Machine Learning, 22, 59–94 (1996)

Feature-Based Methods for Large Scale Dynamic Programming

JOHN N. TSITSIKLIS AND BENJAMIN VAN ROY jnt@mit.edu, bvr@mit.edu

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

Editor: Leslie Pack Kaelbling

Abstract. We develop a methodological framework and present a few different ways in which dynamic programming and compact representations can be combined to solve large scale stochastic control problems. In particular, we develop algorithms that employ two types of feature-based compact representations; that is, representations that involve feature extraction and a relatively simple approximation architecture. We prove the convergence of these algorithms and provide bounds on the approximation error. As an example, one of these algorithms is used to generate a strategy for the game of Tetris. Furthermore, we provide a counter-example illustrating the difficulties of integrating compact representations with dynamic programming, which exemplifies the shortcomings of certain simple approaches.

Keywords: Compact representation, curse of dimensionality, dynamic programming, features, function approximation, neuro-dynamic programming, reinforcement learning.

1. Introduction

Problems of sequential decision making under uncertainty (stochastic control) have been studied extensively in the operations research and control theory literature for a long time, using the methodology of dynamic programming (Bertsekas, 1995). The "planning problems" studied by the artificial intelligence community are of a related nature although, until recently, this was mostly done in a deterministic setting leading to search or shortest path problems in graphs (Korf, 1987). In either context, realistic problems have usually proved to be very difficult mostly due to the large size of the underlying state space or of the graph to be searched. In artificial intelligence, this issue is usually addressed by using heuristic *position evaluation functions*; chess playing programs are a prime example (Korf, 1987). Such functions provide a rough evaluation of the quality of a given state (or board configuration in the context of chess) and are used in order to rank alternative actions.

In the context of dynamic programming and stochastic control, the most important object is the *cost-to-go function*, which evaluates the expected future cost to be incurred, as a function of the current state. Similarly with the artificial intelligence context, cost-to-go functions are used to assess the consequences of any given action at any particular state. Dynamic programming provides a variety of methods for computing cost-to-go functions. Due to the curse of dimensionality, however, the practical applications of dynamic programming are somewhat limited; they involve certain problems in which

the cost–to–go function has a simple analytical form (e.g., controlling a linear system subject to a quadratic cost) or to problems with a manageably small state space.

In most of the stochastic control problems that arise in practice (control of nonlinear systems, queueing and scheduling, logistics, etc.) the state space is huge. For example, every possible configuration of a queueing system is a different state, and the number of states increases exponentially with the number of queues involved. For this reason, it is essentially impossible to compute (or even store) the value of the cost–to–go function at every possible state. The most sensible way of dealing with this difficulty is to generate a compact parametric representation (compact representation, for brevity), such as an artificial neural network, that approximates the cost–to–go function and can guide future actions, much the same as the position evaluators are used in chess. Since a compact representation with a relatively small number of parameters may approximate a cost-to-go function, we are required to compute only a few parameter values rather than as many values as there are states.

There are two important preconditions for the development of an effective approximation. First, we need to choose a compact representation that can closely approximate the desired cost-to-go function. In this respect, the choice of a suitable compact representation requires some practical experience or theoretical analysis that provides some rough information on the shape of the function to be approximated. Second, we need effective algorithms for tuning the parameters of the compact representation. These two objectives are often conflicting. Having a compact representation that can approximate a rich set of functions usually means that there is a large number of parameters to be tuned and/or that the dependence on the parameters is nonlinear, and in either case, there is an increase in the computational complexity involved.

It is important to note that methods of selecting suitable parameters for standard function approximation are inadequate for approximation of cost-to-go functions. In function approximation, we are given training data pairs $\{(x_1, y_1), \ldots, (x_K, y_K)\}$ and must construct a function $y = f(x)$ that "explains" these data pairs. In dynamic programming, we are interested in approximating a cost-to-go function $y = V(x)$ mapping states to optimal expected future costs. An ideal set of training data would consist of pairs $\{(x_1, y_1), \ldots, (x_K, y_K)\}$, where each x_i is a state and each y_i is a sample of the future cost incurred starting at state x_i when the system is optimally controlled. However, since we do not know how to control the system at the outset (in fact, our objective is to figure out how to control the system), we have no way of obtaining such data pairs. An alternative way of making the same point is to note that the desirability of a particular state depends on how the system is controlled, so observing a poorly controlled system does not help us tell how desirable a state will be when the system is well controlled. To approximate a cost-to-go function, we need variations of the algorithms of dynamic programming that work with compact representations.

The concept of approximating cost-to-go functions with compact representations is not new. Bellman and Dreyfus (1959) explored the use of polynomials as compact representations for accelerating dynamic programming. Whitt (1978) and Reetz (1977) analyzed approaches of reducing state space sizes, which lead to compact representations. Schweitzer and Seidmann (1985) developed several techniques for approximating

cost-to-go functions using linear combinations of fixed sets of basis functions. More recently, reinforcement learning researchers have developed a number of approaches, including temporal-difference learning (Sutton, 1988) and Q–learning (Watkins and Dayan, 1992), which have been used for dynamic programming with many types of compact representation, especially artificial neural networks.

Aside from the work of Whitt (1988) and Reetz (1977), the techniques that have been developed largely rely on heuristics. In particular, there is a lack of formal proofs guaranteeing sound results. As one might expect from this, the methods have generated a mixture of success stories and failures. Nevertheless, the success stories – most notably the world-class backgammon player of Tesauro (1992) – inspire great expectations in the potential of compact representations and dynamic programming.

The main aim of this paper is to provide a methodological foundation and a rigorous assessment of a few different ways that dynamic programming and compact representations can be combined to form the basis of a rational approach to difficult stochastic control problems. Although heuristics have to be involved at some point, especially in the selection of a particular compact representation, it is desirable to retain as much as possible of the non-heuristic aspects of the dynamic programming methodology. A related objective is to provide results that can help us assess the efficacy of alternative compact representations.

Cost-to-go functions are generally nonlinear, but often demonstrate regularities similar to those found in the problems tackled by traditional function approximation. There are several types of compact representations that one can use to approximate a cost–to–go function. (a) Artificial neural networks (e.g., multi-layer perceptrons) present one possibility. This approach has led to some successes, such as Tesauro's backgammon player which was mentioned earlier. Unfortunately, it is very hard to quantify or analyze the performance of neural-network-based techniques. (b) A second form of compact representation is based on the use of feature extraction to map the set of states onto a much smaller set of feature vectors. By storing a value of the cost–to–go function for each possible feature vector, the number of values that need to be computed and stored can be drastically reduced and, if meaningful features are chosen, there is a chance of obtaining a good approximation of the true cost-to-go function. (c) A third approach is to choose a parametric form that maps the feature space to cost–to–go values and then try to compute suitable values for the parameters. If the chosen parametric representation is simple and structured, this approach may be amenable to mathematical analysis. One such approach, employing linear approximations, will be studied here.

In this paper, we focus on dynamic programming methods that employ the latter two types of compact representations, i.e., the feature–based compact representations. We provide a general framework within which one can reason about such methods. We also suggest variants of the value iteration algorithm of dynamic programming that can be used in conjunction with the representations we propose. We prove convergence results for our algorithms and then proceed to derive bounds on the difference between optimal performance and the performance obtained using our methods. As an example, one of the techniques presented is used to generate a strategy for Tetris, the arcade game.

This paper is organized as follows. In Section 2, we introduce the Markov decision problem (MDP), which provides a mathematical setting for stochastic control problems, and we also summarize the value iteration algorithm and its properties. In Section 3, we propose a conceptual framework according to which different approximation methodologies can be studied. To illustrate some of the difficulties involved with employing compact representations for dynamic programming, in Section 4, we describe a "natural" approach for dynamic programming with compact representations and then present a counter-example demonstrating the shortcomings of such an approach. In Section 5, we propose a variant of the value iteration algorithm that employs a look-up table in feature space rather than in state space. We also discuss a theorem that ensures its convergence and provides bounds on the accuracy of resulting approximations. Section 6 discusses an application of the algorithm from Section 5 to the game of Tetris. In Section 7, we present our second approximation methodology, which employs feature extraction and linear approximations. Again, we provide a convergence theorem as well as bounds on the performance it delivers. This general methodology encompasses many types of compact representations, and in Sections 8 and 9 we provide two subclasses: interpolative representations and localized basis function architectures. Two technical results that are central to our convergence theorems are presented in the Appendices A and B. In particular, Appendix A proves a theorem involving transformations that preserve contraction properties of an operator, and Appendix B reviews a result on stochastic approximation algorithms involving maximum norm contractions. Appendices C and D provide proofs of the convergence theorems of Sections 5 and 7, respectively.

2. Markov Decision Problems

In this section, we introduce Markov decision problems, which provide a model for sequential decision making problems under uncertainty (Bertsekas, 1995).

We consider infinite horizon, discounted Markov decision problems defined on a finite state space S. Throughout the paper, we let n denote the cardinality of S and, for simplicity, assume that $S = \{1, \ldots, n\}$. For every state $i \in S$, there is a finite set $U(i)$ of possible control actions and a set of nonnegative scalars $p_{ij}(u)$, $u \in U(i)$, $j \in S$, such that $\sum_{j \in S} p_{ij}(u) = 1$ for all $u \in U(i)$. The scalar $p_{ij}(u)$ is interpreted as the probability of a transition to state j, given that the current state is i and the control u is applied. Furthermore, for every state i and control u, there is a random variable c_{iu} which represents the one-stage cost if action u is applied at state i. We assume that the variance of c_{iu} is finite for every $i \in S$ and $u \in U(i)$. In this paper, we treat only Markov decision problems for which transition probabilities $p_{ij}(u)$ and expected immediate costs $E[c_{iu}]$ are known. However, the ideas presented generalize to the context of algorithms such as Q-learning, which assume no knowledge of transition probabilities and costs.

A *stationary policy* is a function π defined on S such that $\pi(i) \in U(i)$ for all $i \in S$. Given a stationary policy, we obtain a discrete-time Markov chain $s^\pi(t)$ with transition probabilities

$$\Pr(s^\pi(t+1) = j \mid s^\pi(t) = i) = p_{ij}(\pi(i)).$$

Let $\beta \in [0,1)$ be a discount factor. For any stationary policy π and initial state i, the cost-to-go V_i^π is defined by

$$V_i^\pi = E\Big[\sum_{t=0}^{\infty} \beta^t c(t) \Big| s^\pi(0) = i\Big],$$

where $c(t) = c_{s^\pi(t),\pi(s^\pi(t))}$. In much of the dynamic programming literature, the mapping from states to cost-to-go values is referred to as the cost-to-go function. However, since the state spaces we consider in this paper are finite, we choose to think of the mapping in terms of a cost-to-go vector whose components are the cost-to-go values of various states. Hence, given the cost-to-go vector V^π of policy π, the cost-to-go value of policy π at state i is the ith component of V^π. The optimal cost-to-go vector V^* is defined by

$$V_i^* = \min_\pi V_i^\pi, \qquad i \in S.$$

It is well known that the optimal cost-to-go vector V^* is the unique solution to Bellman's equation:

$$V_i^* = \min_{u \in U(i)} \Big(E[c_{iu}] + \beta \sum_{j \in S} p_{ij}(u) V_j^*\Big), \qquad \forall i \in S. \tag{1}$$

This equation simply states that the optimal cost–to–go starting from a state i is equal to the minimum, over all actions u that can be taken, of the immediate expected cost $E[c_{iu}]$ plus the suitably discounted expected cost–to–go V_j^* from the next state j, assuming that an optimal policy will be followed in the future.

The Markov decision problem is to find a policy π^* such that

$$V_i^{\pi^*} = V_i^*, \qquad \forall i \in S$$

This is usually done by computing V_i^*, and then choosing π^* as a function which satisfies

$$\pi^*(i) = \arg\min_{u \in U(i)} \Big(E[c_{iu}] + \beta \sum_{j \in S} p_{ij}(u) V_j^*\Big), \qquad \forall i \in S.$$

If we can not compute V^* but can obtain an approximation V to V^*, we might generate a reasonable control policy π_V satisfying

$$\pi_V(i) = \arg\min_{u \in U(i)} \Big(E[c_{iu}] + \beta \sum_{j \in S} p_{ij}(u) V_j\Big), \qquad \forall i \in S.$$

Intuitively, this policy considers actual immediate costs and uses V to judge future consequences of control actions. Such a policy is sometimes called a *greedy policy* with respect to the cost-to-go vector V, and as V approaches V^*, the performance of a greedy policy π_V approaches that of an optimal policy π^*.

There are several algorithms for computing V^* but we only discuss the value iteration algorithm which forms the basis of the algorithms to be considered later on. We start with some notation. We define $T_i : \Re^n \mapsto \Re$ by

$$T_i(V) = \min_{u \in U(i)} \left(E[c_{iu}] + \beta \sum_{j \in S} p_{ij}(u) V_j \right), \qquad \forall i \in S. \tag{2}$$

We then define the *dynamic programming operator* $T : \Re^n \mapsto \Re^n$ by

$$T(V) = (T_1(V), \ldots, T_n(V))$$

In terms of this notation, Bellman's equation simply asserts that $V^* = T(V^*)$ and V^* is the unique fixed point of T. The value iteration algorithm is described by

$$V(t+1) = T(V(t)),$$

where $V(0)$ is an arbitrary vector in $\Re^n$ used to initialize the algorithm. Intuitively, each $V(t)$ is an estimate (though not necessarily a good one) of the true cost–to–go function V^*, which gets replaced by the hopefully better estimate $T(V(t))$.

Let $\|\cdot\|_\infty$ be the maximum norm defined for every vector $x = (x_1, \ldots, x_n) \in \Re^n$ by $\|x\|_\infty = \max_i |x_i|$. It is well known (Bertsekas, 1995) and easy to check that T is a contraction with respect to the maximum norm, that is, for all $V, V' \in \Re^n$,

$$\|T(V) - T(V')\|_\infty \leq \beta \|V - V'\|_\infty.$$

For this reason, the sequence $V(t)$ produced by the value iteration algorithm converges to V^*, at the rate of a geometric progression. Unfortunately, this algorithm requires that we maintain and update a vector V of dimension n and this is essentially impossible when n is extremely large.

For notational convenience, it is useful to define for each policy π the operator $T_i^\pi : \Re^n \mapsto \Re$:

$$T_i^\pi(V) = E[c_{i\pi(i)}] + \beta \sum_{j \in S} p_{ij}(\pi(i)) V_j,$$

for each $i \in S$. The operator T^π is defined by

$$T^\pi(V) = (T_1^\pi(V), \ldots, T_n^\pi(V)).$$

It is well known that T^π is also a contraction of the maximum norm and that V^π is its unique fixed point (Bertsekas, 1995). Note that, for any vector $V \in \Re^n$ we have

$$T(V) = T^{\pi_V}(V),$$

since the cost-minimizing control action in Equation (2) is given by the greedy policy.

3. Compact Representations and Features

As mentioned in the introduction, the size of state spaces typically grows exponentially with the number of variables involved. Because of this, it is often impractical to compute and store every component of a cost-to-go vector. We set out to overcome this limitation by using compact representations to approximate cost-to-go vectors. In this section, we

develop a formal framework for reasoning about compact representations and features as groundwork for subsequent sections, where we will discuss ways of using compact representations for dynamic programming. The setting is in many respects similar to that in (Schweitzer and Seidman, 1985).

A compact representation can be thought of as a scheme for recording a high-dimensional cost-to-go vector $V \in \Re^n$ using a lower-dimensional parameter vector $W \in \Re^m$ ($m \ll n$). Such a scheme can be described by a mapping $\tilde{V} : \Re^m \to \Re^n$ which to any given parameter vector $W \in \Re^m$ associates a cost-to-vector $\tilde{V}(W)$. In particular, each component $\tilde{V}_i(W)$ of the mapping is the ith component of a cost-to-go vector represented by the parameter vector W. Note that, although we may wish to represent an arbitrary vector V in $\Re^n$, such a scheme allows for exact representation only of those vectors V which happen to lie in the range of $\tilde{V}$.

Let us define a *feature* f as a function from the state space S into a finite set Q of feature values. For example, if the state i represents the number of customers in a queueing system, a possible and often interesting feature f is defined by $f(0) = 0$ and $f(i) = 1$ if $i > 0$. Such a feature focuses on whether a queue is empty or not.

Given a Markov decision problem, one may wish to use several features $f_1, \ldots, f_K$, each one being a function from the state space S to a finite set Q_k, $k = 1, \ldots, K$. Then, to each state $i \in S$, we associate the feature vector $F(i) = (f_1(i), \ldots, f_K(i))$. Such a feature vector is meant to represent the most salient properties of a given state. Note that the resulting set of all possible feature vectors is the Cartesian product of the sets Q_k and its cardinality increases exponentially with the number of features.

In a feature-based compact representation, each component $\tilde{V}_i$ of the mapping $\tilde{V}$ is a function of the corresponding feature vector $F(i)$ and the parameter vector W (but not an explicit function of the state value i). Hence, for some function $g : (\prod_{k=1}^{K} Q_k) \times \Re^m \mapsto \Re$,

$$\tilde{V}_i(W) = g(F(i), W). \tag{3}$$

If each feature takes on real values, we have $Q_k \subset \Re$ for all k, in which case it may be natural to define the function g over all possible real feature values, $g : \Re^K \times \Re^m \to \Re$, even though g will only ever be computed over a finite domain. Figure 1 illustrates the structure of a feature-based compact representation.

In most problems of interest, V_i^* is a highly complicated function of i. A representation like the one in Equation (3) attempts to break the complexity of V^* into less complicated mappings g and F. There is usually a trade-off between the complexity of g and F and different choices lead to drastically different structures. As a general principle, the feature extraction function F is usually hand crafted and relies on whatever human experience or intelligence is available. The function g represents the choice of an *architecture* used for approximation and the vector W are the free parameters (or weights) of the chosen architecture. When a compact representation is used for static function approximation, the values for the parameters W are chosen using some optimization algorithm, which could range from linear regression to backpropagation in neural networks. In this paper, however, we will develop parameter selection techniques for dynamic programming (rather than function approximation). Let us first discuss some alternative architectures.

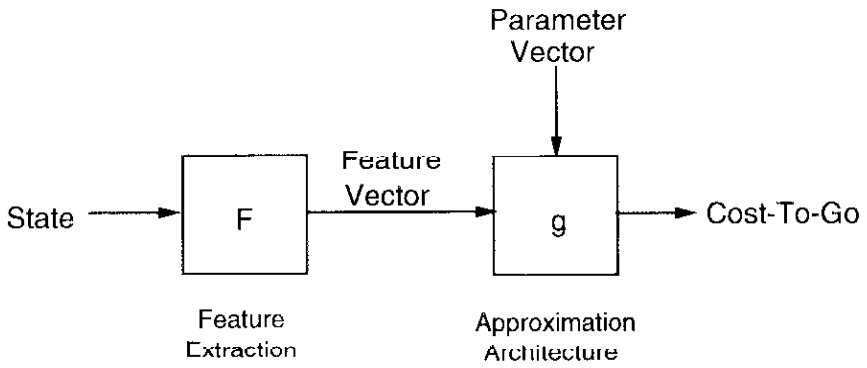

Figure 1. Block structure of a feature-based compact representation.

Look-Up Tables

One possible compact representation can be obtained by employing a look-up table in feature space, that is, by assigning one value to each point in the feature space. In this case, the parameter vector W contains one component for each possible feature vector. The function g acts as a hashing function, selecting the component of W corresponding to a given feature vector. In one extreme case, each feature vector corresponds to a single state, there are as many parameters as states, and $\tilde{V}$ becomes the identity function. On the other hand, effective feature extraction may associate many states with each feature vector so that the optimal cost-to-go values of states associated to any particular feature vector are close. In this scenario, the feature space may be much smaller that the state space, reducing the number of required parameters. Note, however, that the number of possible feature vectors increases exponentially with the number of features. For this reason, look-up tables are only practical when there are very few features.

Using a look-up table in feature space is equivalent to partitioning the state space and then using a common value for the cost-to-go from all the states in any given partition. In this context, the set of states which map to a particular feature vector forms one partition. By identifying one such partition per possible feature vector, the feature extraction mapping F defines a partitioning of the state space. The function g assigns each component of the parameter vector to a partition. For conceptual purposes, we choose to view this type of representation in terms of state aggregation, rather than feature-based look-up tables. As we will see in our formulation for Tetris, however, the feature-based look-up table interpretation is often more natural in applications.

We now develop a mathematical description of state aggregation. Suppose that the state space $S = \{1, \ldots, n\}$ has been partitioned into m disjoint subsets $S_1, \ldots, S_m$, where m is the same as the dimension of the parameter vector W. The compact representations we consider take on the following form:

$$\tilde{V}_i(W) = W_j,$$

for any $i \in S_j$.

There are no inherent limitations to the representational capability of such an architecture. Whatever limitations this approach may have are actually connected with the availability of useful features. To amplify this point, let us fix some $\epsilon > 0$ and let us define, for all j,

$$S_j = \{i \mid j\epsilon \leq V_i^* < (j+1)\epsilon\}.$$

Using this particular partition, the function V^* can be approximated with an accuracy of ϵ. The catch is of course that since V^* is unknown, we are unable to form the sets S_j. A different way of making the same point is to note that the most useful feature of a state is its optimal cost–to–go but, unfortunately, this is what we are trying to compute in the first place.

Linear Architectures

With a look-up table, we need to store one parameter for every possible value of the feature vector $F(i)$, and, as already noted, the number of possible values increases exponentially with the number of features. As more features are deemed important, look-up tables must be abandoned at some point and a different kind of parametric representation is now called for. For instance, a representation of the following form can be used:

$$\tilde{V}_i(W) = \sum_{k=1}^{K} W_k f_k(i). \tag{4}$$

This representation approximates a cost-to-go function using a linear combination of features. This simplicity makes it amenable to rigorous analysis, and we will develop an algorithm for dynamic programming with such a representation. Note that the number of parameters only grows linearly with the number of features. Hence, unlike the case of look-up tables, the number of features need not be small. However, it is important to choose features that facilitate the linear approximation.

Many popular function approximation architectures fall in the class captured by Equation (4). Among these are radial basis functions, wavelet networks, polynomials, and more generally all approximation methods that involve a fixed set of basis functions. In this paper, we will discuss two types of these compact representations that are compatible with our algorithm – a method based on linear interpolation and localized basis functions.

Nonlinear Architectures

The architecture, as described by g, could be a nonlinear mapping such as a feedforward neural network (multi-layer perceptron) with parameters W. The feature extraction mapping F could be either entirely absent or it could be included to facilitate the job of the neural network. Both of these options were used in the backgammon player of Tesauro and, as expected, the inclusion of features led to improved performance. Unfortunately, as was mentioned in the introduction, there is not much that can be said analytically in this context.

4. Least-Squares Value Iteration: A Counter-Example

Given a set of k samples $\{(i_1, V_{i_1}^*), (i_2, V_{i_2}^*), ..., (i_K, V_{i_K}^*)\}$ of an optimal cost-to-go vector V^*, we could approximate the vector with a compact representation $\tilde{V}$ by choosing parameters W to minimize an error function such as

$$\sum_{k=1}^{K} \left(\tilde{V}_{i_k}(W) - V_{i_k}^*\right)^2,$$

i.e., by finding the "least-squares fit." Such an approximation conforms to the spirit of traditional function approximation. However, as discussed in the introduction, we do not have access to such samples of the optimal cost-to-go vector. To approximate an optimal cost-to-go vector, we must adapt dynamic programming algorithms such as the value iteration algorithm so that they manipulate parameters of compact representations.

For instance, we could start with a parameter vector $W(0)$ corresponding to an initial cost-to-go vector $\tilde{V}(W(0))$, and then generate a sequence $\{W(t)|t = 1, 2, ...\}$ of parameter vectors such that $\tilde{V}(W(t+1))$ approximates $T(\tilde{V}(W(t)))$. Hence, each iteration approximates a traditional value iteration. The hope is that, by approximating individual value iterations in such a way, the sequence of approximations converges to an accurate approximation of the optimal cost-to-go vector, which is what value iteration converges to.

It may seem as though any reasonable approximation scheme could be used to generate each approximate value iteration. For instance, the "least-squares fit" is an obvious candidate. This involves selecting $W(t+1)$ by setting

$$W(t+1) = \arg\min_{W} \sum_{i=1}^{n} \left(\tilde{V}(W) - T(\tilde{V}(W(t)))\right)^2. \tag{5}$$

However, in this section we will identify subtleties that make the choice of criterion for parameter selection crucial. Furthermore, an approximation method that is compatible with one type of compact representation may generate poor results when a different compact representation is employed.

We will now develop a counter-example that illustrates the shortcomings of such a combination of value iteration and least-squares approximation. This analysis is particularly interesting, since the algorithm is closely related to Q learning (Watkins and Dayan, 1992) and temporal-difference learning (TD(λ)) (Sutton, 1988), with λ set to 0. The counter-example discussed demonstrates the short-comings of some (but not all) variants of Q learning and temporal difference learning that are employed in practice. [1]

Bertsekas (1994) described a counter-example to methods like the one defined by Equation (5). His counter-example involves a Markov decision problem and a compact representation that could generate a close approximation (in terms of Euclidean distance) of the optimal cost-to-go vector, but fails to do so when algorithms like the one we have described are used. In particular, the parameter vector does converge to some $W^* \in \Re^m$, but, unfortunately, this parameter vector generates a poor estimate of the

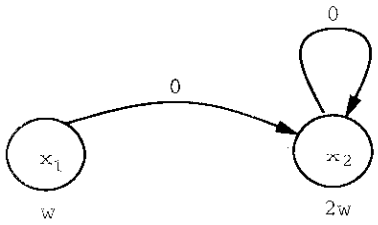

Figure 2. A counter-example.

optimal cost-to-go vector (in terms of Euclidean distance), that is,

$$\|\tilde{V}(W^*) - V^*\|_2 \gg \min_{W \in \Re^n} \|\tilde{V}(W) - V^*\|_2,$$

where $\|\cdot\|_2$ denotes the Euclidean norm. With our upcoming counter-example, we show that much worse behavior is possible: even when the compact representation can generate a perfect approximation of the optimal cost-to-go function (i.e., $\min_W \|\tilde{V}(W) - V^*\|_2 = 0$), the algorithm may diverge.

Consider the simple Markov decision problem depicted in Figure 2. The state space consists of two states, x_1 and x_2, and at state x_1 a transition is always made to x_2, which is an absorbing state. There are no control decisions involved. All transitions incur 0 cost. Hence, the optimal cost to-go function assigns 0 to both states.

Suppose a feature f is defined over the state space so that $f(x_1) = 1$ and $f(x_2) = 2$, and a compact representation of the form

$$\tilde{V}_i(w) = wf(i), \qquad i \in \{x_1, x_2\},$$

is employed, where w is scalar. When we set w to 0, we get $\tilde{V}(w) = V^*$, so a perfect representation of the optimal cost-to-go vector is possible.

Let us investigate the behavior of the least-squares value iteration algorithm with the Markov decision problem and compact representation we have described. The parameter w evolves as follows:

$$\begin{aligned} w(t+1) &= \arg\min_{w \in \Re^m} \sum_{i \in S} \left(\tilde{V}_i(w) - T_i(\tilde{V}(w(t)))\right)^2 \\ &= \arg\min_{w \in \Re} \left((w - \beta 2w(t))^2 + (2w - \beta 2w(t))^2\right), \end{aligned}$$

and we obtain

$$w(t+1) = \frac{6}{5}\beta w(t). \tag{6}$$

Hence, if $\beta > \frac{5}{6}$ and $w(0) \neq 0$, the sequence diverges. Counter-examples involving Markov decision problems that allow several control actions at each state can also be

produced. In that case, the least-squares approach to value iteration can generate poor control strategies even when the optimal cost-to-go vector can be represented.

The shortcomings of straightforward procedures such as least-squares value iteration characterize the challenges involved with combining compact representations and dynamic programming. The remainder of this paper is dedicated to the development of approaches that guarantee more graceful behavior.

5. Value Iteration with Look-Up Tables

As a starting point, let us consider what is perhaps the simplest possible type of compact representation. This is the feature-based look-up table representation described in Section 3. In this section, we discuss a variant of the value iteration algorithm that has sound convergence properties when used in conjunction with such representations. We provide a convergence theorem, which we formally prove in Appendix C. We also point out relationships between the presented algorithm and previous work in the fields of dynamic programming and reinforcement learning.

5.1. *Algorithmic Model*

As mentioned earlier, the use of a look-up table in feature space is equivalent to state aggregation. We choose this latter viewpoint in our analysis. We consider a partition of the state space $S = \{1, \ldots, n\}$ into subsets $S_1, S_2, \ldots, S_m$; in particular, $S = S_1 \cup S_2 \cup \cdots \cup S_m$ and $S_i \cap S_j = \emptyset$ if $i \neq j$. Let $\tilde{V} : \Re^m \mapsto \Re^n$, the function which maps a parameter vector W to a cost-to-go vector V, be defined by:

$$\tilde{V}_i(W) = W_j, \qquad \forall i \in S_j.$$

Let $\mathcal{N}$ be the set of nonnegative integers. We employ a discrete variable t, taking on values in $\mathcal{N}$, which is used to index successive updates of the parameter vector W. Let $W(t)$ be the parameter vector at time t. Let Γ^j be an infinite subset of $\mathcal{N}$ indicating the set of times at which an update of the jth component of the parameter vector is performed. For each set S_j, $j = 1, \ldots, m$, let $p^j(\cdot)$ be a probability distribution over the set S_j. In particular, for every $i \in S_j$, $p^j(i)$ is the probability that a random sample from S_j is equal to i. Naturally, we have $p^j(i) \geq 0$ and $\sum_{i \in S_j} p^j(i) = 1$.

At each time t, let $X(t)$ be an m–dimensional vector whose jth component is a random representative of the set S_j, sampled according to the probability distribution $p^j(\cdot)$. We assume that each such sample is generated independently from everything else that takes place in the course of the algorithm.[2]

The value iteration algorithm applied at state $X_j(t)$ would update the value $V_{X_j(t)}$, which is represented by W_j, by setting it equal to $T_{X_j(t)}(V)$. Given the compact representation that we are using and given the current parameter vector $W(t)$, we actually need to set W_j to $T_{X_j(t)}(\tilde{V}(W(t)))$. However, in order to reduce the sensitivity of the algorithm to the randomness caused by the random sampling, W_j is updated in that direction with a small stepsize. We therefore end up with the following update formula:

$$W_j(t+1) = (1-\alpha_j(t))W_j(t) + \alpha_j(t)T_{X_j(t)}(\tilde{V}(W(t))), \qquad t \in \Gamma^j, \tag{7}$$

$$W_i(t+1) = W_i(t), \qquad t \notin \Gamma^i. \tag{8}$$

Here, $\alpha_j(t)$ is a stepsize parameter between 0 and 1. In order to bring Equations (7) and (8) into a common format, it is convenient to assume that $\alpha_j(t)$ is defined for every j and t, but that $\alpha_j(t) = 0$ for $t \notin \Gamma^j$.

In a simpler version of this algorithm, we could define a single probability distribution $p(\cdot)$ over the entire state space S such that for each subset S_j, we have $\sum_{i \in S_j} p_X(i) > 0$. Then, defining $x(t)$ as a state sampled according to the $p(\cdot)$, updates of the form

$$W_i(t+1) = (1-\alpha_i(t))W_i(t) + \alpha_i(t)T_{x(t)}(\tilde{V}(W(t))), \qquad \text{if } x(t) \in S_i, \tag{9}$$

$$W_j(t+1) = W_j(t), \qquad \text{if } x(t) \notin S_j, \tag{10}$$

can be used. The simplicity of this version – primarily the fact that samples are taken from only one distribution rather than many – makes it attractive for implementation. This version has a potential shortcoming, though. It does not involve any adaptive exploration of the feature space; that is, the choice of the subset S_j to be sampled does not depend on past observations. This rules out the possibility of adapting the distribution to concentrate on a region of the feature space that appears increasingly significant as approximation of the cost-to-go function ensues. Regardless, this simple version is the one chosen for application to the Tetris playing problem which is reported in Section 5.

We view all of the variables introduced so far, namely, $\alpha_j(t)$, $X_j(t)$, and $W(t)$, as random variables defined on a common probability space. The reason for $\alpha_j(t)$ being a random variable is that the decision whether W_j will be updated at time t (and, hence, whether $\alpha_j(t)$ will be zero or not) may depend on past observations. Let $\mathcal{F}(t)$ be the set of all random variables that have been realized up to and including the point at which the stepsize $\alpha_j(t)$ is fixed but just before $X_j(t)$ is generated.

5.2. *Convergence Theorem*

Before stating our convergence theorem, we must introduce the following standard assumption concerning the stepsize sequence:

Assumption 1 *a) For all i, the stepsize sequence satisfies*

$$\sum_{t=0}^{\infty} \alpha_i(t) = \infty, \qquad \text{w.p.1.} \tag{11}$$

b) There exists some (deterministic) constant C such that

$$\sum_{t=0}^{\infty} \alpha_i^2(t) < C, \qquad \text{w.p.1.} \tag{12}$$

Following is the convergence theorem:

THEOREM 1 *Let Assumption 1 hold.*
(a) With probability 1, the sequence $W(t)$ converges to W^*, the unique vector whose components solve the following system of equations:

$$W_j^* = \sum_{i \in S_j} p^j(i) T_i(\tilde{V}(W^*)), \qquad \forall j. \tag{13}$$

Define V^* as the optimal cost-to-go vector and $e \in \Re^m$ by

$$e_i = \max_{j,l \in S_i} |V_j^* - V_l^*|, \qquad \forall i \in \{1, ..., m\}.$$

Recall that $\pi_{\tilde{V}(W^*)}$ denotes a greedy policy with respect to cost-to-go vector $\tilde{V}(W^*)$, i.e.,

$$\pi_{\tilde{V}(W^*)}(i) = \arg \min_{u \in U(i)} \left(E[c_{iu}] + \beta \sum_{j \in S} p_{ij}(u) \tilde{V}_j(W^*) \right).$$

The following hold:
(b)

$$\|\tilde{V}(W^*) \quad V^*\|_\infty \leq \frac{\|e\|_\infty}{1 - \beta},$$

(c)

$$\|V^{\pi_{\tilde{V}(W^*)}} \quad V^*\|_\infty \leq \frac{2\beta \|e\|_\infty}{(1 - \beta)^2},$$

(d) there exists an example for which the bounds in (b) and (c) both hold with equality.

A proof of Theorem 1 is provided in Appendix C. We prove the theorem by showing that the algorithm corresponds to a stochastic approximation involving a maximum norm contraction, and then appeal to a theorem concerning asynchronous stochastic approximation due to Tsitsiklis (1994) (see also (Jaakola, Jordan, and Singh, 1994)), which is discussed in Appendix B, and a theorem concerning multi-representation contractions presented and proven in Appendix A.

5.3. *The Quality of Approximations*

Theorem 1 establishes that the quality of approximations is determined by the quality of the chosen features. If the true cost–to–go function V^* can be accurately represented in the form $\tilde{V}(W)$, then the computed parameter values deliver near optimal performance. This is a desirable property.

The distressing aspect of Theorem 1 is the wide margin allowed by the worst case bound. As the discount factor approaches unity, the $\frac{1}{1-\beta}$ term explodes. Since discount factors close to one are most common in practice, this is a severe weakness. However, achieving or nearly achieving the worst case bound in real world applications may be a rare event. These weak bounds are to be viewed as the minimum desired properties for a method to be sound. As we have seen in Section 4, even this is not guaranteed by some other methods in current practice.

5.4. *Role of the Sampling Distributions*

The worst-case bounds provided by Theorem 1 are satisfied for any set of state–sampling distributions. The distribution of probability among states within a particular partition may be arbitrary. Sampling only a single state per partition constitutes a special case which satisfies the requirement. For this special case, a decaying stepsize is unnecessary. If a constant stepsize of one is used in such a setting, the algorithm becomes an asynchronous version of the standard value iteration algorithm applied to a reduced Markov decision problem that has one state per partition of the original state space; the convergence of such an algorithm is well known (Bertsekas, 1982; Bertsekas and Tsitsiklis, 1989). Such a state space reduction is analogous to that brought about by state space discretization, which is commonly applied to problems with continuous state spaces. Whitt (1978) considered this method of discretization and derived the bounds of Theorem 1, for the case where a single state is sampled in each partition. Our result can be viewed as a generalization of Whitt's, allowing the use of arbitrary sampling distributions.

When the state aggregation is perfect in that the true optimal cost-to-go values for all states in any particular partition are equal, the choice of sampling function is insignificant. This is because, independent of the distribution, the error bound is zero when there is no fluctuation of optimal cost-to-go values within any partition. In contrast, when V^* fluctuates within partitions, the error achieved by a feature-based approximation can depend on the sampling distribution. Though the derived bound limits the error achieved using any set of state distributions, the choice of distributions may play an important role in attaining errors significantly lower than this worst case bound. It often appears desirable to distribute the probability among many representative states in each partition. If only a few states are sampled, the error can be magnified if these states do not happen to be representative of the whole partition. On the other hand, if many states are chosen, and their cost-to-go values are in some sense averaged, a cost-to-go value representative of the entire partition may be generated. It is possible to develop heuristics to aid in choosing suitable distributions, but the relationship between sampling distributions and approximation error is not yet clearly understood or quantified.

5.5. *Related Work*

As was mentioned earlier, Theorem 1 can be viewed as an extension to the work of Whitt (1978). However, our philosophy is much different. Whitt was concerned with discretizing a continuous state space. Our concern here is to exploit human intuition concerning useful features and heuristic state sampling distributions to drastically reduce the dimensionality of a dynamic programming problem.

Several other researchers have considered ways of aggregating states to facilitate dynamic programming. Bertsekas and Castañon (1989) developed an adaptive aggregation scheme for use with the policy iteration algorithm. Rather than relying on feature extraction, this approach automatically and adaptively aggregates states during the course of an algorithm based on probability transition matrices under greedy policies.

The algorithm we have presented in this section is closely related to Q-learning and temporal-difference learning (TD(λ)) in the case where λ is set to 0. In fact, Theorem 1 can easily be extended so that it applies to TD(0) or Q-learning when used in conjunction with feature-based look-up tables. Since the convergence and efficacy of TD(0) and Q-learning in this setting have not been theoretically established in the past, our theorem sheds new light on these algorithms.

In considering what happens when applying the Q-learning algorithm to partially observable Markov decision problems, Jaakola, Singh and Jordan (1995) prove a convergence theorem similar to part (a) of Theorem 1. Their analysis involves a scenario where the state aggregation is inherent because of incomplete state information – i.e., a policy must choose the same action within a group of states because there is no way a controller can distinguish between different states within the group – and is not geared towards accelerating dynamic programming in general.

6. Example: Playing Tetris

As an example, we used the algorithm from the previous section to generate a strategy for the game of Tetris. In this section we discuss the process of formulating Tetris as a Markov decision problem, choosing features, and finally, generating and assessing a game strategy. The objective of this exercise was to verify that feature based value iteration can deliver reasonable performance for a rather complicated problem. Our objective was not to construct the best possible Tetris player, and for this reason, no effort was made to construct and use sophisticated features.

6.1. Problem Formulation

We formulated the game of Tetris as a Markov decision problem, much in the same spirit as the Tetris playing programs of Lippman, Kukolich and Singer (1993). Each state of the Markov decision problem is recorded using a two–hundred–dimensional binary vector (the *wall vector*) which represents the configuration of the current wall of bricks and a seven–dimensional binary vector which identifies the current falling piece.[3] The Tetris screen is twenty squares high and ten squares wide, and each square is associated with a component of the wall vector. The component corresponding to a particular square is assigned 1 if the square is occupied by a brick and 0 otherwise. All components of the seven–dimensional vector are assigned 0 except for the one associated with the piece which is currently falling (there are seven types of pieces).

At any time, the set of possible decisions includes the locations and orientations at which we can place the falling piece on the current wall of bricks. The subsequent state is determined by the resulting wall configuration and the next random piece that appears. Since the resulting wall configuration is deterministic and there are seven possible pieces, there are seven potential subsequent states for any action, each of which occurs with equal probability. An exception is when the wall is higher than sixteen rows. In this circumstance, the game ends, and the state is absorbing.

Each time an entire row of squares is filled with bricks, the row vanishes, and the portion of the wall previously supported falls by one row. The goal in our version of Tetris is to maximize the expected number of rows eliminated during the course of a game. Though we generally formulate Markov decision problems in terms of minimizing costs, we can think of Tetris as a problem of maximizing rewards, where rewards are negative costs. The reward of a transition is the immediate number of rows eliminated. To ensure that the optimal cost–to–go from each state is finite, we chose a discount factor of $\beta = 0.9999$.

In the vast majority of states, there is no scoring opportunity. In other words, given a random wall configuration and piece, chances are that no decision will lead to an immediate reward. When a human being plays Tetris, it is crucial that she makes decisions in anticipation of long–term rewards. Because of this, simple policies that play Tetris such as those that make random decisions or even those that make greedy decisions (i.e., decisions that maximize immediate rewards with no concern for the future) rarely score any points in the course of a game. Decisions that deliver reasonable performance reflect a degree of "foresight."

6.2. *Some Simple Features*

Since each combination of wall configuration and current piece constitute a separate state, the state space of Tetris is huge. As a result, classical dynamic programming algorithms are inapplicable. Feature-based value iteration, on the other hand, *can* be used. In order to demonstrate this, we chose some simple features and applied the algorithm.

The two features employed in our experiments were the height of the current wall and the number of holes (empty squares with bricks both above and below) in the wall. Let us denote the set of possible heights by $H = \{0, ..., 20\}$, and the set of possible numbers of holes by $L = \{0, ..., 200\}$. We can then think of the feature extraction process as the application of a function $F : S \rightarrow H \times L$.

Note that the chosen features do not take into account the shape of the current falling piece. This may initially seem odd, since the decision of where to place a piece relies on knowledge of its shape. However, the cost–to–go function actually only needs to enable the assessment of alternative decisions. This would entail assigning a value to each possible placement of the current piece on the current wall. The cost–to–go function thus needs only to evaluate the desirability of each resulting wall configuration. Hence, features that capture salient characteristics of a wall configuration are sufficient.

6.3. *A Heuristic Evaluation Function*

As a baseline Tetris–playing program, we produced a simple Tetris player that bases state assessments on the two features. The player consists of a quadratic function $g : H \times L \rightarrow \Re$ which incorporates some heuristics developed by the authors. Then, although the composition of feature extraction and the rule based system's evaluation function,

$g \circ F$, is not necessarily an estimate of the optimal cost-to-go vector, the expert player follows a greedy policy based on the composite function.

The average score of this Tetris player on a hundred games was 31 (rows eliminated). This may seem low since arcade versions of Tetris drastically inflate scores. To gain perspective, though, we should take into account the fact that an experienced human Tetris player would take about three minutes to eliminate thirty rows.

6.4. *Value Iteration with a Feature-Based Look-Up Table*

We synthesized two Tetris playing programs by applying the feature-based value iteration algorithm. These two players differed in that each relied on different state–sampling distributions.

The first Tetris player used the states visited by the heuristic player as sample states for value iterations. After convergence, the average score of this player on a hundred games was 32. The fact that this player does not do much better than the heuristic player is not surprising given the simplicity of the features on which both players base position evaluations. This example reassures us, nevertheless, that feature-based value iteration converges to a reasonable solution.

We may consider the way in which the first player was constructed unrealistic, since it relied on a pre-existing heuristic player for state sampling. The second Tetris player eliminates this requirement by uses an *ad hoc* state sampling algorithm. In sampling a state, the sampling algorithm begins by sampling a maximum height for the wall of bricks from a uniform distribution. Then, for each square below this height, a brick is placed in the square with probability $\frac{3}{4}$. Each unsupported row of bricks is then allowed to fall until every row is supported. The player based on this sampling function gave an average score of 11 (equivalent to a human game lasting about one and a half minutes).

The experiments performed with Tetris provide some assurance that feature-based value iteration produces reasonable control policies. In some sense, Tetris is a worst-case scenario for the evaluation of automatic control algorithms, since humans excel at Tetris. The goal of algorithms that approximate dynamic programming is to generate reasonable control policies for large scale stochastic control problems that we have no other reasonable way of addressing. Such problems would not be natural to humans, and any reasonable policy generated by feature-based value iteration would be valuable. Furthermore, the features chosen for this study were very crude; perhaps with the introduction of more sophisticated features, feature-based value iteration would excel in Tetris. As a parting note, an additional lesson can be drawn from the fact that two strategies generated by feature–based value iteration were of such disparate quality. This is that the sampling distribution plays an important role.

7. Value Iteration with Linear Architectures

We have discussed the use of feature-based look-up tables with value iteration, and found that their use can significantly accelerate dynamic programming. However, employing a

look-up table with one entry per feature vector is viable only when the number of feature vectors is reasonably small. Unfortunately, the number of possible feature vectors grows exponentially with the dimension of the feature space. When the number of features is fairly large, alternative compact representations, requiring fewer parameters, must be used. In this section, we explore one possibility which involves a linear approximation architecture. More formally, we consider compact representations of the form

$$\tilde{V}_i(W) = \sum_{k=1}^{K} W_k f_k(i) = W^T F(i), \qquad \forall i \in S, \tag{14}$$

where $W \in \Re^K$ is the parameter vector, $F(i) = (f_1(i), ..., f_K(i)) \in \Re^K$ is the feature vector associated with state i, and the superscript T denotes transpose. This type of compact representation is very attractive since the number of parameters is equal to the number of dimensions of, rather than the number of elements in, the feature space.

We will describe a variant of the value iteration algorithm that, under certain assumptions on the feature mapping, is compatible with compact representations of this form, and we will provide a convergence result and bounds on the quality of approximations. Formal proofs are presented in Appendix D.

7.1. *Algorithmic Model*

The iterative algorithm we propose is an extension to the standard value iteration algorithm. At the outset, K representative states $i_1, ..., i_K$ are chosen, where K is the dimension of the parameter vector. Each iteration generates an improved parameter vector $W(t+1)$ from a parameter vector $W(t)$ by evaluating $T_i(\tilde{V}(W(t)))$ at states $i_1, ..., i_K$ and then computing $W(t+1)$ so that $\tilde{V}_i(W(t+1)) = T_i(\tilde{V}(W(t)))$ for $i \in \{i_1, ..., i_K\}$. In other words, the new cost-to-go estimate is constructed by fitting the compact representation to $T(V)$, where V is the previous cost to go estimate, by fixing the compact representation at $i_1, ..., i_K$. If suitable features and representative states are chosen, $\tilde{V}(W(t))$ may converge to a reasonable approximation of the optimal cost-to-go vector V^*. Such an algorithm has been considered in the literature (Bellman (1959), Reetz (1977), Morin (1979)). Of these references, only (Reetz (1977)), establishes convergence and error bounds. However, Reetz's analysis is very different from what we will present and is limited to problems with one dimensional state spaces.

If we apply an algorithm of this type to the counter-example of Section 4, with $K = 1$ and $i_1 = x_1$, we obtain $w(t+1) = 2\beta w(t)$, and if $\beta > \frac{1}{2}$, the algorithm diverges. Thus, an algorithm of this type is only guaranteed to converge for a subclass of the compact representations described by Equation (14). To characterize this subclass, we introduce the following assumption which restricts the types of features that may be employed:

Assumption 2 *Let $i_1, ..., i_K \in S$ be the pre-selected states used by the algorithm.*
(a) The vectors $F(i_1), ..., F(i_K)$ are linearly independent.
(b) There exists a value $\beta' \in [\beta, 1)$ such that for any state $i \in S$ there exist $\theta_1(i), ...,$

$\theta_K(i) \in \Re$ *with*

$$\sum_{k=1}^{K} |\theta_k(i)| \leq 1,$$

and

$$F(i) = \frac{\beta'}{\beta} \sum_{k=1}^{K} \theta_k(i) F(i_k).$$

In order to understand the meaning of this condition, it is useful to think about the feature space defined by $\{F(i) | i \subset S\}$ and its convex hull. In the special case where $\beta = \beta'$ and under the additional restrictions $\sum_{k=1}^{K} \theta_k(i) = 1$ for all i, and $\theta_k(i) \geq 0$, the feature space is contained in the $(K-1)$-dimensional simplex with vertices $F(i_1), \ldots, F(i_K)$. Allowing β' to be strictly greater than β introduces some slack and allows the feature space to extend a bit beyond that simplex. Finally, if we only have the condition $\sum_{k=1}^{K} |\theta_k(i)| \leq 1$, the feature space is contained in the convex hull of the vectors $\pm\frac{\beta'}{\beta} F(i_1), \pm\frac{\beta'}{\beta} F(i_2), \ldots, \pm\frac{\beta'}{\beta} F(i_K)$.

The significance of the geometric interpretation lies in the fact that the extrema of a linear function within a convex polyhedron must be located at the corners. Formally, Assumption 2 ensures that

$$\|\tilde{V}(W)\|_\infty < \frac{\beta'}{\beta} \max_k |\tilde{V}_{i_k}(W)|.$$

The upcoming convergence proof capitalizes on this property.

To formally define our algorithm, we need to define a few preliminary notions. First, the representation described by Equation (14) can be rewritten as

$$\tilde{V}(W) = MW, \tag{15}$$

where $M \in \Re^{n \times K}$ is a matrix with the ith row equal to $F(i)^T$. Let $L \in \Re^{K \times K}$ be a matrix with the kth row being $F(i_k)^T$. Since the rows of L are linearly independent, there exists a unique matrix inverse $L^{-1} \in \Re^{K \times K}$. We define $M^\dagger \in \Re^{K \times n}$ as follows. For $k \subset \{1, \ldots, K\}$, the i_kth column is the same as the kth column of L^{-1}; all other entries are zero. Assuming, without loss of generality, that $i_1 = 1, \ldots, i_k = K$, we have

$$M^\dagger M = [L^{-1}\ 0] \begin{bmatrix} L \\ G \end{bmatrix} = L^{-1}L = I,$$

where $I \in \Re^{K \times K}$ is the identity matrix and G represents the remaining rows of M. Hence, $M^\dagger$ is a left inverse of M.

Our algorithm proceeds as follows. We start by selecting a set of K states, $i_1, \ldots, i_K$, and an initial parameter vector $W(0)$. Then, defining T' as $M^\dagger \circ T \circ M$, successive parameter vectors are generated using the following update rule:

$$W(t+1) = T'(W(t)) \tag{16}$$

7.2. *Computational Considerations*

We will prove shortly that the operation T' applied during each iteration of our algorithm is a contraction in the parameter space. Thus, the difference between an intermediate parameter vector $W(t)$ and the limit W^* decays exponentially with the time index t. Hence, in practice, the number of iterations required should be reasonable.[1]

The reason for using a compact representation is to alleviate the computational time and space requirements of dynamic programming, which traditionally employs an exhaustive look-up table, storing one value per state. Even when the parameter vector is small and the approximate value iteration algorithm requires few iterations, the algorithm would be impractical if the computation of T' required time or memory proportional to the number of states. Let us determine the conditions under which T' can be computed in time polynomial in the number of parameters K rather than the number of states n.

The operator T' is defined by

$$T'(W) = M^{\dagger}T(MW).$$

Since $M^{\dagger}$ only has K nonzero columns, only K components of $T(MW)$ must be computed: we only need to compute $T_i(MW)$ for $i = i_1, \ldots, i_k$. Each iteration of our algorithm thus takes time $O(K^2 t_T)$ where t_T is the time taken to compute $T_i(MW)$ for a given state i. For any state i, $T_i(MW)$ takes on the form

$$T_i(MW) = \min_{u \in U(i)} \Big(E[c_{iu}] + \sum_{j \in S} p_{ij}(u) W^T F(i) \Big).$$

The amount of time required to compute $\sum_{j \in S} p_{ij}(u) W^T F(i)$ is $O(N_s K)$, where N_s is the maximum number of possible successor states under any control action (i.e., states j such that $p_{ij}(u) > 0$). By considering all possible actions $u \in U(i)$ in order to perform the required minimization, $T_i(MW)$ can be computed in time $O(N_u N_s K)$ where N_u is maximum number of control actions allowed at any state. The computation of T' thus takes time $O(N_u N_s K^3)$.

Note that for many control problems of practical interest, the number of control actions allowed at a state and the number of possible successor states grow exponentially with the number of state variables. For problems in which the number of possible successor states grows exponentially, methods involving Monte-Carlo simulations may be coupled with our algorithm to reduce the computational complexity to a manageable level. We do not discuss such methods in this paper since we choose to concentrate on the issue of compact representations. For problems in which the number of control actions grows exponentially, on the other hand, there is no satisfactory solution, except to limit choices to a small subset of allowed actions (perhaps by disregarding actions that seem "bad" *a priori*). In summary, our algorithm is suitable for problems with large state spaces and can be modified to handle cases where an action taken at a state can potentially lead to any of a large number of successor states, but the algorithm is not geared to solve problems where an extremely large number of control actions is allowed.

7.3. *Convergence Theorem*

Let us now proceed with our convergence result for value iteration with linear architectures.[5]

THEOREM 2 *Let Assumption 2 hold.*
(a) There exists a vector $W^* \in \Re^K$ *such that* $W(t)$ *converges to* W^*.
(b) T' *is a contraction, with contraction coefficient* β', *with respect to a norm* $\|\cdot\|$ *on* $\Re^K$ *defined by*

$$\|W\| = \|MW\|_\infty.$$

Let V^* *be the optimal cost-to-go vector, and define* ϵ *by letting*

$$\epsilon = \inf_{W \in \Re^K} \|V^* - \tilde{V}(W)\|_\infty,$$

where V^* *is the optimal cost-to-go vector. Recall that* $\pi_{\tilde{V}(W^*)}$ *denotes a greedy policy with respect to cost-to-go vector* $\tilde{V}(W^*)$, *i.e.,*

$$\pi_{\tilde{V}(W^*)}(i) = \arg\min_{u \in U(i)} \Big(E[c_{iu}] + \beta \sum_{j \in S} p_{ij}(u) \tilde{V}_j(W^*) \Big).$$

The following hold:
(c)

$$\|V^* - \tilde{V}(W^*)\|_\infty \leq \frac{\beta + \beta'}{\beta(1 - \beta')} \epsilon,$$

(d)

$$\|V^{\pi_{\tilde{V}(W^*)}} - V^*\|_\infty \leq \frac{2(\beta + \beta')}{(1 - \beta)(1 - \beta')} \epsilon,$$

(e) there exists an example for which the bounds of (c) and (d) hold with equality.

This result is analogous to Theorem 1. The algorithm is guaranteed to converge and, when the compact representation can perfectly represent the optimal cost-to-go vector, the algorithm converges to it. Furthermore, the accuracy of approximations generated by the algorithm decays gracefully as the propriety of the compact representation diminishes. The proof of this Theorem involves a straightforward application of Theorem 3 concerning multi-representation contractions, which is presented in Appendix D.

Theorem 2 provides some assurance of reasonable behavior when feature-based linear architectures are used for dynamic programming. However, the theorem requires that the chosen representation satisfies Assumption 2, which seems very restrictive. In the next two sections, we discuss two types of compact representations that satisfy Assumption 2 and may be of practical use.

8. Example: Interpolative Representations

One possible compact representation can be produced by specifying values of K states in the state space, and taking weighted averages of these K values to obtain values of

other states. This approach is most natural when the state space is a grid of points in a Euclidean space. Then, if cost-to-go values at states sparsely distributed in the grid are computed, values at other points can be generated via interpolation. Other than the case where the states occupy a Euclidean space, interpolation-based representations may be used in settings where there seems to be a small number of "prototypical" states that capture key features. Then, if cost-to-go values are computed for these states, cost-to-go values at other states can be generated as weighted averages of cost-to-go values at the "prototypical" states.

For a more formal presentation of interpolation-based representations, let $S = \{1, \ldots, n\}$ be the states in the original state space and let $i_1, \ldots, i_K \in S$ be the states for which values are specified. The kth component of the parameter vector $W \in \Re^K$ stores the cost-to-go value of state i_k. We are then dealing with the representation

$$\tilde{V}_i(W) = \begin{cases} W_i, & \text{if } i \in \{i_1, \ldots, i_K\}, \\ W^T F(i), & \text{otherwise,} \end{cases} \tag{17}$$

where $F(i) \in \Re^K$ is a vector used to interpolate at state i. For any $i \in S$, the vector $F(i)$ is fixed; it is a part of the interpolation architecture, as opposed to the parameters W which are to be adjusted by an algorithm. The choice of the components $f_k(i)$ of $F(i)$ is generally based on problem-specific considerations. For the representations we consider in this section, we require that each component $f_k(i)$ of $F(i)$ be nonnegative and $\sum_{k=1}^{K} f_k(i) = 1$ for any state i.

In relation to feature–based methods, we could view the vector $F(i)$ as the feature vector associated with state i. To bring Equation (17) into a uniform format, let us define vectors $\{F(i_1), \ldots, F(i_K)\}$ as the usual basis vectors of $\Re^m$ so that we have

$$\tilde{V}_i(W) = W^T F(i), \quad \forall i \in S.$$

To apply the algorithm from Section 6, we should show that Assumption 2 of Theorem 2 is satisfied. Assumption 2(a) is satisfied by the fact that $F(i_1), \ldots, F(i_K)$ are the basis vectors of $\Re^m$. This fact also implies that $F(i) = \sum_{k=1}^{K} \theta_k(i) F(i_k)$ for $\theta_k(i) = f_k(i)$. Since the components of $F(i)$ sum to one, Assumption 2(b) is satisfied with $\beta' = \beta$. Hence, this interpolative representation is compatible with the algorithm of Section 6.

9. Example: Localized Basis Functions

Compact representations consisting of linear combinations of localized basis functions have attracted considerable interest as general architectures for function approximation. Two examples are radial basis function (Poggio and Girosi, 1990) and wavelet networks (Bakshi and Stephanopoulos, 1993). With these representations, states are typically contained in a Euclidean space $\Re^d$ (typically forming a finite grid). Let us continue to view the state space as $S = \{1, \ldots, n\}$. Each state index is associated with a point $x^i \in \Re^d$. With a localized basis function architecture, the cost-to-go value of state $i \in S$ takes on the following form:

$$\tilde{V}_i(W) = \sum_{k=1}^{K} W_k \phi(x^i, \mu_k, \sigma_k), \tag{18}$$

where $W \subset \Re^d$ is the parameter vector, and the function $\phi : \Re^d \times \Re^d \times \Re \mapsto \Re$ is the chosen basis function. In the case of radial basis functions, for instance, ϕ is a Gaussian, and the second and third arguments, $\mu_k \in \Re^d$ and $\sigma_k \in \Re$, specify the center and dilation, respectively. More formally,

$$\phi(x, \mu, \sigma) = ae^{-\frac{\|x-\mu\|_2^2}{2\sigma^2}}, \qquad \forall x, \mu \in \Re^d, \sigma \in \Re,$$

where $\|\cdot\|_2$ denotes the Euclidean norm and $a \in \Re$ is a normalization factor. Without loss of generality, we assume that the height at the center of each basis function is normalized to one. In the case of radial basis functions, this means $a = 1$. For convenience, we will assume that $\mu_k = x^{i_k}$ for $k \in \{1, \ldots, K\}$, where $i_1, \ldots, i_K$ are preselected states in S. In other words, each basis function is centered at a point that corresponds to some state.

Architectures employing localized basis functions are set apart from other compact representations by the tendency for individual basis functions to capture only local characteristics of the function to be approximated. This is a consequence of the fact that $\phi(x, \mu, \sigma)$ generates a significant value only when x is close to μ. Otherwise, $\phi(x, \mu, \sigma)$ is extremely small. Intuitively, each basis function captures a feature that is local in Euclidean space. More formally, we use locality to imply that a basis function, ϕ, has maximum magnitude at the center, so $\phi(\mu, \mu, \sigma) = 1$ while $|\phi(x, \mu, \sigma)| < 1$ for $x \neq \mu$. Furthermore, $|\phi(x, \mu, \sigma)|$ generally decreases as $\|x - \mu\|_2$ increases, and the dilation parameter controls the rate of this decrease. Hence, as the dilation parameter is decreased, $|\phi(x, \mu, \sigma)|$ becomes increasingly localized, and formally, for all $x \neq \mu$, we have

$$\lim_{\sigma \to 0} \phi(x, \mu, \sigma) = 0.$$

In general, when a localized basis function architecture is used for function approximation, the centers and dilations are determined via some heuristic method which employs data and any understanding about the problem at hand. Then, the parameter vector W is determined, usually via solving a least-squares problem. In this section, we explore the use of localized basis functions to solve dynamic programming, rather than function approximation, problems. In particular, we show that, under certain assumptions, the algorithm of Section 6 may be used to generate parameters for approximation of a cost-to-go function.

To bring localized basis functions into our feature-based representation framework, we can view an individual basis function, with specified center and dilation parameter, as a feature. Then, given a basis function architecture which linearly combines K basis functions, we can define

$$f_k(i) = \phi(x^i, \mu_k, \sigma_k), \qquad \forall i \in S,$$

and a feature mapping $F(i) = (f_1(i), \ldots f_K(i))$. The architecture becomes a special case of the familiar feature-based representation from Section 6.

$$\tilde{V}(W) = g(F(i), W) = \sum_{k=1}^{K} W_k f_k(i), \qquad \forall i \in S. \tag{19}$$

We now move on to show how the algorithm introduced in Section 6 may be applied in conjunction with localized basis function architectures. To do this, we will provide conditions on the architecture that are sufficient to ensure satisfaction of Assumption 2. The following formal assumption summarizes the sufficient condition we present.

Assumption 3 *(a) For all* $k \in \{1, \ldots, K\}$,

$$f_k(i_k) = 1.$$

(b) For all $j \in \{1, \ldots, K\}$,

$$\sum_{k \neq j} |f_k(i_j)| < 1.$$

(c) With δ *defined by*

$$\delta = \max_{j \in \{1, \ldots, K\}} \sum_{k \neq j} |f_k(i_j)|,$$

there exists a $\beta' \in [\beta, 1)$ *such that, for all* $i \in S$,

$$\sum_{k=1}^{K} |f_k(i)| \leq \frac{\beta'}{\beta}(1 - \delta).$$

Intuitively, δ is a bound on the influence of other basis functions on the cost-to-go value at the center of a particular basis function. By decreasing the dilation parameters of the basis functions, we can make δ arbitrarily small. Combined with the fact that $\max_{i \in S} \sum_{k=1}^{K} F(i)$ approaches unity as the dilation parameter diminishes, this implies that we can ensure satisfaction of Assumption 3 by choosing sufficiently small dilation parameters. In practice, a reasonable size dilation parameter may be desirable, and Assumption 3 may often be overly restrictive.

We will show that Assumption 3 guarantees satisfaction of Assumption 2 of Theorem 2. This will imply that, under the given restrictions, localized basis function architectures are compatible with the algorithm of Section 6. We start by choosing the states $\{i_1, \ldots, i_K\}$ to be those corresponding to node centers. Hence, we have $x^{i_k} = \mu_k$ for all k.

Define $B \in \Re^{K \times K}$ as a matrix whose kth column is the feature vector $F(i_k)$. Define $\|\cdot\|_1 : \Re^K \mapsto \Re$ as the l_1 norm on $\Re^K$. Suppose we choose a vector $\theta \in \Re^K$ with $\|\theta\|_1 = 1$. Using Assumptions 3(a) and 3(b), we obtain

$$\begin{aligned} \|B\theta\|_1 &= \left\| \sum_{j=1}^{K} \theta_j F(i_j) \right\|_1 \\ &\geq \sum_{k=1}^{K} \left| \sum_{j=1}^{K} \theta_j f_k(i_j) \right| \end{aligned}$$

$$\begin{aligned}
&\geq \sum_{k=1}^{K} \Big(|\theta_k| - \sum_{j \neq k} |\theta_j| |f_k(i_j)| \Big) \\
&= 1 - \sum_{j=1}^{K} |\theta_j| \sum_{k \neq j} |f_k(i_j)| \\
&\geq 1 - \sum_{j=1}^{K} |\theta_j| \delta \\
&= 1 - \delta \\
&> 0.
\end{aligned}$$

Hence, B is nonsingular. It follows that the columns of B, which are the vectors $F(i_1), \ldots, F(i_K)$, are linearly independent. Thus, Assumption 2(a) is satisfied.

We now place an upper bound on $\|B^{-1}\|_1$, the l_1–induced norm on B^{-1}.

$$\begin{aligned}
\|B^{-1}\|_1 &= \max_{x \in \Re^K} \frac{\|B^{-1}x\|_1}{\|x\|_1} \\
&= \min_{\theta \in \Re^K} \frac{\|\theta\|_1}{\|B\theta\|_1} \\
&< \frac{1}{1-\delta}
\end{aligned}$$

Let us define $\theta(i) = B^{-1}F(i)$ so that

$$F(i) = \sum_{k=1}^{K} \theta_k(i) F(i_k).$$

For any i, we can put a bound on $\|\theta(i)\|_1$ as follows:

$$\begin{aligned}
\|\theta(i)\|_1 &= \|B^{-1}F(i)\|_1 \\
&\leq \|B^{-1}\|_1 \|F(i)\|_1 \\
&\leq \frac{\|F(i)\|_1}{1-\delta} \\
&\leq \frac{\beta'}{\beta}.
\end{aligned}$$

Hence, Assumption 2(b) is satisfied. It follows that the algorithm of Section 6 may be applied to localized basis function architectures that satisfy Assumption 5.

10. Conclusion

We have proved convergence and derived error bounds for two algorithms that employ feature–based compact representations to approximate cost–to–go functions. The use of

compact representations can potentially lead to the solution of many stochastic control problems that are computationally intractable to classical dynamic programming.

The algorithms described in this paper rely on the use of features that summarize the most salient characteristics of a state. These features are typically hand-crafted using available experience and intuition about the underlying Markov decision problem. If appropriate features are chosen, the algorithms lead to good solutions. When it is not clear what features are appropriate, several choices may be tried in order to arrive at a set of features that enables satisfactory performance. However, there is always a possibility that a far superior choice of features exists but has not been considered.

The approximation architectures we have considered are particularly simple. More complex architectures such as polynomials or artificial neural networks may lead to better approximations. Unfortunately, the algorithms discussed are not compatible with such architectures. The development of algorithms that guarantee sound behavior when used with more complex architectures is an area of open research.

Acknowledgments

The use of feature extraction to aggregate states for dynamic programming was inspired by Dimitri Bertsekas. We thank Rich Sutton for clarifying the relationship of the counter-example of Section 4 with TD(0). The choice of Tetris as a test-bed was motivated by earlier developments of Tetris learning algorithms by Richard Lippman and Linda Kukolich at Lincoln Laboratories. Early versions of this paper benefited from the proofreading of Michael Branicky and Peter Marbach. This research was supported by the NSF under grant ECS 9216531 and by EPRI under contract 8030-10.

Appendix A

Multi-Representation Contractions

Many problems requiring numerical computation can be cast in the abstract framework of fixed point computation. Such computation aims at finding a fixed point $V^* \in \Re^n$ of a mapping $T : \Re^n \rightarrow \Re^n$; that is, solving the equation $V = T(V)$. One typical approach involves generating a sequence $\{V(t)|t = 0, 1, 2, ...\}$ using the update rule $V(t+1) = T(V(t))$ with the hope that the sequence will converge to V^*. In the context of dynamic programming, the function T could be the value iteration operator, and the fixed point is the optimal cost-to-go vector.

In this appendix, we deal with a simple scenario where the function T is a contraction mapping – that is, for some vector norm $\|\cdot\|$, we have $\|T(V) - T(V')\| \leq \beta\|V - V'\|$ for all $V, V' \in \Re^n$ and some $\beta \in [0, 1)$. Under this assumption, the fixed point of T is unique, and a proof of convergence for the iterative method is trivial.

When the number of components n is extremely large (n often grows exponentially with the number of variables involved in a problem), the computation of T is inherently slow. One potential way to accelerate the computation is to map the problem onto a smaller space $\Re^m$ ($m \ll n$), which can be thought of as a parameter space. This can be done by

defining a mapping $\tilde{V} : \Re^m \mapsto \Re^n$ and a pseudo-inverse $\tilde{V}^\dagger : \Re^n \mapsto \Re^m$. The mapping $\tilde{V}$ can be thought of as a compact representation. A solution can be approximated by finding the fixed point of a mapping $T' : \Re^m \mapsto \Re^m$ defined by $T' = \tilde{V}^\dagger \circ T \circ \tilde{V}$. The hope is that $\tilde{V}(W^*)$ is close to a fixed point of T if W^* is a fixed point of T'. Ideally, if the compact representation can exactly represent a fixed point $V^* \in \Re^n$ of T – that is, if there exists a $W \in \Re^m$ such that $\tilde{V}(W) = V^*$ – then W should be a fixed point of T'. Furthermore, if the compact representation cannot exactly, but can closely, represent the fixed point $V^* \in \Re^n$ of T then W^* should be close to V^*. Clearly, choosing a mapping $\tilde{V}$ for which $\tilde{V}(W)$ may closely approximate fixed points of T requires some intuition about where fixed points should generally be found in $\Re^n$.

A.1. Formal Framework

Though the theorem we will prove generalizes to arbitrary metric spaces, to promote readability, we only treat normed vector spaces. We are given the mappings $T : \Re^n \mapsto \Re^n$, $\tilde{V} : \Re^m \mapsto \Re^n$, and $\tilde{V}^\dagger : \Re^n \mapsto \Re^m$. We employ a vector norm on $\Re^n$ and a vector norm on $\Re^m$, denoting both by $\|\cdot\|$. We have $m < n$, so the norm being used in a particular expression can be determined by the dimension of the argument. Define a mapping $T' : \Re^m \mapsto \Re^m$ by $T' = \tilde{V}^\dagger \circ T \circ \tilde{V}$. We make two sets of assumptions. The first concerns the mapping T.

Assumption 4 *The mapping T is a contraction with contraction coefficient $\beta \in [0, 1)$ with respect to $\|\cdot\|$. Hence, for all $V, V' \in \Re^n$,*

$$\|T(V) - T(V')\| \leq \beta \|V - V'\|.$$

Our second assumption defines the relationships between $\tilde{V}$ and $\tilde{V}^\dagger$.

Assumption 5 *The following hold for the mappings $\tilde{V}$ and $\tilde{V}^\dagger$:*

(a) For all $W \in \Re^m$,

$$W = \tilde{V}^\dagger(\tilde{V}(W)).$$

(b) There exists a $\beta' \in [\beta, 1)$ such that, for all $W, W' \in \Re^m$,

$$\|\tilde{V}(W) - \tilde{V}(W')\| \leq \frac{\beta'}{\beta} \|W - W'\|.$$

(c) For all $V, V' \in \Re^n$,

$$\|\tilde{V}^\dagger(V) - \tilde{V}^\dagger(V')\| \leq \|V - V'\|.$$

Intuitively, part (a) ensures that $\tilde{V}^\dagger$ is a pseudo-inverse of $\tilde{V}$. Part (b) forces points that are close in $\Re^m$ to map to points that are close in $\Re^n$, and part (c) ensures the converse, nearby points in $\Re^n$ must project onto points in $\Re^m$ that are close.

A.2. Theorem and Proof

Since T is a contraction mapping, it has a unique fixed point V^*. Let

$$\epsilon = \inf_{W \in \Re^m} \|V^* - \tilde{V}(W)\|.$$

THEOREM 3 *Let Assumptions 4 and 5 hold.*
(a) We have

$$\|T'(W) - T'(W')\| \leq \beta' \|W - W'\|,$$

for all $W, W' \in \Re^m$.
(b) If W^* *is the fixed point of* T', *then*

$$\|V^* - \tilde{V}(W^*)\| \leq \frac{\beta + \beta'}{\beta(1 - \beta')}\epsilon.$$

This theorem basically shows that T' is a contraction mapping, and if V^* can be closely approximated by the compact representation then W^* provides a close representation of V^*.

Proof of Theorem 3 (a) Take arbitrary $W, W' \in \Re^m$. Then,

$$\begin{aligned} \|T'(W) - T'(W')\| &= \|\tilde{V}^\dagger(T(\tilde{V}(W))) - \tilde{V}^\dagger(T(\tilde{V}(W')))\| \\ &< \|T(\tilde{V}(W)) - T(\tilde{V}(W'))\| \\ &\leq \beta \|\tilde{V}(W) - \tilde{V}(W')\| \\ &\leq \beta' \|W - W'\|. \end{aligned}$$

Hence, T' is a contraction mapping with contraction coefficient β'.
(b) Let $\epsilon' = \epsilon + \delta$ for some $\delta > 0$. Choose $W_{opt} \in \Re^m$ such that $\|V^* - \tilde{V}(W_{opt})\| < \epsilon'$. Then,

$$\begin{aligned} \|W_{opt} - T'(W_{opt})\| &= \|\tilde{V}^\dagger(\tilde{V}(W_{opt})) - \tilde{V}^\dagger(T(\tilde{V}(W_{opt})))\| \\ &\leq \|\tilde{V}(W_{opt}) - T(\tilde{V}(W_{opt}))\| \\ &\leq \|\tilde{V}(W_{opt}) - V^*\| + \|T(\tilde{V}(W_{opt})) - V^*\| \\ &< \epsilon' + \beta\epsilon' \\ &= (1 + \beta)\epsilon'. \end{aligned}$$

Now we can place a bound on $\|W^* - W_{opt}\|$:

$$\begin{aligned} \|W^* - W_{opt}\| &\leq \|W^* - T'(W_{opt})\| + \|T'(W_{opt}) - W_{opt}\| \\ &< \beta' \|W^* - W_{opt}\| + (1 - \beta)\epsilon', \end{aligned}$$

and it follows that

$$\|W^* - W_{opt}\| < \frac{1 + \beta}{1 - \beta'}\epsilon'.$$

Next, a bound can be put on $\|V^* - \tilde{V}(W^*)\|$:

$$\begin{aligned}\|V^* - \tilde{V}(W^*)\| &\leq \|V^* - \tilde{V}(W_{opt})\| + \|\tilde{V}(W_{opt}) - \tilde{V}(W^*)\| \\ &< \epsilon' + \frac{\beta'}{\beta}\|W_{opt} - W^*\| \\ &< \epsilon' + \frac{\beta'}{\beta}\left(\frac{1+\beta}{1-\beta'}\right)\epsilon' \\ &= \frac{\beta+\beta'}{\beta(1-\beta')}\epsilon'.\end{aligned}$$

Since δ can be arbitrarily small, the proof is complete. □

Appendix B

Asynchronous Stochastic Approximation

Consider an algorithm that performs noisy updates of a vector $V \in \Re^n$, for the purpose of solving a system of equations of the form $T(V) = V$. Here T is a mapping from $\Re^n$ into itself. Let $T_1, \ldots, T_n : \Re^n \mapsto \Re$ be the corresponding component mappings; that is, $T(V) = (T_1(V), \ldots, T_n(V))$ for all $V \in \Re^n$.

Let $\mathcal{N}$ be the set of nonnegative integers, let $V(t)$ be the value of the vector V at time t, and let $V_i(t)$ denote its ith component. Let Γ^i be an infinite subset of $\mathcal{N}$ indicating the set of times at which an update of V_i is performed. We assume that

$$V_i(t+1) = V_i(t), \qquad t \notin \Gamma^i. \tag{B.1}$$

and

$$V_i(t+1) = V_i(t) + \alpha_i(t)\Big(T_i(V(t)) - V_i(t) + \eta_i(t)\Big), \qquad t \in \Gamma^i. \tag{B.2}$$

Here, $\alpha_i(t)$ is a stepsize parameter between 0 and 1, and $\eta_i(t)$ is a noise term. In order to bring Equations (B.1) and (B.2) into a unified form, it is convenient to assume that $\alpha_i(t)$ and $\eta_i(t)$ are defined for every i and t, but that $\alpha_i(t) = 0$ for $t \notin \Gamma^i$.

Let $\mathcal{F}(t)$ be the set of all random variables that have been realized up to and including the point at which the stepsizes $\alpha_i(t)$ for the tth iteration are selected, but just before the noise term $\eta_i(t)$ is generated. As in Section 2, $\|\cdot\|_\infty$ denotes the maximum norm. The following assumption concerns the statistics of the noise.

Assumption 6 *(a) For every i and t, we have $E[\eta_i(t) \mid \mathcal{F}(t)] = 0$.*
(b) There exist (deterministic) constants A and B such that

$$E[\eta_i^2(t) \mid \mathcal{F}(t)] < A + B\|V(t)\|_\infty^2, \qquad \forall i, t.$$

We then have the following result (Tsitsiklis, 1994) (related results are obtained in (Jaakola, Jordan, and Singh, 1994)).

THEOREM 4 *Let Assumption 6 and Assumption 1 of Section 5 on the stepsizes $\alpha_i(t)$ hold and suppose that the mapping T is a contraction with respect to the maximum norm. Then, $V(t)$ converges to the unique fixed point V^* of T, with probability 1.*

Appendix C

Proof of Theorem 1

(a) To prove this result, we will bring the aggregated state value iteration algorithm into a form to which Theorems 3 and 4 (from the Appendices A and B) can be applied and we will then verify that the assumptions of these theorems are satisfied.

Let us begin by defining a function $T' : \Re^m \mapsto \Re^m$, which in some sense is a noise-free version of our update procedure on $W(t)$. In particular, the jth component T'_j of T' is defined by

$$T'_j(W) = E\Big[T_{X_j}(\tilde{V}(W))\Big] = \sum_{i \in S_j} p^j(i) T_i(\tilde{V}(W)). \tag{C.1}$$

The update equations (7) and (8) can be then rewritten as

$$W_j(t+1) = (1 - \alpha_j(t))W_j(t) + \alpha_j(t)(T'_j(W) + \eta_j(t)). \tag{C.2}$$

where the random variable $\eta_j(t)$ is defined by

$$\eta_j(t) = T_{X_j(t)}(\tilde{V}(W(t))) - E\Big[T_{X_j(t)}(\tilde{V}(W(t)))\Big].$$

Given that each $X_j(t)$ is a random sample from S_j whose distribution is independent of $\mathcal{F}(t)$ we obtain

$$E[\eta_j(t) \mid \mathcal{F}(t)] = E[\eta_j(t)] = 0.$$

Our proof consists of two parts. First, we use Theorem 3 to establish that T' is a maximum norm contraction. Once this is done, the desired convergence result follows from Theorem 4.

Let us verify the assumptions required by Theorem 3. First, let us define a function $\tilde{V}^\dagger : \Re^n \to \Re^m$ as

$$\tilde{V}^\dagger_j(V) = \sum_{i \in S_j} p^j(i) V_i.$$

This function is a pseudo-inverse of $\tilde{V}$ since, for any $W \in \Re^M$,

$$\tilde{V}^\dagger_j(\tilde{V}(W)) = \sum_{i \in S_j} p^j(i) \tilde{V}_i(W) = W_j.$$

We can express T' as $T' = \tilde{V}^\dagger \circ T \circ \tilde{V}$, to bring it into the form of Theorem 3. In this context, the vector norm we have in mind for both $\Re^n$ and $\Re^m$ is the maximum norm, $\|\cdot\|_\infty$.

Assumption 4 is satisfied since T is a contraction mapping. We will now show that $\tilde{V}^{\dagger}$, $\tilde{V}$, and T, satisfy Assumption 5. Assumption 5(a) is satisfied since $\tilde{V}^{\dagger}$ is a pseudo-inverse of $\tilde{V}$. Assumption 5(b) is satisfied with $\beta' = \beta$ since

$$\begin{aligned}\|\tilde{V}(W) - \tilde{V}(W')\|_{\infty} &= \max_{i \in S} \left|\tilde{V}_i(W) - \tilde{V}_i(W')\right| \\ &= \max_{j \in \{1,\ldots,m\}} |W_j - W'_j| \\ &= \|W - W'\|_{\infty}.\end{aligned}$$

Assumption 5(c) is satisfied because

$$\begin{aligned}\|\tilde{V}^{\dagger}(V) - \tilde{V}^{\dagger}(V')\|_{\infty} &= \max_{j \in \{1,\ldots,m\}} \left|\sum_{i \in S_j} p^j(i)(V_i - V'_i)\right| \\ &\leq \max_{j \in \{1,\ldots,m\}} \max_{i \in S_j} |V_i - V'_i| \\ &= \|V - V'\|_{\infty}.\end{aligned}$$

Hence, Theorem 3 applies, and T' must be a maximum norm contraction with contraction coefficient β.

Since T' is a maximum norm contraction, Theorem 4 now applies as long as its assumptions hold. We have already shown that

$$E[\eta_j(t) \mid \mathcal{F}(t)] = 0,$$

so Assumption 6(a) is satisfied. As for Assumption 6(b) on the variance of $\eta_j(t)$, the conditional variance of our noise term satisfies

$$\begin{aligned}E[\eta_j^2(t) \mid \mathcal{F}(t)] &= E\left[\left(T_{X_j(t)}(\tilde{V}(W(t))) - E[T_{X_j(t)}(\tilde{V}(W(t)))]\right)^2\right] \\ &\leq 4\left(\max_{i \in S} T_i(\tilde{V}(W(t)))\right)^2 \\ &\leq 8 \max_{i,u}(E[c_{iu}])^2 + 8\|W(t)\|_{\infty}^2.\end{aligned}$$

Hence, Theorem 4 applies and our proof is complete.

(b) If the maximum fluctuation of V^* within a particular partition is e_i then the minimum error that can be attained using a single constant to approximate the cost-to-go of every state within the partition is $\frac{e_i}{2}$. This implies that $\min_W \|\tilde{V}(W) - V^*|_{\infty}$, the minimum error that can be attained by an aggregated state representation, is $\frac{\|e\|_{\infty}}{2}$. Hence, by substituting ϵ with $\frac{\|e\|_{\infty}}{2}$, and recalling that we have $\beta' = \beta$, the result follows from Theorem 3(b).

(c) Now that we have a bound on the maximum norm between the optimal cost-to-go estimate and the true optimal cost-to-go vector, we can place a bound on the maximum norm between the cost of a greedy policy with respect to $\tilde{V}(W^*)$ and the optimal policy as follows:

$$|V^{\pi_{\tilde{V}(W^*)}} - V^*|_{\infty} \leq |V^{\pi_{\tilde{V}(W^*)}} - T(\tilde{V}(W^*))\|_{\infty} + |T(\tilde{V}(W^*)) - V^*\|_{\infty}.$$

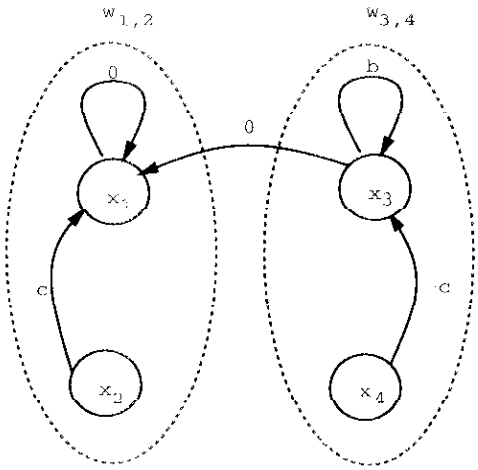

Figure C.1. An example for which the bound holds with equality.

Since $T(V) = T^{\pi_V}(V)$ for all $V \in \Re^n$, we have

$$\begin{aligned}\|V^{\pi_{\tilde{V}}(W^*)} - V^*\|_\infty &\leq \|V^{\pi_{\tilde{V}}(W^*)} - T^{\pi_{\tilde{V}}(W^*)}(\tilde{V}(W^*))\|_\infty + \|T(\tilde{V}(W^*)) - V^*\|_\infty \\ &\leq \beta\|V^{\pi_{\tilde{V}}(W^*)} - \tilde{V}(W^*)\|_\infty + \beta\|\tilde{V}(W^*) - V^*\|_\infty \\ &\leq \beta\|V^{\pi_{\tilde{V}}(W^*)} - V^*\|_\infty(1+\beta)\|V^* - \tilde{V}(W^*)\|_\infty.\end{aligned}$$

It follows that

$$\begin{aligned}\|V^{\pi_{\tilde{V}}(W^*)} - V^*\|_\infty &\leq \frac{2\beta}{1-\beta}\|\tilde{V}(W^*) - V^*\|_\infty \\ &\leq \frac{2\beta}{(1-\beta)^2}\|e\|_\infty.\end{aligned}$$

(d) Consider the four–state Markov decision problem shown in Figure C.1. The states are x_1, x_2, x_3, and x_4, and we form two partitions, the first consisting of x_1 and x_2, and the second containing the remaining two states. All transition probabilities are one. No control decisions are made at states x_1, x_2, or x_4. State x_1 is a zero-cost absorbing state. In state x_2 a transition to state x_1 is inevitable, and, likewise, when in state x_4, a transition to state x_3 always occurs. In state x_3 two actions are allowed: *move* and *stay*. The transition cost for each state–action pair is deterministic, and the arc labels in Figure C.1 represent the values. Let c be an arbitrary positive constant and, let b, the cost of staying in state x_3, be defined as $b = \frac{2\beta c - \delta}{1-\beta}$, with $\delta < 2\beta c$. Clearly, the optimal cost-to-go values at x_1, x_2, x_3, and x_4 are 0, c, 0, $-c$, respectively, and $\|e\| = c$.

Now, let us define sampling distributions, within each partition, that will be used with the algorithm. In the first partition, we always sample x_2 and in the second partition, we

always sample x_4. Consequently, the algorithm will converge to partition values $w^*_{1,2}$ and $w^*_{3,4}$ satisfying the following equations:

$$\begin{aligned} w^*_{1,2} &= c + \beta w^*_{1,2} \\ w^*_{3,4} &= -c + \beta w^*_{3,4}. \end{aligned}$$

It is not hard to see that the unique solution is

$$\begin{aligned} w^*_{1,2} &= \frac{c}{1-\beta} \\ w^*_{3,4} &= \frac{-c}{1-\beta}. \end{aligned}$$

The bound of part (b) is therefore satisfied with equality.

Consider the greedy policy with respect to w. For $\delta > 0$, the *stay* action is chosen at state x_3, and the total discounted cost incurred starting at state x_3 is $\frac{2\beta c - \delta}{(1-\beta)^2}$. When $\delta = 0$, both actions, *stay* and *move*, are legitimate choices. If *stay* is chosen, the bound of part (c) holds with equality. □

Appendix D

Proof of Theorem 2

(a) By defining $\tilde{V}^\dagger(V) = M^\dagger V$, we have $T' = \tilde{V}^\dagger \circ T \circ \tilde{V}$, which fits into the framework of multi-representation contractions. Our proof consists of a straightforward application of Theorem 3 from Appendix A (on multi-representation contractions). We must show that the technical assumptions of Theorem 3 are satisfied. To complete the multi-representation contraction framework, we must define a norm in our space of cost-to-go vectors and a norm in the parameter space. In this context, as a metric for parameter vectors, let us define a norm $\|\cdot\|$ by

$$\|W\| = \frac{\beta}{\beta'}\|MW\|_\infty.$$

Since M has full column rank, $\|\cdot\|$ has the standard properties of a vector norm. For cost-to-go vectors, we employ the maximum norm in $\Re^n$ as our metric.

We know that T is a maximum norm contraction, so Assumption 4 is satisfied. Assumption 5(a) is satisfied since, for all $W \in \Re^K$,

$$\begin{aligned} \tilde{V}_i^\dagger(\tilde{V}(W)) &= M^\dagger MW \\ &= W. \end{aligned}$$

Assumption 5(b) follows from our definition of $\|\cdot\|$ and the fact that $\beta' \in [\beta, 1)$:

$$\begin{aligned} \|W - W'\| &= \frac{\beta}{\beta'}\|MW - MW'\|_\infty \\ &= \frac{\beta}{\beta'}\|\tilde{V}(W) - \tilde{V}(W')\|_\infty \end{aligned}$$

Showing that Assumption 2 implies Assumption 5(c) is the heart of this proof. To do this, we must show that, for arbitrary cost-to-go vectors V and V',

$$\|V - V'\|_\infty \geq |V^\tau(V) - V^\tau(V')|. \tag{D.1}$$

Define $D = \frac{\beta}{\beta'} M(V^\dagger(V) - V^\dagger(V'))$. Then, for arbitrary $i \in S$ we have

$$|D_i| = \frac{\beta}{\beta'} |F^T(i)(V^\dagger(V) - V^\dagger(V'))|.$$

Under Assumption 2 there exist positive constants $\theta_1(i), \ldots, \theta_K(i) \in \Re$, with $\sum_{k=1}^K |\theta_k(i)| \leq 1$, such that $F(i) = \frac{\beta'}{\beta} \sum_{k=1}^K \theta_k(i) F(i_k)$. It follows that, for such $\theta_1(i), \ldots, \theta_K(i) \in \Re$,

$$\begin{aligned}
|D_i| &\leq \frac{\beta}{\beta'} |(\frac{\beta'}{\beta} \sum_{k=1}^K \theta_k(i) F^T(i_k))(V^\dagger(V) - V^\dagger(V'))| \\
&\leq \max_k |F^T(i_k)(V_i^\dagger(V) - V_i^\dagger(V'))| \\
&\leq |D_{i_k}| \\
&= |V_{i_k} - V'_{i_k}| \\
&\leq \|V - V'\|_\infty.
\end{aligned}$$

Inequality (D.1) follows. Hence, Theorem 3 applies, implying parts (a), (b), and (c), of Theorem 2.

Part (d) can be proven using the same argument as in the proof of Theorem 1(c). For part (e), we can use the same example as that used to prove part (d) of Theorem 1.

□

Notes

1. To those familiar with Q-learning or temporal-difference learning: the counter-example applies to cases where temporal-difference or Q-learning updates are performed at states that are sampled uniformly from the entire state space. Often times, however, temporal-difference methods assume that sample states are generated by following a randomly produced complete trajectory. In our example, this would correspond to starting at state x_1, moving to state x_2, and then doing an infinite number of self-transitions from state x_2 to itself. If this mechanism is used, our example is no longer divergent, in agreement with results of Dayan (1992).
2. We take the point of view that each of these samples is independently generated from the same probability distribution. If the samples were generated by a simulation experiment, as Monte-Carlo simulation under some fixed policy, independence would fail to hold. This would complicate somewhat the convergence analysis, but can be handled as in (Jaakola, Singh and Jordan, 1995).
3. The way in which state is recorded is inconsequential, so we have made no effort to minimize the number of vector components required.
4. To really ensure a reasonable order of growth for the number of required iterations, we would have to characterize a probability distribution for the difference between the initial parameter vector $W(0)$ and the goal W^* as well as how close to the goal W^* the parameter vector $W(t)$ must be in order for the error bounds to hold.
5. Related results have been obtained independently by Gordon (1995).

References

Bakshi, B. R. & Stephanopoulos G., (1993)."Wave-Net: A Multiresolution, Hierarchical Neural Network with Localized Learning," AIChE Journal, vol. 39, no. 1, pp. 57-81.

Barto, A. G., Bradtke, S. J., & Singh, S. P., (1995). "Real-time Learning and Control Using Asynchronous Dynamic Programming," Artificial Intelligence, vol. 72, pp. 81-138.

Bellman, R. E. & Dreyfus, S. E., (1959)."Functional Approximation and Dynamic Programming," Math. Tables and Other Aids Comp., Vol. 13, pp. 247-251.

Bertsekas, D. P., (1995).*Dynamic Programming and Optimal Control*, Athena Scientific, Bellmont, MA.

Bertsekas, D. P., (1994)."A Counter-Example to Temporal Differences Learning," Neural Computation, vol. 7, pp. 270-279.

Bertsekas D. P. & Castañon, D. A., (1989). "Adaptive Aggregation for Infinite Horizon Dynamic Programming," IEEE Transactions on Automatic Control, Vol. 34, No. 6, pp. 589-598.

Bertsekas, D. P. & Tsitsiklis, J. N., (1989). *Parallel and Distributed Computation: Numerical Methods*, Prentice Hall, Englewood Cliffs, NJ.

Dayan, P. D., (1992). "The Convergence of TD(λ) for General λ," Machine Learning, vol. 8, pp. 341-362.

Gordon, G. J., (1995)."Stable Function Approximation in Dynamic Programming," Technical Report: CMU-CS-95-103, Carnegie Mellon University.

Jaakola, T., Jordan M. I., & Singh, S. P., (1994)."On the Convergence of Stochastic Iterative Dynamic Programming Algorithms," Neural Computation, Vol. 6, No. 6.

Jaakola T., Singh, S. P., & Jordan, M. I., (1995). "Reinforcement Learning Algorithms for Partially Observable Markovian Decision Processes," in *Advances in Neural Information Processing Systems 7*, J. D. Cowan, G. Tesauro, and D. Touretzky, editors, Morgan Kaufmann.

Korf, R. E., (1987). "Planning as Search: A Quantitative Approach," Artificial Intelligence, vol. 33, pp. 65-88.

Lippman, R. P., Kukolich, L. & Singer, E., (1993). "LNKnet: Neural Network, Machine-Learning, and Statistical Software for Pattern Classification," The Lincoln Laboratory Journal, vol. 6, no. 2, pp. 249-268.

Morin, T. L., (1987). "Computational Advances in Dynamic Programming," in *Dynamic Programming and Its Applications*, edited by Puterman, M.L., pp. 53-90.

Poggio, T. & Girosi, F., (1990). "Networks for Approximation and Learning," Proceedings of the IEEE, vol. 78, no. 9, pp. 1481-1497.

Reetz, D., (1977). "Approximate Solutions of a Discounted Markovian Decision Process," Bonner Mathematische Schriften, vol. 98: Dynamische Optimierung, pp. 77-92.

Schweitzer, P. J., & Seidmann, A., (1985). "Generalized Polynomial Approximations in Markovian Decision Processes," Journal of Mathematical Analysis and Applications, vol. 110, pp. 568-582.

Sutton, R. S., (1988). "Learning to Predict by the Method of Temporal Differences," Machine Learning, vol. 3, pp. 9-44.

Tesauro, G., (1992). "Practical Issues in Temporal Difference Learning," Machine Learning, vol. 8, pp. 257-277.

Tsitsiklis, J. N., (1994). "Asynchronous Stochastic Approximation and Q-Learning," Machine Learning, vol. 16, pp. 185-202.

Watkins, C. J. C. H., Dayan, P., (1992). "Q-learning," Machine Learning, vol. 8, pp. 279-292.

Whitt, W., (1978). Approximations of Dynamic Programs I *Mathematics of Operations Research*, vol. 3, pp. 231-243.

Received Decmeber 2, 1994
Accepted March 29, 1995
Final Manuscript October 13, 1995