A Tour of the Jungle of Approximate Dynamic Programming

Warren B. Powell

Department of Operations Research and Financial Engineering
Princeton University, Princeton, NJ 08544

July 6, 2011

Abstract

Approximate dynamic programming has evolved, initially independently, within operations research, computer science and the engineering controls community, all searching for practical tools for solving sequential stochastic optimization problems. More so than other communities, operations research continued to develop the theory behind the basic model introduced by Bellman with discrete states and actions, even while authors as early as Bellman himself recognized its limits due to the "curse of dimensionality" inherent in discrete state spaces. In response to these limitations, subcommunities in computer science, control theory and operations research developed practical methods for solving stochastic, dynamic optimization problems which has emerged as a seemingly disparate family of algorithmic strategies. In this article, we show that there is actually a common theme to these strategies, and underpinning the entire field remains the fundamental algorithmic strategies of value and policy iteration that were first introduced in the 1950's and 60's.

1 Introduction

In 1957, Bellman published his seminal volume that laid out a simple and elegant model and algorithmic strategy for solving sequential stochastic optimization problems. This problem can be stated as one of finding a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ that maps a discrete state $s \in \mathcal{S}$ to an action $a \in \mathcal{A}$, generating a contribution $C(s, a)$. The system then evolves to a new state s' with probability $p(s'|s, a)$. If $V(s)$ is the value of being in state s, then Bellman showed that

$$V(s) = \max_{a \in \mathcal{A}} \left(C(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) V(s') \right). \tag{1}$$

where γ is a discount factor. This has become widely known as Bellman's optimality equation, which expresses Bellman's "principle of optimality" that characterizes an optimal solution. It provides a compact and elegant solution to a wide class of problems that would otherwise be computationally intractable, which is true even for problems with relatively small numbers of states and actions. To understand the significance of this breakthrough, imagine solving sequential optimal control problems with even a small number of states, actions and random outcomes as a decision tree. A problem with as little as 10 actions and 10 possible random outcomes grows by a factor of 100 with every stage (consisting of a decision followed by a random outcome). First imagine solving such a problem over a planning horizon of just 10 time periods, which would produce a decision tree with 10^{20} nodes. Now imagine solving the problem over an infinite horizon.

The problem with decision trees is that they do not exploit the property that multiple trajectories can lead back to a single state. Bellman's breakthrough was exploiting the value of computing the value of being in a state. Once we know the value of being in a state, then we have only to evaluate the value of a decision by computing its reward plus the value of the next state the decision might take us to. For a finite-horizon problem, the result was equations that looked like

$$V_t(S_t) = \max_{a \in \mathcal{A}} \left(C(S_t, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|S_t, a) V_{t+1}(s') \right). \tag{2}$$

where $\mathcal{S}$ is the set of all states. Equation (2) is executed by starting with some terminal condition such as $V_T(S_T) = 0$ for all ending states T, and then stepping backward in time. Equation (2) has to be calculated for every state S_t. For infinite horizon problems, Howard (1960) introduced value

iteration, which requires iteratively computing

$$V^n(s) = \max_{a \in \mathcal{A}} \left(C(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V^{n-1}(s') \right). \tag{3}$$

This algorithm converges in the limit with provable bounds to provide rigorous stopping rules (Puterman (2005)). Both (1) and (2) require at least three nested loops: 1) over all states S_t, then over all actions a, and finally over all future states S_{t+1}. Of course, both algorithms require that the one-step transition matrix $p(s'|s,a)$ be known, which involves a summation (or integral) over the random information W.

The alternative to value iteration is policy iteration. In an ideal world of infinitely powerful computers, assume we can create an $\mathcal{S} \times \mathcal{S}$ matrix P^π with element $P^\pi(s,s') = p(s'|s,\pi(s))$ where the action $a = \pi(s)$ is determined by policy π (this is best envisioned as a lookup table policy that specifies a discrete action for each state). Let c^π be an $\mathcal{S}$-dimensional vector of contributions, with one element per state given by $C(s,\pi(s))$.

Finally let v^π be an $\mathcal{S}$-dimensional vector where element s corresponds to the steady state value of starting in state s and then following policy π from now to infinity (all of this theory assumes an infinite horizon). We can (in theory) compute the steady state value of starting in each state using

$$v^\pi = (I - \gamma P^\pi)^{-1} c^\pi, \tag{4}$$

where I is the identity matrix. Not surprisingly, we often cannot compute the matrix inverse, so as an alternative we can iteratively compute

$$v^m = c^\pi + \gamma P^\pi v^{m-1}. \tag{5}$$

For some $m = M$, we would stop and let $v^\pi = v^N$. Once we have evaluated the value of a policy, we can update the policy by computing, for each state s, the optimal action $a(s)$ using

$$a(s) = \arg\max_a \big(C(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v^\pi(s') \big). \tag{6}$$

The vector $a(s)$ constitutes a policy that we designate by π. Equation (6) (along with (4) or (5)) is known as policy iteration.

The study of discrete Markov decision problems quickly evolved in the operations research community into a very elegant theory. Beginning with the seminal volumes by Richard Bellman (Bellman (1957)) and Ron Howard (Howard (1960)), there have been numerous, significant textbooks on the subject, including Nemhauser (1966), White (1969), Derman (1970), Bellman (1971), Dreyfus & Law (1977), Dynkin & Yushkevich (1979), Denardo (1982), Ross (1983) and Heyman & Sobel (1984). The capstone for this line of research was the landmark volume Puterman (1994) (a second edition appeared in 2005). When we take advantage of the compact structure of a dynamic program, finite horizon problems become trivial since the size of our tree is limited by the number of states. For infinite horizon problems, algorithmic strategies such as value iteration and policy iteration proved to be powerful tools.

The problem with this theory is that it quickly became "widely known" that Bellman's equation "does not work" because of the "curse of dimensionality." These issues were recognized by Bellman himself, who published one of the very earliest papers on approximations of dynamic programs (Bellman & Dreyfus (1959*a*)). Applications with a hundred thousand states were considered ultra-large scale, while the number of discrete actions rarely exceeded a few hundred (and these were considered quite large). However, seemingly small problems produced state spaces that were far larger than this. Imagine a retailer optimizing the inventories of 1000 products within limited shelf space. Imagine that the retailer might have up to 50 items of each product type. The size of the joint state space of all the products would be 50^{1000}. And this barely describes the inventory of a small corner store. The study of Markov decision processes in operations research quickly became a mathematician's playground characterized by elegant (but difficult) mathematics but few practical applications.

Now advance the clock to a recent project optimizing a fleet of 5,000 drivers for Schneider National, one of the largest truckload carriers in the U.S. Each driver is described by a 15-dimensional vector of attributes. The state variable is a vector with about 10^{20} *dimensions.* The action is a vector with 50,000 *dimensions.* The random information is a vector with 10,000 *dimensions.* If the problem were deterministic, it could be engineered to produce an integer program with 10^{23} variables. Yet a practical solution was developed, implemented and accepted by the company, which documented over 30 million dollars in annual savings. The problem was solved using dynamic programming, with an algorithm that approximated value iteration (Simao et al. (2009), Simao et al. (2010)).

How did this happen? It is helpful to understand the history of how the field evolved, because

different communities have contributed important ideas to this field. While the essential ideas started in operations research, the OR community tended to focus on the analytical elegance, leaving the development of practical ideas to the control theory community in engineering, and the artificial intelligence community in computer science. Somewhat later, the operations research community became re-engaged, producing breakthroughs such as the application to Schneider National.

But perhaps the more important question is: do we now have tools that can solve large-scale dynamic programs? Not exactly. To be sure, some of the breakthroughs are quite real. However, the field that is emerging under names like approximate (or adaptive) dynamic programming, reinforcement learning, neuro-dynamic programming and heuristic dynamic programming offers a powerful toolbox of algorithms, but the design of successful algorithms for specific problem classes remains an art form.

This paper represents a tour of approximate dynamic programming, providing an overview of the communities that have contributed to this field along with the problems that each community has contributed. The diversity of problems has produced what can sometimes seem like a jungle of algorithmic strategies. We provide a compact representation of the different classes of algorithms that have been proposed for the many problems that fall in this broad problem class. In the process, we highlight parallels between disparate communities such as stochastic programming, stochastic search, simulation optimization, optimal control and reinforcement learning. We cover strategies for discrete actions a and vector actions x, and review a powerful idea, mostly overlooked by both the operations research and reinforcement learning communities, of the post-decision state. The remainder of the paper reviews strategies for approximating value functions and the challenges that arise when trying to find good policies using value function approximations.

2 The evolution of dynamic programming

Dynamic programming, and approximate dynamic programming, has evolved from within different communities, reflecting the breadth and importance of dynamic optimization problems. The roots of dynamic programming can be traced to the control theory community, motivated by problems of controlling rockets, aircraft and other physical processes, and operations research, where the seminal work of Richard Bellman in the 1950's laid many of the foundations of the field. Control theory focused primarily on problems in continuous time, with continuous states and actions. In operations

research, the focus was on modeling problems in discrete time, with discrete states and actions. The fundamental equation (1) is known in control theory as the Hamilton-Jacobi equation, while in operations research it is known as Bellman's equation of optimality (or simply Bellman's equation). Many refer to equation (1) as the Hamilton-Jacobi-Bellman equations (or HJB for short).

After recognizing the "curse of dimensionality," Bellman made what appears to be the first contribution to the development of approximations of dynamic programs in Bellman & Dreyfus (1959*a*). However, subsequent to this important piece of research, the operations research community focused primarily on the mathematics of discrete Markov decision processes, including important contributions by Cy Derman (Derman (1962), Derman (1966), Derman (1970)), leading to a deep understanding of a very elegant theory, but limited in practical applications. The first sustained efforts at developing practical computational methods for dynamic programs appear to have started in the field of control theory with the Ph.D. dissertation of Paul Werbos (Werbos (1974)), with numerous contributions to the research literature (Werbos (1974), Werbos (1989), Werbos (1990), Werbos (1992*a*), Werbos (1992*b*)). The edited volumes Werbos et al. (1990) and Si et al. (2004) summarize many contributions that were made in control theory which originally used the name "heuristic dynamic programming" to describe the emerging field.

A second line of research started in computer science around 1980 with the early research of Andy Barto and his Ph.D. student, Richard Sutton, where they introduced the term "reinforcement learning." This work is summarized in numerous publications (for a small sample, see Barto et al. (1981), Sutton & Barto (1981), Barto et al. (1983), Sutton (1988)) leading up to their landmark volume, *Reinforcement Learning* (Sutton & Barto (1998)). In contrast with the mathematical sophistication of the theory of Markov decision processes, and the analytical complexity of research in control theory, the work in reinforcement learning was characterized by relatively simple modeling and an emphasis on a wide range of applications. Most of the work in reinforcement learning focuses on small, discrete action spaces (one of the first attempts to use a computer to solve a problem was the work by Samuel to play checkers in Samuel (1959)), there has been a long tradition in reinforcement learning solving the types of continuous problems that arise in engineering applications. One of the most famous is the use of reinforcement learning to solve the problem of balancing an inverted pendulum on a cart that can only be moved to the right and left (Barto et al. (1983)). However, these problems are almost always solved by discretizing both the states and actions.

One of the major algorithmic advances in reinforcement learning was the introduction of an

algorithm known as Q-learning. The steps are extremely simple. Assume we are in some state s^n at iteration n, and we use some rule (policy) for choosing an action a^n. We next find a downstream state s^{n+1}, which we can compute in one of two ways. One is that we simply observe it from a physical system, given the pair (s, a). The other is that we assume we have a transition function $S^M(\cdot)$ and we compute $s^{n+1} = S^M(s^n, a^n, W^{n+1})$ where W^{n+1} is a sample realization of some random variable which is not known when we choose a^n (in the reinforcement learning community, a common assumption is that s^{n+1} is observed from some exogenous process). Then compute

$$\hat{q}^n = C(s^n, a^n) + \gamma \max_{a'} \bar{Q}^{n-1}(s^{n+1}, a'), \tag{7}$$

$$\bar{Q}^n(s^n, a^n) = (1 - \alpha_{n-1})\bar{Q}^{n-1}(s^n, a^n) + \alpha_{n-1}\hat{q}^n. \tag{8}$$

The quantities $\bar{Q}^n(s, a)$ are known as Q-factors, and they represent estimates of the value of being in a state s and taking action a. We let $Q(s, a)$ be the true values. Q-factors are related to value functions through

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

Q-learning was first introduced in Watkins (1989), and published formally in Watkins & Dayan (1992). Given assumptions on the stepsize α_n, and the way in which the state s and action a is selected, this algorithm has been shown to converge to the true value in Tsitsiklis (1994) and Jaakkola et al. (1994). These papers were the first to bring together the early field of reinforcement learning with the field of stochastic approximation theory from Robbins & Monro (1951) (see Kushner & Yin (2003) for an in-depth treatment). The research-caliber book Bertsekas & Tsitsiklis (1996) develops the convergence theory for reinforcement learning (under the name "neuro-dynamic programming") in considerable depth, and remains a fundamental reference for the research community.

Problems in reinforcement learning are almost entirely characterized by relative small, discrete action spaces, where 100 actions is considered fairly large. Problems in control theory are typically characterized by low-dimensional but continuous decisions (such as velocity, acceleration, position, temperature, density). Five or ten dimensions is considered large, but the techniques generally do not require that these be discretized. By contrast, there are many people in operations research who work on vector-valued problems, where the number of dimensions may be as small as 100, but easily range into the thousands or more. A linear program with 1,000 variables (which means 1,000 dimensions to the decision variable) are considered suitable for classroom exercises. These problems

appeared to be hopelessly beyond the ability of the methods being proposed for Markov decision processes, reinforcement learning or approximate dynamic programming as it was evolving within the control theory community.

There are, of course, many applications of high-dimensional optimization problems that involve uncertainty. One of the very earliest research papers was by none other than the inventor of linear programming, George Dantzig (see Dantzig (1955), Dantzig & Ferguson (1956)). This research has evolved almost entirely independently of the developments in computational methods for dynamic programming, and lives primarily as a subcommunity within the larger community focusing on deterministic math programming. The links between dynamic programming and stochastic programming are sparse, and this can be described (as of this writing) as a work in progress. There is by now a substantial community working in stochastic programming (see Kall & S.W.Wallace (1994), Higle & Sen (1996), Birge & Louveaux (1997), Sen & Higle (1999)) which has enjoyed considerable success in certain problem classes. However, while this field can properly claim credit to solving stochastic problems with high-dimensional decision vectors and complex state vectors, the price it has paid is that it is limited to very few time periods (or "stages," which refer to points in time when new information becomes available). The difficulty is that the method depends on a concept known as a "scenario tree" where random outcomes are represented as a tree which enumerates all possible sample paths. Problems with more than two or three stages (and sometimes with even as few as two stages) require the use of Monte Carlo methods to limit the explosion

In the 1990's, this author undertook the task of merging math programming and dynamic programming, motivated by very large scale problems in freight transportation. These problems were very high-dimensional, stochastic and often required explicitly modeling perhaps 50 time periods, although some applications were much larger. This work had its roots in models for dynamic fleet management Powell (1987), but evolved through the 1990's under the umbrella of "adaptive dynamic programming" (see Powell & Frantzeskakis (1990), Cheung & Powell (1996), Powell & Godfrey (2002)) before maturing under the name "approximate dynamic programming" (Powell & Van Roy (2004), Topaloglu & Powell (2006), Powell (2007), Simao et al. (2009), Powell (2010)).

Each subcommunity has enjoyed dramatic successes in specific problem classes, as algorithms have been developed that addresses the challenges of individual domains. Researchers in reinforcement learning have taught computers how to play chess and, most recently, master the complex Chinese game of Go. Researchers in robotics have taught robots to climb stairs, balance on a moving platform

and play soccer. In operations research, we have used approximate dynamic programming to manage the inventory of high value spare parts for an aircraft manufacturer (Powell & Simão (2009)), plan the movements of military airlift for the air force (Wu et al. (2009)), and solve an energy resource allocation problem with over 175,000 time periods (Powell et al. (2011)).

Despite theses successes, it is frustratingly easy to create algorithms that simply do not work, even for very simple problems. Below we provide an overview of the most popular algorithms, and then illustrate how easily we can create problems where the algorithms do not work.

3 Modeling a dynamic program

There is a diversity of modeling styles spanning reinforcement learning, stochastic programming and control theory. We propose modeling a dynamic program in terms of five core elements: states, actions, exogenous information, the transition function and the objective function. The community that studies Markov decision processes denote states by S, actions are a, rewards are $r(s,a)$ (we use a contribution $C(s,a)$) with transition matrix $p(s'|s,a)$. In control theory, states are x, controls are u, and rewards are $g(x,u)$. Instead of a transition matrix, in control theory they will use a transition function $x' = f(x,u,w)$. In math programming (and stochastic programming), a decision vector is x; they do not use a state variable, but instead define scenario trees, where a node n in a scenario tree captures the history of the process up to that node (typically a point in time).

We adopt the notational style of Markov decision processes and reinforcement learning, with two exceptions. We use the convention in control theory of using a transition function, also known as a system model, which we depict using $S_{t+1} = S^M(S_t, a_t, W_{t+1})$. We assume that any variable indexed by time t is known deterministically at time t. By contrast, control theorists, who conventionally work in continuous time, would write $x_{t+1} = f(x_t, u_t, w_t)$, where w_t is random at time t. We use two notational styles for actions. We use a_t for discrete actions, and x_t for vector-valued decisions, which will be familiar to the math programming community.

Some communities maximize rewards while others minimize costs. To conserve letters, we use $C(S,a)$ for contribution if we are maximizing, or cost if we are minimizing, which creates a minor notational conflict with reinforcement learning.

The goal in math programming is to find optimal decisions. In stochastic optimization, the goal

is to find optimal policies (or more realistically, the best policy within a well defined class), where a policy is a function that determines a decision given a state. Conventional notation is to write a policy as $\pi(s)$, but we prefer to use $A^\pi(S)$ to represent the function that returns an action a, or $X^\pi(S)$ for the function that returns a feasible vector x. Here, π specifies both the class of function, and any parameters that determine the particular function within a class Π.

Our optimization problem can then be written

$$\max_{\pi \in \Pi} \mathbb{E}^\pi \sum_{t=0}^{T} \gamma^t C(S_t, A^\pi(S_t)). \tag{9}$$

We index the expectation by π because the exogenous process (random variables) can depend on the decisions that were made. The field of stochastic programming assumes that this is never the case. The Markov decision process community, on the other hand, likes to work with an *induced stochastic process* (Puterman (2005)), where a sample path consists of a set of states and actions, which of course depends on the policy.

It is not unusual to see authors equating "dynamic programming" with Bellman's equation (1). In fact, equation (9) is the dynamic program, while as we show below, Bellman's equation is only one of several algorithmic strategies for solving our dynamic program.

4 Policies

Scanning the literature in stochastic optimization can produce what seems to be a daunting array of algorithmic strategies, which are then compounded by differences in notation and mathematical styles. Cosmetic differences in presentation can disguise similarities of algorithms, hindering the cross-fertilization of ideas.

For newcomers to the field, perhaps one of the most subtle concepts is the meaning of the widely used term "policy," which is a mapping from a state to an action. The problem is that from a computational perspective, policies come in different forms. For many, an example of a policy might be "if the number of filled hospital beds is greater than θ, do not admit any low priority patients." In another setting, a policy might involve solving a linear program, which can look bizarre to people who think of policies as simple rules.

There are many variations of policies, but our reading of the literature across different commu-

nities has identified four fundamental classes of policies: 1) myopic policies, 2) lookahead policies, 3) policy function approximations and 4) policies based on value function approximations. We describe these briefly below. Recognizing that there are entire communities focusing on the last three classes (the first is a bit of a special case), we then focus our attention for the remainder of the paper on value function approximations.

4.1 Myopic policies

A myopic policy maximizes contributions (or minimizes costs) for one time period, ignoring the effect of a decision now on the future. If we have a discrete action a, we would write this policy as

$$A^\pi(S_t) = \arg\max_{a \in \mathcal{A}} C(S_t, a).$$

Myopic policies can be optimal. Consider a portfolio balancing problem, where we have R_{ti} dollars in asset class i. Let x_{tij} be the movement of funds from class i to class j at time t. Let ρ_{ti} be the expected return from asset class i given what we know at time t, and let σ^2_{tij} be the covariance of returns from classes i and j (estimated at time t). A policy that balances risk and reward is given by

$$X^\pi(S_t) = \arg\max_{x \in \mathcal{X}_t} \left(\sum_j R_{t+1,j}\rho_{tj} - \beta \sum_i \sum_j R_{t+1,i} R_{t+1,j} \sigma^2_{tij} \right), \tag{10}$$

where β provides the weight that we want to put on the variability of our portfolio. The feasible region $\mathcal{X}_t$ is defined by the constraints

$$\sum_j x_{tij} = R_{ti}, \tag{11}$$

$$\sum_i x_{tij} = R_{t+1,j}, \tag{12}$$

$$x_{tij} \geq 0. \tag{13}$$

Note that there are no transaction costs in this model. We can move from asset class i to j at no cost. We then receive the expected return ρ_{tj} from class j. We also take the allocation $R_{t+1,j} = \sum_i x_{tij}$ and add a term that computes the variance of the resulting portfolio. This policy is optimal, because we can move from one state to any other state at no cost.

Myopic policies can be effective approximations for some problems. In the truckload trucking industry, the most widely used commercial package optimizes the assignment of drivers to loads without regard to the downstream impact. If $c_{td\ell}$ is the contribution from assigning driver d to load ℓ at time t. Our myopic policy might be written

$$X^\pi(S_t) = \arg\max_x \sum_d \sum_\ell c_{td\ell} x_{td\ell}.$$

Often, tunable parameters can be added to a myopic model to help overcome some of the more serious limitations. For example, in truckload trucking there may be loads that cannot be moved right away. There is an obvious desire to put a higher priority on moving loads that have been delayed, so we can add a bonus proportional to how long a load has been held. Let θ be the bonus per unit time that rewards moving a load that has been held. In this case, we would write the contribution $c_{td\ell}(\theta)$ as a function of θ, and we might write our policy $X^\pi(S_t|\theta)$ to express its dependence on θ. We might then write our objective as

$$F(\theta) = \mathbb{E} \sum_{t=0}^{T} \gamma^t c_t X^\pi(S_t|\theta).$$

The problem of optimizing over policies π in our objective function in equation (9) now consists of searching for the best value of θ.

We note that our policy with the bonus on rewarding the movement of loads that have been held the longest is not, strictly speaking, a myopic policy because the choice of θ is designed to capture, in an indirect way, the impact of a decision now on the future. But this also highlights the ease with which we can produce hybrid policies.

4.2 Lookahead policies

There is a substantial and growing literature which designs policies by looking into the future to determine the best decision to make now. The simplest illustration of this strategy is the use of tree search in an algorithm to find the best chess move. If we have to evaluate 10 possible moves each time, looking five steps into the future requires evaluating 10^5 moves. Now introduce the uncertainty of what your opponent might do. If she might also make 10 possible moves (but you are unsure which she will make), then for each move you might make, you have to consider the 10 moves your opponent might make, producing 100 possibilities for each step. Looking five moves into the future

now requires evaluating 10^{10} possibilities. The problem explodes dramatically when the decisions and random information are vectors.

The most common lookahead strategy uses a deterministic approximation of the future. Let $x_{tt'}$ be a vector of decisions that we determine now (at time t) to be implemented at time t' in the future. Let $c_{t'}$ be a deterministic vector of costs. We would solve the deterministic optimization problem

$$X^\pi(S_t) = \arg\max_t \sum_{t'=t}^{t+T} c_{t'} x_{tt'},$$

where $\arg\max_t$ returns only x_{tt}. We optimize over a horizon T, but implement only the decision that we wish to make now.

This strategy is widely known in operations research as a rolling horizon procedure, in computer science as a receding horizon procedure, and in engineering control as model predictive control. However, these ideas are not restricted to deterministic approximations of the future. We can also pose the original stochastic optimization problem over a restricted horizon, giving us

$$X^\pi(S_t) = \arg\max_{\pi'} \mathbb{E}\left(\sum_{t'=t}^{t+T} C(S_{t'}, Y^{\pi'}(S_{t'}))\right), \tag{14}$$

where $Y^{\pi'}(S_{t'})$ represents an approximation of the decision rule that we use within our planning horizon, but where the only purpose is to determine the decision we make at time t. Normally, we choose T small enough that we might be able to solve this problem (perhaps as a decision tree). Since these problems can explode even for small values of T, the stochastic programming community has adopted the strategy of breaking multiperiod problems into *stages* representing points in time where new information is revealed. The most common strategy is to use two stages. The first stage is "here and now" where all information is known. The second stage, which can cover many time periods, assumes that there has been a single point where new information has been revealed. Let $t = 0$ represent the first stage, and then let $t = 1, \ldots, T$ represent the second stage (this means that decisions at time $t = 1$ get to "see" the future, but we are only interested in the decisions to be implemented now). Let ω be a sample realization of what might be revealed in the second stage, and let Ω be a sample. Our stochastic programming policy would be written

$$X^\pi(S_0) = \arg\max_{x_0,(x_1(\omega),\ldots,x_T(\omega))} \left(c_0 x_0 + \sum_{\omega\in\Omega} p(\omega) \sum_{t=1}^{T} c_t(\omega) x_t(\omega)\right). \tag{15}$$

This problem has to be solved subject to constraints that link x_0 with the decisions that would be implemented in the future $x_t(\omega)$ for each outcome ω. In the stochastic programming community, an outcome ω is typically referred to as a *scenario*.

For problems with small action spaces, the deterministic rolling horizon procedure in equation (14) is trivial. However, for vector-valued problems, even this deterministic problem can be computationally daunting. However, even when this deterministic problem is not too hard, the expanded problem in equation (15) can be quite demanding, because it is like solving $|\Omega|$ versions of (14) all at the same time. Not surprisingly, then, stochastic programming has attracted considerable interest from specialists in large-scale computing.

A number of papers will break a problem into multiple stages. This means that for every outcome ω_1 in the first stage, we have a decision problem leading to a new sampling of outcomes ω_2 for the second stage. It is extremely common to see authors sampling perhaps 100 samples in Ω_1 for the first stage, but then limit the number of samples Ω_2 in the second stage (remember that the total number of samples is $|\Omega_1| \times |\Omega_2|$) to a very small number (under 10).

Because of the dramatic growth of problem size as Ω grows, the stochastic programming community has been devoting considerable effort to the science of sampling scenarios very efficiently. A sample of this research is given in Dupačová et al. (2000), Kaut & Wallace (2003), Growe-Kuska et al. (2003), Romisch & Heitsch (2009)). This area of research has recently become visible in the reinforcement learning community. Silver & Tesauro (2009) describes a method for optimizing the generation of Monte Carlo samples in a lookahead policy used to solve the Chinese game of Go.

A different strategy is to evaluate an action a by drawing on the knowledge of a "good" policy, and then simply simulating a single sample path starting with the ending state from action a. This is a noisy and certainly imperfect estimate of the value of taking action a, but it is extremely fast to compute, making it attractive for problems with relatively large numbers of actions. Such policies are known as roll-out policies (see Bertsekas & Castanon (1999)).

Lookahead policies represent a relatively brute-force strategy that takes little or no advance of any problem structure. They are well suited to problems with complex state variables. Lookahead policies were used in the highly successful efforts to develop computerized chess, and the recent breakthrough to develop a computer that could play the Chinese game of Go at an expert level.

The remaining two strategies require, in varying degrees, the ability to exploit problem structure

in some way.

4.3 Policy function approximations

There are numerous applications where the structure of a policy is fairly obvious. Sell the stock when the price goes over some limit θ; dispatch the shuttle when it has at least θ passengers (this could vary by time of day, implying that θ is a vector); if the inventory goes below some point q, order up to Q. We might write our inventory policy as

$$A^\pi(S_t) = \begin{cases} 0 & \text{If } S_t \geq q \\ Q - S_t & \text{If } S_t < q. \end{cases}$$

Finding the best policy means searching for the best values of q and Q.

In other cases, we might feel that there is a well-defined relationship between a state and an action. For example, we may feel that the release rate a_t is related to the level of water in a reservoir S_t that is described by the quadratic formula

$$A^\pi(S_t) = \theta_0 + \theta_1 S_t + \theta_2 (S_t)^2.$$

Here, the index π captures that the policy function is a quadratic. The search for the best policy in this class means finding the best vector $(\theta_0, \theta_1, \theta_2)$.

A very popular strategy in the engineering literature is to represent a policy as a neural network, depicted in figure 1. The neural network takes as input each dimension of the state variable. This is then filtered by a series of multiplicative weights (which we represent by θ for consistency) and signum functions to produce an estimate of what the action should be (see Haykin (1999) for a thorough introduction to the field of neural networks).

If we represent a policy using a statistical approximation such as a regression equation or neural network, we need some way to train our policy. These are often described as "actor-critic" methods, where some method is used to suggest an action, and then classical statistical methods are used to fit our policy to "predict" this action. We might use a lookahead policy to suggest an action, but the most common strategy is to use a policy based on a value function approximation (discussed next). The value function approximation is known as the critic, while the policy is known as the actor. The policy function approximation is desirable because it is typically very fast to compute (once it has

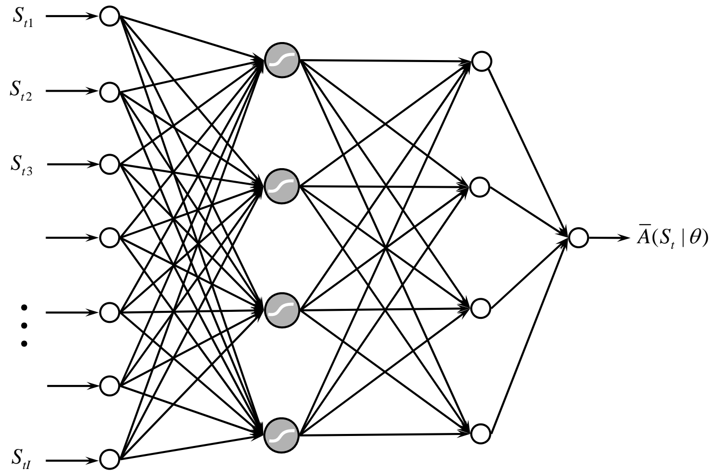

Figure 1: Illustration of a neural network with a single hidden layer.

been fitted), which is an important feature for some applications. Actor-critic methods are basically a form of policy iteration, familiar to the Markov decision process community since the 1950's (see in particular Howard (1960)).

A policy that is widely used in the reinforcement learning community for discrete actions is based on the Boltzmann distribution (also known as Gibbs sampling), where, given we are in state s, we choose an action a with probability $p(s, a|\theta)$ using

$$p(s, a|\theta) = \frac{e^{-\theta C(s,a)}}{\sum_{a' \in \mathcal{A}} e^{-\theta C(s,a')}}. \tag{16}$$

Here, θ is a tunable parameter. If $\theta = 0$, we are choosing actions at random. As θ grows, we would choose the action that appears to be best (based, for example, on the one-period contribution $C(s, a)$). In dynamic programming, we would use the one-period contribution plus an approximation of the downstream value function (discussed below). This policy is popular because it provides for a level of exploration of actions that do not appear to be the best, but may be the best. The parameter θ is used to control the tradeoff between exploration and exploitation. Not surprisingly, this policy is often described as a "soft-max" policy. See Silver & Tesauro (2009) for a nice illustration of the use of a Boltzmann policy.

The term "policy function approximation" is not common (but there is a small literature using this term). We use it because it nicely parallels "value function approximation," which is widely used. The key feature of a policy function approximation is that once it has been computed, finding

an action given a state does not involve solving an optimization problem (although we may have solved many optimization problems while fitting the policy).

Policy function approximations are particularly useful when we can identify obvious structure in the policy and exploit it. However, this strategy is limited to problems where such a structure is apparent (the Boltzmann policy is an exception, but this is limited to discrete action spaces).

4.4 Policies based on value function approximations

The fourth class of policy starts with Bellman's equation which we first stated in equation (1) in the familiar form which uses a one-step transition matrix. For our purposes, it is useful to start with the expectation form of Bellman's equation, given by

$$V(S_t) = \max_{a \in \mathcal{A}} \left(C(S_t, a) + \gamma \mathbb{E}\{V(S_{t+1})|S_t\} \right), \tag{17}$$

where $S_{t+1} = S^M(S_t, a, W_{t+1})$. For the purposes of our presentation, we can assume that the expectation operator is with respect to the random variable W_{t+1}, which may depend on the fact that we are in state S_t, and may also depend on the action a. There is a subtle distinction in the interpretation of the conditional expectation, which may be *conditionally dependent on* S_t (and possibly a), or a *function of* S_t and a.

The most widely cited problem with Bellman's equation (whether we are using (1) or (17)) is the "curse of dimensionality." These equations assume that are given $V_{t+1}(s)$, from which we can compute $V_t(s)$ for each state s. This algorithmic strategy assumes as a starting point that s is discrete. If s is a scalar, then looping over all states is generally not too difficult. However, if s is a vector, the number of potential states grows dramatically with the number of dimensions. The original insight (first tested in Bellman & Dreyfus (1959*b*)) was to replace the value function with a statistical approximation. This strategy received relatively little attention until the 1970's, when the control theory community started using neural networks to fit approximations of value functions.

This idea gathered considerable momentum when the idea of replacing the value function with a linear regression. In the language of approximate dynamic programming, let $\phi_f(S_t)$ be a *basis function*, which captures some feature of the underlying system. A feature might be, for example, the number of X's that a tic-tac-toe player has in corner positions, or the square of the amount of energy stored in a battery. Let $(\phi_f(S_t)), f \in \mathcal{F}$ be the set of features. We might then write our value

function approximation as

$$\bar{V}(S_t|\theta) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(S_t). \tag{18}$$

We have now reduced the problem of estimating the value $V(S_t)$ for each state S_t with the reduced problem of estimating the vector $(\theta_f)_{f \in \mathcal{F}}$.

We now have to develop a method for estimating this approximation. The original idea was that we could replace the backward dynamic programming equations such as (2) and (3) with equations that avoided the need to loop over all the states. Rather than stepping backward in time, the idea was to step forward in time, using an approximation of the value function to guide decisions.

Assume we are solving a finite horizon problem to make the indexing of time explicit. Let

$$\bar{V}_t^{n-1}(S_t) = \sum_{f \in \mathcal{F}} \theta_{tf}^{n-1} \phi_{tf}(S_t) \tag{19}$$

be an approximation of the value function at time t after $n-1$ observations. Now assume that we are in a single state S_t^n. We can compute an estimate of the value of being in state S_t^n using

$$\hat{v}_t^n = \max_a \left(C(S_t^n, a) + \gamma \sum_{s'} p(s'|S^n, a) \bar{V}_{t+1}^{n-1}(S_{t+1}) \right). \tag{20}$$

Further let a_t^n be the action that solves (20). In equations (2) and (3), we used the right hand side of (20) to update our estimate of the value of being in state. Now we propose to use $\hat{v}_t^n$ as a sample observation to update our approximate of the value function. If we were using a discrete, lookup table representation, we could update the estimate $\bar{V}_t^{n1}(S_t^n)$ of the value of being in state S_t^n using

$$\bar{V}_t^n(S_t^n) = (1 - \alpha_{n-1}) \bar{V}_t^{n-1}(S_t^n) + \alpha_{n-1} \hat{v}_t^n, \tag{21}$$

where α_{n-1} is a stepsize (also known as a smoothing factor or learning rate) that is less than 1. If we are using a linear regression model (also known as a linear architecture) as given in (19), we would use recursive least squares to update θ_t^{n-1}. The attraction of linear regression is that we do not need to visit every state since all we are doing is estimating the regression coefficients θ_t. This algorithmic strategy closely mirrors value iteration, so it is known as *approximate value iteration.*

Using this idea, we step forward in time, where the next state that we would visit might be given by

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

where ω^n is a sample path, and $W_{t+1}(\omega^n)$ is the information that becomes available between t and $t+1$. We now have an algorithm where we avoid the need to loop over states, and where we use linear regression to approximate the entire value function. It would seem that we have fixed the curse of dimensionality!

Unfortunately, while this strategy sometimes works spectacularly, there are no guarantees and it can fail miserably. The simplicity of the idea is behind its tremendous appeal, but getting it to work reliably has proved to be a difficult (if exciting) area of research. Some of the challenges include

- While it is tempting to make up a group of basis functions $(\phi_f), f \in \mathcal{F}$, it is quite important that they be chosen so that the right choice of θ in (19) produces an accurate approximation of the value function. This is the art of designing value function approximations, but it has to be done well, and requires a good understanding of how to capture the important qualities of the value function.
- The transition matrix is almost never available (if we can compute a matrix for every pair of states and every action, we do not need approximate dynamic programming). Often it is much easier to compute the expectation form of Bellman's equation in (17), but for most problems, the expectation cannot be computed. As a result, even if we could approximate the value function, we still cannot compute (20).
- Some communities focus exclusively on discrete action spaces, but there are many problems where the decision is a vector x. In such cases, we can no longer simply search over all possible actions to find the best one. We will have to solve some sort of high-dimensional optimization problem. Keep in mind that we still have the complication of computing an expectation within the optimization problem.
- While forward dynamic programming does not require that you loop over all states, it does require that you be able to approximate the value of every state that you *might* visit. Obtaining a good approximation of a state often requires either visiting the state, or visiting nearby states.

Often we may have to visit states just to approximate the value function in the vicinity of these states.

This algorithmic strategy has produced some tremendous successes. The engineering controls community has enjoyed significant breakthroughs designing controllers for aircraft, helicopters and robots using neural networks to approximate value functions. The computer science community has taught computers to play backgammon at an advanced level, and has recently developed a system that can play the Chinese game of Go at an expert level. In operations research, we have developed a system to optimize a fleet of 5,000 trucks with a very high level of detail to model drivers and loads, producing a state variable that is effectively infinite-dimensional and a decision vector with 50,000 dimensions (see Simao et al. (2009) for a technical description, and Simao et al. (2010) for a nontechnical description).

At the same time, it is frustratingly easy to create an algorithm that works poorly. For example:

- Consider a "dynamic program" with a single state and action. Each transition incurs a random reward $\hat{R}^n$. We would like to compute the value of being in our single state. We quickly recognize that this (scalar) value is given by

  $$\begin{aligned} V &= \mathbb{E}\sum_{n=0}^{\infty}\gamma^n\hat{R}^n \\ &= \frac{1}{1-\gamma}\mathbb{E}\hat{R}. \end{aligned}$$

  If we know the expected value of $\hat{R}$, we are done. But let's try to estimate this value using our approximate value iteration algorithm. Let $\hat{v}^n$ be given by

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

  Because $\hat{v}^n$ is a noisy observation, we smooth it to obtain an updated estimate

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

  We know that this algorithm is provably convergent if we use a stepsize $\alpha_{n-1} = 1/n$. However, we can *prove* that this algorithm requires 10^{20} iterations to converge within one percent of optimal if $\gamma = .95$. This is a trivial algorithm to code in a spreadsheet, so we encourage the reader to try it out.

- Now consider a simple but popular algorithm known as Q-learning, where we try to learn the value of a state-action pair. Let $Q(s,a)$ be the true value of being in state s and taking action a. If we had these Q-factors, the value of being in state s would be $V(s) = \max_a Q(s,a)$. The basic Q-learning algorithm is given by

  $$\hat{q}^n(s^n, a^n) = \hat{C}(s^n, a^n) + \gamma \max_a \bar{Q}^{n-1}(s', a').$$

  where $s' = S^M(s^n, a^n, W^{n+1})$, and where we have randomly sampled W^{n+1} from some distribution. Given the state s^n, the action a^n is sampled according to some policy that ensures that all actions are sampled infinitely often. Because $\hat{v}^n$ is a noisy observation, we smooth it to obtain an updated estimate

  $$\bar{Q}^n(s^n, a^n) = (1 - \alpha_{n-1})\bar{Q}^{n-1}(s^n, a^n) + \alpha_{n-1}\hat{q}^n(s^n, a^n).$$

  This algorithm has been shown to converge to the optimal Q-factors, from which we can infer an optimal policy by choosing the best action (given a state) from these factors. In our previous example, the poor performance was due to the fact that the stepsize $1/n$ works unusually poorly. Now we are going to use a carefully tuned stepsize to give the fastest possible convergence. However, if the observations $\hat{R}^n$ are noisy, and if the discount factor γ is close enough to 1, Q-learning will appear to diverge for what can be tens of millions of iterations, growing endlessly past the optimal solution.

- If we use linear basis functions, it is well known that approximate value iteration can diverge, even if we start with the true, optimal solution! The biggest problem with linear basis functions is that we rarely have any guarantees that we have chosen a good set of basis functions. Figure 2 demonstrates that if we try to learn the shape of a nonlinear function using a linear approximation when the state sampling distribution is shifted to the left, then we may learn an upward sloping function that encourages larger actions (which may lead to larger states). This may shift our sampling distribution to the right, which changes the slope of the approximation, which may then encourage smaller actions.

4.5 Approximating functions

The use of policy function approximations and value function approximations both require approximating a function. There are three fundamental ways of accomplishing this: lookup tables, parametric models, and nonparametric models.

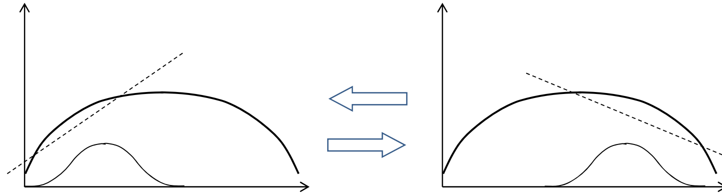

Figure 2: Learning a nonlinear function using a linear function with two different state sampling strategies.

Lookup tables require estimating a tables that maps a discrete state to an action (for policy function approximations) or to a value of being in a state (for value function approximations). In classical textbook treatments of discrete Markov decision processes, a "policy" is almost always intended to mean a lookup table that specifies an action for every state (see Puterman (2005)). Lookup tables offer the feature that if we visit each state infinitely often, we will eventually learn the right value of being in a state (or the best action). However, this means estimating a parameter (a value or action) for each state, which does not scale past very small problems.

Parametric models have attracted the most attention because they are the simplest way to approximate an entire value function (or policy) using a small number of parameters. We might approximate a value function using

$$\bar{V}(S|\theta) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(S), \tag{22}$$

where $\phi_f(S)$ is known generally as a *feature* (because it extracts a specific piece of information from the state variable S) or a *basis function* (in the hopes that the set $(\phi_f(S))$, $f \in \mathcal{F}$ span the space of value functions). Generally the number of features $|\mathcal{F}|$ will be dramatically smaller than the size of the state space, making it much easier to work in terms of estimating the vector $(\theta_f)_{f \in \mathcal{F}}$. We might also have a parametric representation of the policy. If S is a scalar such as the amount of energy in a battery and a is the amount of energy to store or withdraw, we might feel that we can express this relationship using

$$A^\pi(S) = \theta_0 + \theta_1 S + \theta_2 S^2.$$

The challenge of nonparametric models is that it is necessary to design the functional form (such as in (22)), which not only introduces the need for human input, it also introduces the risk that we

do a poor job of choosing basis functions. For this reason, there has been growing interest in the use of nonparametric models. Most forms of nonparametric modeling estimate a function as some sort of weighting of observations in the region of the function that we are interested in evaluating. For example, let S^n be an observed state, and let $\hat{v}^n$ be the corresponding noisy observation of the value of being in that state. Let $(\hat{v}^m, S^m)_{m=1}^n$ be all the observations we have made up through iteration n of our algorithm. Now assume that we would like an estimate of the value of some query state s. We might write this using

$$\bar{V}(s) = \sum_{m=1}^{n} \frac{\hat{v}^m k(s, S^m)}{\sum_{m'=1}^{n} k(s, S^{m'})}.$$

Here, $k(s, S^m)$ is known as a *kernel function* which determines the weight that we will put on $\hat{v}^m$ based on the distance between the observed state S^m and our query state s.

4.6 Direct policy search

Each of the four major classes of policies (combined with the three ways of approximating policy functions or value functions) offers a particular algorithmic strategy for solving a dynamic programming problem. However, cutting across all of these strategies is the ability to tune parameters using stochastic search methods. We refer to this general approach as direct policy search.

To illustrate this idea, we start by noting that any policy can be modified by tunable parameters, as indicated by the following examples:

Myopic policies Let $x_{tij} = 1$ if we assign resource i to a task j at time t, returning a contribution c_{ij} (this could be the negative of a cost). A purely myopic policy would minimize costs by choosing x_t to maximize $\sum_i \sum_j c_{ij} x_{tij}$ subject to flow conservation constraints. Further assume that if we do not assign any resource to a task j at time t, then the task is held to the next time period. Our myopic policy might result in a particular task being delayed a long time. Let τ_{tj} be the number of time periods that task j has been held as of time t. We might use a modified contribution $c^{\pi}_{tij}(\theta) = c_{ij} - \theta \tau_{tj}$, where θ is a positive coefficient that reduces costs for tasks that have been delayed. Our policy $X^{\pi}(S_t|\theta)$ is computed using

$$X^{\pi}(S_t|\theta) = \arg\max_{x_t} \sum_i \sum_j c^{\pi}_{tij}(\theta) x_{tij}$$

for a given parameter θ.

Lookahead policies Lookahead policies have to be designed given decisions on the planning horizon T, and the number of scenarios S that are used to approximate random outcomes. Let $\theta = (T, S)$, and let $A^\pi(S_t|\theta)$ be the first-period decision produced by solving the lookahead problem.

Policy function approximations These are the easiest to illustrate. Imagine that we have an inventory problem where we order inventory if it falls below a level q, in which case we order enough to bring it up to Q. $A^\pi(S_t|\theta)$ captures this rule, with $\theta = (q, Q)$. Another example arises if we feel that the decision of how much energy to storage in a battery, a_t, is related to the amount of energy in the battery, S_t, according to the function

$$A^\pi(S_t|\theta) = \theta_0 + \theta_1 S_t + \theta_2 S_t^2.$$

Policies based on value function approximations A popular approximation strategy is to write a value function using basis functions as we illustrate in equation (18). This gives us a policy of the form

$$A^\pi(S_t|\theta) = \arg\max_a \left(C(S_t, a) + \gamma \mathbb{E} \sum_f \theta_f \phi_f(S_{t+1}) \right).$$

These examples show how each of our four classes of policies can be influenced by a parameter vector θ. If we use policy function approximations, we would normally find θ so that our value function approximation fits observations of the value of being in a state (this is a strategy we cover in more depth later in this paper). However, we may approach the problem of finding θ as a stochastic search problem, where we search for θ to solve

$$\max_\theta F^\pi(\theta) = \mathbb{E} \sum_{t=0}^{T} \gamma^t C(S_t, A^\pi(S_t|\theta)), \tag{23}$$

where $C(S_t, a_t)$ is the contribution we earn at time t when we are in state S_t and use action $a_t = A^\pi(S_t|\theta)$. We assume that the states evolve according to $S_{t+1} = S^M(S_t, a_t, W_{t+1}(\omega))$ where ω represents a sample path.

Typically, we cannot compute the expectation in (23), so we resort to stochastic search techniques. For example, assume that we can compute a gradient $\nabla_\theta F(\theta, \omega)$ for a particular sample path ω. We might use a stochastic gradient algorithm such as

$$\theta^n = \theta^{n-1} + \alpha_{n-1} \nabla_\theta F(\theta^{n-1}, \omega^n).$$

Of course, there are many problems where the stochastic gradient is not easy to compute, in which case we have to resort to other classes of stochastic search policies. An excellent overview of stochastic search methods can be found in Spall (2003). An overview of methods can be found in chapter 7 of Powell (2011).

4.7 Comments

All of these four types of policies, along with the three types of approximation strategies, have attracted attention in the context of different applications. Furthermore, there are many opportunities to create hybrids. A myopic policy to assign taxicabs to the closest customer might be improved by adding a penalty for holding taxis that have been waiting a long time. Lookahead policies, which optimize over a horizon, might be improved by adding a value function approximation at the end of the horizon. Policy function approximations, which directly recommend a specific action, can be added to any of the other policies as a penalty term for deviating from what appears to be a recommended action.

This said, policy function approximations tend to be most attractive for relatively simpler problems where we have a clear idea of what the nature of the policy function might look like (exceptions to this include the literature where the policy is a neural network). Lookahead policies have been attractive because they do not require coming up with a policy function or value function approximation, but they can be computationally demanding.

For the remainder of this paper, we focus on strategies based on value function approximations, partly because they seem to be applicable to a wide range of problems, and partly because they have proven frustratingly difficult to develop as a general purpose strategy.

5 The three curses of dimensionality and the post-decision state variable

There has been considerable attention given to "the" curse of dimensionality in dynamic programming, which always refers to the explosion in the number of states as the number of dimensions in the state variable grows. Of course, this growth in the size of the state space with the dimensions refers to discrete states, and ignores the wide range of problems in fields such as engineering, economics and operations research where the state variable is continuous.

At the same time, the discussion of the curse of dimensionality ignores the fact that for many applications in operations research, there are three curses of dimensionality. These are

1. The state space - Let $S_t = (S_{t1}, S_{t2}, \ldots, S_{td})$. If S_{ti} as N possible values, our state space has N^d states. Of course, if states are continuous, even a scalar problem has an infinite number of states, but there are powerful techniques for approximating problems with a very small number of dimensions (see Judd (1998)).

2. The outcome space - Assume that our random variable W_t is a vector. It might be a vector of prices, or demands for different products, or a vector of different components such as the price of electricity, energy from wind, behavior of drivers, temperature and other parameters that affect the behavior of an energy system. Computing the expectation (which is implicit in the one-step transition matrix) will generally involve nested summations over each dimension. Spatial resource allocation problems that arise in transportation can have random variables with hundreds or thousands of dimensions.

3. The action space - The dynamic programming community almost exclusively thinks of an action a_t as discrete, or perhaps a continuous variable with a very small number of dimensions which can be discretized. In operations research, we typically let x_t be a vector of decisions, which may have thousands or tens of thousands of dimensions. In this paper, we use a to represent discrete actions, while x refers to vectors of actions that may be discrete or continuous (either way, the set of potential values of x will always be too large to enumerate).

One of the most problematic elements of a dynamic program is the expectation. Of course there are problems where this is easy to compute. For example, we may have a problem of controlling the arrival of customers to a hospital queue, where the only random variable is binary, indicating whether a customer has arrived. But there are many applications where the expectation is difficult or impossible to compute. The problem may be computational, because we cannot handle the dimensionality of a vector of random variables. Or it may be because we simply do not know the probability law of the random variables.

The reinforcement learning community often works on problems where the dynamic system involves humans or animals making decisions. The engineering controls community might work on a problem to optimize the behavior of a chemical plant, where the dynamics are so complex that they cannot be modeled. These communities have pioneered research on a branch of optimization known

as *model-free* dynamic programming. This work assumes that given a state S_t and action a_t, we can only observe the next state S_{t+1}. The reinforcement learning community uses the concept of Q-learning (equations (7)-(8)) to learn the value of a state-action pair. An action is then determined using

$$a_t = \arg\max_a \bar{Q}^n(S_t, a)$$

to determine the best action to be taken now. This calculation does not require knowing a transition function, or computing an expectation. However, it does require approximating the Q-factors, which depends on both a state *and* an action. Imagine if the action is a vector!

An alternative strategy that works on many applications (sometimes dramatically well) is to use the idea of the post-decision state variable. If S_t is the state just before we make a decision (the "pre-decision state"), then the post-decision state S_t^a is the state at time t, immediately after we have made a decision. The idea of post-decision states has been around for some time, often under different names such as the "after-state" variable (Sutton & Barto (1998)), or the "end of period" state (Judd (1998)). We prefer the term post-decision state which was first introduced by Van Roy et al. (1997). Unrealized at the time, the post-decision state opens the door to solving problems with very high-dimensional decision vectors. This idea is discussed in much greater depth in chapter 4 of Powell (2011), but a few examples include:

- Let R_t be the resources on hand (water in a reservoir, energy in a storage device, or retail inventory), and let $\hat{D}_{t+1}$ be the demand that needs to be satisfied with this inventory. The inventory equation might evolve according to

  $$R_{t+1} = \max\{0, R_t + a_t - \hat{D}_{t+1}\}.$$

  The post-decision state would be $R_t^a = R_t + a_t$ while the next pre-decision state is $R_{t+1} = R_t^a + a_t$.

- Let S_t be the state of a robot, giving its location, velocity (with speed and direction) and acceleration. Let a_t be a force applied to the robot. The post-decision state S_t^a is the state that we intend the robot to be in at some time $t+1$ in the future. However, the robot may not be able to achieve this precisely due to exogenous influences (wind, temperature and external interference). It is often the case that we might write $S_{t+1} = S_t^a + \epsilon_{t+1}$, where ϵ_{t+1} captures

the external noise, but it may also be the case that this noise depends on the current state and action.

- A driver is trying to find the best path through a network with random arc costs. When she arrives at a node i, she is allowed to see the actual cost $\hat{c}_{ij}$ on each link out of node i. For other links in the network (such as those out of node j), she only knows the distribution. If she is at node i, her state variable would be $S_t = (i, (\hat{c}_{ij})_j)$, which is fairly complex to deal with. If she makes the decision to go to node j (but before she has actually arrived to node j), her post-decision state is $S_i^a = (j)$.

- A cement manufacturer allocates its inventory of cement among dozens of construction jobs in the region. At the beginning of a day, he knows his inventory R_t and the vector of demands $D_t = (D_{ti})_i$ that he has to serve that day. He then has to manage delivery vehicles and production processes to cover these demands. Any demands that cannot be covered are handled by a competitor. Let x_t be the vector of decisions that determine which customers are satisfied, and how much new cement is made. The pre-decision state is $S_t = (R_t, (D_{ti})_i)$, while the post-decision state is the scalar $S^x = (R_t^x)$, giving the amount of cement in inventory at the end of the day.

The concept of a post-decision state can also be envisioned using a decision tree, which consists of decision nodes (points where decisions are made), and outcome nodes, from which random outcomes occur, taking us to a new decision node. The decision node is a pre-decision state, while the outcome node is a post-decision state. However, the set of post-decision nodes is, for some problems, very compact (the amount of cement left over in inventory, or the set of nodes in the network). At the same time, there are problems where a compact post-decision state does not exist. In the worst case, the post-decision state consists of $S_t^a = (S_t, a_t)$, which is to say the state-action pair. This is just what is done in Q-learning.

If we use the idea of the post-decision state, Bellman's equation can be broken into two steps:

$$V_t(S_t) = \max_a \left(C(S_t, a) + \gamma V_t^a(S_t^a) \right), \quad (24)$$

$$V_t^a(S_t^a) = \mathbb{E}\{V_{t+1}(S_{t+1}) | S_t^a\}. \quad (25)$$

Here, we assume there is a function $S_t^a = S^{M,a}(S_t, a)$ that accompanies our standard transition function $S_{t+1} = S^M(S_t, a_t, W_{t+1})$. It is very important to recognize that equation (24) is deterministic,

which eliminates the second curse of dimensionality. For most applications, we will never be able to compute $V_t^a(S_t^a)$ exactly, so we would replace it with an approximation $\bar{V}_t(S_t^a)$. If this approximation is chosen carefully, we can replace small actions a_t with large vectors x_t. Indeed, we have done this for problems where x_t has tens of thousands of dimensions (see Simao et al. (2009) and Powell et al. (2011)).

It is easiest to illustrate the updating of a value function around the post-decision state if we assume a lookup table representation. Assume that we are in a discrete, pre-decision state S_t^n at time t while following sample path ω^n. Further assume that we transitioned there from the previous post-decision state $S_{t-1}^{a,n}$. Now assume that we obtain an observation of the value of being in state S_t^n using

$$\hat{v}_t^n = \max_a \left(C(S_t^n, a) + \gamma \bar{V}_t^{n-1}(S_t^a) \right).$$

We would update our value function approximation using

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

So, we are using an observation $\hat{v}_t^n$ of the value of being in a pre-decision state S_t^n to update the value function approximation around the previous, post-decision state.

The post-decision state allows us to solve problems with vector-valued decision variables and expectations that cannot be calculated (or even directly approximated). We have also described four major classes of policies that might be used to make decisions. For the remainder of the paper, we are going to focus on the fourth class, which means we return to the problem of approximating value functions.

Designing policies by approximating value functions has, without question, attracted the most attention as a strategy for solving complex dynamic programming problems, although the other three policies are widely used in practice. Lookahead policies and policy function approximations also continue to be active areas of research for specific problem classes. But the strategy of designing policies based on value function approximations carries tremendous appeal. However, the early promise of this strategy has been replaced with the realization that it is much harder than people originally thought.

There are three steps to the process of designing policies based on value function approxima-

tions. These include: 1) designing a value function approximation, 2) updating the value function approximation for a fixed policy, and 3) learning value function approximations while simultaneously searching for a good policy. The next three sections deal with each of these steps.

6 Value function approximations

The first step in approximating a value function is to choose an architecture. In principle we can draw on the entire field of statistics, and for this reason we refer the reader to references such as Hastie et al. (2009) for an excellent overview of the vast range of parametric and nonparametric learning methods. Below, we describe three powerful methods. The first is a nonparametric method that uses hierarchical aggregation, while the second is recursive least squares using a linear model, which is the method that has received the most attention in the ADP/RL communities. The third describes the use of concave value function approximations that are useful in the setting of resource allocation.

6.1 Hierarchical aggregation

Aggregation involves mapping sets of states into a single state whose value is then used as an approximation of all the states in the set. Not surprisingly, this introduces aggregation errors, and requires that a tradeoff be made in choosing the level of aggregation. A more powerful strategy is to use a family of aggregation functions which, for the purposes of our discussion, we will assume is hierarchical. We are going to estimate the value of states at different levels of aggregation, and then use a weighted sum to estimate the value function.

Let $G^g(s)$ be a mapping from a state s to an aggregated state $s^{(g)}$. We assume we have a family of these aggregation functions $g \in \mathcal{G}$, where $g = 0$ corresponds to the most disaggregate level (and this may be arbitrarily fine). We propose to estimate the value of being in a state s using

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

where $\bar{v}^{(g,n)}(s)$ is an estimate of the value of being in aggregated state $G^g(s)$ after n observations have been made. This estimate is weighted by $w^{(g)}(s)$, which depends on both the level of aggregation and the state being observed. This is important, because as we obtain more observations of an

aggregated state, we want to give it a higher weight. We compute the weights using

$$w_s^{(g,n)} \propto \left((\bar{\sigma}^2(s))^{(g,n)} + \left(\bar{\mu}^{(g,n)}(s) \right)^2 \right) \tag{26}$$

where $(\bar{\sigma}^2(s))^{(g,n)}$ is an estimate of the variance of $\bar{v}^{(g,n)}(s)$, and $\bar{\mu}^{(g,n)}(s)$ is an estimate of the bias due to aggregation error. These quantities are quite easy to estimate, but we refer the reader to George et al. (2008) for the full derivation (see also chapter 8 in Powell (2011)).

Hierarchical aggregation offers the advantage that as the number of observations grows, the approximation steadily improves. It also provides rough approximations of the value function after just a few observations, since most of the weight at that point will be on the highest levels of aggregation.

A popular form of aggregation in the reinforcement learning community is known as tilings, which uses overlapping aggregations to create a good approximation. When we use hierarchical aggregation, we assume that the function $G^g(s)$ represents states more coarsely than $G^{g-1}(s)$, and at the same time we assume that if two states are aggregated together at aggregation level g, then they will also be aggregated together at level $g+1$. With tilings, there is a family of aggregations, but they are all at the same level of coarseness. However, if one aggregation can be viewed as a series of squares ("tiles") that cover an area, then the next aggregation might be squares of the same size that are shifted or tilted in some way so that the surface is being approximated in a different way.

6.2 Basis functions

The approximation strategy that has received the most attention is the use of linear parametric models of the form given in (22), where we assume that we have designed a set of basis functions $\phi_f(s)$. For ease of reference, we repeat the approximation strategy which is given by

$$\bar{V}(S_t|\theta) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(S_t). \tag{27}$$

We note that the functions $\phi_f(S_t)$ are known in the ADP/RL communities as basis functions, but elsewhere in statistics they would be referred to as independent variables or covariates. The biggest challenge is designing the set of basis functions (also known as features) $\phi_f(s)$, but for the moment we are going to take these as given. In this section, we tackle the problem of estimating the parameter vector θ using classical methods from linear regression.

Assume we have an observation $\hat{v}^m$ of the value of being in an observed state S^m. At iteration n, we have the vector $(\hat{v}^m, S^m)_{m=1}^n$. Let ϕ^m be the vector of basis functions evaluated at S^m. Now let Φ^n be a matrix with n rows (one corresponding to each observed state S^m), and $|\mathcal{F}|$ columns (one for each feature). Let $\hat{V}^n$ be a column vector with element $\hat{v}^m$. If we used batch linear regression, we could compute the regression vector using

$$\theta^n = ((\Phi^n)^T \Phi^n)^{-1} (\Phi^n)^T \hat{V}^n. \tag{28}$$

Below we describe how this can be done recursively without performing the matrix inverse.

6.3 Piecewise linear, separable functions

There is a broad class of problems that can be best described as "resource allocation problems" where there is a quantity of a resource (blood, money, water, energy, people) that need to be managed in some way. Let R_{ti} be the amount of the resource in a state i at time t, where i might refer to a blood type, an asset type, a water reservoir, an energy storage device, or people with a particular type of skill. Let x_{tij} be the amount of resource in state i that is moved to state j at time t. We may receive a reward from having a resource in state i at time t, and we may incur a cost from moving resources between states. Either way, we can let $C(R_t, x_t)$ be the total contribution earned from being in a resource state R_t and implementing action x_t.

Assume that our system evolves according to the equation

$$R_{t+1,j} = \sum_i x_{tij} + \hat{R}_{t+1,j},$$

where $\hat{R}_{t+1,j}$ is exogenous changes to the resource level in state j (blood donations, rain fall, financial deposits or withdrawals). We note that the post-decision state is given by $R^x_{tj} = \sum_i x_{tij}$. Using a value function approximation, we would make decisions by solving

$$X^\pi(R_t) = \arg\max_x \big(C(R_t, x_t) + \gamma \bar{V}_t(R^x_t) \big), \tag{29}$$

subject to the constraints

$$\sum_j x_{tij} = R_{ti}, \tag{30}$$

$$x_{tij} \geq 0. \tag{31}$$

For this problem, it is straightforward to show that the value function is concave in R_t. A convenient way to approximate the value function is to use a separable approximation, which we can write as

$$\bar{V}_t(R_t^x) = \sum_i \bar{V}_{ti}(R_{ti}^x),$$

where $\bar{V}_{ti}(R_{ti}^x)$ is piecewise linear and concave.

We estimate $\bar{V}_{ti}(R_{ti}^x)$ by using estimates of the slopes. If we solve (29) − (31), we would obtain dual variable for the resource constraint (30), which gives the marginal value of the resource vector R_{ti}. Thus, rather than observing the value of being in a state which we have denoted by $\hat{v}_t^n$, we would find the dual variable $\hat{\nu}_{ti}^n$ for constraint (30).

Chapter 13 of Powell (2011) describes several methods for approximating piecewise linear, concave value function approximations for resource allocation problems. The key idea is that we are using derivative information (dual variables) to update the slopes of the function. This strategy scales to very high-dimensional problems, and has been used in production systems to optimize freight cars and locomotives at Norfolk Southern Railroad (Powell & Topaloglu (2005)), spare parts for Embraer (Powell & Simao (2009)), and truck drivers for Schneider National (Simao et al. (2009)), as well as planning systems for transformers (Enders et al. (2010)) and energy resource planning (Powell (2010)).

7 Updating the value function for a fixed policy

Updating the value function requires computing an observation $\hat{v}^n$ of the value of being in a state S^n, and then using this observation to update the value function approximation itself. We deal with each of these aspects of the updating process below. Throughout this section, we assume that we have fixed the policy, and are simply trying to update an estimate of the value function for this policy.

7.1 Policy simulation

The most direct way of observing the value of a state S^n is simply to simulate a policy into the future. This is easiest to envision for a finite horizon problem. Assume that we are at time t while

following sample path ω^n. If $A^\pi(S_t)$ is our policy, we would compute

$$\hat{v}_t^{\pi,n} = \sum_{t'=t}^{T} \gamma^{t'-t} C(S_{t'}, A^\pi(S_{t'})).$$

where $S_{t+1} = S^M(S_t^n, A^\pi(S_t^n), W_{t+1}(\omega^n))$. This is generally calculated by performing a forward pass through time (traversing backward through the same sequence of states), and then calculating $\hat{v}_t^n$ using a backward pass, that can also be written

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

This method works, of course, only for finite horizon problems.

7.2 Stochastic gradient updates

Assume that we are trying to estimate a value v which is the true value of being in state s, and further assume that we can observe a random variable $\hat{v}$ that is an unbiased estimate of the value of being in state s. We would like to find v by solving

$$\min_v \mathbb{E} F(v, \hat{v}), \tag{32}$$

where

$$F(v, \hat{v}) = \frac{1}{2}(v - \hat{v})^2.$$

A simple algorithm for solving this problem uses a stochastic gradient where we find an estimate of v using

$$\begin{aligned} v^n &= v^{n-1} - \alpha_{n-1} \nabla F(v^{n-1}, \hat{v}^n) \qquad (33) \\ &= v^{n-1} - \alpha_{n-1}(v^{n-1} - \hat{v}^n). \end{aligned}$$

Now assume that we are updating a lookup table approximation $\bar{V}^n(s)$. If we observe $s = S_t^n$, we would use the updating equation

$$\bar{V}_t^n(S_t^n) = \bar{V}_t^{n-1}(S_t^n) - \alpha_{n-1}(\bar{V}_t^{n-1}(S_t^n) - \hat{v}^n). \tag{34}$$

Now replace our lookup table representation with a linear model $\bar{V}(s|\theta) = \theta^T\phi = \sum_{f\in\mathcal{F}} \theta_f \phi_f(s)$. We can search for the best value of θ using a stochastic gradient algorithm applied to the problem

$$\min_\theta \mathbb{E}\frac{1}{2}(\bar{V}(s|\theta) - \hat{v})^2.$$

The stochastic gradient updating equation is

$$\theta^n = \theta^{n-1} - \alpha_{n-1}(\bar{V}(s|\theta^{n-1}) - \hat{v}^n)\nabla_\theta \bar{V}(s|\theta^n). \tag{35}$$

Since $\bar{V}(s|\theta^n) = \sum_{f\in\mathcal{F}} \theta^n_f \phi_f(s) = (\theta^n)^T\phi(s)$, the gradient with respect to θ is given by

$$\nabla_\theta \bar{V}(s|\theta^n) = \begin{pmatrix} \frac{\partial \bar{V}(s|\theta^n)}{\partial \theta_1} \\ \frac{\partial \bar{V}(s|\theta^n)}{\partial \theta_2} \\ \vdots \\ \frac{\partial \bar{V}(s|\theta^n)}{\partial \theta_F} \end{pmatrix} = \begin{pmatrix} \phi_1(s^n) \\ \phi_2(s^n) \\ \vdots \\ \phi_F(s^n) \end{pmatrix} = \phi(s^n).$$

Equation (35) can now be written as

$$\begin{aligned} \theta^n &= \theta^{n-1} - \alpha_{n-1}(\bar{V}(s|\theta^{n-1}) - \hat{v}^n)\phi(s^n) \\ &= \theta^{n-1} - \alpha_{n-1}(\bar{V}(s|\theta^{n-1}) - \hat{v}^n) \begin{pmatrix} \phi_1(s^n) \\ \phi_2(s^n) \\ \vdots \\ \phi_F(s^n) \end{pmatrix}. \end{aligned} \tag{36}$$

Stochastic gradient algorithms underlie many algorithms used in approximate dynamic programming, as we see below. A key issue is the choice of stepsize rule, a topic that is important for certain algorithmic strategies. For an in-depth discussion of stepsize rules, see chapter 11 of Powell (2011).

7.3 Temporal difference learning

Perhaps the most popular approach for approximating the value of being in a state is by observing the one-step contribution from an action, and then adding an approximation of the downstream value of being in the next state we visit. This is an approximate form of value iteration, where we would use

$$\hat{v}^{\pi,n}_t = C(S^n_t, A^\pi(S^n_t)) + \gamma \bar{V}^{\pi,n-1}_{t+1}(S_{t+1}),$$

where $S_{t+1} = S^M(S_t^n, A^\pi(S_t^n), W_{t+1}(\omega^n))$ is a simulation of the next state that we might visit. This is known as bootstrapping, because our estimate of the value of being in state S_t^n depends on our current approximation of the value of being in state S_{t+1}. If we use $\hat{v}_t^n$ to update the value function around the pre-decision state (we do this to keep the presentation as simple as possible), we would perform the following calculation:

$$\begin{aligned}
\bar{V}^{\pi,n}(S_t^{a,n}) &= (1-\alpha_{n-1})\bar{V}^{\pi,n-1}(S_t^{a,n}) + \alpha_{n-1}\hat{v}_t^{\pi,n} \\
&= (1-\alpha_{n-1})\bar{V}^{\pi,n-1}(S_t^{a,n}) + \alpha_{n-1}(C(S_t^n, A^\pi(S_t^n)) + \gamma\bar{V}_{t+1}^{\pi,n-1}(S_{t+1})) \\
&= \bar{V}^{\pi,n-1}(S_t^{a,n}) + \alpha_{n-1}(C(S_t^n, A^\pi(S_t^n)) + \gamma\bar{V}_{t+1}^{\pi,n-1}(S_{t+1}) - \bar{V}^{\pi,n-1}(S_t^{a,n})). \quad (37)
\end{aligned}$$

Let

$$\delta_t^{\pi,n} = C(S_t^n, A^\pi(S_t^n)) + \gamma\bar{V}_{t+1}^{\pi,n}(S_{t+1}) - \bar{V}^{\pi,n-1}(S_t^{a,n}).$$

The quantity $\delta^{\pi,n}(s)$ is widely known in the reinforcement learning community as the *temporal difference*. The name derives from the interpretation of the iteration counter n as representing time. $\bar{v}^{\pi,n-1}(s)$ can be thought of as the current estimate of the value of being in state s, while $C(s,a) + \gamma\bar{v}^{\pi,n-1}(s)$ can be viewed as an updated estimate. The difference is the change in the estimate of the value of being in a state over "time." We can now write our updating formula (37) as

$$\bar{V}^{\pi,n}(S_t^{a,n}) = \bar{V}^{\pi,n-1}(S_t^{a,n}) + \alpha_{n-1}\delta_t^{\pi,n}.$$

In the reinforcement learning literature, temporal difference learning is generally viewed in the context of learning a fixed policy (as we have done). It is the same updating we would do if we were using approximate value iteration, which is often associated with algorithms where we are simultaneously optimizing a policy. However, the basic updating step is the same.

7.4 Recursive least squares

The use of basis functions (linear regression) is one of the most popular approaches for avoiding the problems of large state spaces. In section 6.2 we provided the basic equations for estimating the regression vector θ using batch methods, which requires finding $((\Phi^n)^T\Phi^n)^{-1}$. It would be

computationally cumbersome both to store Φ^n as well as to compute the inverse. Fortunately, we can use recursive least squares. This is done using

$$\theta^{n+1} = \theta^n - \frac{1}{\gamma^n} B^n \phi^n (\bar{V}_s(\theta^n) - \hat{v}^{n+1}). \tag{38}$$

B^n is an $|\mathcal{F}|$ by $|\mathcal{F}|$ matrix which is equivalent to $((\Phi^n)^T \Phi^n)^{-1}$. We do not have to perform the inverse explicitly. Instead, it is well known that this can be updated recursively using

$$B^n = B^{n-1} - \frac{1}{\gamma^n}(B^{n-1}\phi^n(\phi^n)^T B^{n-1}). \tag{39}$$

γ^n is a scalar computed using

$$\gamma^n = 1 + (\phi^n)^T B^{n-1} \phi^n. \tag{40}$$

This method is well known. We refer the reader to chapter 9 of Powell (2011) for a more detailed description, and a full derivation of the updating equations.

Recursive least squares is particularly attractive because it has very low storage requirements. We only need to carry the $|\mathcal{F}|$-dimensional vector θ^n, and the $|\mathcal{F}|$ by $|\mathcal{F}|$ matrix B^n.

7.5 Least squares temporal differencing

Least squares temporal differencing (LSTD) is a different strategy for estimating the regression vector θ when using basis functions. Designed for infinite-horizon applications, LSTD was first introduced by Bradtke & Barto (1996), but our presentation is based on chapter 6 of Bertsekas (2011). Recall that $\phi(S^i)$ is a column vector of basis functions (one element for each feature $f \in \mathcal{F}$) if we are in state S^i. Assume that at iteration i, we also observe a contribution $C(S^i, a^i, W^{i+1})$ (the RL community may think of this as observations of $C(S^i, a^i, S^{i+1})$). We first compute a matrix A^n using

$$A^n = \frac{1}{n}\sum_{i=0}^{n-1} \phi(S^i)(\phi(S^i) - \gamma\phi(S^{i+1}))^T. \tag{41}$$

Next compute a column vector b^n using

$$b^n = \frac{1}{n}\sum_{i=0}^{n-1} \phi(S^i) C(S^i, a^i, W^{i+1}). \tag{42}$$

The LSTD strategy estimates the regression vector θ^n using

$$\theta^n = (A^n)^{-1} b^n. \tag{43}$$

Of course, this assumes that A^n is invertible, which is not guaranteed, but steps can be taken to overcome this.

LSTD is a method for solving the projected Bellman equations, which requires optimizing

$$\min_\theta \left(\Phi\theta - (c^\pi + \gamma P^\pi \Phi \theta^n)\right)^T D^\pi \left(\Phi\theta - (c^\pi + \gamma P^\pi \Phi \theta^n)\right) \tag{44}$$

where D^π is a $|\mathcal{S}| \times |\mathcal{S}|$ diagonal matrix with elements d_s^π which is the steady state probability of visiting (discrete) state s while following policy π, and P^π is the one-step transition matrix under policy π. Differentiating (44) with respect to θ gives the optimality condition

$$\Phi^T D^\pi \left(\Phi\theta^{n+1} - (c^\pi + \gamma P^\pi \Phi \theta^n)\right) = 0. \tag{45}$$

This can be stated as solving the equations

$$A\theta^* = b, \tag{46}$$

where $A = \Phi^T D^\pi (I - \gamma P^\pi)\Phi$ and $b = \Phi^T D^\pi c^\pi$. This allows us, in theory at least, to solve for θ^* using

$$\theta^* = A^{-1} b, \tag{47}$$

The LSTD algorithm is a simulation-based version of this equation, recognizing that we cannot actually compute A and b. The simulation-based version scales because it never actually requires that we generate any large matrices.

The use of basis functions disguises the underlying calculations. Imagine we have one basis function per state, where $\phi_f(s) = 1$ if feature f corresponds to state s. This is equivalent to a lookup table representation. In this case, θ is equivalent to a vector v with one element per state, which means that θ_f is the value of being in the state s for which $\phi_f(s) = 1$. For this problem, $A = D^\pi(I - \gamma P^\pi)$, and $b = D^\pi c^\pi$, where an element of c^π is $C(s, A^\pi(s))$. The scaling by the state probabilities in D^π makes it possible to calculate quantities via simulation, because we naturally capture the probability of being in each state through the simulation.

7.6 Least squares policy evaluation

LSTD is effectively a batch algorithm which limits its computational efficiency for applications with a large number of iterations. At each step we execute

$$\theta^n = \theta^{n-1} - \frac{\alpha}{n} G^n \sum_{i=0}^{n-1} \phi(S^i) \delta^i(n), \tag{48}$$

where G^n is a scaling matrix. G^n can be calculated in different ways, but the easiest to implement is a simulation-based estimate of $(\Phi^T D^\pi \Phi)^{-1}$ given by

$$G^n = \left(\frac{1}{n+1} \sum_{i=0}^{n} \phi(S^i) \phi(S^i)^T \right)^{-1}.$$

If we return to our model where there is a basis function $\phi_f(s)$ for each state s, where $\phi_f(s) = 1$ if feature f corresponds to state s, then $\phi(S^i)\phi(S^i)^T$ is an $|\mathcal{S}|$ by $|\mathcal{S}|$ matrix with a 1 on the diagonal for row S^i and column S^i. As n approaches infinity, the matrix

$$\left(\frac{1}{n+1} \sum_{i=0}^{n} \phi(S^i) \phi(S^i)^T \right)$$

approaches the matrix D^π of the probability of visiting each state, stored in elements along the diagonal.

8 Learning while optimizing

By now we have been given a brief tutorial on methods for approximating value functions, and we have seen some of the more popular algorithms for updating value functions while following a single policy. In this section, we review some of the algorithms that have been used to find good (ideally optimal) policies, while simultaneously trying to approximate the value of being in a state. We emphasize that throughout our discussion, we assume that our policies are based on value function approximations (our fourth class of policy).

There are two broad algorithmic strategies that have been proposed for optimizing policies based on value function approximations: approximate value iteration and approximate policy iteration, mirroring the classical value iteration and policy iteration algorithms proposed for Markov decision processes where the one-step probability transition matrix is known (and can be computed).

Within these strategies, we have to decide how we are going to approximate the value function (the "approximation architecture"), and how we are going to update the approximation.

We begin our discussion by presenting approximate value iteration, since this is the simplest and, we suspect, most widely used. However, approximate value iteration can also be extremely difficult to work with, especially when using parametric approximation architectures. We then present approximate policy iteration.

8.1 Approximate value iteration

The essence of approximate value iteration is the use of an updating mechanism based on

$$\hat{v}_t^n = \max_a \left(C(S_t^n, a) + \gamma \bar{V}_t^{n-1}(S_t^a) \right).$$

That is, we depend on a value function approximation for the downstream state. Note that this is closely related to Q-learning (see equations (7)-(8)). If states and actions are discrete and we use a lookup table representation, Q-learning has been proven to converge to the optimal Q-factors, which gives us an optimal policy by choosing $a = \arg\max Q(s, a)$ for any given state s (see Tsitsiklis (1994)). The algorithm requires that we use a policy that samples states and actions infinitely often. In essence, this is an exploration policy to learn the Q-factors, which then yields an optimal policy.

Figure 3 depicts a basic implementation of an approximate value iteration algorithm which assumes that we are using a lookup table representation for the value of being in a state. The algorithm is presented for a finite horizon problem so that the indexing of variables over time and iterations is clear. For an infinite horizon implementation, you can just drop the time index t everywhere.

A problem with the algorithm in figure 3 is that it is unlikely to work in practice. The problem is that we use the optimal action a_t^n in Step 2a to determine the next state that we visit in step 2c. Consider the two-stage dynamic program in figure 4, where we have currently estimated the value of being in each of the two states as $\bar{v}_1 = \bar{v}_2 = 0$. Now imagine that we are in state 1, and we are trying to decide whether to stay in state 1 or move to state 2. Using the current value function approximation, staying in state 1 earns 0, while moving to 2 would earn $-1 + \bar{v}_2 = -1$, so it appears that we should stay in state 1. Doing what *appears* to be best is known as *exploitation*, since we are using (exploiting) what we know. But if we did move to state 2, we would see that we would earn \$10 by moving from state 2 back to state 1, allowing us to update $\bar{v}_2$. The only way we are

Step 0. Initialization:

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

Step 0b. Set $n = 1$.

Step 0c. Initialize $S_0^{a,1}$.

Step 1. Choose a sample path ω^n.

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

Step 2a: Determine the action using ϵ-greedy. With probability ϵ, choose an action a^n at random from $\mathcal{A}$. With probability $1 - \epsilon$, choose a^n using

$$\hat{v}_t^n = \max_{a_t \in \mathcal{A}_t} \left(C(S_t^n, a_t) + \gamma \bar{V}_t^{n-1}(S^{M,a}(S_t^n, a_t)) \right).$$

Let a_t^n be the action that solves the maximization problem.

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

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

Step 2c. Sample $W_{t+1}^n = W_{t+1}(\omega^n)$ and compute the next state $S_{t+1}^n = S^M(S_t^n, a_t^n, W_{t+1}^n)$.

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

going to learn that this is the case is by making the decision while in state 1 to make the decision to *explore* state 2. Striking a balance between exploring and exploiting is one of the major challenges of approximate dynamic programming.

To address the exploration vs. exploitation problem, we need to identify two policies. The first is the policy we are trying to learn, given by

$$a_t^n = \arg\max_{a_t \in \mathcal{A}_t} \left(C(S_t^n, a_t) + \gamma \bar{V}_t^{n-1}(S^{M,a}(S_t^n, a_t)) \right). \tag{49}$$

The *learning policy* (known as the *target policy* in the reinforcement learning community) is the policy we are trying to optimize. It is determined by the value function approximation, so the premise is that if we can do a better job of approximating the value of being in a state, we should obtain a better policy.

The second policy is called the *sampling policy* (known as the *behavior policy* in the reinforcement learning community), which determines which states are sampled next. Most of the time, after computing a_t^n using equation (49), we would then choose a different action $a^{SP} = A^{SP}(S_t)$ which is our sampling policy, and we would then determine the next state using $S_{t+1} = S^M(S_t, a^{SP}, W_{t+1}(\omega^n))$.

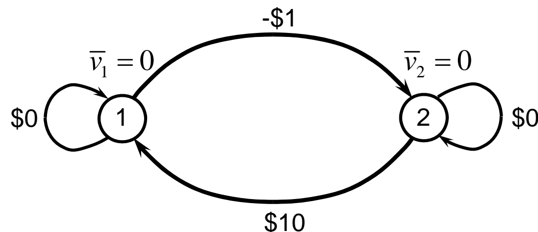

Figure 4: Two-state dynamic program with contributions and state value function estimates.

If the sampling policy is the same as the learning policy (that is, if we use a_t^n from (49) to determine the next state to visit), then we are using what is known as *on-policy learning*. If the sampling policy is different than the learning policy, then we are using *off-policy learning*. The distinction is important. If we are using lookup table representations, the only way to obtain an algorithm that will converge to the optimal policy is one where we use a sampling policy that ensures that every action (and therefore every reachable state) is sampled infinitely often.

The situation changes completely if we are using a linear architecture (basis functions) to approximate the value of being in a state. As shown by Tsitsiklis & Van Roy (1997), temporal-difference learning using basis functions and a fixed policy is known to converge to the best possible value function approximation in terms of a mean-square metric over the states, weighted by the probability of being in each state. At the same time, the same authors showed that off-policy learning may diverge if using a weighted Bellman error minimization as the objective.

If we were to modify our approximate value iteration in figure 3 to use basis functions, the result would be an algorithm with no convergence guarantees at all. The problem is that evaluation errors in $\hat{v}^n$ result in changes to the regression vector θ^n that can completely change the policy. Even if we use on-policy learning, the resulting interaction between approximating the value of a policy and using the approximate value function to determine the policy creates a fundamentally unstable system without the presence of structural properties such as convexity.

Recently, Sutton et al. (2009*b*) and Sutton et al. (2009*a*) propose an off-policy learning algorithm (that is, for a fixed policy) based on the principle of minimizing the *projected* Bellman error. Maei et al. (2010) extend this general idea to an off-policy algorithm that learns an optimal policy. This work overcomes the well known divergence of prior off-policy algorithms using parametric function approximations, but the work is quite new and time is needed to understand the limitations of the assumptions required in the proof and the empirical performance on actual problems. Just

the same, these papers represent a potentially significant breakthrough for general, discrete action problems which lack properties such as convexity. Off-policy learning makes it possible to introduce exploration policies that give the algorithm designer more control over the states that are visited. We note, however, that these ideas do not extend to high-dimensional action spaces, where off-policy learning is much harder to implement.

The situation changes again if we make the transition to nonparametric value function approximations. Here, an observation of a state-value pair $(S^n, \hat{v}^n)$ does not affect the entire value function approximation as occurs with a parametric model. Now, updates are more localized, which makes the behavior more similar to lookup table representations. The literature on learning issues for nonparametric approximation architectures remains quite young. For a review of some of this literature, see Powell & Ma (2011).

8.2 Approximate policy iteration

Perhaps the biggest challenge of approximate value iteration when using functional approximations is that a single observation can change the entire value function approximation, thereby changing the policy which then distorts future observations. One way to overcome this limitation is through approximate policy iteration which is illustrated in figure 5 using a lookup table representation. The major difference between this and approximate value iteration is the introduction of an inner loop (step 2) which iteratively simulates over the planning horizon to develop a good approximation of the value function for a fixed policy.

Approximate policy iteration is generally more stable than approximate value iteration, but the price is the need to execute the inner loop repeatedly. Even with this step, we generally do not have guarantees that we will converge to an optimal policy without satisfying some strong assumptions. However, we can get stronger convergence results with approximate policy iteration than we can get with approximate value iteration.

9 Conclusions

This paper provides a broad overview of approximate dynamic programming. In the operations research community, ADP has been equated with the use of value function approximations which has separated it from the stochastic programming community (a form of lookahead policy) or simulation

Step 0. Initialization:

Step 0a. Initialize $\bar{V}^{\pi,0}$.

Step 0b. Set a look-ahead parameter T and inner iteration counter M.

Step 0c. Set $n = 1$.

Step 1. Sample a state S_0^n and then do:

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

Step 3. Choose a sample path ω^m (a sample realization over the lookahead horizon T).

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

Step 4a. Compute

$$a_t^{n,m} = \arg\max_{a_t \in \mathcal{A}_t^{n,m}} \left(C(S_t^{n,m}, a_t) + \gamma \bar{V}^{\pi,n-1}(S^{M,a}(S_t^{n,m}, a_t)) \right).$$

Step 4b. Compute

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

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

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

Step 6a: Accumulate $\hat{v}^{n,m}$:

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

Step 6b: Update the approximate value of the policy:

$$\bar{v}^{n,m} = (\frac{m-1}{m})\bar{v}^{n,m-1} + \frac{1}{m}\hat{v}_0^{n,m}.$$

Step 8. Update the value function at S^n:

$$\bar{V}^{\pi,n} = (1 - \alpha_{n-1})\bar{v}^{n-1} + \alpha_{n-1}\hat{v}_0^{n,M}.$$

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

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

Figure 5: A policy iteration algorithm for infinite horizon problems

optimization (which typically involves policy search). By contrast, the reinforcement learning literature has embraced value function approximations, "policy gradient algorithms" (policy search), and a variety of methods under names such as roll-out heuristics and Monte Carlo tree search (lookahead policies).

Dynamic programming is widely associated, both in computer science and operations research, with small action spaces. We review the concept of the post-decision state variable, and illustrate how this can be used, with a properly designed value function approximation, to handle sequential optimization problem where the decision vector x_t has thousands (even tens of thousands) of dimensions.

Our presentation closes with a discussion of algorithmic issues, including the exploration-exploitation problem, that arise when using value function approximations. We divide the steps involved us-

ing value function approximations between choosing an architecture (lookup table, parametric or nonparametric), estimating the value of being in a state and then updating the value function approximation. We close with a brief discussion of the two major algorithmic strategies, approximate value iteration and approximate policy iteration, and the convergence issues that arise with these algorithms.

References

Barto, A. G., Sutton, R. S. & Brouwer, P. (1981), 'Associative search network: A reinforcement learning associative memory', *Biological cybernetics* **40**(3), 201—-211.

Barto, A., Sutton, R. S. & Anderson, C. (1983), 'Neuronlike adaptive elements that can solve difficult learning control problems', *IEEE Transactions on systems, man, and cybernetics* **13**(5), 834–846.

Bellman, R. (1971), *Introduction to the Mathematical Theory of Control Processes, Vol. II*, Academic Press, New York.

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

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

Bellman, R. E. & Dreyfus, S. E. (1959*b*), 'Functional approximations and dynamic programming', *Mathematical Tables and Other Aids to Computation* **13**, 247–251.

Bertsekas, D. P. (2011), *Approximate Dynamic Programming*, Vol. II, 3 edn, Athena Scientific, Belmont, MA, chapter 6.

Bertsekas, D. P. & Castanon, D. A. (1999), 'Rollout Algorithms for Stochastic Scheduling Problems', *J. Heuristics* **5**, 89–108.

Bertsekas, D. P. & Tsitsiklis, J. N. (1996), *Neuro-dynamic programming*, Athena Scientific, Belmont, MA.

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

Bradtke, S. J. & Barto, A. G. (1996), 'Linear least-squares algorithms for temporal difference learning', *Machine Learning* **22**(1), 33–57.

Cheung, R. K.-M. & Powell, W. B. (1996), 'An Algorithm for Multistage Dynamic Networks with Random Arc Capacities, with an Application to Dynamic Fleet Management', *Operations Research* **44**, 951–963.

Dantzig, G. (1955), 'Linear Programming Under Uncertainty', *Management Science* **1**, 197–206.

Dantzig, G. & Ferguson, A. (1956), 'The Allocation of Aircrafts to Routes: An Example of Linear Programming Under Uncertain Demand', *Management Science* **3**, 45–73.

Denardo, E. V. (1982), *Dynamic Programming*, Prentice-Hall, Englewood Cliffs, NJ.

Derman, C. (1962), 'On sequential decisions and Markov chains', *Management Science* **9**(1), 16–24.

Derman, C. (1966), 'Denumerable state Markovian decision processes-average cost criterion', *Statistics, The Annals of Mathematical* **37**(6), 1545—-1553.

Derman, C. (1970), *Finite State Markovian Decision Processes*, Academic Press, New York.

Dreyfus, S. & Law, A. M. (1977), *The Art and Theory of Dynamic Programming*, Academic Press, New York.

Dupaçová, J., Consigli, G. & Wallace, S. W. (2000), 'Scenarios for multistage stochastic programs', *Annals of Operations Research* **100**, 25–53.

Dynkin, E. B. & Yushkevich, A. A. (1979), 'Controlled Markov processes', *in volume Grundlehren der mathematischen Wissenschaften 235 of A Series of Comprehensive Studies in Mathematics. New York: SpringerVerlag.*

Enders, J., Powell, W. B. & Egan, D. M. (2010), 'Robust policies for the transformer acquisition andallocation problem', *Energy Systems* **1**(3), 245–272.

George, A., Powell, W. B. & Kulkarni, S. (2008), 'Value Function Approximation using Multiple Aggregation for Multiattribute Resource Management', *J. Machine Learning Research* **9**, 2079–2111.

Growe-Kuska, N., Heitsch, H. & Romisch, W. (2003), 'Scenario reduction and scenario tree construction for power management problems', *IEEE Bologna Power Tech Proceedings (Borghetti, A., Nucci, CA, Paolone, M. eds.)*.

Hastie, T., Tibshirani, R. & Friedman, J. (2009), *The elements of statistical learning: data mining, inference and prediction*, Springer, New York.

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

Heyman, D. P. & Sobel, M. (1984), *Stochastic Models in Operations Research, Volume II: Stochastic Optimization*, McGraw Hill, New York.

Higle, J. & Sen, S. (1996), *Stochastic Decomposition: A Statistical Method for Large Scale Stochastic Linear Programming*, Kluwer Academic Publishers.

Howard, R. A. (1960), *Dynamic programming and Markov process*, MIT Press, Cambridge, MA.

Jaakkola, T., Jordan, M. I. & Singh, S. (1994), 'On the convergence of stochastic iterative dynamic programming algorithms', *Neural Computation* **6**(6), 1185–1201.

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

Kall, P. & S.W.Wallace (1994), *Stochastic programming*, John Wiley & Sons.

Kaut, M. & Wallace, S. W. (2003), 'Evaluation of scenario-generation methods for stochastic programming', *Stochastic Programming E-Print Series*.

Kushner, H. J. & Yin, G. G. (2003), *Stochastic Approximation and Recursive Algorithms and Applications*, Springer.

Maei, H. R., Szepesvari, C., Bhatnagar, S. & Sutton, R. S. (2010), Toward Off-Policy Learning Control with Function Approximation, *in* 'ICML-2010'.

Nemhauser, G. L. (1966), *Introduction to dynamic programming*, John Wiley & Sons, New York.

Powell, W. B. (1987), 'An operational planning model for the dynamic vehicle allocation problem with uncertain demands', *Transportation Research 21B* pp. 217–23253.

Powell, W. B. (2007), *Approximate Dynamic Programming: Solving the curses of dimensionality*, John Wiley & Sons, Hoboken, NJ.

Powell, W. B. (2010), 'Merging AI and OR to Solve High-Dimensional Stochastic Optimization Problems Using Approximate Dynamic Programming', *INFORMS Journal on Computing* **22**(1), 2–17.

Powell, W. B. (2011), *Approximate Dynamic Programming: Solving the curses of dimensionality*, 2nd. edn, John Wiley & Sons, Hoboken, NJ.

Powell, W. B. & Frantzeskakis, L. F. (1990), 'A Successive Linear Approximation Procedure for Stochastic Dynamic Vehicle Allocation Problems', *Transportation Science* **24**, 40–57.

Powell, W. B. & Godfrey, G. (2002), 'An adaptive dynamic programming algorithm for dynamic fleet management, I: Single period travel times', *Transportation Science* **36**(1), 21–39.

Powell, W. B. & Ma, J. (2011), 'A review of stochastic algorithms with continuous value function approximation and some new approximate policy iteration algorithms for multidimensional continuous applications', *J. Control Theory and Applications* **9**(3), 336–352.

Powell, W. B. & Simão, H. (2009), 'Approximate dynamic programming for management of high-value spare parts', *Journal of Manufacturing Technology Management* **20**(2), 147–160.

Powell, W. B. & Simao, H. P. (2009), 'Approximate Dynamic Programming for Management of High Value Spare Parts', *J. of Manufacturing Technology Management* **20**(2), 147—-160.

Powell, W. B. & Topaloglu, H. (2005), Fleet Management, *in* S. Wallace & W. Ziemba, eds, 'Applications of Stochastic Programming', Math Programming Society - SIAM Series in Optimization, Philadelphia, pp. 185–216.

Powell, W. B. & Van Roy, B. (2004), Approximate Dynamic Programming for High Dimensional Resource Allocation Problems, *in* J. Si, A. G. Barto, W. B. Powell & D. W. II, eds, 'Handbook of Learning and Approximate Dynamic Programming', IEEE Press, New York.

Powell, W. B., George, A., Lamont, A. & Stewart, J. (2011), 'SMART: A Stochastic Multiscale Model for the Analysis of Energy Resources, Technology and Policy', *Informs J. on Computing*.

Puterman, M. L. (1994), *Markov Decision Processes*, 1st edn, Johnn Wiley and Sons, Hoboken.

Puterman, M. L. (2005), *Markov Decision Processes*, 2nd edn, John Wiley and Sons, Hoboken, NJ.

Robbins, H. & Monro, S. (1951), 'A stochastic approximation method', *The Annals of Mathematical Statistics* **22**(3), 400–407.

Romisch, W. & Heitsch, H. (2009), 'Scenario tree modeling for multistage stochastic programs', *Mathematical Programming* **118**, 371–406.

Ross, S. (1983), 'Introduction to Stochastic Dynamic Programming', *Academic Press, New York*.

Samuel, A. L. (1959), 'Some studies in machine learning using the game of checkers', *IBM Journal of Research and Development* **3**, 211—-229.

Sen, S. & Higle, J. (1999), 'An introductory tutorial on stochastic linear programming models', *Interfaces* **terfaces29**(2), 33–6152.

Si, J., Barto, A. G., Powell, W. B. & Wunsch, D. (2004), 'Handbook of learning and approximate dynamic programming', *Wiley-IEEE Press*.

Silver, D. & Tesauro, G. (2009), 'Monte-Carlo simulation balancing', *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09* pp. 1–8.

Simao, H. P., Day, J., George, A. P., Gifford, T., Powell, W. B. & Nienow, J. (2009), 'An Approximate Dynamic Programming Algorithm for Large-Scale Fleet Management: A Case Application', *Transportation Science* **43**(2), 178–197.

Simao, H. P., George, A., Powell, W. B., Gifford, T., Nienow, J. & Day, J. (2010), 'Approximate Dynamic Programming Captures Fleet Operations for Schneider National', *Interfaces* **40**(5), 1–11.

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

Sutton, R. S. (1988), 'Learning to predict by the methods of temporal differences', *Machine Learning* **3**(1), 9–44.

Sutton, R. S. & Barto, A. G. (1981), 'Toward a modern theory of adaptive networks', *Psychological Review* **88**(2), 135–170.

Sutton, R. S. & Barto, A. G. (1998), *Reinforcement Learning*, Vol. 35, MIT Press, Cambridge, MA.

Sutton, R. S., Maei, H. R., Precup, D., Bhatnagar, S., Silver, D., Szepesvari, C. & Wiewiora, E. (2009*a*), Fast gradient-descent methods for temporal-difference learning with linear function approximation, *in* 'Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09', ACM Press, New York, New York, USA, pp. 1–8.

Sutton, R. S., Szepesvári, C. & Maei, H. (2009*b*), A convergent O (n) algorithm for off-policy temporal-difference learning with linear function approximation, *in* 'Advances in Neural Information Processing Systems', Vol. 21, Citeseer, pp. 1609–1616.

Topaloglu, H. & Powell, W. B. (2006), 'Dynamic Programming Approximations for Stochastic, Time-Staged Integer Multicommodity Flow Problems', *Informs Journal on Computing* **18**, 31–42.

Tsitsiklis, J. N. (1994), 'Asynchronous stochastic approximation and Q-learning', *Machine Learning* **16**, 185–202.

Tsitsiklis, J. N. & Van Roy, B. (1997), 'An Analysis of Temporal-Difference Learning with Function Approximation', *IEEE Transactions on Automatic Control* **42**, 674–690.

Van Roy, B., Bertsekas, D. P., Lee, Y. & 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.

Watkins, C. (1989), 'Learning from delayed rewards', *PhD thesis, Kings College, Cambridge, England.*

Watkins, C. & Dayan, P. (1992), 'Q-learning', *Machine Learning* **8**(3-4), 279–292.

Werbos, P. J. (1974), Beyond regression: new tools for prediction and analysis in the behavioral sciences, PhD thesis, Harvard University.

Werbos, P. J. (1989), 'Backpropagation and neurocontrol: A review and prospectus', *Neural Networks* pp. 209—-216.

Werbos, P. J. (1990), 'Consistency of HDP applied to a simple reinforcement learning problem', *Neural Networks* **3**, 179–189.

Werbos, P. J. (1992*a*), Approximate Dynamic Programming for Real-Time Control and Neural Modelling, *in* D. J. White & D. A. Sofge, eds, 'Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches'.

Werbos, P. J. (1992*b*), Neurocontrol and Supervised Learning: an Overview and Valuation, *in* D. A. White & D. A. Sofge, eds, 'Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches'.

Werbos, P. J., Miller, W. T. & Sutton, R. S., eds (1990), *Neural Networks for Control*, MIT Press, Cambridge, MA.

White, D. J. (1969), *Dynamic Programming*, Holden-Day, San Francisco.

Wu, T., Powell, W. B. & Whisman, A. (2009), 'The Optimizing-Simulator: An Illustration using the Military Airlift Problem', *ACM Transactions on Modeling and Simulation* **19**(3), 1–31.