. Needless to say, the usefulness of dynamic programming is looking quite limited.

Stochastic Programming

Stochastic programming evolved out of the math programming community when interest grew (starting in the 1950's) to introduce uncertainty. A large body of research has evolved to solve the so-called two-stage problem that can be generally written as

$$\min_x \mathbb{E}(F(x, W)).$$

In the stochastic programming community, this is most commonly written

replaced 'E' with \Exp

$$\min_{x_1} c_1 x_1 + \mathbb{E}(Q(x_1)) \qquad \text{(WP.3)}$$

subject to various constraints. $Q(x_1)$ is referred to as the *recourse function* and is given by

$$Q(x_1, \omega) = \min_{x_2} c_2(\omega) x_2(\omega)$$

also subject to various constraints that depend on a random outcome ω. x_1 refers to the first-stage decisions while x_2 refers to the second stage. Note that $Q(x_1)$ is comparable to the value function of dynamic programming, except that x_1 is used as the state variable.

This general idea can be extended to multistage problems, but here is where the stochastic programming community splits. One branch uses the concept of scenario trees, while the other uses Benders decomposition. With scenario trees, the state of the system consists of the entire history, which explodes quickly in size, quickly producing intractably large problems.

With Benders' decomposition, equation (WP.3) is replaced with

$$\min_{x_1} c_1 x_1 + z \tag{WP.4}$$

subject to the same constraints on x_1 as before plus

$$z \leq \alpha_i + \beta_i^T x_1, \quad i \in \mathcal{I}^n, \tag{WP.5}$$

where α_i is a set of scalars and β_i is a set of vectors over the set $\mathcal{I}^n$ which is generated iteratively by the algorithm. Here, the recourse function is replace by a series of cuts that are generated by solving the dual of the problem for the next time period. A particularly powerful algorithm is the stochastic decomposition algorithm [3] (see also [2]), which offers a convergence proof (if the only source of randomness is in the right hand side constraint).

Approximate Dynamic Programming

There are three views of approximate dynamic programming: 1) a framework for solving complex dynamic programs, 2) a framework for making simulations more intelligent, and 3) a decomposition technique for very large-scale deterministic math programs. One of our messages is that ADP is a valid method for solving deterministic or stochastic, multistage mathematical programs.

There are many variations of approximate dynamic programming. This presentation only briefly illustrates the basic ideas.

General Strategy

At the risk of oversimplifying the diversity of algorithmic strategies that fall under the name approximate dynamic programming, the central idea of ADP is to replace the value function $V_{t+1}(S_{t+1})$ with some sort of statistical model that we call $\bar{V}_{t+1}(S_{t+1})$. Then, instead of solving (WP.2), we would make decisions by solving

$$X^\pi(S_t) = \underset{x_t \in \mathcal{X}_t}{\text{argmax}} \left\{ C(S_t, x_t) + \gamma \mathbb{E}\left(\bar{V}_{t+1}(S_{t+1}) | S_t\right) \right\}. \tag{WP.6}$$

Then, instead of stepping backward through time (forcing us to compute the value of being in every state), we step forward through time, sampling a single sequence of states. We use information gathered from solving these decision problems to update the value function approximation.

There are **many** ways to estimate the value function approximation. The simplest way to illustrate the idea is to use a lookup-table representation, where there is a value $\bar{V}_t(S_t)$ for each state S_t. Assume we are in state S_t^n at iteration n of our algorithm, and let $\bar{V}_t^{n-1}(S_t)$ be our estimate (from the previous iteration) of the value of being in each state. Finally let $\hat{v}_t^n$ be the value from solving the maximization problem in equation (WP.6) (this would be the value if we used max instead of argmax). We could update the value function approximation using

'are a number of' replaced by 'are many'

$$\bar{V}_t^n(S_t^n) = (1 - \alpha_{n-1})\bar{V}_t^{n-1}(S_t^n) + \alpha_{n-1}\hat{v}_t^n, \tag{WP.7}$$

where α_{n-1} is a stepsize between 0 and 1. In practice, lookup-table representations are not practical, but they illustrate the basic idea.

We are not out of the woods. The difficulty is the other two curses of dimensionality, which means computing the expectation and then solving the resulting optimization problem (which itself may be fairly hard, especially if x_t is integer and we face a difficult integer programming problem). We address this problem by using a concept that goes under many names, but I prefer the name first proposed in [12] which is the post-decision state variable. This is the state of the system immediately after a decision has been made, but before any new information has arrived. In decision trees (where you make a decision at a decision node), this is called the outcome node. It has also been called the end-of-period state [4] and the after-state variable [10]. All that matters is that it is a deterministic function of S_t and x_t.

The post-decision state takes different forms depending on the nature of the application. Some examples are:

changed itemize to description — ok?

Blood management. Let R_{tb} be the amount of blood of type b on hand to be used during week t. Let x_{tb} be the amount of blood of type b used in week t, and let $\hat{R}_{t+1,b}$ be random donations of blood that will be available to be used during week $t+1$. R_t is the pre-decision state. $R_{tb}^x = R_{tb} - x_{tb}$ is the post-decision state, while $R_{t+1,b} = R_{tb}^x + \hat{R}_{t+1,b}$ is the next pre-decision state.

Managing expensive equipment. Let a_t be the vector of attributes of a business jet, which includes attributes such as location, repair status and time of availability. If we make the decision to move the aircraft, it might arrive late due to weather delays, and an equipment problem might arise. The post-decision state a_t^x might assume the aircraft will arrive on time with no equipment problems. The next pre-decision state, a_{t+1}, would reflect the weather delays and equipment problems.

Let $S_t^x = S^M(S_t, x_t)$ be the post-decision state (that is, the state after x_t has been determined). Instead of coming up

with a value function approximation around S_t, we will instead come up with a value function approximation around S_t^x. If we were using a lookup-table representation, the update in equation (WP.7) becomes

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

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 change is subtle but significant. Now, our policy looks like

$$X^\pi(S_t) = \underset{x_t \in \mathcal{X}_t}{\text{argmax}} \{C(S_t, x_t) + \gamma \bar{V}_t(S_t^x)\}. \tag{WP.8}$$

where $S_t^x = S^{M,x}(S_t^n, x_t)$ is the post-decision state if we are currently in state S_t^n and if we were to take action x_t.

Note that we no longer have an expectation in the decision problem in equation (WP.8). This is a major change that also helps us solve the third curse of dimensionality. We are primarily interested in problems where x_t is a vector. Depending on the characteristics of the problem, we might naturally want to solve our decision problem as a linear program, nonlinear or integer program, or using our favorite metaheuristic. If we use a metaheuristic, the value function approximation can take virtually any form (as long as it can be quickly computed). If the contribution function $C(S_t, x_t)$ is nonlinear, we could use a nonlinear value function approximation.

After we solve our problem at time t, we simulate our way to the next state by randomly sampling the exogenous information (which we represent using $W_t(\omega^n)$), and then compute

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

This is pure simulation — nothing fancy here, and it scales to really large problems.

Value Function Approximations

Formulating and estimating a value function approximation is at the heart of any ADP algorithm. A popular strategy in the ADP literature is to use linear regression, where the independent variables are referred to as *basis functions* $\phi_f(S_t)$, $f \in \mathcal{F}$, where each basis function $\phi_f(S_t)$ is also known as a feature. A basis function is any scalar function of the state variable. If we use this strategy, the value function approximation would look like

$$\begin{aligned} V_t(S_t) &\approx \bar{V}_t(S_t|\theta) \\ &= \sum_{f\in\mathcal{F}} \theta_f \phi_f(S_t). \end{aligned}$$

The appeal of basis functions is that the strategy is quite general. You take a physical problem, design a set of features (basis functions) that you think capture important properties of the problem, and then string them together linearly into an approximation. Our own experience, however, is that it is critical to take advantage of problem structure.

Our projects in CASTLE Lab all involve the management of physical resources. Let $a \in \mathcal{A}$ be an attribute vector describing a type of resource (e.g. the location and type of equipment) and let R_{ta} be the number of resources with attribute a at time t. Also let d be a type of decision (move, clean, repair, modify) that acts on a resource with type a, producing a resource with attribute $a' = a^M(a,d)$. Let $R_t = (R_{ta})_{a\in\mathcal{A}}$ be the state of all the resources, and let ρ_t be the state of other parameters (prices, weather, technology) which evolve randomly over time. Our system state variable is $S_t = (R_t, \rho_t)$. Let $\delta_{a'}(a,d) = 1$ if decision d acting on a resource with attribute a produces a resource with attribute a', and let x_{tad} be the number of resources with attribute a that decision $d \in \mathcal{D}$ acts on. The decision vector x_t must satisfy

$$\sum_{d\in\mathcal{D}} x_{tad} = R_{ta}. \tag{WP.9}$$

The post-decision resource vector is given by

$$R_{ta'}^x = \sum_{a\in\mathcal{A}}\sum_{d\in\mathcal{D}} \delta_{a'}(a,d)x_{tad}.$$

Now let $\hat{R}_{t+1,a}$ be exogenous changes to R_{ta}^x (arrivals, departures, delays, breakdowns). The next pre-decision state is given by

$$R_{t+1,a} = R_{ta}^x + \hat{R}_{t+1,a}.$$

For this problem class, there are some natural value function approximations. We can start by ignoring the exogenous parameter vector ρ_t. Possible value function approximations include linear in the resource vector:

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

and separable

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

where $\bar{V}_{ta}(R_{ta}^x)$ may be piecewise linear, or any of a set of concave, nonlinear functions. Figure WP.1 illustrates the decision problem where we can assign a reusable resource to either a set of known tasks (which have the effect of modifying the resource) or we can modify the resource (move it, clean it, repair it, or set up the machine for a type of task). Separable, nonlinear functions (of the post-decision state) capture the value of resources in the future.

This strategy has been applied to several transportation applications where the resource is a vehicle (trailer, locomotive, freight car) which can serve a task (move a load of freight, pull a train) or be modified (move the equipment empty to another location, repair the locomotive). ADP compares favorably against a rolling-horizon model, where a decision at time t is based on what we know at time t and forecasts over a specific horizon. These results are then compared against the optimal posterior bound (where we optimize given we know the

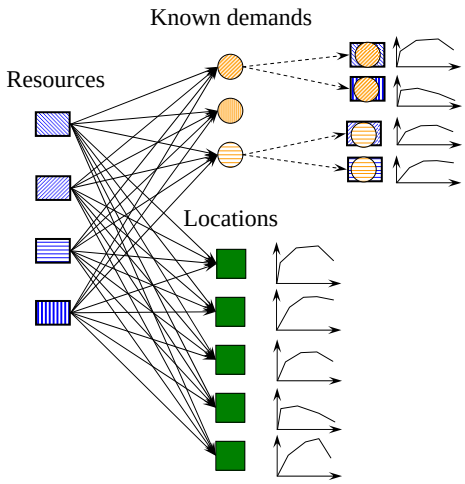

Figure WP.1: Illustration of a linear program for assigning resources to known tasks (with nonlinear value functions after task is completed) and to new locations (with nonlinear value functions capturing value of resources in the future)

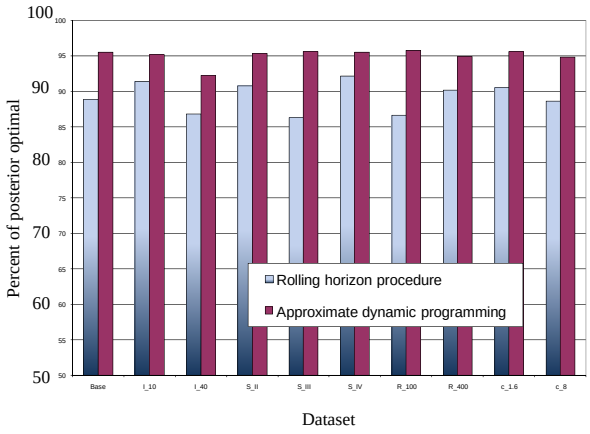

Figure WP.2: Comparison of ADP to a rolling horizon procedure for a stochastic, multicommodity flow network (from [11])

future), shown in Figure WP.2. Furthermore, once the value functions are estimated, it is very easy to solve the "here and now" problem (at time $t = 0$) which is useful for real-time applications.

Of course, these solutions are all approximate. A powerful strategy is to use Benders' decomposition (WP.4)—(WP.5) (see [3], which offers a convergence proof). In [5, Chapter 11], Benders' decomposition is presented as a type of value function approximation.

It is common to illustrate the process of estimating the value function by using the value of being in a state to update the value function approximation. For resource allocation problems, it is much more effective to use an estimate of the derivative of the maximization problem in equation (WP.6) with respect to the resource variable R_{ta}. Often this is available as a dual variable of equation (WP.9), although we sometimes have to resort to numerical derivatives.

It is important to realize that developing and estimating a value function approximation is a classical statistical modeling exercise (with some twists). It is not necessary to use all the elements of the post-decision state variable. The art of ADP is identifying the variables that are the most important. As with other fields of statistics, econometrics and machine learning, the art is designing the functional approximation, while the science is estimating the best possible function.

Some Applications

Our work in ADP has been motivated by problems in transportation and logistics, including major systems that are now running in production at several companies. Some of these projects include:

- We recently developed a large-scale fleet management system for optimizing the movements of drivers and loads at Schneider National. The model uses ADP to estimate the value of different types of drivers in the future. Drivers are modeled with a 15-dimensional attribute vector, although the linear value function approximation only uses four attributes (which still produces 600,000 parameters to be estimated). The system is in production at Schneider for performing a broad range of policy studies [8].

- A few years ago we implemented a model for managing freight cars at Norfolk Southern. The model assign cars to known orders, and can reposition cars to different locations in anticipation of unknown demands. The system has been in production for a number of years [6].

- We have recently completed a decade-long development project, resulting in the successful development and implementation of an optimization model using ADP for Norfolk Southern Railroad (paper in preparation).

- We recently completed a planning system for high-value spare parts for Embraer. The system uses ADP to determine how many spares of over 400 types of parts should be allocated over 19 service centers, subject to budget and service constraints [9].

In addition to these production projects, ADP has been used in numerous senior theses and projects at Princeton University. However, it has been the ability of ADP to handle the complexities of real-world problems that seems to set it apart from other methods.

Closing Remarks

So, ADP is truly a melting pot of methods. It involves statistics/machine learning (estimating the value function) and simulation, all within a dynamic programming framework that allows you to use your favorite optimization algorithm for solving decision problems at each point in time. This strategy has

allowed us to solve some extremely large problems that arise in freight transportation, as well as applications in health and finance.

ADP is more than just a technique for solving stochastic problems. We often use it as a decomposition method for very large-scale deterministic problems, including some hard integer programming problems. We have found that commercial solvers such as CPLEX® may struggle with long horizons, but can often solve very large problems as long as the horizon is fairly short. Thus, problems that might otherwise be solved using a heuristic can be solved optimally at a point in time, with solutions that are quite good over time.

References

[1] D. P. Bertsekas and J. Tsitsiklis, *Neuro-Dynamic Programming*, Athena Scientific, Belmont, MA, 1996.

[2] J. Birge and F. Louveaux, *Introduction to Stochastic Programming*, Springer-Verlag, New York, NY, 1997.

[3] J. Higle and S. Sen, *Stochastic Decomposition: A Statistical Method for Large Scale Stochastic Linear Programming*, Kluwer Academic Publishers, Norwell, MA, 1996.

[4] K. Judd, *Numerical Methods in Economics*, MIT Press, Cambridge, MA, 1998.

[5] W. B. Powell, *Approximate Dynamic Programming: Solving the Curses of Dimensionality*, John Wiley & Sons, New York, NY, 2007.

[6] W. B. Powell and H. Topaloglu, Fleet Management, in S. W. Wallace and W. T. Ziemba (eds.), *Applications of Stochastic Programming*, pages 185–216, MPS-SIAM Series on Optimization, Philadelphia, PA., 2005.

[7] M. L. Puterman, *Markov Decision Processes*, John Wiley & Sons, New York, NY, 1994.

[8] H. P. Simao, J. Day, A. P. George, T. Gifford, J. Nienow, and W. B. Powell, An Approximate Dynamic Programming Algorithm for Large-Scale Fleet Management: A Case Application, *Transportation Science* (to appear).

[9] H. P. Simao and W. B. Powell, Approximate Dynamic Programming for Management of High Value Spare Parts, *Journal of Manufacturing Technology Management* 20:9 (2009).

[10] R. Sutton and A. Barto, *Reinforcement Learning*, MIT Press, Cambridge, MA, 1998.

[11] H. Topaloglu and W. B. Powell, Dynamic Programming Approximations for Stochastic, Time-Staged Integer Multicommodity Flow Problems, *INFORMS Journal on Computing* 18:1 (2006), 31–42.

[12] B. Van Roy, D. P. Bertsekas, Y. Lee, and J. N. Tsitsiklis, A Neuro-Dynamic Programming Approach to Retailer Inventory Management, in *Proceedings of the 36th IEEE Conference on Decision & Control*, Vol. 4, 1997, 4052–4057.