Intelligent Optimal Control

M. Athans, D. Bertsekas, W. McDermott, J. Tsitsiklis, B. Van Roy

Laboratory for Information and Decision Systems
Massachusetts Institute of Technology
Cambridge, MA 02139

September 1995

1 Introduction

In this chapter we consider nonlinear systems where decisions are made in stages, as is the usual case in feedback control. The outcome of each decision is not fully predictable but can be anticipated to some extent before the next decision is made. Each decision results in some immediate cost but also affects the context in which future decisions are to be made and therefore affects the cost incurred in future stages. We are interested in decision making policies that minimize the total cost over a number of stages. Such problems are challenging primarily because of the trade–off between immediate and future costs.

Dynamic programming (DP for short) provides a mathematical formalization for resolving this trade–off [4]. Numerical methods associated with DP compute the expected future cost to be incurred as a function of the current state of the system, given that optimal decisions will be made at every future stage. By providing a basis for efficient assessment of alternative decisions, this *optimal cost function* facilitates the selection of optimal decisions. It is well known, however, that for many important problems the computational requirements involved with obtaining the optimal cost function are overwhelming, mainly because of a very large number of states and controls (Bellman's "curse of dimensionality"). In such situations, a suboptimal solution is required. This chapter focuses on methods that generate suboptimal control strategies through approximating the optimal cost function.

Approximations of the optimal cost function have been used in the past in a variety contexts. Chess playing programs represent a successful example. A key idea in these programs is to use a *position evaluator* to rank different chess positions and to select at each turn a move that results in the position with the best rank. The position evaluator assigns a numerical value to each position, according to a heuristic formula that includes weights for the various features of the position (material balance, piece mobility, king safety, and other factors). Usually, the general structure of the position evaluator is selected first (this is largely an art that has evolved over many years, based on experimentation and human knowledge about chess), and the numerical weights are chosen by trial and error or (as in the case of the champion program Deep Thought) by "training" using a large number of sample grandmaster games.

As the chess program paradigm suggests, intuition about the problem, heuristics, and trial and error are all important ingredients for constructing cost approximations. However, it is important to supplement heuristics and intuition with more systematic techniques that are broadly applicable and retain as much as possible the nonheuristic aspects of DP. This chapter will describe several recent efforts to develop a methodological foundation for combining dynamic programming, compact representations, and simulation to provide the basis for a rational approach to complex stochastic decision problems.

1.1 Approximating the Optimal Cost

In this chapter, we focus on methods for the approximation of the optimal cost function J^*, possibly through the use of neural networks and/or simulation. Suboptimal control strategies can be generated using these approximations in the same way that optimal control strategies are generated using the true optimal cost function. In particular, we replace the optimal cost $J^*(j)$ with a suitable approximation $\tilde{J}(j,r)$, where r is a vector of parameters, and we use at state i the (suboptimal) control $\tilde{\mu}(i)$ that attains the minimum in the (approximate) right-hand side of Bellman's equation (see Section 2)

$$\tilde{\mu}(i) = \arg\min_u E\{g(i,u,j) + \tilde{J}(j,r) \mid i,u\}, \tag{1}$$

where u is the control, j is a random next state, and $g(i,u,j)$ is the cost of a transition from state i to state j under control u. The function $\tilde{J}$ will be called the *scoring function*, or *cost-to-go function* and the value $\tilde{J}(j,r)$ will be called the *score* of state j. Once the form of $\tilde{J}$ is selected and the parameter vector r is determined, the evaluation of $\tilde{J}(j,r)$ of any state j is presumed to be a simple computation.

We note that in some problems the minimization over u of the expression

$$E\{g(i,u,j) + \tilde{J}(j,r) \mid i,u\} \tag{2}$$

may be too complicated or too time-consuming for making decisions in real-time, even if the scores $\tilde{J}(j,r)$ are simply calculated. In such problems we may use a related technique, whereby we approximate the expression minimized in Bellman's equation,

$$Q(i,u) = E\{g(i,u,j) + J^*(j) \mid i,u\}, \tag{3}$$

which is known as the *Q-factor corresponding to* (i,u). In particular, we replace $Q(i,u)$ with a suitable approximation $\tilde{Q}(i,u,r)$, where r is a vector of parameters. We then use at state i the (suboptimal) control that minimizes the approximate Q-factor corresponding to i:

$$\tilde{\mu}(i) = \arg\min_u \tilde{Q}(i,u,r). \tag{4}$$

Although our discussion focuses on approximation of the optimal cost function J^*, the same considerations also apply to the approximation of Q-factors.

We are interested in problems with a large number of states, but we want scoring functions $\tilde{J}$ that can be described with relatively few numbers (a vector r of small dimension). Scoring functions involving few parameters will be called *compact representations*. In contrast, a tabular description of J^*, in which the values $J^*(j)$ are stored in a table with one entry dedicated to each state j, is called a *lookup table representation*. In a typical compact representation, only the vector r and the general structure of the scoring function $\tilde{J}(\cdot, r)$ are stored; the scores $\tilde{J}(j, r)$ are generated only when needed. For example, $\tilde{J}(j, r)$ may be the output of a neural network in response to the input state j, and r is the associated vector of weights or parameters of the neural network. Alternatively, $\tilde{J}(j, r)$ may involve a lower dimensional description of the state j in terms of its "significant features", with r the associated vector of relative weights of the features. Thus, generating a useful scoring function $\tilde{J}(j, r)$ requires two complementary steps:

1. deciding on the general structure of the function $\tilde{J}(j, r)$, and

2. calculating the parameter vector r so as to minimize in some sense the error between the optimal cost function $J^*(\cdot)$ and the approximation $\tilde{J}(\cdot, r)$.

In the chess example discussed earlier, the position evaluator corresponds to the scoring function $\tilde{J}(j, r)$ above, while determining the weights of the features correspond to calculating the parameter vector r.

1.2 Types of Approximation

An important issue in function approximation is the *selection of architecture*, that is, the choice of a parametric class of functions $\tilde{J}(\cdot, r)$ or $\tilde{Q}(\cdot, \cdot, r)$ that suits the problem at hand. One interesting type of cost approximation is provided by *feature extraction*, a process that maps the state into some vector, called the *feature vector* associated with the state. Feature vectors summarize, in a heuristic sense, what is considered to be important characteristics of the state, and they are very useful in incorporating the designer's prior knowledge or intuition about the problem and about the structure of the optimal controller. For example in a queueing system involving several queues, a feature vector may involve for each queue a three-value indicator, that specifies whether the queue is "nearly empty", "moderately busy", or "nearly full". Note that in many cases, analysis can be used to suggest the right features for the problem at hand.

Another interesting type of approximation, which appears to be well suited for many practical problems, is provided by neural network architectures of various types. We should point out that we use the term "neural network" in a very broad sense, essentially as a synonym to "approximation architecture." In particular, we do not restrict ourselves to the classical multilayer perceptron structure with sigmoidal nonlinearities. Any type of universal approximator of nonlinear mappings could be used in our context. The nature of the approximating structure is left open in our discussion, and it could involve, for example, radial basis functions, wavelets, polynomials, splines, feature extraction mappings, etc.

Finally, one may consider approximation architectures where both features and neural networks are used together. In particular, the state may be mapped to a feature vector, which is then used as input to a neural network that produces the score of the state (see Figure 1).

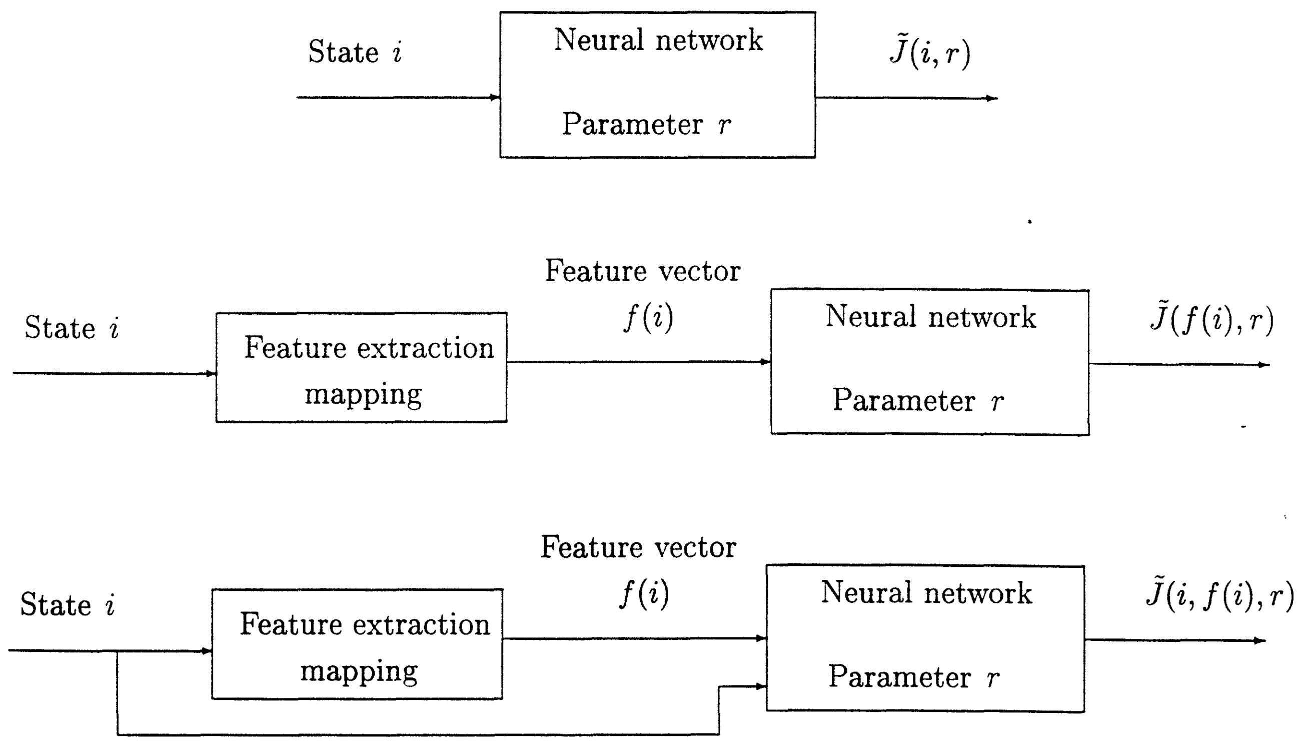

Figure 1: Several approximation architectures involving feature extraction and neural networks

1.3 Building an Approximation

Neural networks have been applied in a variety of contexts, but their most successful applications are in the areas of pattern recognition, nonlinear regression, and nonlinear system identification. In these applications the neural network is used as a universal approximator: the input-output mapping of the neural network is matched to an unknown nonlinear mapping F of interest using least-squares optimization. This optimization is known as *training the network*. To perform training, one must have some training data, that is, a set of pairs $(i, F(i))$, which is representative of the mapping F that is approximated.

It is important to note that in contrast with these neural network applications, in the DP context there is no readily available training set of input-output pairs $(i, J^*(i))$, which can be used to approximate J^* with a least squares fit. More sophisticated algorithms, such as those discussed later in this chapter, are required to generate approximations to J^*.

One particularly interesting possibility is to evaluate (exactly or approximately) by simulation the cost functions of given (suboptimal) policies, and to try to iteratively improve these policies based on the simulation outcomes. A rigorous analysis of such an approach involves complexities that do not arise in classical neural network training contexts. Indeed the use of simulation to evaluate approximately the optimal cost function is a key new idea, that distinguishes the methodology of this chapter from earlier methods for approximate DP.

Using simulation offers another major advantage: it allows the methods of this chapter to be used for systems that are hard to model but easy to simulate; that is, in problems where an explicit model is not available, and the system can only be observed, either as it operates in real time or through a software simulator. For such problems, the traditional DP techniques are inapplicable, and estimation of the transition probabilities to construct a detailed mathematical model is often cumbersome or impossible.

There is a third potential advantage of simulation: it can implicitly identify the "most important" or "most representative" states of the system. It appears plausible that if these states are the ones most often visited during the simulation, the scoring function will tend to approximate better the optimal cost for these states, and the suboptimal policy obtained will perform better.

In view of the reliance on both DP and neural network concepts, we use the name *neuro-dynamic programming* (NDP for short) to describe collectively the methods of this chapter. In the artificial intelligence community, where the methods originated, the name *reinforcement learning* is also used. In common artificial intelligence terms, the methods of this chapter allow systems to "learn how to make good decisions by observing their own behavior, and use built-in mechanisms for improving their actions through a reinforcement mechanism." In the less anthropomorphic DP terms used in this chapter, "observing their own behavior" relates to simulation, and "improving their actions through a reinforcement mechanism" relates to iterative schemes for improving the quality of approximation of the optimal cost function, or the Q-factors, or the optimal policy. There has been a gradual realization that reinforcement learning techniques can be fruitfully motivated and interpreted in terms of classical DP concepts such as value and policy iteration. Werbos points out the relationship between early reinforcement learning methods and DP in [28], spawning a cross-fertilization of ideas. A recommended survey is [1], which explores the connections between the artificial intelligence/reinforcement learning viewpoint and the control theory/DP viewpoint, and gives many references.

We finally mention a conceptually different approximation possibility that aims at direct optimization over policies of a given type. Here we hypothesize a stationary policy of a certain structural form, say $\tilde{\mu}(i,r)$, where r is a vector of unknown parameters/weights that is subject to optimization. We also assume that for a fixed r, the cost of starting at i and using the stationary policy $\tilde{\mu}(\cdot,r)$, call it $\tilde{J}(i,r)$, can be evaluated by simulation. We may then minimize over r

$$E\{\tilde{J}(i,r)\}, \tag{5}$$

where the expectation is taken with respect to some probability distribution over the set of initial states i. Generally, the minimization of the cost function (5) tends to be quite difficult, particularly if the dimension of the parameter vector r is large (say over 10). As a result, this approach is typically effective only when adequate optimal policy approximation is possible with very few parameters. However, deterministic control problems yield a simplification; we can directly calculate optimal time-control sample pairs by solving the (computationally expensive) problem for a number of initial conditions. A near-optimal policy can then be obtained by using these samples and least-squares fitting. This idea will be discussed in Section 5.

1.4 Overview

In this chapter, we attempt to clarify some aspects of the current NDP methodology. In particular, we overview available results (including several recent ones) within a common framework, suggest some new algorithmic approaches, and identify some open questions. A detailed exposition can be found in [8]. Despite the great interest in NDP, there is little solid theory at present to guide the user, and the corresponding literature is often confusing. Yet, there have been many reports of successes with problems too large and complex to be treated in any other way. A particularly impressive success is the development of a backgammon playing program as reported in [22]. Here a neural network was trained to approximate the optimal cost function of the game of backgammon by using simulation; that is, by letting the program play against itself. After training for several months, the program nearly defeated the human world champion.

Our own limited experience has confirmed that NDP methods can be impressively effective in problems where traditional DP methods would be hardly applicable and other heuristic methods would have a limited chance of success. We note, however, that the practical application of NDP is computationally very intensive, and often requires a considerable amount of trial and error. Furthermore, success is often obtained using methods whose properties are not well understood. Fortunately, all the computation and experimentation with different approaches can be done off-line. Once the approximation is obtained, it can be used to generate decisions fast enough for use in real time. In this context, we mention that in the artificial intelligence literature, reinforcement learning is often viewed as an "on-line" method, whereby the cost approximation is improved as the system operates in real time. This is reminiscent of the methods of traditional adaptive control. We will not discuss this viewpoint in this chapter, as we prefer to focus on applications involving a large and complex system. A lot of training data is required for such a system. These data typically cannot be obtained in sufficient volume as the system is operating; even if they can, the corresponding processing requirements are typically too large for effective use in real time.

The chapter is organized as follows. Section 2 introduces dynamic programming as a basis for the approximation algorithms that will be discussed later. Section 3 addresses the use of simulation to build lookup table representations of costs. The $TD(\lambda)$ and Q–learning algorithms are introduced and the issue of algorithm convergence is discussed. Section 4 describes algorithms, issues, and results on using compact representations to find suboptimal control policies for problems where a lookup table is infeasible. Finally, Section 5 describes the direct approximation of optimal control policies when the generation of a value function can be bypassed.

2 Dynamic Programming

Out of several possible DP models, we focus our initial attention on infinite horizon finite-state controlled Markov chains with undiscounted costs [5]. (Discounted problems can be converted to undiscounted problems by viewing the discount factor as a probability that the process terminates at any given time step.)

We are given a controlled discrete-time dynamic system whereby, given the state i_k at time k, the use of a control u_k specifies the transition probability $p_{i_k i_{k+1}}(u_k)$ to the next state i_{k+1}. Here for all k, the state i_k is measured directly and is an element of a finite state space, and the control u_k is constrained to take values in a given finite constraint set $U(i_k)$, which may depend on the current state i_k [$u_k \in U(i_k)$, for all i_k]. There is a cost $g(i_k, u_k, i_{k+1})$ associated with the use of control u_k at state i_k and a transition to state i_{k+1}.

We are interested in admissible policies, that is, sequences $\pi = \{\mu_0, \mu_1, \ldots\}$ where each μ_k is a function mapping states into controls with $\mu_k(i) \in U(i)$ for all states i. The total expected cost associated with an initial state i_0 and a policy $\pi = \{\mu_0, \mu_1, \ldots\}$ is

$$J^\pi(i_0) = \lim_{N\to\infty} E\left\{\sum_{k=0}^{N-1} g(i_k, \mu_k(i_k), i_{k+1})\right\}.$$

(If there is doubt regarding the existence of the limit, one can use "lim inf" above.) A stationary policy is an admissible policy of the form $\pi = \{\mu, \mu, \ldots\}$, and its corresponding cost function is denoted by J^μ. Finally, the optimal expected cost associated with an initial state i is the minimum $J^\pi(i)$ over all policies π, and is denoted by $J^*(i)$.

In the absence of discounting and in order to obtain a meaningful problem, we assume that *there is a special cost-free and absorbing destination state, denoted as state* 0. Once the system reaches that state it remains there at no further cost. We are interested in problems where either reaching the destination is inevitable or else there is an incentive to reach the destination in a finite expected number of stages. Thus, the essence of the problem is how to reach the destination with minimum expected cost. We call this problem the *stochastic shortest path problem*. The deterministic shortest path problem is obtained as the special case where for each state-control pair (i, u), the transition probability $p_{ij}(u)$ is equal to 1 for a unique state j that depends on u.

The analysis of the stochastic shortest path problem requires certain assumptions that guarantee that the problem is well-posed. We first need to define the notion of a *proper policy*, that is, a stationary policy that leads to the destination with probability one, regardless of the initial state.

Let $1, \ldots, n$ denote the states other than the termination state 0.

Definition 2.1 A stationary policy $\pi = \{\mu, \mu, \ldots\}$ is said to be *proper* if there exists an integer m such that, using this policy, there is positive probability that the destination will be reached after m stages, regardless of the initial state x_0, that is, if

$$\rho_\mu = \max_{i=1,\ldots,n} P\{x_m \neq 0 \mid x_0 = i, \pi\} < 1. \tag{6}$$

A stationary policy that is not proper is said to be *improper*.

We assume the following:

Assumption 2.1 (a) There exists at least one proper policy.
(b) For every improper policy $\{\mu, \mu, \ldots\}$, the corresponding cost $J^\mu(i)$ is ∞ for at least one state i.

Note that in many problems of interest, all policies are proper, in which case the above assumptions are trivially satisfied. In other problems where the cost $g(i,u,j)$ is nonnegative, it may be easily verifiable that for every improper policy, there is a state with positive expected cost that is visited infinitely often with probability 1, in which case Assumption 2.1(b) holds.

Because there is only a finite number of states, functions J with components $J(i)$, where i is one of the states $1, \ldots, n$, will also be viewed as vectors in $\Re^n$. The basic results for stochastic shortest path problems can be expressed in terms of the DP mapping T that transforms vectors $J \in \Re^n$ into vectors $TJ \in \Re^n$ and is defined by

$$TJ(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + J(j)), \qquad i = 1, \ldots, n, \tag{7}$$

as well as the mappings T_μ defined for each stationary policy μ by

$$T_\mu J(i) = \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i,\mu(i),j) + J(j)), \qquad i = 1, \ldots, n. \tag{8}$$

In summary, these results are (see [6], [7], [5]):

(a) The optimal cost vector J^* is the unique solution of Bellman's equation $J = TJ$.

(b) A stationary policy $\{\mu, \mu, \ldots\}$ is optimal if and only if for every state i, $\mu(i)$ attains the minimum in the definition of $TJ^*(i)$, that is, $T_\mu J^* = TJ^*$.

Furthermore, it is possible to show the validity of the following two main computational methods for stochastic shortest path problems. These are:

1. **Value Iteration:** Given any vector J, we generate successively $TJ, T^2J, \ldots$. It can be shown that the generated sequence $\{T^kJ\}$ converges to the optimal cost function J^*. It can also be proved that the Gauss-Seidel version of the value iteration method works, and in fact the same is true for parallel asynchronous versions of value iteration [3], [6].

2. **Policy Iteration:** Starting with a proper policy μ^0, we generate a sequence of new policies $\mu^1, \mu^2, \ldots$. In the kth iteration, given the policy μ^{k-1}, we perform a *policy evaluation step*, that computes $J^{\mu^{k-1}}$ by solving for J the equation $J = T_{\mu^{k-1}} J$, which can also be written as the system of n linear equations

$$J(i) = \sum_{j=1}^{n} p_{ij}(\mu(i))(g(i, \mu(i), j) + J(j)), \qquad i = 1, \ldots, n, \tag{9}$$

in the n unknowns $J(1), \ldots, J(n)$. We then perform a *policy improvement step*, which computes a new policy μ^k using the equation $T_{\mu^k} J^{\mu^{k-1}} = T J^{\mu^{k-1}}$, or equivalently

$$\mu^k(i) = \arg \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)(g(i, u, j) + J^{\mu^{k-1}}(j)), \qquad i = 1, \ldots, n. \tag{10}$$

Given that the initial policy μ^0 is proper as stated above, the policy iteration algorithm terminates after a finite number of iterations with an optimal proper policy.

Both of these methods are useful in practice; both can form the basis of computational methods that use neural network approximations in cases where we lack an explicit system model.

3 Simulation Methods for a Lookup Table Representation

Classical computational methods for dynamic programming apply when there is an explicit model of the cost structure and the transition probabilities of the system to be controlled. In many problems, however, such a model is not available, but instead, the system can be simulated. It is then of course possible to use repeated simulation to calculate (at least approximately) the transition probabilities of the system and the expected costs per stage by averaging, and then to apply the methods discussed earlier.

The methodology discussed in this section, however, is geared towards an alternative possibility which is much more attractive when one contemplates approximations: the transition probabilities are not explicitly estimated, but instead the cost–to–go function of a given policy is progressively calculated by generating several sample system trajectories and associated costs. There are a number of possible techniques within this context, which may be viewed as suboptimal control methods. We discuss here several possibilities, which will be revisited in Section 4 in conjunction with approximation methods. While the methods of this section involve a *lookup table representation* of the cost–to–go function, and are practical only when the number of states is moderate, they are still of more general interest for several reasons:

(a) Unlike the traditional computational methods, these methods are applicable even when there is lack of an exact model.

(b) The use of simulation to guide exploration of the state space can significantly increase the efficiency of dynamic programming.

(c) Some approximate methods that employ compact representations can be viewed as natural extensions of the simulation–based methods considered in this section. For this reason, the algorithms studied in this section provide a baseline against which the results of Section 4, where compact representations are employed, are to be compared.

(d) Finally, some of the approximate methods studied in Section 4 can be viewed as lookup table methods (of the type considered here) for a suitable auxiliary (and usually much smaller) problem. Thus, the understanding gained here can be transferred to ideas in Section 4.

3.1 Policy Evaluation by Monte-Carlo Simulation

Suppose that we have fixed a *proper* stationary policy μ and that we wish to calculate by simulation the corresponding cost vector J^{μ}. One possibility is to generate, starting from each i, many sample state trajectories, and average the corresponding costs to obtain an approximation to $J^{\mu}(i)$. While this can be done separately for each state i, a possible alternative is to use each trajectory to obtain cost samples for each state visited by the trajectory. In other words, given a trajectory, the portion starting at any particular visited state is considered to be an additional simulated trajectory. Thus, a single simulated trajectory contributes to the cost–to–go estimates of many states, rather than only one.

One implementation of this method can be described as follows. We generate a random trajectory $(i_0, i_1, \ldots, i_N)$, where i_N is the terminal state and define the *temporal differences* d_k by letting

$$d_k = g(i_k, i_{k+1}) + J(i_{k+1}) - J(i_k). \tag{11}$$

We then update $J(i_k)$ for each state i_k encountered by the trajectory by letting

$$J(i_k) := J(i_k) + \gamma(d_k + d_{k+1} + \cdots + d_{N-1}), \tag{12}$$

where γ is a stepsize parameter. Note that the ℓth temporal difference d_ℓ becomes known as soon as the transition from i_ℓ to $i_{\ell+1}$ is simulated. This raises the possibility of carrying out the update (12) incrementally, that is, by setting

$$J(i_k) := J(i_k) + \gamma d_\ell, \qquad k = 0, 1, \ldots, \ell \tag{13}$$

as soon as d_ℓ becomes available.

The method based on the update rule (13) is known as TD(1). A generalization proposed by Sutton [20] and known as TD(λ), is obtained by introducing a scalar parameter $\lambda \in [0, 1]$ and modifying equation (13) to

$$J(i_k) := J(i_k) + \gamma\lambda^{\ell-k} d_\ell, \qquad k = 0, 1, \ldots, \ell, \tag{14}$$

The TD(λ) algorithm can be viewed as a Robbins-Monro stochastic approximation algorithm for solving a certain system of equations, related to the policy evaluation equation

$J^\mu = T_\mu J^\mu$. It turns out that this system of equations involves a contraction mapping with respect to a suitable weighted maximum norm [6], and for this reason, under some natural assumptions, the method converges with probability 1 to J^μ, for all values of λ in the closed interval $[0, 1]$ [14]. However, there is currently little theoretical understanding of how λ should be chosen to make the most efficient use of simulations.

Any algorithm, like TD(λ), for evaluating the vector J^μ can be embedded within the policy iteration algorithm. In a typical iteration of the algorithm, one fixes a proper policy μ, performs enough simulations and iterations of the TD(λ) algorithm to evaluate J^μ with sufficient accuracy, and then obtains a new policy $\overline{\mu}$ according to the policy iteration update rule $T_{\overline{\mu}} J^\mu = T J^\mu$. In an alternative method, which we call *optimistic policy iteration*, every TD(λ) update is followed by a policy update. That is, given the current vector J we obtain the policy μ determined by $T_\mu J = TJ$ and use this policy to simulate the next trajectory (or the next transition, if policies are updated after every transition). While this method is often used in practice, its properties were far from understood until very recently. We have shown, however, that the following is true [8]:

(a) A somewhat restricted ("synchronous") version of the method converges with probability 1 to J^*.

(b) The method does not converge in general and even if it converges, the limit can be different than J^*.

Note that performing a policy iteration update given J^μ requires knowledge of the transition probabilities and cost structure of the underlying system. Thus, although the policy evaluation via TD(λ) *can* be performed in the absence of an explicit model, the method we have described for deriving an optimal policy can not. Nevertheless, the computational method is of practical interest since it naturally generalizes to the context of compact representations, which we will discuss in Section 4. In that context, this approach to computing a cost–to–go function potentially reduces requirements on computation time, although the necessity of an explicit model is not alleviated.

3.2 Q–Learning

We now discuss a computational method that can be used to derive an optimal policy when there is no explicit model of the system. This method is analogous to value iteration, but updates values called Q-factors, rather than cost–to–go values.

We define the optimal Q-factor $Q^*(i, u)$ by $Q^*(i, u) = 0$ if $i = 0$ and

$$Q^*(i,u) = \sum_{j=0}^{n} p_{ij}(u)(g(i,u,j) + J^*(j)), \qquad i = 1, \dots, n, \tag{15}$$

otherwise. Note that Bellman's equation can be written as

$$J^*(i) = \min_{u \in U(i)} Q^*(i,u). \tag{16}$$

Using equation (16) to eliminate $J^*(j)$ from equation (15), we obtain

$$Q^*(i,u) = \sum_{j=0}^{n} p_{ij}(u) \left(g(i,u,j) + \min_{v \in U(j)} Q^*(j,v) \right). \tag{17}$$

The optimal Q-factors $Q^*(i,u)$ are the unique solution of the system equation (17), as long as we only consider Q–factors that obey the natural condition $Q(0,u) = 0$, which is something that will be assumed throughout.

In terms of the Q-factors, the value iteration algorithm can be written as

$$Q(i,u) := \sum_{j=0}^{n} p_{ij}(u) \left(g(i,u,j) + \min_{v \in U(j)} Q(j,v) \right), \qquad \text{for all } (i,u). \tag{18}$$

A more general version of this iteration is

$$Q(i,u) := (1-\gamma)Q(i,u) + \gamma \sum_{j=0}^{n} p_{ij}(u) \left(g(i,u,j) + \min_{v \in U(j)} Q(j,v) \right), \tag{19}$$

where γ is a stepsize parameter with $\gamma \in (0,1]$, that may change from one iteration to the next. The Q-learning method is an approximate version of this iteration, whereby the expected value is replaced by a single sample, i.e.,

$$Q(i,u) := (1-\gamma)Q(i,u) + \gamma \left(g(i,u,j) + \min_{v \in U(j)} Q(j,v) \right). \tag{20}$$

Here j and $g(i,u,j)$ are generated from the pair (i,u) by simulation, that is, according to the transition probabilities $p_{ij}(u)$. Thus, Q-learning can be viewed as a combination of value iteration and simulation. Equivalently, Q–learning is the Robbins–Monro method based on equation (17). The Q-learning algorithm was proposed by Watkins [26], and its convergence was established in [27] for the case of discounted problems. In [23] the connection with stochastic approximation was established and a convergence proof was obtained in greater generality, including stochastic shortest path problems. (Related results are discussed in [14].) In fact, for the case where some of the policies are allowed to be improper, convergence has only been proved under the additional assumption that the algorithm stays bounded, but this is a condition that is easily enforced using the common device of projecting back into a bounded set whenever the Q–factor iterate becomes too large [9, 16].

4 Approximate DP for a Compact Representation

In this section, we discuss several possibilities for approximate or suboptimal control. Generally we are interested in approximation of the cost function J^μ of a given policy μ, the optimal cost function J^*, or the optimal Q-factors $Q^*(i,u)$. This is done using a function, which given a state i, produces an approximation $\tilde{J}^\mu(i,r)$ of $J^\mu(i)$, or an approximation

$\tilde{J}(i,r)$ of $J^*(i)$, or, given also a control u, an approximation $\tilde{Q}(i,u,r)$ of $Q^*(i,u)$. The approximating function involves a parameter/weight vector r, and may be implemented using a neural network, a feature extraction mapping, or any other suitable architecture. The parameter/weight vector r is determined by optimization using some type of least squares framework.

Most of the discussion in this section is geared towards the case where the set $U(i)$ of possible decisions at each state i has small or moderate cardinality and therefore the required minimizations do not present any intrinsic difficulties. We will occasionally discuss, however, some issues that arise if the set $U(i)$ is very large or infinite, even though the choices available tend to be problem specific.

There are two main choices that have to be made in approximate dynamic programming: the choice of approximation architecture and the choice of an algorithm used to update the parameter vector r. Sometimes these issues are coupled because some algorithms are guaranteed to work properly only for certain types of architectures.

4.1 Bellman Error Methods

One possibility for approximation of the optimal cost by a function $\tilde{J}(i,r)$, where r is a vector of unknown parameters/weights, is based on minimizing the error in Bellman's equation; for example, by solving the problem

$$\min_r \sum_{i \in \tilde{S}} \left| \tilde{J}(i,r) - \min_{u \in U(i)} \sum_j p_{ij}(u)\big(g(i,u,j) + \tilde{J}(j,r)\big) \right|^2, \tag{21}$$

where $\tilde{S}$ is a suitably chosen subset of "representative" states. This minimization may be attempted by using some type of gradient or Gauss-Newton method. We observe that if $\tilde{S}$ is the entire state space and if the cost in the problem (21) can be brought down to zero, then $\tilde{J}(i,r)$ solves Bellman's equation and we have $\tilde{J}(i,r) = J^*(i)$ for all i. Regarding the set $\tilde{S}$ of representative states, it may be selected by means of regular or random sampling of the state space, or by using simulation to help us focus on the more significant parts of the state space. For the special case where we are dealing with a single policy, the gradient of the objective function (21) can be generated by simulation, and a stochastic gradient algorithm and stochastic approximation theory can be used to study convergence issues.

Suppose now that a model of the system is unavailable or too complex to be useful. We then need to work in terms of approximate Q-factors. Let us introduce an approximation $\tilde{Q}(i,u,r)$ to the optimal Q-factor $Q^*(i,u)$, where r is an unknown parameter vector. Bellman's equation for the Q-factors is given by (recall equation (19))

$$Q^*(i,u) = \sum_j p_{ij}(u)\Big(g(i,u,j) + \min_{u' \in U(j)} Q^*(j,u')\Big). \tag{22}$$

In analogy with problem (21), we determine the parameter vector r by solving the least squares problem

$$\min_r \sum_{(i,u) \in \tilde{V}} \left| \tilde{Q}(i,u,r) - \sum_j p_{ij}(u)\Big(g(i,u,j) + \min_{u' \in U(i)} \tilde{Q}(j,u',r)\Big) \right|^2, \tag{23}$$

where $\tilde{V}$ is a suitably chosen subset of "representative" state-control pairs. The incremental gradient method for this problem is given by

$$\begin{aligned} r &:= r - \gamma E\{d^u(i,j,r) \mid i,u\} E\{\nabla d^u(i,j,r) \mid i,u\} \\ &= r - \gamma E\{d^u(i,j,r) \mid i,u\}\Big(\sum_j p_{ij}(u) \nabla_r \tilde{Q}(j,\overline{u},r) - \nabla_r \tilde{Q}(i,u,r)\Big), \end{aligned} \tag{24}$$

where $d^u(i,j,r)$ is given by

$$d^u(i,j,r) = g(i,u,j) + \min_{u' \in U(i)} \tilde{Q}(j,u',r) - \tilde{Q}(i,u,r), \tag{25}$$

$\overline{u}$ is obtained by

$$\overline{u} = \arg \min_{u' \in U(j)} \tilde{Q}(j,u',r), \tag{26}$$

and γ is a stepsize parameter. A simulation-based version of this iteration is given by

$$r := r - \gamma d^u(i,j,r)(\nabla_r \tilde{Q}(\overline{j},\overline{u},r) - \nabla_r \tilde{Q}(i,u,r)), \tag{27}$$

where j and $\overline{j}$ are two states independently generated from i according to the transition probabilities corresponding to u, and

$$\overline{u} = \arg \min_{u' \in U(j)} \tilde{Q}(\overline{j},u',r). \tag{28}$$

The convergence of the resulting stochastic gradient algorithm can be studied using stochastic approximation theory.

4.2 Approximate Policy Iteration

In approximate policy iteration, one fixes a stationary policy $\pi = \{\mu, \mu, \mu, \ldots\}$ and evaluates an approximation $\tilde{J}^\mu(\cdot, r)$ of the function J^μ. This is followed by a policy update step whereby one obtains a new stationary policy $\overline{\pi} = \{\overline{\mu}, \overline{\mu}, \overline{\mu}, \ldots\}$ by choosing $\overline{\mu}$ to satisfy $T_{\overline{\mu}} \tilde{J}^\mu = T\tilde{J}^\mu$.

The above described method has been shown recently [8] to be consistent in the following sense: if the approximation architecture is rich enough and the training information is sufficiently rich so that $\tilde{J}^\mu$ is within ϵ of the true function J^μ for all policies encountered in the course of the algorithm, then the algorithm converges to a neighborhood of J^* whose radius is $O(\epsilon)$.

The most important element of the approximate policy iteration algorithm is the way that the approximate policy evaluation is carried out. This can be accomplished using one of several methods.

1. We can use the Bellman error method described in the preceding section, specialized to the case where we are dealing with a single policy.

2. We can collect sample data pairs $\{(i_1, c_1), (i_2, c_2), \ldots, (i_K, c_K)\}$, each composed of a representative state i_k and the total cost of a simulated trajectory starting at state i_k, under the current policy $\pi = \{\mu, \mu, \mu, \ldots\}$. Then an approximate cost function can be acquired by solving the least squares problem
$$\min_r \sum_{k=1}^{K} (\tilde{J}^\mu(i_k, r) - c_k)^2. \tag{29}$$

3. We can generalize TD(λ) to the present context, involving compact representations of the cost-to-go function. The following generalization has been introduced by Sutton [20]: we sample a trajectory $i_0, \ldots, i_K, 0$, define the temporal differences $d_k = g(i_k, \mu(i_k), i_{k+1}) + \tilde{J}_\mu(i_{k+1}) - \tilde{J}(i_k)$, and update the parameter vector r according to
$$r := r + \gamma d_k \sum_{m=0}^{k} \lambda^{k-m} \nabla_r \tilde{J}^\mu(i_m, r). \tag{30}$$
This procedure might be repeated several times, using new trajectories, to improve the accuracy of $\tilde{J}^\mu$ prior to a policy update. For the case of a lookup table representation, it is easy to check that this method reduces to the TD(λ) method of Section 3.

The third option, which employs the TD(λ) algorithm, has received wide attention. However, its convergence behavior is unclear, unless $\lambda = 1$. Basically, for a compact representation, TD(λ) can be viewed as a form of incremental gradient method where there are some error terms in the gradient direction. These error terms depend on r as well as λ, and do not necessarily diminish when r is equal to the value where TD(1) converges, unless $\lambda = 1$ or a lookup table representation is used. Thus, in general, the limit obtained by TD(λ) depends on λ.

To enhance our understanding, let us consider in more detail the algorithm TD(0) obtained by setting $\lambda = 0$. In the on line version of the algorithm, a transition from i to j leads to an update of the form
$$r := r + \gamma d(i, j, r) \nabla_r \tilde{J}^\mu(i, r), \tag{31}$$
where $d(i, j, r)$ is the temporal difference $g(i, \mu(i), j) + \tilde{J}^\mu(j, r) - \tilde{J}^\mu(i, r)$.

For the case of a linear approximation architecture, TD(λ) always converges [10]. However, there are very few guarantees on the quality of the limit as an approximant of J^μ. We have shown [4] that TD(λ) not only gives in the limit a vector r that depends on λ, but also that the quality of $\tilde{J}^\mu(i, r(\lambda))$ as an approximation to $J^\mu(i)$ may get worse as λ becomes smaller than 1. In particular, the approximation provided by TD(0) can be very poor. Thus, there are legitimate concerns regarding the suitability of TD(0) for obtaining good approximations of the cost-to-go function.

We finally note that, similarly with the case of lookup table representations, there is an optimistic version of approximate policy iteration whereby the policy is updated at each step. While this method has been very successful in some applications, theoretical understanding is far from complete. Nevertheless, there is some preliminary theoretical evidence suggesting that optimistic policy iteration, if it converges, may lead to policies that perform better than those obtained from the non-optimistic variant [8].

4.3 Approximate Value Iteration

The value iteration algorithm is of the form

$$J(i) := \min_{u \in U(i)} \sum_j p_{ij}(u)(g(i,u,j) + J(j)), \tag{32}$$

or, in more abstract notation, $J(i) := (TJ)(i)$, where T is the dynamic programming operator. In this section, we discuss a number of different ways that the value iteration algorithm can be adapted to a setting involving compact representations of the optimal cost function. Our discussion revolves primarily around the case where a model of the system is available, so that the transition probabilities $p_{ij}(u)$ and the one–step costs $g(i,u,j)$ are known. In the absence of such a model, one must work with the Q–factors.

The algorithm is initialized with a parameter vector r_0 and a corresponding cost–to–go function $\tilde{J}(i, r_0)$. At a typical iteration of the algorithm, we have a parameter vector r_k, we select a set S_k of representative states, and we compute estimates of the cost–to–go from the states in S_k by letting

$$\hat{J}_{k+1}(i) = \min_{u \in U(i)} \sum_j p_{ij}(u) \left(g(i,u,j) + \tilde{J}_k(j, r_k)\right), \qquad i \in S_k. \tag{33}$$

We then determine a new set of parameters r_{k+1} by minimizing with respect to r the quadratic cost criterion

$$\sum_{i \in S_k} v(i) |\hat{J}_{k+1}(i) - \tilde{J}_{k+1}(i,r)|^2, \tag{34}$$

where $v(i)$ are some predefined positive weights. In the special case where the approximation architecture $\tilde{J}$ is linear, we are dealing with a linear least squares problem, which can be solved efficiently.

This method has consistency properties that are similar to those of policy iteration: if the approximation error $\hat{J}_k - \tilde{J}_k$ at each iteration is small, the algorithm converges to a neighborhood of J^*. Unfortunately, a simple example from [24] shows that the algorithm suffers, in general, from potential divergence. In particular, consider a three–state system with the structure shown in Figure 2.

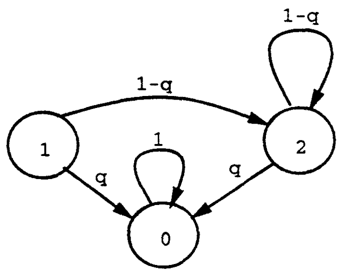

Figure 2: A Markov decision problem in which there is a single policy. The labels next to each arc are the transition probabilities.

State 0 is an absorbing state and all transitions are cost free. Obviously, we have $J^*(i) = 0$ for all i. Consider a compact representation in terms of a single scalar parameter r, of the form $\tilde{J}(1,r) = r$, $\tilde{J}(2,r) = 2r$, and $\tilde{J}(0,r) = 0$. Given a value r_k of r, we obtain $\hat{J}(1) = \hat{J}(2) = 2qr_k$. We form the least squares problem

$$r_{k+1} = \min_{r} \, (r - 2qr_k)^2 + (2r - 2qr_k)^2, \tag{35}$$

and by setting the derivative to zero, we obtain $r_{k+1} = 6qr_k/5$. Hence, if $q > 5/6$, the algorithm diverges.

We note that the compact representation used in our example was rich enough so as to allow us to represent exactly the optimal cost–to–go function J^*. As the example demonstrates, this property is not enough to guarantee the soundness of the algorithm. Apparently, a stronger condition is needed: the parametric representation must be able to closely represent all of the intermediate value functions obtained in the course of the value iteration algorithm. Given that such a condition is in general very difficult to verify, one must either accept the risk of a divergent algorithm or else restrict to particular types of compact representations under which divergent behavior is inherently impossible.

There is an incremental version of approximate value iteration in which the vector r is updated after considering a single state i. For example, if state i is chosen, we carry out the update

$$r := r + \gamma \nabla_r \tilde{J}(i,r) \left(\min_{u \in U(i)} \sum_j p_{ij}(u) \left(g(i,u,j) + \tilde{J}(j,r) \right) - \tilde{J}(i,r) \right). \tag{36}$$

The sum in the preceding equation can be replaced by a single sample estimate, leading to the update equation

$$r := r + \gamma \nabla_r \tilde{J}(i,r)(g(i,u,j) + \tilde{J}(j,r) - \tilde{J}(i,r)), \tag{37}$$

where j is sampled according to the probabilities $p_{ij}(u)$. We note that this has the same form as the update equation used in the TD(0) algorithm. The only difference is that in TD(0), it is assumed that one generates an entire state trajectory and performs updates along all states on the trajectory, whereas the sequence of states used for the updates in incremental value iteration can be arbitrary.

The update equation (36) bears similarities with that of the the Bellman error method. However, there are notable differences. In particular, unlike the Bellman error method, this algorithm does not admit a clean interpretation in terms of minimization of some quadratic cost criterion. Unfortunately, the incremental approximate value iteration algorithm suffers from the same drawbacks as the original approximate value iteration algorithm. There is no clear guarantee of convergence and reasonable accuracy, given a compact representation that can closely approximate the optimal cost–to–go function. In the next few subsections, we consider approximate dynamic programming methods that do deliver such promises.

4.4 Convergent methods

We review here a few variants of approximate value iteration for which some desirable convergence properties are guaranteed.

4.4.1 State Aggregation

Consider a partition of the set $\{0, 1, \ldots, n\}$ of states into disjoint subsets $S_0, S_1, \ldots, S_K$, where we assume that $S_0 = \{0\}$. We consider a K–dimensional parameter vector r whose kth component is meant to approximate the value function for all states $i \in S_k$, $k \neq 0$. In other words, we are dealing with the compact representation

$$\tilde{J}(i, r) = r_k, \qquad \text{if } i \in S_k, \tag{38}$$

together with our usual convention $\tilde{J}(0, r) = 0$. This is feature–based aggregation whereby we assign a common value r_k to all states i that share a common feature vector. Note that such a representation is able to closely approximate J^* as long as J^* does not vary too much within each subset.

We start by noting that $\partial \tilde{J}(i, r)/\partial r_k$ is equal to 1 if $i \in S_k$ and is equal to zero otherwise. Thus, equation (36) becomes

$$r_k := (1 - \gamma) r_k + \gamma \left(\min_{u \in U(i)} \sum_j p_{ij}(u) \Big(g(i, u, j) + \tilde{J}(j, r) \Big) \right), \qquad \text{if } i \in S_k. \tag{39}$$

We only discuss the results that are available for the case of discounted problems. Under minor technical conditions on the stepsizes and the mechanism for choosing at which state to update, the algorithm converges with probability 1 to the solution of a system of nonlinear equations. This system turns out to be the Bellman equation for a modified stochastic shortest path problem. What is more important, if the approximation architecture is capable of closely approximating J^*, the algorithm is guaranteed to converge in the close vicinity of J^*; see [24] for precise results; related results have also been obtained in [11].

4.4.2 Compact Representations Based on Representative States

We now discuss another context in which approximate value iteration converges nicely. Suppose that we have selected K states ($K < n$) considered to be sufficiently representative of the entire state space and assume that the K representative states are states $1, \ldots, K$. We use a K-dimensional parameter vector r, identified with the values of $J(1), \ldots, J(K)$ and let

$$\tilde{J}(i, r) = r_i, \qquad i = 1, \ldots, K, \tag{40}$$

$$\tilde{J}(i, r) = \sum_{k=1}^{K} \theta_k(i) r_k, \tag{41}$$

where $\theta_k(i)$ are some coefficients that are chosen ahead of time. For example, $\theta_k(i)$ could be some measure of similarity of state i to state k.

The natural way of carrying out approximate value iteration in this context is to let

$$r_i = \tilde{J}(i,r) := \min_{u \in U(i)} \sum_{j=0}^{n} p_{ij}(u)\Big(g(i,u,j) + \sum_{k=1}^{K} \theta_k(j)\tilde{J}(k,r)\Big), \qquad i = 1,\ldots,K. \tag{42}$$

It turns out that, under the assumption $\sum_{k=1}^{K} |\theta_k(i)| \leq 1$, and for discounted problems, the algorithm is guaranteed to converge and the limit is a close approximation of J^*, as long as the approximation architecture is capable of closely approximating J^* [24]. The limit obtained by the algorithm can be interpreted as the optimal cost-to-go function for a related auxiliary problem.

4.4.3 Euclidean norm contractions

It is well known that the dynamic programming operator T often has certain contraction properties with respect to the maximum norm or with respect to a suitably weighted maximum norm. It turns out [25] that convergence of approximate value iteration can be guaranteed if one introduces some stronger contraction assumptions. In particular, convergence is obtained if we are dealing with a linear approximation architecture and if T is a contraction with respect to the Euclidean norm on the space of functions $\tilde{J}(\cdot,r)$ that can be represented by the chosen approximation architecture. The search for interesting examples where this stronger contraction assumption is satisfied is a current research topic.

4.5 Applications

There are several successful applications of neuro-dynamic programming that have been reported in the literature, the best known one being Tesauro's backgammon player. Our own experience with applications is that while these methods may require a lot of ingenuity in choosing a good combination of approximation architecture and algorithm, they have great promise of delivering better performance than currently available. Some representative applications we have been investigating involve games (tic-tac-toe [21], Tetris [24]), scheduling in queueing systems, admission control in data networks, and channel allocation in cellular communications.

5 Learning Control Policies Directly

In previous sections, we have discussed algorithms that approximate the cost–to–go associated with states and state/action pairs. A control policy is then implemented by choosing at each step the state/action pair with the smallest approximated cost–to–go.

The above approach to optimal control is better suited to the case where the control set is finite and manageably small. In addition, the controller implementation requires evaluating the cost–to–go for every potential decision at each time step on-line. In many settings, this may be too time consuming and unsuitable for real-time implementation. One alternative is to select off-line a number of representative states i, and use the approximate cost-to-go function $\tilde{J}(i,r)$ to determine good decisions $\tilde{\mu}(i)$ at the representative states. We can then use an approximation architecture, e.g., a neural network, to generalize the policy $\tilde{\mu}$ from the representative states to the entire state space. Once such a generalization is available, it can be employed on-line for the purpose of generating decisions in response to any current state.

The key element in the above outlined approach is the generalization of the policy that has been specified at representative states by the cost-to-go function $\tilde{J}$. This generalization idea can also be applied when the cost-to-go approximations are replaced by other sampled decision information. One approach that has been investigated in the artificial intelligence literature is to use an expert's opinion to generate presumably good decisions at sample states. A last approach, which is the subject of this section, considers cases where optimal decisions can actually be calculated at representative states. This is possible for the special case of deterministic control problems, as discussed below.

5.1 Optimal Control Problem Formulation

In keeping with the traditions of optimal control theory, we consider here a continuous-time formulation. We assume that we are given a continuous time, time invariant plant of he form

$$\dot{x}(t) = f(x(t), u(t)) \tag{43}$$

where $x(t) \in \Re^n$ is the state vector and $u(t) \in \Re^m$ is the control vector. The goal is to guide any initial state $x(0) = x_0$ to the origin while minimizing:

$$J = \int_0^T L(x(t), u(t))\, dt \tag{44}$$

where T is the (free terminal) time at which the state reaches the origin, i.e., $x(T) \equiv 0$ and $u(t)$ is constrained to the set of all bounded measurable piecewise–continuous functions such that

$$u(t) \in \Omega \subset \Re^m$$

where Ω is a given constraint set. There are additional technical assumptions that need to be made in order to obtain well-posed problems [2, Ch.5] but these need not concern us here.

Such optimal control problems can be solved with the help of Pontryagin's maximum principle [2, Ch.5], as follows. We define the *Hamiltonian*

$$H(x,p,u) = L(x,u) + < p, f(x,u) > \tag{45}$$

as a scalar function of $x(t), p(t), u(t)$; $< \cdot, \cdot >$ denotes inner product. Given an initial condition $x(0) = x_0$ let $u^*(t)$ be an optimal control and let $x^*(t)$ denote the corresponding optimal state trajectory. Then there exists a costate vector, $p^*(t) \in \Re^n$, such that

$$\dot{x}^* = \frac{\partial H}{\partial p} \qquad \dot{p}^* = -\frac{\partial H}{\partial x}$$

and

$$H(x^*(t), p^*(t), u^*(t)) \leq H(x^*(t), p^*(t), u(t)) \quad \forall u(t) \in \Omega.$$

Making use of the boundary conditions $x(0) = x_0$, $x(T) = 0$, an optimal trajectory can be found by solving a system of $2n$ differential equations with $2n$ boundary conditions. In the general case where these equations are not readily solved in closed form, this *Two Point Boundary Value* (TPBV) problem can be solved *numerically* using various iterative methods [15] for any given initial condition[1].

The iteration yields a table of *optimal* (x, u) pairs along optimal trajectories in state space. Supervised neural network training techniques can now be used to produce a neural network that directly maps plant states to controls, in contrast to reinforcement learning algorithms that must accumulate information through rewards/penalties and repeated experimentation. The network interpolates the open–loop training information, and acts in a "real time" context where the TPBV solvers could not. Thus, this technique bypasses the evaluation of cost–to–go estimates that occurs in approximate dynamic programming techniques.

In order to make a connection with approximate dynamic programming, we note that $p^*(t)$ is known to be equal to the gradient $\nabla J^*(x^*(t))$ [19]. With the methods in previous sections, we would first approximate J^* and then its gradient in order to estimate the costate, which is then used in the Hamiltonian minimization. In contrast, the method of this section computes the costate directly.

5.2 Algorithm and Architecture

Our experimental work has focused on multilayer neural networks, e.g. with a single hidden layer followed by an output layer, trained by incremental backpropagation [12, 13]. Recall that the network input is the plant state x and the output is the control u. The hidden layer typically consists of a set of sigmoidal units; the output layer might consist of a single sigmoidal unit representing a control constraint Ω limiting the single control to a range $[-1, +1]$. A sigmoidal unit computes:

$$\sigma(z) = -1 + \frac{2}{1 + e^{-z}} \tag{46}$$

where z is the standard scalar input to the node formed by a weighted linear combination of the previous layer. For a detailed description of multilayer neural networks see [12].

The use of sampled trajectories to learn a state feedback control mapping requires addressing several training issues. In particular, the number and distribution of the trajectory

[1]We may treat these iterative methods as a "black box" piece of the overall learning process.

initial conditions and the sampling frequency along trajectories each affect the quantity and distribution of the training data and hence have significant impact on the time required for training and the quality of the trained feedback mapping [17, 18]. Below, we briefly mention another training issue that can occur when learning feedback mappings directly: the choice of output architecture.

5.2.1 Piecewise–Constant Optimal Control

The class of *piecewise-constant* (e.g. bang-bang) optimal control problems involve plant, cost functional, and control restrictions that result in controls switching "hard" between a discrete set of values as the plant state moves along a trajectory toward the origin. The set of optimal control values for these problems may be evident from the form of the Hamiltonian (5.1). For instance, consider a double integrator example of a mass that must be moved to the origin and zero velocity while being penalized by both travel time and expended fuel:

$$\begin{aligned} J &= \int_0^T (1 + |u(t)|)\mathrm{dt} \\ \dot{x}_1(t) &= x_2(t) \\ \dot{x}_2(t) &= u(t) \\ |u(t)| &\leq 1 \end{aligned}$$

where T is the time at which the state reaches the origin. Then, it is an easy consequence of the Maximum Principle that the optimal control need only take values in the set $\{-1, 0, -1\}$.

The discrete set of control values found in piecewise–constant optimal control problems indicates that the neural network is actually learning a *classification* task, forming a static map between each state and a control "class". Thus, we may construct a network output layer with one unit corresponding to each optimal control value. A network architecture corresponding to the example would have outputs y_1^s, y_2^s, and y_3^s corresponding to $u = -1$, $u = 0$, and $u = +1$. Given the weighted input vector to the output layer:

$$z_2 = W_2\sigma(W_1x + b_1) + b_2 \tag{47}$$

the outputs are defined as:

$$y_i^s = \frac{e^{z_{2i}}}{\sum_{r=1}^3 e^{z_{2r}}} \tag{48}$$

The outputs range from zero to one, and sum to one by definition. When used to control a plant, the units can be implemented in a "winner-take-all" fashion:

1. The current state is input to the network,
2. the network outputs are calculated,
3. the control corresponding to the unit with the largest value is applied to the plant.

In the example, if for the current state x, $y_2^s > y_1^s$ and $y_2^s > y_3^s$, then the control $u = 0$ would be applied to the plant.

By selecting a neural network architecture that incorporates such *a priori* information about the solution to the optimal control problem, experimental work has shown that we may obtain control mappings that perform better than mappings obtained using architectures based only on knowledge of the control set Ω [17].

To summarize, in this section we have considered continuous time optimal control problems that can have continuous state and control spaces. Optimal control theory led to the generation of data sets suitable for use with supervised neural network training techniques. This permitted us to generate feedback controllers with the desirable property of *directly* mapping states to controls while avoiding explicit evaluation of the cost–to–go function.

While avoiding the need to generate a value function as done in earlier sections, learning state to control policies directly retains some of the training issues associated with the other approaches, such as a computational burden. Furthermore, the quantity and distribution of the sampled trajectory data impacts the training time and quality of the neural network mapping. Additionally, we note that the output architecture of the neural network, not normally an issue in approximate dynamic programming, can have a significant effect on the training results when learning policies directly.

Acknowledgements

This research was supported by the NSF under grant ECS-9216531 and by the EPRI under contract RP8030-10.

References

[1] A. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence,* Special Volume: Computational Research on Interaction and Agencey, Vol. 72, pp. 81-138.

[2] M. Athans and P. L. Falb. *Optimal Control: An Introduction to the Theory and its Applications.* McGraw-Hill Book Company, 1966.

[3] D. Bertsekas. Distributed dynamic programming. *IEEE Transactions on Automatic Control*, Vol. AC–27, pp. 610–16, 1982.

[4] D. Bertsekas. A counter-example to temporal differences learning. *Neural Computation*, Vol. 7, pp. 270-279, 1995.

[5] D. Bertsekas. *Dynamic Programming and Optimal Control.* Athena Scientific, Belmont, MA, 1995.

[6] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[7] D. Bertsekas and J. Tsitsiklis. An analysis of stochastic shortest path problems. *Mathematics of Operations Research*, Vol. 16, pp. 580-595, 1991.

[8] D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996. (to appear)

[9] E. Chong and P. Ramadge. Convergence of recursive optimization algorithms using infinitesimal perturbation analysis estimates. *Discrete Event Dynamic Systems: Theory and Applications*, Vol. 1, pp. 339-372, 1992.

[10] P. Dayan. The convergence of TD(λ) for general λ. *Machine Learning*, Vol. 8, pp. 341-362, 1992.

[11] G. Gordon. Stable function approximation in dynamic programming. Technical Report: CMU-CS-95-103, Carnegie Mellon University, 1995.

[12] J. Hertz, A. Krogh, and R. Palmer. *Introduction to the Theory of Neural Computation.* Addison-Wesley Publishing, Redwood City, CA, 1991.

[13] K. Hunt, D. Sbarbaro, R. Żbikowski, and P. Gawthrop. Neural networks for control systems – a survey. *Automatica*, Vol. 28, No. 6, pp. 1083–1112, 1992.

[14] T. Jaakola, M. Jordan, and S. Singh. On the convergence of stochastic iterative dynamic programming methods. *Neural Computation*, Vol. 6, pp. 1185-1201, 1994.

[15] D. Kirk. *Optimal Control Theory.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1970.

[16] H. Kushner and D. Clark. *Stochastic Approximation Methods for Constrained and Unconstrained Problems.* Springer Verlag, New York, 1978.

[17] W. McDermott. On the use of neural networks in approximating optimal control solutions. Master's thesis, MIT Laboratory for Information and Decision Systems, May 1994.

[18] W. McDermott and M. Athans. Approximating optimal state feedback using neural networks. In *Proceedings of the 33rd IEEE Conference on Decision and Control, June 1993*, pages 2466–2471. IEEE, December 1994.

[19] A. Sage. *Optimum Systems Control.* Prentice-Hall, Englewood Cliffs, NJ, 1968.

[20] R. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, Vol. 3, pp. 9-44, 1988.

[21] A. Tazi–Riffi. The temporal differences algorithm: parametric representations and simultaneous control-prediction task. Master's thesis, MIT Laboratory for Information and Decision Systems, January 1994.

[22] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, Vol. 8, pp. 257-277, 1992.

[23] J. Tsitsiklis. Asynchronous stochastic approximation and Q–learning. *Machine Learning*, Vol. 16, pp. 185-202, 1994.

[24] J. Tsitsiklis and B. VanRoy. Feature-based methods for large scale dynamic programming. Technical Note 2277, MIT Laboratory for Information and Decision Systems, 1994. Also to Appear in *Machine Learning.*

[25] B. VanRoy and J. Tsitsiklis. Stable linear approximations to dynamic programming for stochastic control problems with local transitions. To appear in *Advances in Neural Information Processing Systems 8*, 1995.

[26] C. Watkins. Learning from delayed rewards. Doctoral dissertation, University of Cambridge, 1989.

[27] C. Watkins and P. Dayan. Q–learning. *Machine Learning*, Vol. 8, pp. 279-292, 1992

[28] P. Werbos. Approximate Dynamic Programming for real time control and neural modeling. In *The Handbook for Intelligent Control*, D. White and D. Sofge, editors, Van Nostrand Reinhold, 1992.