October, 1996 LIDS-P 2350

Research Supported By:

ARO
AASERT-DAAH04-93-GD169

Play Selection in Football: A Case Study in Neuro-Dynamic Programming

Patek, S.D.
Bertsekas, D.P.

Play Selection in Football: a Case Study in Neuro-Dynamic Programming[1]

Stephen D. Patek
Dimitri P. Bertsekas
Laboratory for Information and Decision Systems
Massachusetts Institute of Technology

Abstract

Using a version of the game of American football, this paper presents an extensive computational study of approximate dynamic programming methods. The problem of optimal play selection for the offensive team during a single offensive drive is cast as a stochastic shortest path problem and serves as a medium-scale testbed for algorithms inspired by policy iteration. Our methods are such that at each iteration a new policy is computed on the basis of an approximate evaluation of the policy's reward-to-go function. For a given policy, the approximate evaluation of reward-to-go is obtained either as the solution or as a step toward the solution of a least squares training problem defined by simulated state/reward data pairs. Numerical results for a variety of methods are presented. Because football is a medium-scale problem, a numerical solution can easily be computed, providing a yardstick for meaningful comparisons of the approximate methods.

The main purpose of this paper is to illustrate the application of neuro-dynamic programming methods in solving a concrete problem. Along the way, however, we are able to contrast and compare the methodologies both in terms of performance and complexity of implementation. Finally, our numerical results nicely illustrate some the interesting properties of the algorithms that are becoming recognized as "typical behavior" by various researchers in the learning community.

1 Introduction

We present a case study of a *dynamic optimization* problem where rewards (to be maximized) accumulate in stages as an underlying system transitions from state to state. Control actions for the system are applied at every stage. At a given state, each control action has a corresponding probability distribution which governs the transition to a successor state. The choice of a control at a state also determines a probability distribution for the amount of reward to be earned that stage. The objective is to determine a *policy* (a mapping from states to control actions) which maximizes the expected value of the reward accumulated over time.

The classical framework for solving problems of this type is that of *dynamic programming*, including both the *value iteration* and *policy iteration* algorithms. In this paper we are primarily concerned with methods that relate to policy iteration. In policy iteration, an optimal solution is computed through a sequence of policy evaluations and policy updates (improvements). Each policy evaluation amounts to computing the expected long-term reward (reward-to-go) from each state of the system. Each policy update involves computing the action at each state which is "greedy" with respect to the expected long-term reward of the alternative actions. Unfortunately, due to the "curse of dimensionality", the steps of policy iteration are computationally infeasible for most realistic, large-scale engineering problems. In this sense, policy iteration is really only a conceptual algorithm, not of practical interest.

In recent years, Approximate Policy Iteration (API), has been suggested as a practical approach to solving realistic dynamic optimization problems. In this framework, *approximations* of the reward-to-go function for a fixed policy are trained through simulation and least squares optimization, and policy updates are computed based upon these approximations. Mathematically, the approximations take on a fixed parametric form, known as an *architecture*.

API is one out of several classes of algorithms which comprise the methods of Neuro-Dynamic Programming (NDP). As with API, the methods of NDP generally relate to the classical methods of dynamic

[1] Supported by the US Army Research Office AASERT-DAAH04-93-GD169

programming. The "Neuro-" prefix is attached to indicate the use of (usually neural-network) approximations for reward-to-go. What distinguishes NDP from other forms of approximate dynamic programming is its heavy reliance on simulation as a means for obtaining training data for reward-to-go approximations. In other "machine learning" research communities, NDP is known as Reinforcement Learning (RL).

As an alternative to API (but still within the context of NDP) we consider a related class of algorithms known collectively as Optimistic Policy Iteration (OPI). Before discussing OPI further, note that in API an honest attempt is made to obtain an approximation of the reward-to-go function which is accurate everywhere in the state space. This usually means that a large number of sample trajectories are generated in order to have a representative set of training data. This data is often buffered and is then presented many times (by cycling through the data buffer) to the training algorithms so that each distinct sample state/reward pair can have a significant impact on the outcome of training. In OPI, on the other hand, a relatively small number of sample trajectories are generated, and the data is allowed to impact the reward-to-go approximation in only a very limited fashion before a new policy is computed. (The user of this type of algorithm is *optimistic* about the effectiveness of the data in describing the reward-to-go function.) OPI has become a very popular method, with a number of important success stories recently appearing in the literature (especially [9]).

As a case study, we have applied both API and OPI to a simplified version of American football. We consider play selection for one offensive drive in the middle of an infinitely long game. Begin-game and end-game effects are ignored. Also, in contrast to real American football, we ignore the fact that there is an intelligent opponent which randomizes the optimal strategy and causes the state-transition probabilities to be dependent on field position. In our version of football, state transitions are determined through a probabilistic model (discussed in the Appendix) in which only offensive play decisions can impact the drive trajectory. The objective is to maximize the expected difference between "our" team's drive-score and the opposing team's score from the field position at which they receive the ball. To put it another way, we want to compute a policy which achieves

$$J^*(i) \triangleq \max_{\text{policies, } \pi} E \left\{ \begin{array}{l} \left(\begin{array}{l} \text{Points received at the end of our drive} \\ \text{from initial field-position, } i, \text{ under policy } \pi. \end{array} \right) \\ - \left(\begin{array}{l} \text{anticipated points gained by the opposing} \\ \text{team from our final field position.} \end{array} \right) \end{array} \right\} \tag{1}$$

The rules of this simplified version of football are detailed in the Appendix. Despite the simplicity of the model, this is a moderately large scale problem, having 15,100 states. An interesting aspect of the model is the fact that rewards are earned only upon transitioning the end of the offensive drive (the *terminal state*). Because of this, the nature of the optimization problem is that of choosing policy that drives the system to a desirable region of the state space.

To give a preview of our experimental results, we have found that many of the methods for training reward-to-go approximations perform similarly. For a fixed policy (in API), the final approximations that result from TD(λ), for different values of λ, are not that different from approximations due to the Bellman Error method or even the approximations due to linear least squares regression via the matrix pseudo-inverse. This is true even in the context of OPI, where policies change more on a continual basis. Regarding $TD(\lambda)$, we have found that values of λ closer to one are generally best, but only by a slight margin. Our best results were usually obtained with $\lambda = 1$, supporting the assertion put forth in [1]. The results we have obtained for football have helped to clarify our thinking about how Neuro-Dynamic Programming methods work. In particular, the "greedy subsets" interpretation of the methods (which is discussed in Sections 2.3.1 and 2.3.2) came from our observation of "oscillatory" limiting behavior in both OPI and API.

One of our main conclusions is that football is an effective testbed for Neuro-Dynamic Programming methods. The problem itself is not trivial; the set of allowable policies is quite large, containing a huge number of policies that are "reasonable" (but indistinguishable) in heuristic terms. On the other hand, since an exact (numerical) solution can be computed, we have a useful basis for comparisons of the approximate methods. Additionally, while this is not the focus of our research, our model is such that we may use our understanding of real American football to attach heuristic interpretations to the the suboptimal policies that are produced by the methods.

In Section 2, we give an overview of our methods. We begin by reviewing the pure forms of value iteration and policy iteration which are targets for the approximate methods of NDP. We then discuss methods for training approximations of reward-to-go for a fixed policy. Section 2 ends with a discussion of methods for

computing new policies. This is where the distinction between API and OPI will be clarified. In Section 3, the problem of optimal play selection in football is formulated as a stochastic shortest path problem. We present there the optimal solution for the precise mathematical problem that we pose. We obtain the solution numerically through exact implementation of policy iteration. In Section 4, we discuss technical issues related to our application of NDP to football. We consider there a number of different approximation architectures, as well as issues relating to simulation. In Section 5, the experimental results of our case study are presented and compared to the exact optimal solution. Section 6 finishes the paper with conclusions, and the Appendix gives the exact rules of our simplified version of football.

2 An Overview of the Methods

This section gives an overview of our methods. The discussion is not specific to Football, nor is it specific to a particular architecture for function approximation. (Our model for football is presented in Section 3, and the details of our methods particular to football are given in Section 4.) To keep the presentation compact, we limit the discussion primarily to the case of stochastic shortest path problems. For the appropriate extensions to discounted and average cost problems, the reader is referred to [4].

2.1 Exact Dynamic Programming

The general class of problems we consider is that of Markov Decision Processes. We are generally interested in choosing a policy (a map from states to controls) that maximizes long-term total reward. The control we apply at a given state determines the probability distributions that govern both immediate reward and the transition to a successor state. Our objective, in somewhat loose mathematical terms, is as follows: Given that we are at a state i, achieve the long-term discounted reward

$$J^*(i_0) = \max_{\pi=\{\mu^0,\mu^1,\ldots\}\in\Pi} E\left\{\sum_{k=0}^{\infty} \alpha^k g(i_k,\mu^k(i_k),i_{k+1}) \mid i_0,\ \pi\right\} \tag{2}$$

where

1. the trajectory $\{i_k\}$ is a sequence of states in a finite state space S,
2. the policy $\pi = \{\mu^0,\mu^1,\ldots\} \in \Pi$ is a sequence of functions $\mu^k \in M$ which map the state space S to a finite set of allowable controls U,
3. $g(i,u,j)$ is the reward for transitioning from i to j under the control u,
4. $\alpha \in (0,1]$ is a discount factor for future rewards, and
5. the expectation (which is conditional on the initial state i_0 and the policy π) is taken over all of the state trajectories $\{i_k\}$ that are possible under π.

We assume that the mechanism for state transitions is Markov. That is, we assume that the probability of transitioning to state j in one step is dependent only on the current state i and the control u applied at i. This probability is denoted $p_{ij}(u)$. For notational convenience, if a policy $\pi \in \Pi$ is such that $\pi = \{\mu,\mu,\ldots\}$, then we shall refer to this as the *stationary policy* μ.

We are primarily interested in problems where either

1. the reward function is bounded (i.e. $|g(i,u,j)| \leq K_g$ for all i,u,j) and $\alpha < 1$, or
2. all of the following are true:
   (a) there is no discounting (i.e. $\alpha = 1$),
   (b) there exists a stationary policy such that for all initial states i_0 the system reaches a terminal, zero-reward state with probability one (this is a "proper" policy), and
   (c) for all stationary policies that are not proper, the expected cost (negative reward) from some initial state is infinite.

The first type of problem is called a *discounted reward problem*; the second is called a *stochastic shortest path problem*. For a careful definition of these problems and corresponding analysis, the reader is referred to [2]. When the Markov Decision Process falls into either of these categories, the optimal expected reward-to-go from each state is characterized by "Bellman's equation":

$$J^*(i) = \max_{u \in U} \left[\sum_{j \in S} p_{ij}(u)(g(i,u,j) + \alpha J^*(j)) \right] \quad \forall\, i \in S \tag{3}$$

The value of u which achieves the maximum in Bellman's equation for each $i \in S$, determines a stationary optimal policy μ^*. Bellman's equation can thus be viewed as a coupled set of nonlinear equations which have to be solved simultaneously. Solving this set of equations can be difficult to do explicitly, especially when the state space is large. On the other hand, there are a number of algorithms that can be used to find J^* iteratively. One of these is *value iteration*. The form of this algorithm is suggested by Bellman's equation itself. We start with an initial guess for J^*, denoted J^0, defined for all $i \in S$. The k^{th} iterate function is computed according to,

$$J^k(i) = \max_{u \in U} \left[\sum_{j \in S} p_{ij}(u)(g(i,u,j) + \alpha J^{k-1}(j)) \right] \quad \forall\, i \in S \tag{4}$$

It can be shown that $J^*(i) = \lim_{k \to \infty} J^k(i) \;\; \forall\, i \in S$.

If one has a suboptimal policy μ, for which it is necessary to determine the expected reward-to-go function J^μ, an analogous form of Bellman's equation holds. Here, U is replaced with $\{\mu(i)\}$, eliminating the "max" operation in (3). It can be shown that the corresponding form of value iteration (called *successive approximation*) is also valid. The determination of J^μ for a fixed policy μ is referred to as *policy evaluation*.

Another algorithm for determining J^* is known as *policy iteration*. In this algorithm, we start with an initial policy μ^0, and then evaluate J^{μ^0} for all $i \in S$ (using e.g. successive approximation.) The k-th iterate policy is computed according to

$$\mu^k(i) = \arg\max_{u \in U} \left[\sum_{j \in S} p_{ij}(u)(g(i,u,j) + \alpha J^{\mu^{k-1}}(j)) \right] \quad \forall\, i \in S \tag{5}$$

It can be shown that, as long as S and U are finite and the evaluations of J^{μ^k} are *exact*, then J^{μ^k} converges to the optimal reward-to-go function J^* in a finite number of iterations.

From this point on in the paper we shall be concerned exclusively with stochastic shortest path problems. Since $\alpha = 1$ for these problems, we shall drop α from the discussion.

2.2 Approximating Reward-to-Go for a Fixed Policy (Approximate Policy Evaluation)

A central idea for the methods of this paper is that exact evaluations of the reward-to-go function in policy iteration can be replaced with *approximations*, denoted $\tilde{J}(\cdot, r)$. As the notation suggests, the approximations are selected from a parametric family of functions (e.g. neural networks), where the parameter $r \in R^d$ is chosen on the basis of simulation data and least squares regression. Throughout this paper specific forms for the approximation are referred to as "architectures." We assume that, for each state $i \in S$, $\tilde{J}(i,r)$ is continuously differentiable with respect to r. The various architectures we have used in football are detailed in Section 4.1. In computer implementations of NDP, the approximations of reward-to-go are implemented as subroutines. The process of executing the code for a reward approximation is referred to as "evaluating the approximation."

Given a batch of simulation data, a suitable parameter vector can be found by solving a least squares optimization problem of the form

$$\min_{r \in R^d} \frac{1}{2} \sum_{k=1}^{N} \left(c_k - \tilde{J}(i_k, r) \right)^2 \tag{6}$$

where c_k is the sample total (discounted) reward from state i_k, and N is the number of data pairs (i_k, c_k). Unfortunately, unless $\tilde{J}$ is linear in the parameter vector r, this optimization problem may have many local minima. Despite this, the usual approach to optimizing the parameter vector is to use gradient-like methods to minimize least squares cost. The application of these methods is referred to as "training". In many cases the training methods may be used "on-line" in the sense that it is not necessary to have generated the complete data set before training may begin.

Throughout this paper, approximations of the reward-to-go function are "feature-based", meaning that $\tilde{J}(i,r)$ is of the form $H(f(i),r)$ where $f = (f_1, \ldots, f_d)$ is a vector of *features*. Each feature f_l is a function that maps the raw state (an element of the finite set S) to a real number, giving a quantitative description of an important aspect of the system. For example, in our model for football there is a finite number of field positions which our team may assume. Depending on how we label these states, it probably isn't very insightful to know only that the system is in some arbitrary state $i \in S$. On the other hand, knowing that i corresponds to some number of yards to the goal gives us an immediate sense for how our team is doing. The function $f(i)$ = "number of yards to the goal from i" is an important feature for football.

It is in the construction of an approximation architecture where "engineering insight" into the problem may be applied most effectively. The most important issue to resolve is what features should the approximation rely upon? (What information is important?) It is possible that the choice of features depends on the type of approximation architecture being used. There are many different kinds of architectures that may be employed, including multilayer perceptrons, radial basis function networks, polynomial (in features), etc. Given that a particular type of architecture has been selected, there are usually many architectural parameters which must be fixed before training may begin. For example, if a multilayer perceptron is to be used, one must decide upon the number and configuration of sigmoidal units within the hidden layers of the network. Clearly, the resolution of these issues requires a great deal of insight into the problem at hand.

2.2.1 Generating Sample Data

Before discussing training algorithms proper, a few words about the generation of sample data are necessary. As mentioned earlier, sample data is obtained through simulation. The goal is to obtain data that will yield an accurate approximation of the reward-to-go function. A data set which is not representative will lead to biased approximations. It may be that the biased approximations are accurate in some regions of the state space yet highly inaccurate (or even misleading) elsewhere.

One important consideration which relates to infinite horizon problems in general is that of termination of individual sample trajectories. For stochastic shortest path problems, it is easiest to simply terminate the simulation whenever the system's terminal state is reached; if more sample data is required then it is easy enough to execute another simulation. For optimization problems where there is no terminal state, the issue of termination is considerably more vexing.

There remains one more important question: How do we choose initial conditions for the simulation? In our experience, the performance of NDP is very sensitive to the method used for picking (or "sampling") initial conditions. One method can lead to success, while another may lead to abject failure. Unfortunately, in checking the literature, there seems to be a total lack of theoretical understanding of this problem, so we have had to rely on our own engineering judgment. The discussion below is based largely on our own computational experience with football and other similar problems.

Given a stationary policy μ, suppose that the initial condition for the system has a strong influence on which parts of the state space get visited. It is important then to choose initial conditions for simulation carefully. The goal is to guarantee that after some number of sample trajectories all states are represented in the data set. For any given fixed set of initial conditions, it may be impossible to provide this guarantee, so the usual practical approach is to choose initial conditions randomly (and hope for the best.) A degenerate form of this rule is to exhaustively cycle through the state space in picking initial conditions, guaranteeing that all states are visited at least once. This may not be necessary, however, since the neural network should be able to generalize from a smaller amount of training data. Moreover this degenerate approach may be impractical if the state space is very large.

To appreciate the subtlety of the initial-condition sample issue, consider the case where there is a particular state of interest which is the natural starting point for the system in real life. One would be tempted in this case to use this state exclusively as an initial condition for simulation, the idea being to focus the simulation

so that data is generated just for the "important" regions of the state space. Unfortunately, because our basic framework is that of policy iteration, all of the regions of the state space are important. To see why, suppose that there is little training data representing a particular region of the state space, and suppose that, after training, the approximation in this region is overly optimistic. When it comes to using this approximation to pick actions (in updating the policy), there is false incentive to visit this region of the state space.

In the end, biased approximations and false policy improvement are intimately connected in way which is difficult to untangle. When the optimization problem comes with a natural starting state, the general rule for success seems to be to pick initial conditions randomly from a "large" ball of states around the special initial state. The selection of a suitable probability distribution is a matter of engineering judgment.

2.2.2 Training algorithms

In this paper, training algorithms come in two flavors: (i) iterative incremental gradient methods (and their cousins) and (ii) non-iterative linear least squares regression using the singular value decomposition. The latter form of training is applicable to architectures where $\tilde{J}(i,r)$ is linear in the parameter vector r. For a general discussion of incremental gradient methods, the reader is referred to [3].

TD(λ) The general class of Temporal Difference (TD) learning methods include a whole family of algorithms parameterized by $\lambda \in [0,\ 1]$. The specific algorithm that correspond to a given value of λ is referred to as "TD(λ)." TD methods are best understood in the context of individual sample trajectories: each sample trajectory results in a single (batch) TD(λ) update. Let $r^t \in R^d$ be the t-th iterate of the algorithm. (r^0 is a suitable initial guess.) We will show how to compute r^t based on the preceding iterate r^{t-1}. Let $\{(i_k, g_k);\ k = 0, \ldots, N\}$ be the sample data for the t-th trajectory, where i_k denotes the k-th state in the trajectory, i_N denotes the last non-terminal state in the trajectory, and g_k is the cost to transition from i_k to i_{k+1}. Defining

$$d_k = g_k + \tilde{J}(i_{k+1}, r^{t-1}) - \tilde{J}(i_k, r^{t-1}), \quad k = 0, \ldots, N. \tag{7}$$

to be the sequence of *temporal differences* for the t-th trajectory, the parameter update for this trajectory is computed as $r^t = S^N$, where

$$\begin{aligned} s^0 &= r^{t-1} + \gamma_t d_0 \nabla \tilde{J}(i_0, r^{t-1}) \\ s^1 &= s^0 + \gamma_t d_1 \left[\nabla \tilde{J}(i_1, r^{t-1}) + \lambda \nabla \tilde{J}(i_0, r^{t-1})\right] \\ &\vdots \\ s^N &= s^{N-1} + \gamma_t d_N \sum_{m=0}^{N} \lambda^{N-m} \nabla \tilde{J}(i_m, r^{t-1}) \end{aligned} \tag{8}$$

The scaling factor γ_t is a stepsize which is chosen to diminish as $1/t$. (More generally, we could allow the stepsize factor to diminish as a function of simulation stage number, but we have omitted this possibility in the notation.) A detailed analysis of TD(λ) for the case of a linear compact representation can be found in [4, 7].

It can be shown that with $\lambda = 1$ the difference $(r^{t-1} - r^t)$ is exactly the gradient of the cost in (6) multiplied by the stepsize factor γ_t. Thus, each TD(1) iteration is equivalent to a gradient descent step where the cost function is defined by the data of the corresponding sample trajectory. Since updates are computed on the basis of complete, independently generated sample trajectories, this algorithm amounts to stochastic gradient descent applied to a least squares problem of the form:

$$\min_{r \in R^d} E\left\{\frac{1}{2}\sum_{k=0}^{N}(c_k - \tilde{J}(i_k, r))^2\right\} \tag{9}$$

The expectation in (9) is over the sequence of random variables $\{(i_k, c_k)\}$ and is conditional on both the policy of interest and the rule for picking initial conditions for simulation.

For the case that $\lambda \neq 1$, the connection between $(r^{t-1} - r^t)$ and the gradient of the cost function in (6) is lost. In fact, it is an open question at present whether TD($\lambda \neq 1$) updates are related to the gradient of *any*

suitable cost function. In addition to this there is some analytical evidence that, with any value for λ (other than one), TD(λ) can lead to arbitrarily bad approximations of reward-to-go. [1] On the other hand, the computational experience of many researchers suggests that $\lambda \neq 1$ is "best" when applied to large-scale/real-world problems [9]. Because of this, we have given considerable attention to the case that $\lambda < 1$ in our football case study.

To finish our description of TD(λ), we note that to compute s^k one needs to have information about only the first $k+1$ stages of the sample trajectory. Because of this, TD(λ) can be applied in real time. (One does not have to wait until sample trajectories end to start computing the corresponding TD update.) We shall address the issue of picking γ_t further when we discuss the variations on policy iteration in section 2.3.

Bellman Error Method The Bellman Error method can be viewed as a variation on TD(0). As with TD(λ), let $r^0 \in R^d$ be an initial guess for the parameter of the approximation of reward-to-go, let γ_t be a stepsize parameter which decreases as $1/t$; and let $\{(i_k, g_k);\ k = 0, \ldots, N\}$ represent the set of sample state and transition reward pairs for the t-th sample trajectory. Also, let $\{d_k;\ \ k = 0, \ldots, N\}$ be the sequence of *temporal differences* for the t-th sample trajectory. The t-th parameter in the Bellman Error method is computed as $r^t = s^N$ where

$$\begin{aligned} s^0 &= r^{t-1} + \gamma_t d_0(\nabla \tilde{J}(i_0, r^{t-1}) - \nabla \tilde{J}(\bar{i}_1, r^{t-1})) \\ s^1 &= s^0 + \gamma_t d_1(\nabla \tilde{J}(i_1, r^{t-1}) - \nabla \tilde{J}(\bar{i}_2, r^{t-1})) \\ &\vdots \\ s^N &= s^{N-1} + \gamma_t d_N(\nabla \tilde{J}(i_N, r^{t-1}) - \nabla \tilde{J}(\bar{i}_{N+1}, r^{t-1})) \end{aligned} \tag{10}$$

where $\{\bar{i}_k; k = 1, \ldots, N+1\}$ is a sequence of independently generated successor states. That is, $\bar{i}_{k+1}$ and i_{k+1} are independent samples of successor states to i_k under the stationary policy μ. To see how this algorithm is motivated, notice that $(\nabla \tilde{J}(i_N, r^{t-1}) - \nabla \tilde{J}(\bar{i}_{N+1}, r^{t-1}))$ is a zero-bias estimate of $E(\nabla \tilde{J}(i_N, r^{t-1}) - \nabla \tilde{J}(\bar{i}_{N+1}, r^{t-1}))$. With is interpretation in mind, it is possible to check that each complete iteration is a zero-bias approximation to one gradient step toward solving the following problem:

$$\min_{r \in R^d} \sum_{k=0}^{N} \frac{1}{2} \left(\tilde{J}(i_k, r) - E[g(i_k, \mu(i_k), \bar{i}) + \tilde{J}(\bar{i}_{k+1}) \mid i_k, \mu] \right)^2 \tag{11}$$

where μ is the stationary policy of interest. As this iteration is applied to many sample trajectories, the overall flavor of the algorithm is that of stochastic gradient descent applied to the minimization of the expected value of a least squares objective function.

Linear least squares approximation via the Singular Value Decomposition (SVD) This training method assumes that the approximation is linear in the parameter vector, as follows:

$$\tilde{J}(i, r) = \sum_{l=1}^{d} f_l(i) r_l = f(i)' r \tag{12}$$

The functions $f_l : S \mapsto R$ are features. Let $\{(i_k, c_k)\}$ represent the set of state/reward-to-go data pairs generated by many sample trajectories under the stationary policy of interest. A closed form solution to (6) can be obtained as

$$r^* = \left[\sum_{k=1}^{N} f(i_k) f(i_k)' \right]^+ \left(\sum_{k=1}^{N} c_k f(i_k) \right) \tag{13}$$

where $[\cdot]^+$ denotes the matrix pseudo-inverse. Such a solution is very easy to compute numerically using the matrix Singular Value Decomposition (SVD) as a basic building block.

2.3 Policy Updates. API and OPI.

Methods for computing approximations of the reward-to-go function for fixed policies were discussed in the preceding subsection. Here, we discuss methods of computing *new* policies based on these approximations. In particular, we present two variations on classical (exact) Policy Iteration: Approximate Policy Iteration (API) and Optimistic Policy Iteration (OPI). While there are significant differences between the two, they both use the same fundamental technique for computing new policies: given the approximation $\tilde{J}(\cdot, r^{k-1})$ for a stationary policy μ^{k-1}, compute a new policy μ^k as

$$\mu^k(i) = G(r)(i) \triangleq \arg\max_{u \in U} \left[\sum_{j \in S} p_{ij}(u)(g(i,u,j) + \tilde{J}(j, r^{k-1})) \right] \quad \forall\, i \in S. \tag{14}$$

We have defined here the "greedy" operator G which maps parameter vectors $r \in R^d$ to state-action mappings $G(r) \in M$. (To make G well defined, we stipulate that a deterministic method of breaking ties between equally greedy actions is employed.) Notice that (14) differs from (5) in that an approximation is used in place of the exact reward function $J_{\mu^{k-1}}$.

In the subsections which follow we give detailed descriptions of both API and OPI, highlighting the differences between the two. Before preceding with that discussion, however, there remain some additional details to cover that relate to the incremental training methods (TD(λ) and the Bellman Error method.) First regarding stepsizes, we have used the following rule for both API and OPI.

$$\gamma_{t,p} = \max\left\{ \frac{\gamma_0}{(1+\gamma_{ra}t)(1+\gamma_{ro}p)}, \gamma_f \right\} \tag{15}$$

is the stepsize for the t-th trajectory of the p-th policy, where $\gamma_0, \gamma_{ra}, \gamma_{ro}$, and γ_f are free parameters. Different values for these parameters are appropriate, depending on whether the method is API or OPI. As a second note, the final approximation parameter received in evaluating policy μ^k is always used as the starting point in the evaluation of the updated policy μ^{k+1}.

2.3.1 Approximate Policy Iteration (API)

As discussed in the Introduction, Approximate Policy Iteration can be (viewed) as an honest attempt to emulate policy iteration. Generally, many sample data pairs are generated and used in the approximate evaluation of reward-to-go, any of the training algorithms discussed previously may be used. For TD(λ) and the Bellman Error methods, a fixed (large) number of sample trajectories are generated and stored in a data buffer. The sample data is repeatedly processed by training algorithms. (This is called "cycling through the samples buffer.") Each time a sample trajectory is seen, it is treated as though it were a new trajectory. Recalling the stepsize rule given by (15), γ_{ro} is generally set to 0. For the SVD method a large amount of training data is generated, then the regression matrices are computed, and the least squares solution is computed all in one shot. Generally, a predetermined number of policies are computed, and the weights of the approximation corresponding to the best policy are stored for future use. The API algorithm is summarized as follows:

1. Start with an initial stationary policy μ^0.
2. At the t-th iteration, generate a large number of sample trajectories to get the training data for the policy μ^{t-1}.
   (a) If using TD(λ) or the Bellman Error method, store the data as complete (and distinct) sample trajectories, keeping the data in the form (i_k, g_k), where i_k is the state at the k-th stage of a trajectory with corresponding transitional reward g_k.
   (b) If using SVD, store the data as a running sum of $f(i_k)f'(i_k)$ and a running sum of $c_k f(i_k)$, where c_k is the sample reward-to-go from the state i_k.
3. Use the sample data to compute the "best" approximation parameter r^{t-1}.

(a) If using TD(λ) or the Bellman Error method, cycle through the sample data buffer updating the parameter according to either (8) or (10) with a stepsize given by (15), where $\gamma_{ro} = 0$.

(b) If using SVD, compute r^{t-1} as in (13).

4. Compute μ^t using (14). That is, $\mu^t = G(r^{t-1})$.

5. Either stop or go back to step 2.

As for analytical guarantees of performance, it has been shown in [2] that if the approximations of reward-to-go are close to the actual reward function (in a specific mathematical sense) and if the policy updates are computed accurately with respect to the approximations, then infinitely often in the limit the resulting policies will yield actual rewards which are close to optimal. The bounds on limiting performance are proportional to the accuracy of the approximations. This type of mathematical result serves mainly as a "sanity-check", guaranteeing that there aren't any hidden bugs in the methodology which can lead poor performance a priori. While results of this type are available for API, this method sees very little use in practice.

To gain a heuristic understanding of how API works, it is useful to recall the "greedy" operator G which maps parameter vectors r to stationary policies $G(r) \in M$. Given that there are finitely many control actions for each state (in a finite state space), there is a finite number of stationary policies for the system. Because of this the respective values of G form a partition of the parameter space R^d. Given a convergent algorithm for training reward-to-go approximations, each policy μ has a "target" parameter value r^μ which is the limit of the training algorithm applied to sample data generated by that policy. If, in implementing API, enough training data is generated so that these limiting parameter values are closely approached, then the API iteration can be expressed roughly as $\mu^{k+1} \approx G(r^{\mu^k})$. Clearly, the iterations of API are heavily dependent on the nature of G and the relationship between μ and r^μ. (These are in turn determined by both the architecture for reward-to-go approximation and the method used for choosing simulation initial conditions.) A priori, there is no guarantee (in fact, it is unlikely) that the method will converge to a single policy. It is more likely that an "oscillatory" mode will arise where, after enough iterations, the method starts generating with some periodicity the same policies over and over again.

2.3.2 Optimistic Policy Iteration (OPI)

OPI distinguishes itself from API in that policy updates are computed on the basis of very rough (optimistic) approximations to the reward-to-go function. Generally, very little sample data is generated for each policy (usually just one sample trajectory). Also, the training algorithms are allowed to have very few (usually just one) passes through the data set, so the effect of training is very limited. The intuition here is that the corresponding policy update represents an *incremental* change from the old policy. (This is viewed as a beneficial property, since small changes in the policy hopefully will not result in entirely new regions of the state space being visited from policy to policy. In the context of API, such drastic changes tend to over-extend the approximation's region of accuracy.) To make up for the limited amount of training data per policy, usually a very large number of policy updates are computed.

OPI has one very important, practical difficulty: there is no automatic mechanism for evaluating the policies that are computed. By the optimistic nature of OPI, very little data is required to compute new policies. However, to gain a practical evaluation of a policy's effectiveness, many additional sample trajectories are required. (We would need such an evaluation to decide, for example, whether to terminate the iterations of OPI.) Generating lots of "extra" sample trajectories is contrary to the spirit of OPI, so evaluation is an inherent difficulty of the methodology. One way to circumvent this is to evaluate policies only periodically within the OPI iterations. This technique will unfortunately allow many policies (some of which may be very close to optimal) to slip by between evaluations. On the other hand, this technique allows *most* of the computational effort to be directed toward the underlying OPI method. In contrast to OPI,, the evaluation of individual policies is an integral feature of the API method.

We now give a formal description of the OPI methodology. As with API, optimistic policy iteration can use any of the approximation training methods discussed in the preceding section. If either the TD(λ) or Bellman error methods are used, then the only difference from API (in addition to the amount of training data) is that the stepsizes here are computed with γ_{ra} set to zero as opposed to γ_{ro} in (15). Otherwise, the flow of the OPI algorithm is identical to that of API as given in the steps of the preceding subsection:

1. Start with an initial stationary policy μ^0.

2. At the t-th iteration, if $(t-1)$ is a multiple of the desired number of policies per evaluation, then generate a large number of sample trajectories under μ^{t-1}; estimate reward-to-go by computing sample means. Store the results for post-processing.

3. Generate a small number of sample trajectories to get the training data for the policy μ^{t-1}. Store the data as complete (and distinct) sample trajectories, keeping the data in the form (i_k, g_k), where i_k is the state at the k-th stage of the trajectory with corresponding transitional reward g_k.

4. Use the sample data to compute parameter r^{t-1}. Cycle (few times) through the sample data buffer updating the parameter according to either (8) or (10) with a stepsize given by (15), where $\gamma_{ra} = 0$.

5. Compute μ^t using (14). That is, $\mu^t = G(r^{t-1})$.

6. Either stop or go back to step 2.

Significant differences between OPI and API arise when the SVD method of training is used. To see where these differences come from, recall that in API (with the SVD method), the parameters of the reward-to-go approximation for μ^k are completely determined by the sample data generated under μ^k. The effect of data generated for earlier policies is completely lost. This is acceptable the API scheme since very many sample trajectories are computed per policy. On the other hand, a fundamental characteristic of OPI is that the policies do not change very quickly and, while very little data is generated for each policy, a meaningful aggregate approximation can be built after a number of policies are considered. The effect of past policies is accumulated in the parameter vector, so that the SVD method as implemented in API will not work. An appropriate OPI/SVD methodology is given by the following Kalman filtering scheme:

1. Start with $H^{-1} = 0 \in R^{d \times d}$, $\eta^{-1} = 0 \in R^d$, $r^{-1} = 0 \in R^d$, an initial stationary policy μ^0, and a scalar parameter $\lambda \in (0, 1]$.

2. At the t-th iteration, if $(t-1)$ is a multiple of the desired number of policies per evaluation, then generate a large number of sample trajectories under μ^{t-1}; estimate reward-to-go by computing sample means. Store the results for post-processing.

3. Generate a small number of sample trajectories to get the sample data pairs (i_k, c_k) for the policy μ^{t-1}. Let N be the (random) number of data pairs.

4. Compute H^{t-1} and η^{t-1} as

$$H^{t-1} = \left[\sum_{k=1}^{N} \lambda^{N-k} f(i_k) f(i_k)'\right] + \lambda^N H^{t-2} \tag{16}$$

$$\eta^{t-1} = \left[\sum_{k=1}^{N} \lambda^{N-k} f(i_k)(c_k - f(i_k)' r^{t-2})\right] + \lambda^N \eta^{t-2} \tag{17}$$

5. Compute r^{t-1} as

$$r^{t-1} = r^{t-2} + \left[H^{t-1}\right]^+ \eta^{t-1} \tag{18}$$

6. Compute μ^t using (14) with $r = r^{t-1}$. That is, $\mu^t = G(r^{t-1})$.

7. Either stop or go back to step 2.

The interpretation of OPI as an incremental form of approximate policy iteration is very appealing. Unfortunately, the method suffers from a lack of mathematical analysis guaranteeing either convergence or performance. The main analytical difficulty stems from the fact that the approximations at each iteration bear little resemblance to the corresponding exact reward-to-go functions. The performance bounds for API given in [2] are so weak here that the theory cannot explain any success of the method. Despite this, OPI has become very popular in the machine learning research community.

To gain a heuristic understanding of OPI, recall the "greedy" partitioning of the parameter space R^d provided by the operator G. Assume that parameter vectors generated by the iteration converge. (A formal proof of convergence remains to be found.) Given convergence, the changes in the parameter vector that result from each iteration become smaller and smaller. If there is convergence to a point interior to one of the subsets of R^d, then eventually the policies generated by OPI will converge (since $r^k \approx r^{k-1}$ implies that $\mu^{k+1} \triangleq G(r^k) = G(r^{k-1}) \triangleq \mu^k$.) OPI exhibits more interesting behavior when the parameter vectors converge to a point on the boundary of two or more of the greedy subsets of R^d. In this case, it is no longer true that the corresponding policies generated by the method will converge. Rather, as the parameter vectors bounce from one subset to another (even as the size of the excursions into the subsets gets smaller and smaller), the corresponding greedy policies vary discretely. This provides a mechanism for the "oscillatory" limiting behavior often exhibited by the methodology. It is possible for the parameter vectors of the algorithm to converge while the sequence of policies generated by the method may vary radically (even qualitatively) in the limit. A priori, there are no guarantees that this behavior will not occur. (Again, what actually happens depends on the architecture used for reward-to-go approximations and the method used for choosing simulation initial conditions.) It is our belief that convergence to a boundary is the usual way in which OPI plays out; convergence of the policies seems to be the exception, not the rule.

3 The Football Model

3.1 Problem Formulation

We present here a simplified version of American football and cast it as a stochastic shortest path problem. The "system" in this model is the current offensive drive whose net score we are trying to maximize. The state of the system is characterized by three quantities: x = the number of yards to goal (field position), y = yards to go until next first down, and d = down number. Note that x and y are constrained to taking on integer values. Thus, there is a finite number of states at which the quarterback must have some control action in mind. Let these states be identified by $i \in S$, where S is a finite set. To keep the notation consistent, the states of the system are the integer-valued elements i of the set S. The triple (x_i, y_i, d_i) denotes the field position, yards to go until next first down, and down number corresponding to state $i \in S$. We shall sometimes abuse notation and refer to (x_i, y_i, d_i) or simply (x, y, d) as the state. This should not lead to confusion since for each state $i \in S$ there is only one triple (x_i, y_i, d_i) and vice-versa.

Transitions from state to state are governed by the probabilistic model discussed in the Appendix. At each state, the quarterback must choose one out of four play options: 0 = Run, 1 = Pass, 2 = Punt, and 3 = Kick (field goal). The quarterback's *policy* is a function $\mu : S \rightarrow U$, where $U = \{0, 1, 2, 3\}$ denotes the set of control options.

"Our" team transitions to an absorbing, zero-reward termination state T whenever it looses possession of the ball. Rewards in this framework are earned only upon first transitioning to T. The amount of reward g is exactly the score received at the end of the our team's drive *minus* the expected score to be received by the opponent at the end of his drive, which is a function of where he receives the ball. As is the case in real football, termination of the current offensive drive here is inevitable under all policies. (More precisely, for the model described in the Appendix, this is a finite-horizon dynamic programming problem, with 40 being the maximum number of transitions to termination.) The problem of maximizing expected total reward can be viewed as a stochastic shortest path problem. Note that we use the terminology "reward" here, instead of "cost" which is prevalent in the literature on stochastic shortest paths. Since termination is inevitable under all policies, this is just a matter of semantics. Our objective is to "maximize net rewards" as opposed to "minimize net costs".

We believe football is an appropriate testbed for methods of Neuro-Dynamic programming because it lies on the boundary between medium-scale (numerically tractable) problems and truly large-scale (computationally infeasible) problems. The model for football that we consider is obviously numerically tractable. However, simple enhancements to the model can be implemented that make the computational requirements for solving the problem much more significant. For example, by more finely discretizing the playing field, say to half-yard units, we would suddenly have 60200 states, an increase by a factor of four (i.e. not linear). Additionally, suppose we wanted to factor in time. If we discretized time in 10 second intervals and then

played for the last two minutes of the game (even without giving the other team the opportunity to make decisions), we would suddenly have 602000 states. For these larger problems, it becomes impractical or even impossible to solve the problem using numerical implementations of the classical methods. The memory and processing requirements are simply too great. This is what motivates us to consider API and OPI. On the other hand, because we have the optimal reward-to-go and optimal policy for our present formulation, we may make meaningful comparisons of the algorithms both with respect to each other and to the optimal solution of the problem.

3.2 The Optimal Solution

Our model for football (with 15100 states, four control options per state, and termination inevitable) leads to a problem that can be solved exactly through numerical implementation of the classical methods discussed in Section 2.1. On a 120 MHz Pentium machine running the Linux operating system, it took approximately 2.5 minutes to compute the optimal reward-to-go functions and optimal policies shown in Figure 1 using policy iteration. To evaluate the reward-to-go function for each iterate policy, we applied successive approximation applied until the sup-norm difference between the iterates was less than 10^{-6} football points. Starting from an initial policy that may be described as "always run", six policy iterations were required to determine the optimal policy. The code was written in the C programming language.

In Figure 1, first down is distinctive since the plots there are two-dimensional graphs. This reflects the fact that, at first down, there is only one possible value of y for each value of x. While it is optimal to run from $x = 1$ to $x = 65$, the optimal policy requires that pass attempts be made from (roughly) $x = 66$ to $x = 94$. For the next 5 yards it is optimal to run, and at $x = 100$, the optimal policy is to pass again. (This is not the result of a bug in the software.) We note from the reward-to-go function that, from 80 yards to the goal (which is where we typically expect to gain possession of the ball), the expected net reward is -.9449 points. If our team were to actually receive the ball at 80 yards every possession, we could expect ultimately to loose the game. This is strictly a function of the parameters which we have set arbitrarily in our mathematical model. Another set of parameters could very well lead to an optimal expected net reward from 80 yards to goal which is positive.

The results for the remaining downs are presented as surface plots since for every value of x there are many possible values of y. In theory, y can be as large as x. (However, in practical terms $x > 20$ is an extremely unlikely event.) While $x \approx y$ *is* accounted for in the computations, the plots in the figure show what happens only for values of y from one to 20. At second down, the optimal policy dictates that pass attempts be made for a wide range of values of x and y. The plot also shows that there is a run-attempt region for the remaining values of x and y. At third down it is usually optimal to pass; however, for x and y large enough it is actually optimal to punt. (This is where our team's outlook for this drive is particularly gloomy. The risk is great that the other team will gain possession of the ball in a region disadvantageous to us.) The fourth down optimal policy exhibits the most variety in terms of choosing different play options. If "our" term is close enough to either a new first down or the goal, then a running or passing play is indicated. On the other hand, if a new first down or touchdown is not likely, then either a field goal attempt or punt is specified.

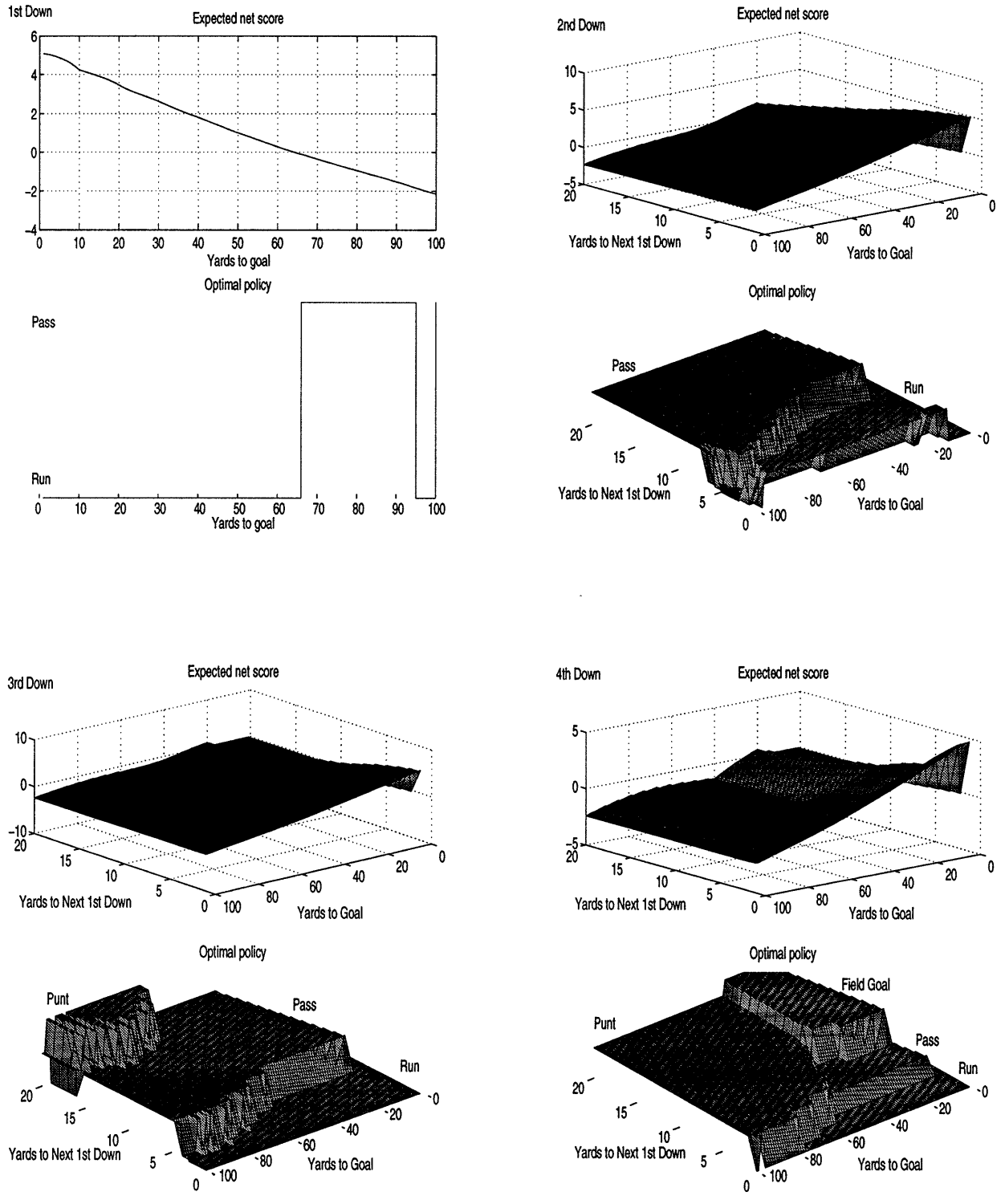

Figure 1: Complete characterization of the optimal policy.

3.3 A Heuristic Solution

To give an idea of the difficulty of football, we hypothesize a class of reasonable policies as follows:

1. At first down, PASS.
2. At second down, if the number of yards to the next first down is less than three, then RUN; otherwise, PASS.
3. At third down,
   (a) if the number of yards to the endzone is less than 41,
      - and if the number of yards to the next first down is less than three, then [RUN or PASS],
      - otherwise, [RUN or PASS]

   (b) if the number of yards to the endzone is greater than 40,
      - and if the number of yards to the next first down is less than three, then [RUN or PASS],
      - otherwise, [RUN or PASS]
4. At fourth down,
   (a) if the number of yards to the endzone is less than 41,
      - and if the number of yards to the next first down is less than three, then [RUN, PASS, or KICK],
      - otherwise, [RUN, PASS, or KICK]

   (b) if the number of yards to the endzone is greater than 40,
      - and if the number of yards to the next first down is less than three, then [RUN, PASS, or PUNT],
      - otherwise, [RUN, PASS, or PUNT]

The particular options chosen for each region of the state space (all combined) define a stationary policy for the quarterback which may be evaluated exactly (numerically), as in the preceding subsection. Each "exact" policy evaluation requires roughly a minute to compute. The number of policies defined in this class is 1296, so evaluating all of them can take close to a full day of compute-time. (This is quite slow compared to (i) the computation of the optimal policy and (ii) many of the NDP runs described in the Section 5.)

To provide a means of comparing the policies in this class, we arbitrarily chose a state of interest:

$$i^* \quad \leftrightarrow \quad (x_{i^*} = 80,\ y_{i^*} = 10,\ d_{i^*} = 1) \tag{19}$$

This is the "typical" state at which our team will receive the ball. (We chose this based on our observation that, in real American football, when a team receives a kickoff, they usually manage to get the ball close to the 80 yards-to-go marker.) The "best" policy in the heuristic class is defined to be the one which has the highest expected reward-to-go from i^*. Figure 2 shows the best heuristic policy, along with its corresponding reward-to-go function. The best heuristic expected reward-to-go from i^* is -1.26, which is .32 game points worse than optimal.

It is important to note that significant effort would be required to improve upon the performance of the best heuristic policy. For example, if we included options for running and passing at first down and in both regions at second down, then the number of policies in this class would jump to 10368. The computations for this enhanced class of policies would require close to ten days!

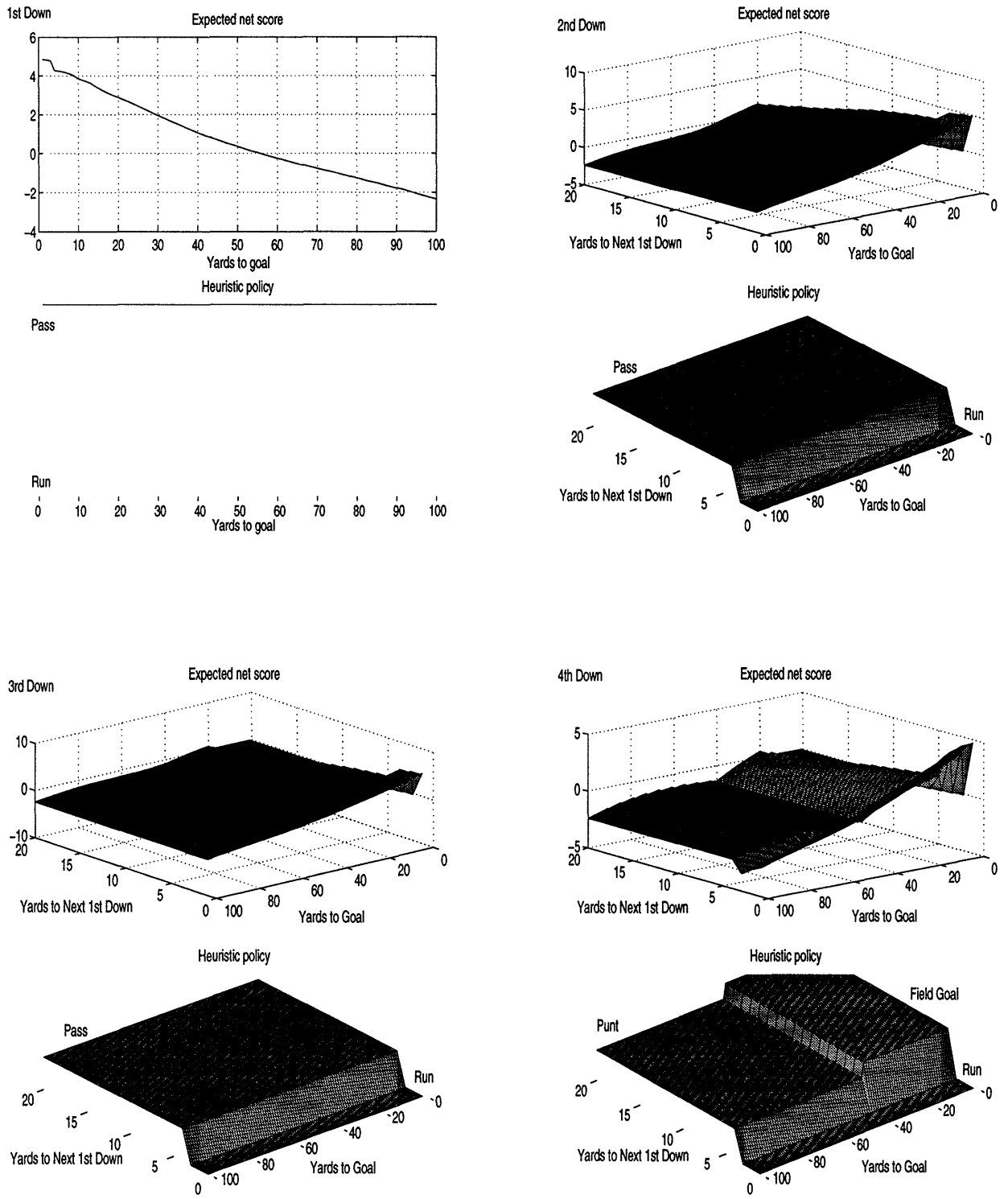

Figure 2: Characterization of the best heuristic policy.

4 Approximate Methods for Football

In this section we provide details about our application of NDP to football which should make our experimental results completely reproducible. This requires that we discuss our approximation architectures and our methods for picking simulation initial conditions.

4.1 The Approximation Architectures

Our goal in this section is to present the architectures we have used for reward-to-go approximation in football. To begin, recall that each state $i \in S$ can be uniquely associated with a triple $(x_i,\ y_i,\ d_i)$, where x_i is the current number of yards to the end-zone (goal), y_i is the current number of yards to the next first down, and d_i is the down number at state i. As functions of i, the quantities x_i, y_i, and d_i can be viewed as features which characterize the state i.

In theory it is possible to use an architecture of the form $\tilde{J}(i,r) = H(x_i, y_i, d_i)$, where H represents a single "global" architecture, such as a multilayer perceptron with three inputs and one output. However, there are problems with this approach. While (for the undiscretized version of football) $J^*(x,y,d)$ is probably smooth or at least continuous as a function of x and y with d fixed, d itself is a truly discrete parameter. Because of this there is no reason to treat d as a "continuous" input to a global architecture. (Why force a continuous architecture to learn discrete relationships?)

With this in mind we decided to use a piecewise continuous architecture, comprised of four independent *subarchitectures*. The appropriate subarchitecture for any state i is determined by d_i. Another way of saying this is that we have *partitioned* the state space such that each state $i \in S$ belongs to the subset characterized by the value of d_i, and there is an independent architecture for each of these subsets. As far as the training algorithms are concerned, this decomposition of the state space has a tangible impact on the details of the training algorithms; however, the changes primarily amount to extra book-keeping. (Each data point is allowed to affect only one subarchitecture.)

In the following subsections we discuss the three main parametric forms we have used in football: MLP, Quadratic, and Recursive. To simply the discussion of these forms without sacrificing clarity, it is convenient to describe the piecewise continuity of the architectures in explicit mathematical notation. The architectures all have the following form:

$$\tilde{J}(i,r) = H(\ \xi^{\Sigma}(i),\ \zeta^{\Sigma}(i),\ r_{d_i}\ ) \tag{20}$$

where H is a generic form for the approximation on the respective subsets, $\xi^{\Sigma}(i)$ is a "standard" feature vector containing scaled versions of x_i and y_i, $\zeta(i) \in R^{n_f}$ is a vector of additional features $(f_1, \ldots, f_{n_f})$, and $r = (r_1, \ldots, r_4)$ is a data structure containing the parameters for the respective subarchitectures. To be more specific, the feature vectors $\xi^{\Sigma}(i)$ and $\zeta^{\Sigma}(i)$ are given by

$$\xi^{\Sigma}(i) \ = \ (\sigma^x_{d_i} \cdot x_i,\ \sigma^y_{d_i} \cdot y_i)' \ \in \ R^2 \tag{21}$$

$$\zeta^{\Sigma}(i) \ = \ (\sigma^{f_1}_{d_i} \cdot f_1(i), \ldots, \sigma^{f_{n_f}}_{d_i} \cdot f_{n_f}(i)\ )' \ \in \ R^{n_f} \tag{22}$$

where

$$\Sigma = \{(\sigma^x_d,\ \sigma^y_d;\ \sigma^{f_1}_d, \ldots,\ \sigma^{f_{n_f}}_d);\quad d = 1, \ldots, 4\}$$

denotes a set of fixed scaling parameters that multiply the input values x_i, y_i, and the feature values $f_l(i)$ for different down numbers d_i. These scaling parameters are not subject to training in approximate policy evaluation. Rather, they are set in advance based on engineering judgment. (The elements of Σ are generally chosen so that $\xi^{\Sigma}(i)$ and $\zeta^{\Sigma}(i)$ lie in the closed interval from zero to one.)

4.1.1 Multilayer Perceptron (MLP)

Multilayer Perceptrons have a history that go way back to the earliest days of artificial neural networks. As a parametric form for function approximation, they are generically comprised of one or more hidden "layers" of sigmoidal activation units. Each layer of activation units is preceded by an affine transformation which fed by the output of the adjacent layer closer to the input layer of the network. The output of the network is formed either by a layer of activation units (whose output levels are constrained) or by a finial affine transformation.

In training, the coefficients of the affine layers are tuned according to a least squares cost criterion to give the best approximation possible. For a comprehensive discussion of neural networks, including the multilayer perceptron, the reader is referred to [8] and [6]. It has been shown elsewhere [5] that, with enough sigmoidal units in a single hidden layer, multilayer perceptrons can be used to approximate arbitrarily closely any continuous function defined on a multi-interval.

The MLP architecture for football reward-to-go approximation uses only a single hidden layer of activation units. Moreover, our MLP's use only the scaled "standard" feature vector ξ^Σ as an input. (This allows us to drop the feature vector ζ^Σ from the notation.) To make the definition explicit, let R be a positive integer equal to the number of hidden nonlinear elements for each multilayer perceptron (on each of the subsets). Let ξ^Σ represent the value of the scaled "standard" feature vector evaluated at some state i, and let $\rho = (W_1, b_1, W_2, b_2)$ be the parameter data structure which applies on the subset d_i, where $W_1 \in R^{R\times 2}$, $b_1 \in R^R$, $W_2 \in R^{1\times R}$, and $b_2 \in R$ are collectively the weights of the multilayer perceptron for subset d_i. The output of the "MLP" architecture is computed as

$$H(\xi^\Sigma, \rho) = W_2\phi(\xi^\Sigma) + b_2 \tag{23}$$

where $\phi(\xi^\Sigma) \in R^R$ is a vector whose elements are computed as

$$\phi_l(\xi^\Sigma) = \tanh(\psi_l(\xi^\Sigma)) \tag{24}$$

and $\psi_l(\xi^\Sigma)$ is the l-th element of the vector $\psi(\xi^\Sigma) \in R^R$, computed as

$$\psi(\xi^\Sigma) = W_1\xi^\Sigma + b_1 \tag{25}$$

In the case study of the following section, we set $R = 20$. In addition, we set the scale factors in Σ to .01. That is, $\sigma_d^x = \sigma_d^y = .01$ for $d = 1, \ldots, 4$. This guarantees that the elements of $\xi^\Sigma(i)$ are in $[0, 1]$ for all states $i \in S$.

4.1.2 Quadratic

The name of this architecture is derived from the fact that the approximation is realized as a quadratic function of the values of the features. The coefficients of this quadratic form are the parameters that get "tuned" in training the approximation. Perhaps the most appealing aspect of this architecture is its simplicity. Computer programs that implement both evaluation and training of the architecture are easy to develop and execute quickly.

Before describing this architecture explicitly, it is useful to define the *quadratic expansion* of a vector. Let $\theta = (\theta_1, \ldots, \theta_{\bar{n}})' \in R^{\bar{n}}$ and let $\hat{n} = \frac{(\bar{n}+1)(\bar{n}+2)}{2}$. Let $Q(\theta)$ be the *quadratic expansion of* θ, where $Q : R^{\bar{n}} \mapsto R^{\hat{n}}$ is defined by the map

$$Q(\theta) = (1,\ \theta_1, \ldots, \theta_{\bar{n}},\ (\theta_1)^2, \theta_1\theta_2, \ldots, \theta_1\theta_{\bar{n}},\ (\theta_2)^2, \theta_2\theta_3, \ldots, \theta_2\theta_{\bar{n}},\ \ldots, (\theta_{\bar{n}})^2)' \in R^{\hat{n}} \tag{26}$$

As before, let ξ^Σ represent the value of the scaled "standard" feature vector evaluated at some state i, and let ρ be the data structure for the parameters which apply on the subset d_i. The "Quadratic" architecture for reward-to-go approximation is given by $H(\xi^\Sigma, \rho) = \rho' Q(\xi^\Sigma)$. Again, as is implicit in the notation, the Quadratic architecture uses only the the "standard" feature vector.

The scale factors in Σ for the case study were chosen as: $\sigma_d^x = .01$ for all d; $\sigma_d^y = .05$ for $d = 2, 3, 4$; and $\sigma_1^y = 0$. To see why we use $\sigma_1^y = 0$, recall that at first down the number of yards to go until the next first down, y, is uniquely determined by the number of yards to the end-zone, x. In training this architecture, we will typically just compute the linear least squares parameter exactly using pseudo-matrix inversion. There is no benefit to including y as an input at first down.

4.1.3 Quadratic with Feature Recursion (Recursive)

Because the Quadratic architecture uses relatively few parameters and is a form which admits an exact optimal solution in training, reward approximations using this architecture can be evaluated and trained very quickly. Unfortunately, for the same reasons, the Quadratic architecture has a limited ability to accurately approximate

very irregular functions. The richness of the Quadratic architecture is limited by the number of elements of the feature vector ξ^{Σ}. If there were more features, then the number of coefficients for the architecture (and thus the architecture's power to approximate reward) would increase sharply. The "Recursive" architecture that we describe here is essentially the Quadratic architecture of the preceding subsection with the additional twist that after every policy update a new feature function is added to the list of features used by the architecture. The new features are themselves the past approximations to the optimal reward-to-go function. The Recursive architecture was originally proposed as a method to decrease the amplitude of the "oscillations" that are usually exhibited in the limit in both API and OPI.

To give a mathematical description of the architecture, let ξ^{Σ} represent the value of the scaled "standard" feature vector evaluated at some state i, and let ρ be the data structure for the parameters which apply on the subset d_i. Suppose that μ^k is the current policy (in either API or OPI) and that we are trying to approximate J^{μ^k}. Let $\{\tilde{J}(\cdot, r^{k-1}), \ldots, \tilde{J}(\cdot, r^{k-n_p})\}$ represent the approximations of the reward-to-go functions for the preceding n_p policies. With the proper scalings, these are the elements of the vector of "additional" features ζ^{Σ}:

$$\zeta^{\Sigma}(i) = (\sigma_{d_i}^{f_1} \cdot \tilde{J}(i, r^{k-1}), \ \ldots \ , \ \sigma_{d_i}^{f_{n_p}} \cdot \tilde{J}(i, r^{k-n_p}) \ ) \tag{27}$$

The Recursive architecture for reward-to-go approximation is given by $H(\xi^{\Sigma}, \zeta^{\Sigma}, \rho) = \rho' Q(\xi^{\Sigma}, \zeta^{\Sigma})$, where $Q(a, b)$ is the quadratic expansion of the elements of both a and b. To make the architecture well-defined for the first n_p iterations, we initialize ζ^{Σ} with zeros, so that the earliest iterations tend to imitate the Quadratic architecture.

Although this architecture is basically "quadratic", there are significant complexities involved with its implementation. The architecture is inherently computationally intense. The reason for this is that *all* of the past approximations of reward-to-go are needed to evaluate the architecture, even if $n_p = 1$. For example, to evaluate the approximation $\tilde{J}(i, r^k)$, one of the features needed in the computation is $\tilde{J}(i, r^{k-1})$. Similarly, the evaluation of $\tilde{J}(i, r^{k-1})$ requires the evaluation of $\tilde{J}(i, r^{k-2})$, and so on. This difficulty has an important impact on the details of implementing OPI with this architecture. (To be discussed shortly.) Despite its computational complexity, the Recursive architecture does have is merits. One of these is the fact that the number of features does not increase as a function of the policy iteration number. That is, as long as n_p is held constant for all policy iterations, the dimension of parameter vector is fixed at roughly $(n_p+3)(n_p+4)/2$. In other methods of automatic feature-extraction, it is not always so clear how to prevent an explosion of the number of features.

Because relatively few policies are ever generated in API, it is practical to the Recursive architecture as described above. With OPI, for the computational issues described above, this architecture would be impractical. A simple modification, which allows the use of the Recursive architecture in OPI, is to not perform feature recursions at every iteration; rather, compute recursions infrequently keeping track of "good" policies in between. Use of the Recursive architecture in OPI is formally described below.

1. Start with an initial stationary policy μ^0.

2. At the t-th iteration, if $(t-1)$ is a multiple of the desired number of policies per evaluation, then

   (a) generate a large number of sample trajectories under μ^{t-1}; estimate reward-to-go by computing sample means. Store the results for post-processing.

   (b) If the number of this sample mean calculation is a multiple of the desired number of mean-calculations per feature recursion, then look back over the last batch of sample means, identify the best policy, and incorporate as a new feature the reward-to-go approximation that induced the best policy.

3. Generate a small number of sample trajectories to get the training data for the policy μ^{t-1}. Store the data as complete (and distinct) sample trajectories, keeping the data in the form (i_k, g_k), where i_k is the state at the k-th stage of the trajectory with corresponding transitional reward g_k.

4. Use the sample data to compute the "best" approximation parameter r^{t-1}. Cycle through the sample data buffer updating the parameter according to either (8) or (10) with a stepsize given by (15), where $\gamma_{ra} = 0$.

5. Compute μ^t using (14). That is, $\mu^t = G(r^{t-1})$.

6. Either stop or go back to step 2.

The scale factors in Σ for the case study for the Recursive architecture were chosen as: $\sigma_d^x = .01$ for all d; $\sigma_d^y = .05$ for $d = 2, 3, 4$; $\sigma_1^y = 0$; and $\sigma_d^{f_k} = 1/7$ for all d and $k = 1, \ldots, n_p$.

4.2 Simulation: sampling initial conditions

In the case study of the following section, we have used two different rules for picking initial conditions for simulation. Both are random, using stationary probability distributions that emphasize different parts of the state space. The first rule (Rule 1) is the one used most often. It selects initial conditions corresponding to just about any field position our team is likely to see. The second rule, which highlights the effect of different sampling methods, is used only once in our case study. It selects initial conditions "close" to $x = 80, y = 10$, and $d = 1$.

Rule 1

1. With probability .35 start at fourth down.
   - Choose yards to go, x, uniformly from 1 to 100.
   - Choose yards to next first down, y, uniformly from 1 to x.
2. With probability .30 start at third down.
   - Choose x uniformly from 1 to 100.
   - Choose y uniformly from 1 to x.
3. With probability .25 start at second down.
   - With probability .25, choose x uniformly from 1 to 50.
   - With probability .75, choose x uniformly from 51 to 100.
   - Choose y uniformly from 1 to x.
4. With probability .10 start at first down.
   - With probability .25, choose x uniformly from 1 to 75.
   - With probability .75, choose x uniformly from 76 to 100.
   - If $x < 10$, choose $y = x$. Else, choose $y = 10$.

Rule 2

1. Pick x uniformly from 75 to 85.
2. Pick y uniformly from 1 to 10.
3. With probability .45 start at 1st down. With probability .25 start at 2nd down. With probability .15 start at 3rd down. With probability .15 start at 4th down.

5 Experimental Results

Figure 3 shows the various NDP-type algorithms we have tested in the football case study. Each row in the table corresponds to

1. a particular scheme for updating policies: API or OPI,
2. an approximation architecture: MLP, Quadratic, or Recursive, and
3. a training algorithm: TD(λ), Bellman Error, or SVD.

In the columns labeled "Training method(s)" and "Details" are shown the parameter settings for the algorithm, some of which may vary for particular trials. The table shows in **bold** the parameter settings which are best with respect to *sample* expected reward-to-go from the typical initial condition i^*. (The sample evaluation is based on 8000 independent sample trajectories, all starting from i^*.) *Exact* evaluations of reward-to-go from i^* for the best runs are shown in the column labeled "Exact RTG of Best". The last column of the table gives the figure number for each algorithm's corresponding trials. In general, for the OPI runs, unless stated otherwise in the table, sample evaluations from i^* are computed every 200 policy updates. For all of the algorithms in the table, we used the "Rule 1" sampling method (for picking simulation initial conditions.)

The algorithmic parameters shown in the table represent the best settings we could find based on a considerable amount of tinkering with the methodologies. We tried to be even handed in this process, not wanting to "sabotage" any particular algorithm. Our goal was to be both comprehensive in scope and objective in evaluation. Regarding the runs of Figure 8, these were originally intended to be API-type runs; the table entry $\gamma_{ra} = .001$ is not a mistake. (Normally, we would have $\gamma_{ra} = 0$ and $\gamma_{ro} \neq 0$ for OPI. We discovered, while tinkering with the parameters, that 1 cycle through a data set of 64000 samples is not enough to make sufficient progress toward the best (approximate) evaluation of reward-to-go. Consequently, these runs are really more like OPI than API.)

Results for the case study are shown in Figures 4 through 11. The figures all generally follow the same format. For each experimental run, we plot

1. the sample evaluations of reward-to-go from i^* as a function of policy number, and
2. first down error from optimal of
   (a) the approximation that yielded the best policy
   (b) the exact evaluation of the best policy

   as a function of x, the number of yards to the goal.

In some cases we also show the exact evaluation of the rollout policy based on (i) the best policy of the trial and (ii) 20000 "rollouts" per state/action pair. The axis scales in the figures are held constant to aid visual comparisons. (For some the the runs the traces go "off-scale.") Results for the API methodology are shown in Figures 4 through 6. Figures 7 through 11 are devoted to OPI.

Figures 12 and 13 show results that augment the main body of the case study. In Figure 12 we show results for both API (SVD) and OPI (Kalman Filtering) for the Quadratic architecture using "Rule 2" for picking simulation initial conditions. In Figure 13, we show the evolution of the weights of the reward-to-go approximation for various NDP-type runs as a function of policy number. The plots on the left correspond to API, while the plots on the right correspond to OPI.

Scheme	Architecture	Training Method	Details	Exact RTG of best	Fig
API	MLP	TD(0, .5, **1**) and Bellman Error	$\gamma_0 = .05, \gamma_{ra} = 5 \cdot 10^{-4}, \gamma_f = 1 \cdot 10^{-6}$ (100 cycles through buffer of 10k trajectories)/policy	-.954	4
API	Quadratic	SVD	4k trajectories/policy **30k trajectories/policy**	-1.172	5
API	Recursive	SVD	30k traj/policy with $n_p = 2$, 30k traj/policy with $n_p = 5$, **45k traj/policy with $n_p = 7$**	-.957	6
OPI	MLP	TD(0, .5, **1**) and Bellman Error	$\gamma_0 = .002, \gamma_{ro} = 1 \cdot 10^{-5}, \gamma_f = 1 \cdot 10^{-5}$ (1 cycle through buffer of 1 traj)/policy	-1.022	7
OPI	MLP	TD(0, .5, **1**) and Bellman Error	$\gamma_0 = .002, \gamma_{ra} = .001, \gamma_f = 1 \cdot 10^{-5}$ (1 cycle through buffer of 64k traj)/policy	-1.042	8
OPI	Quadratic	TD(0, .5, 1) and **Bellman Error**	$\gamma_0 = .001, \gamma_{ro} = 5 \cdot 10^{-6}, \gamma_f = 1 \cdot 10^{-5}$ (1 cycle through buffer of 1 traj)/policy	-1.161	9
OPI	Quadratic	SVD $\lambda = 1$, **.95**	No extra details.	-.988	10
OPI	Recursive	TD(0, **.5**, 1) and Bellman Error	$\gamma_0 = .01, \gamma_{ro} = 2 \cdot 10^{-4}, \gamma_f = 1 \cdot 10^{-5}$ (1 cycle through buffer of 1 traj)/policy 200 policies/evaluation 200 evaluations/recursion	-1.006	11

Figure 3: Table of experimental runs for the football case study. All of these experiments used Rule 1 for picking simulation initial conditions. The best run for each experiment is shown in bold. Note that the optimal reward to go from i^* is -.9449.

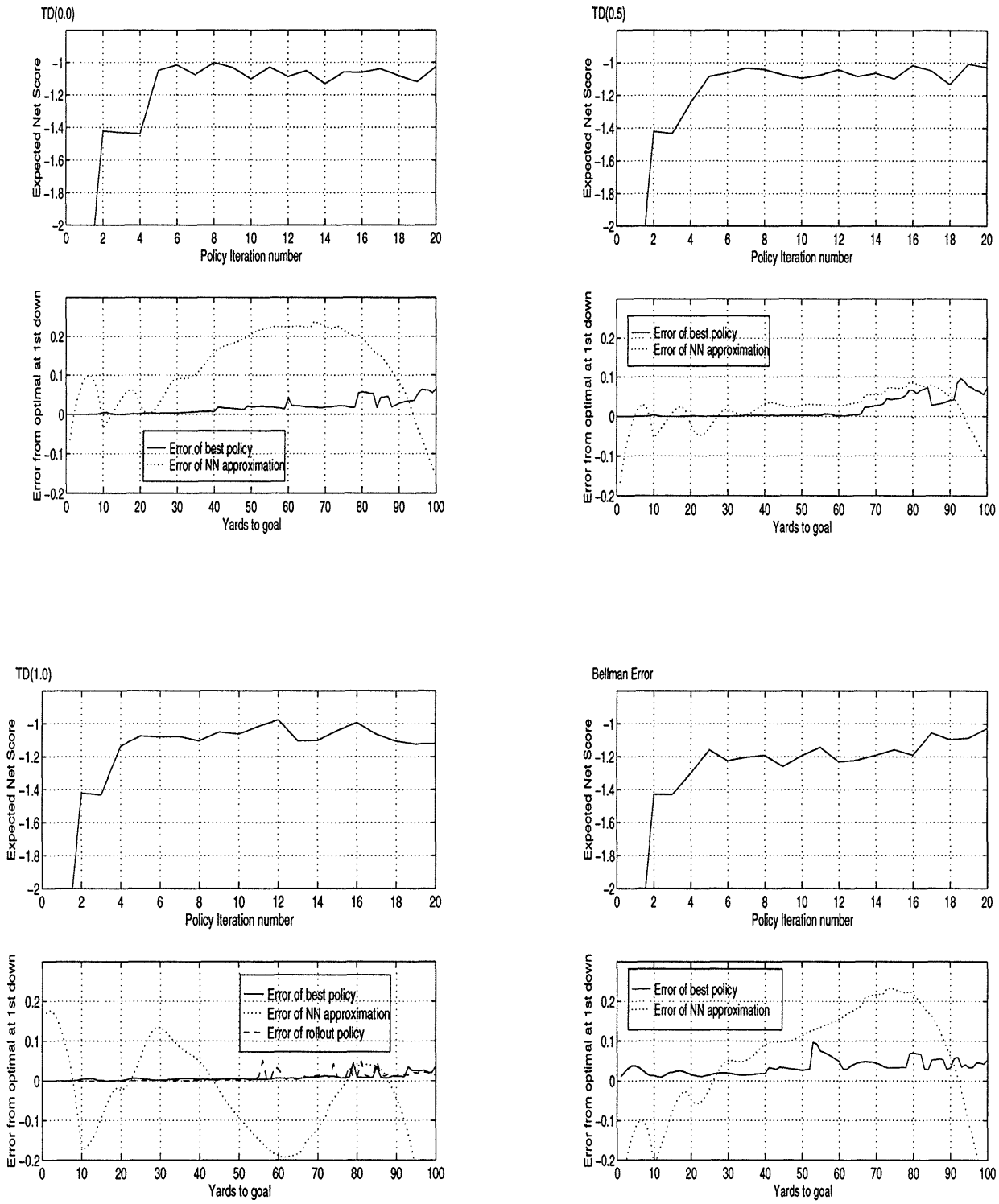

Figure 4: API with the MLP architecture using the TD(λ) ($\lambda = 0, .5, 1$) and the Bellman error methods with 100 cycles through 10000 sample trajectories in training per policy.

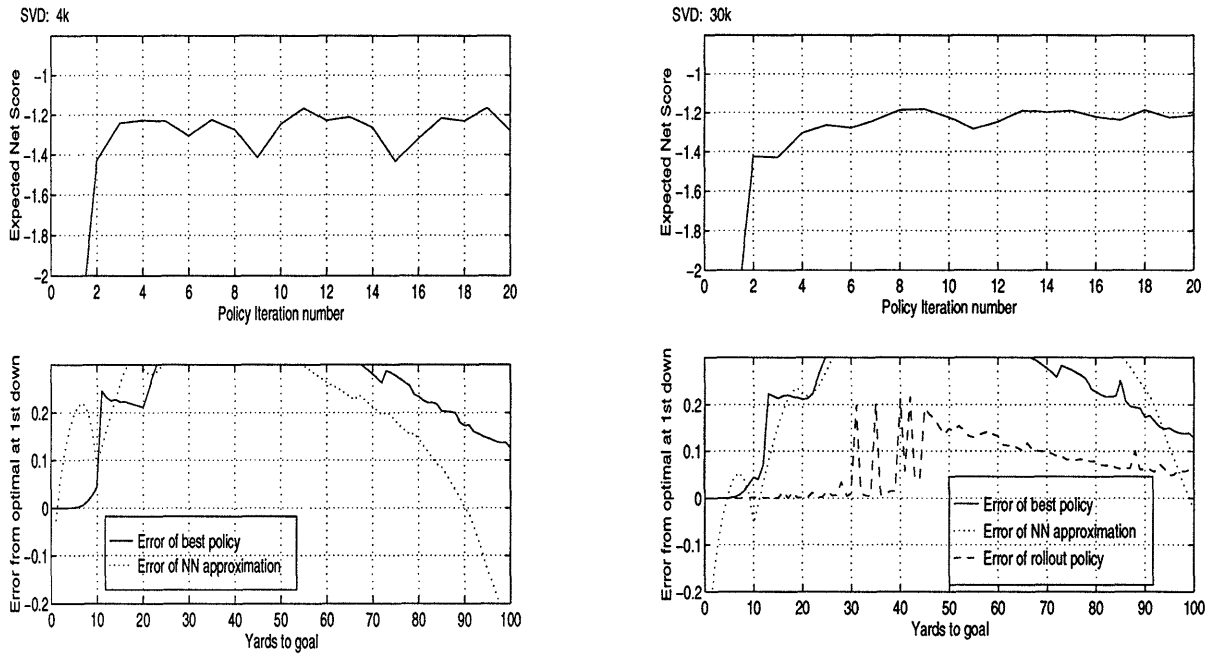

Figure 5: API with the Quadratic architecture using the SVD method of training: 4k and 30k sample trajectories per policy.

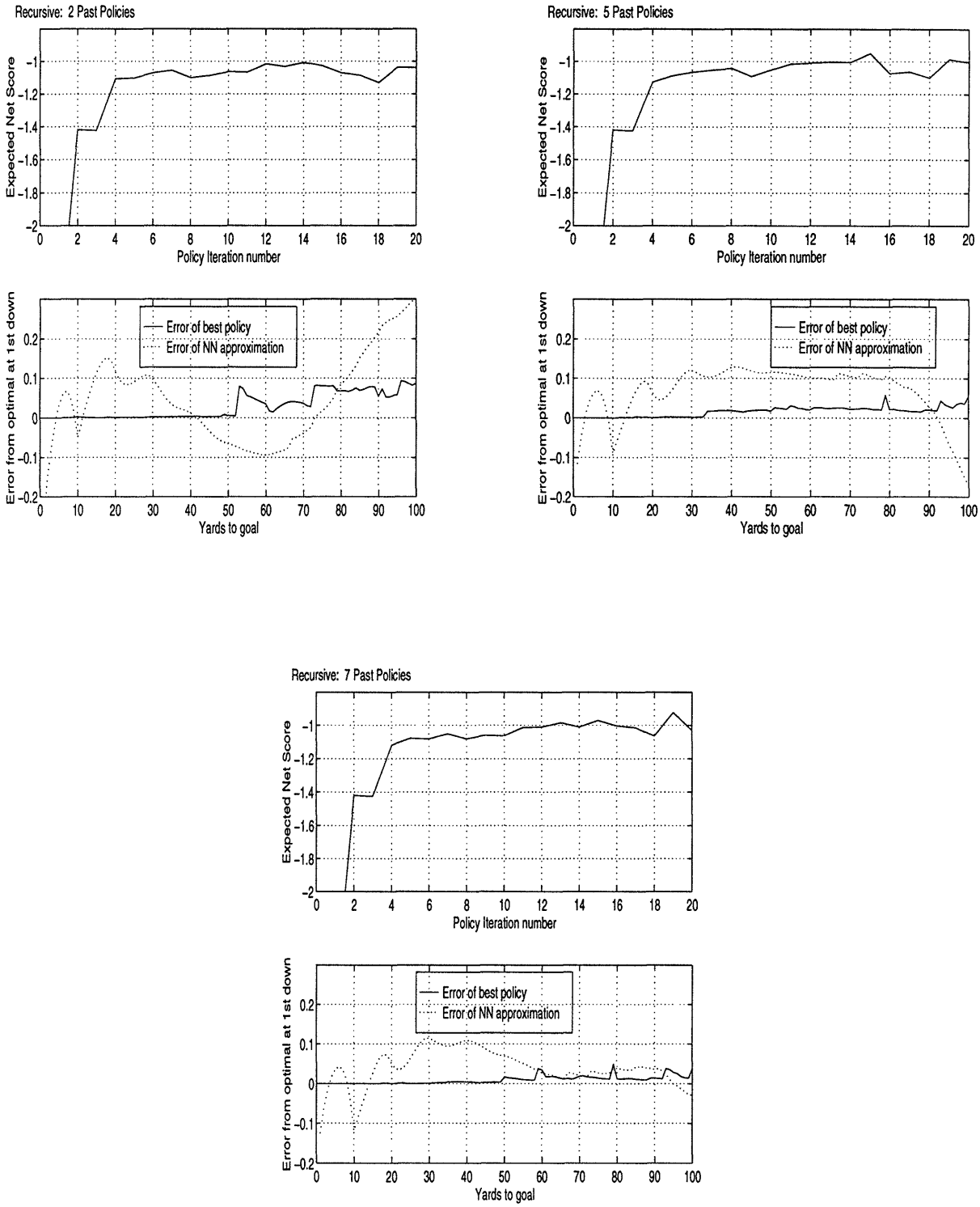

Figure 6: API with the Recursive architecture using the SVD method of training: 2 past policies (30k sample trajectories), 5 past policies (30k sample trajectories), 7 past policies (45k sample trajectories).

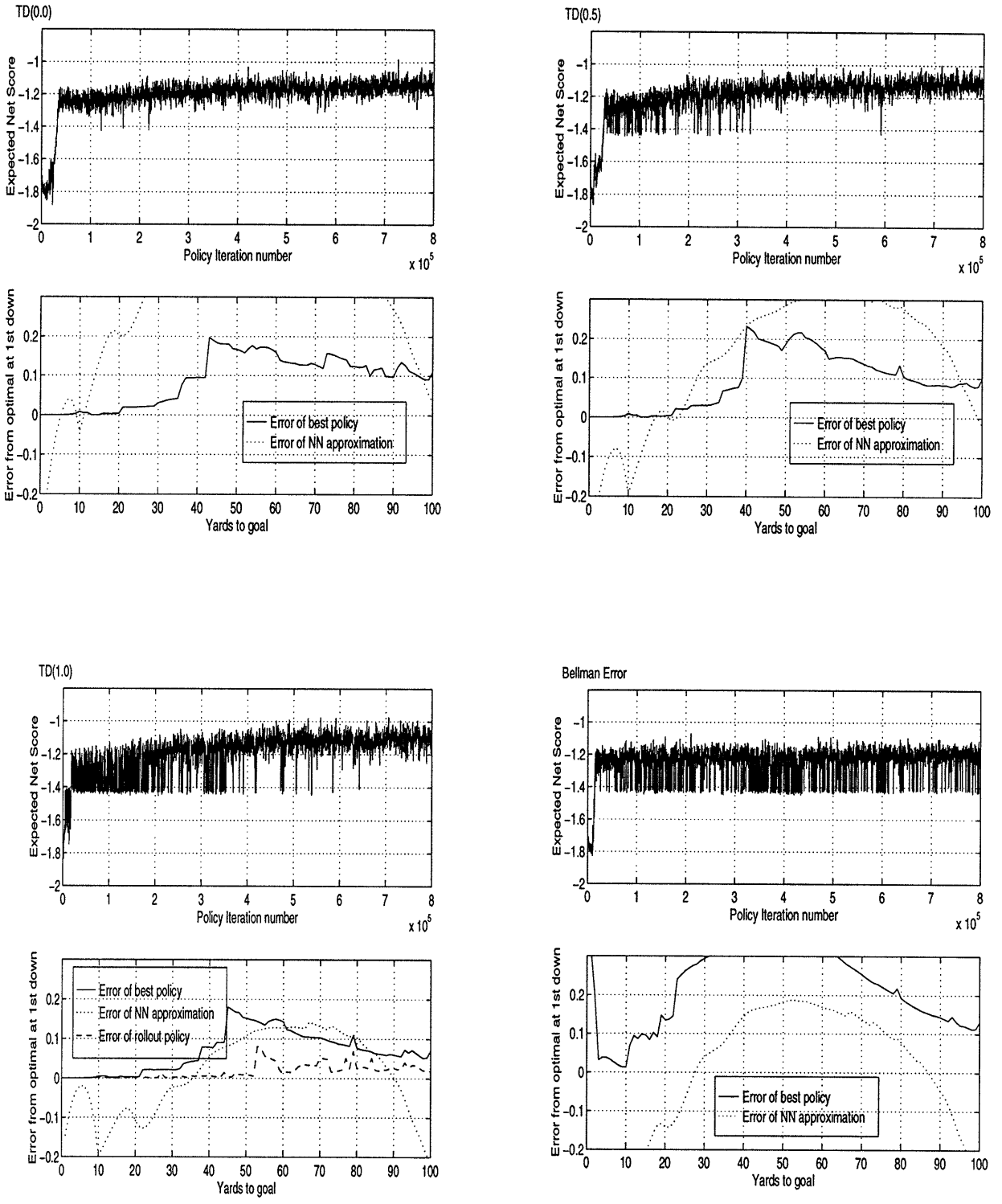

Figure 7: OPI with the MLP architecture using the TD(λ) ($\lambda = 0, .5, 1$) and Bellman error methods of training with one sample trajectory per policy.

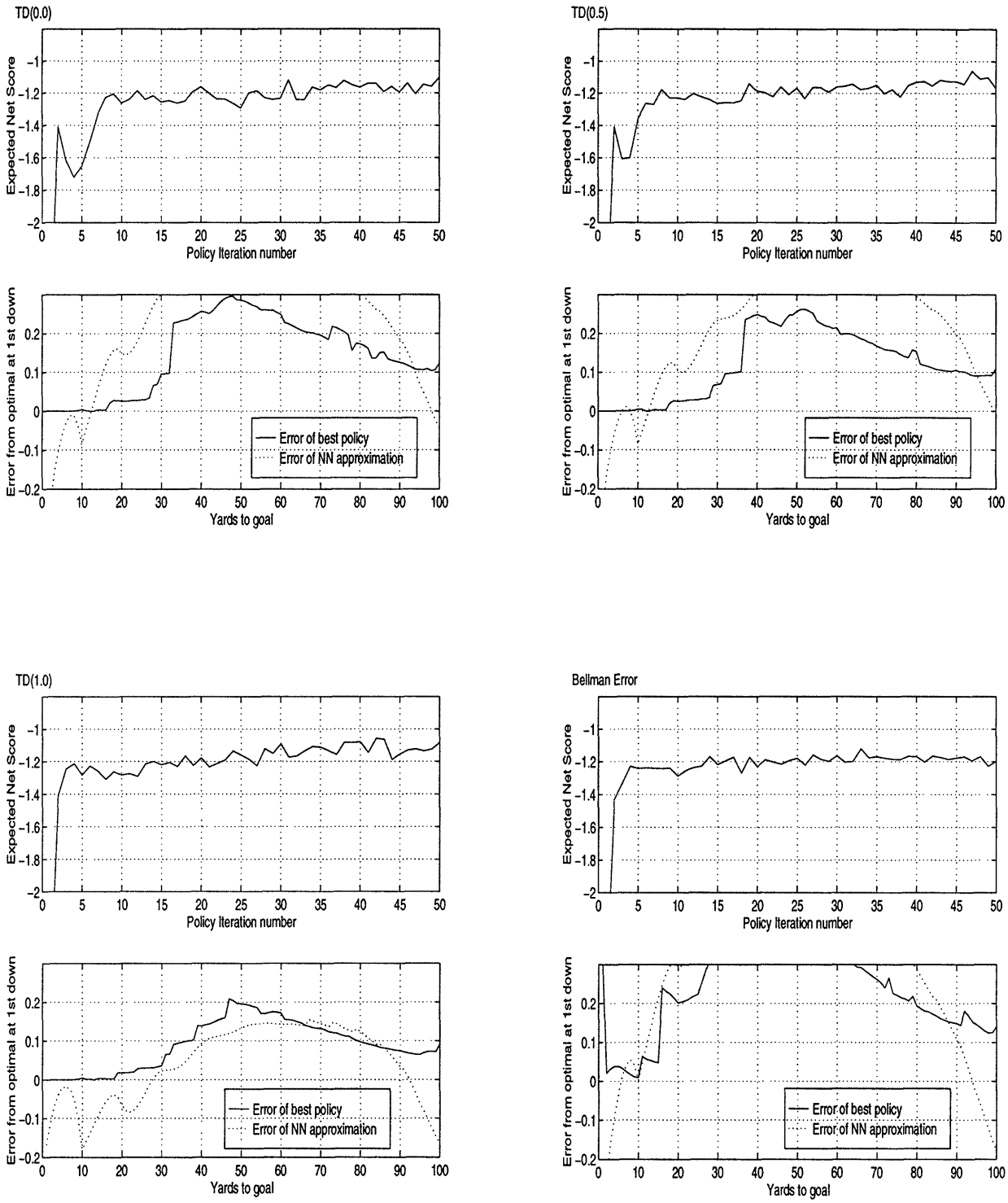

Figure 8: OPI with the MLP architecture using the TD(λ) ($\lambda = 0, .5, 1$) and Bellman error methods of training with 64k sample trajectories per policy.

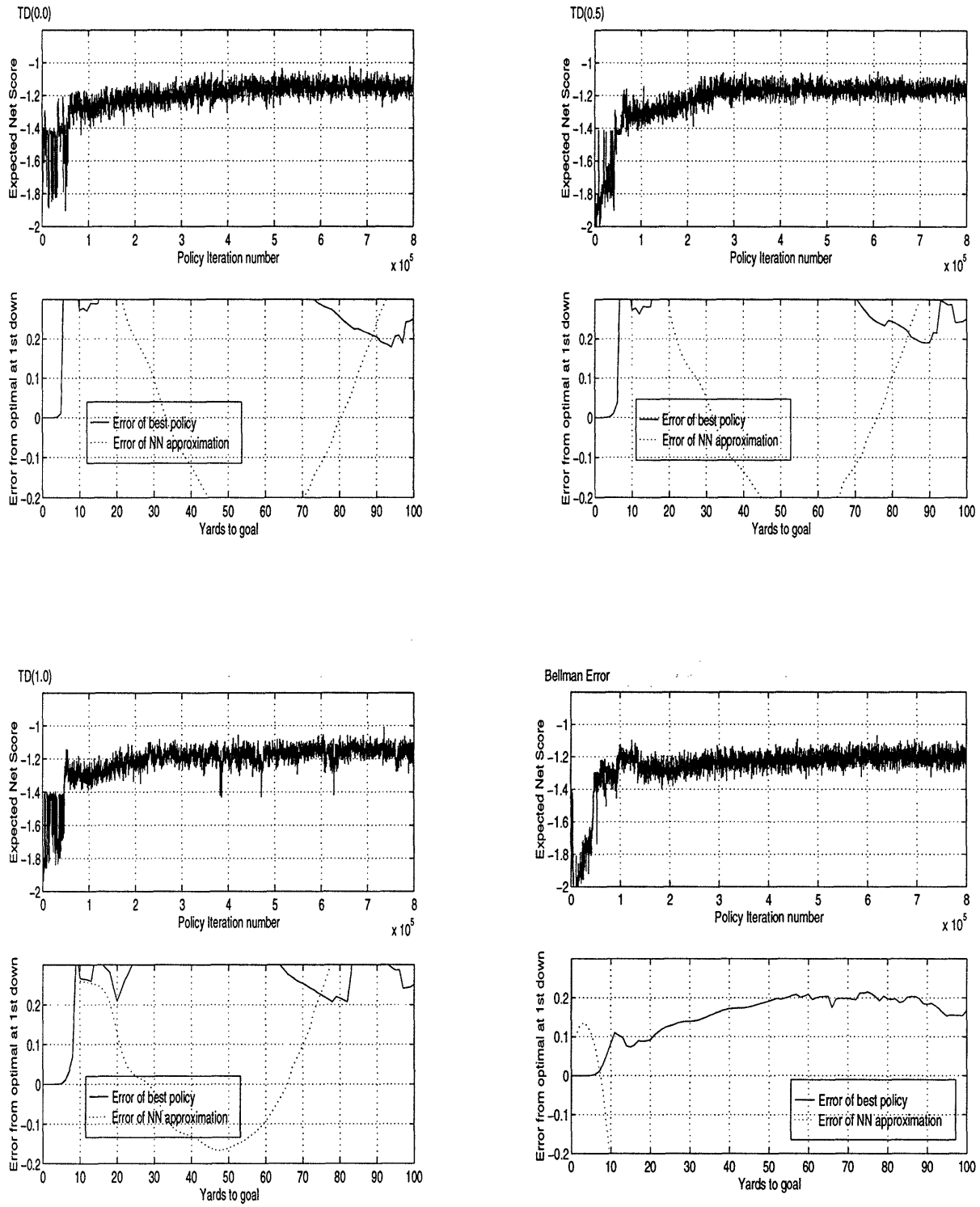

Figure 9: OPI with the Quadratic architecture using the TD(λ) ($\lambda = 0, .5, 1$) and Bellman error methods of training with one sample trajectory per policy.

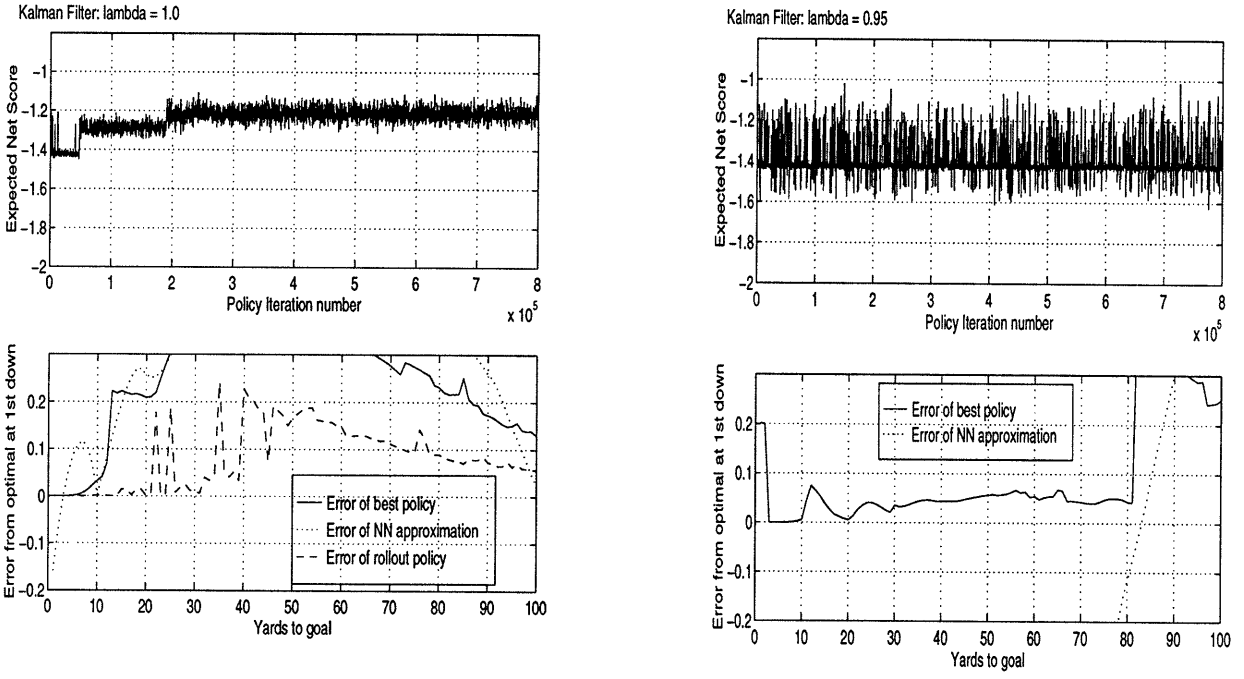

Figure 10: OPI with the Quadratic architecture using the SVD (Kalman Filtering) method of training: $\lambda = 1, .95$.

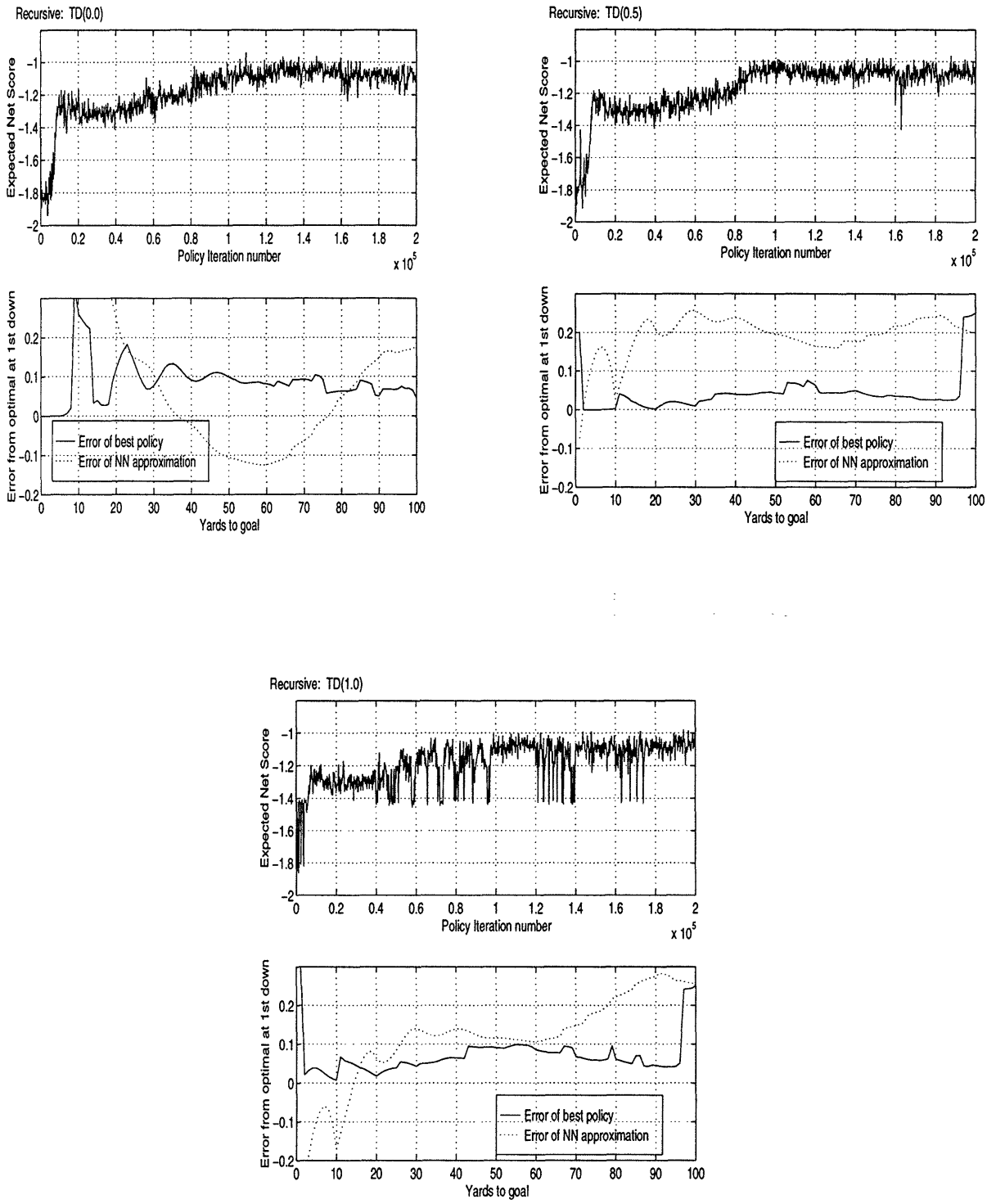

Figure 11: OPI with the Recursive architecture using the TD(λ) method of training: $\lambda = 0, .5, 1$, with one sample trajectory per policy.

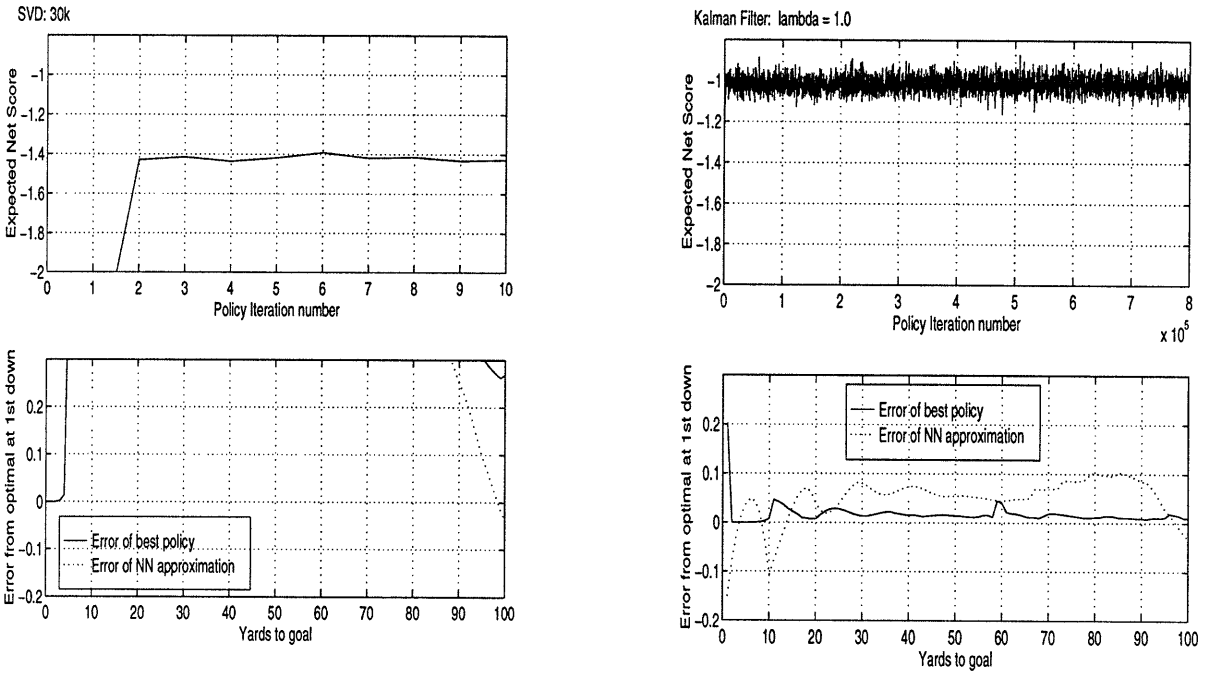

Figure 12: API (with SVD) and OPI (with Kalman Filter) using the Quadratic architecture. We used here "Rule 2" for picking initial conditions for the simulation. Otherwise, the algorithmic settings are the same of those used in the earlier runs. Notice that Rule 2 significantly degrades performance for API and significantly improves performance for OPI.

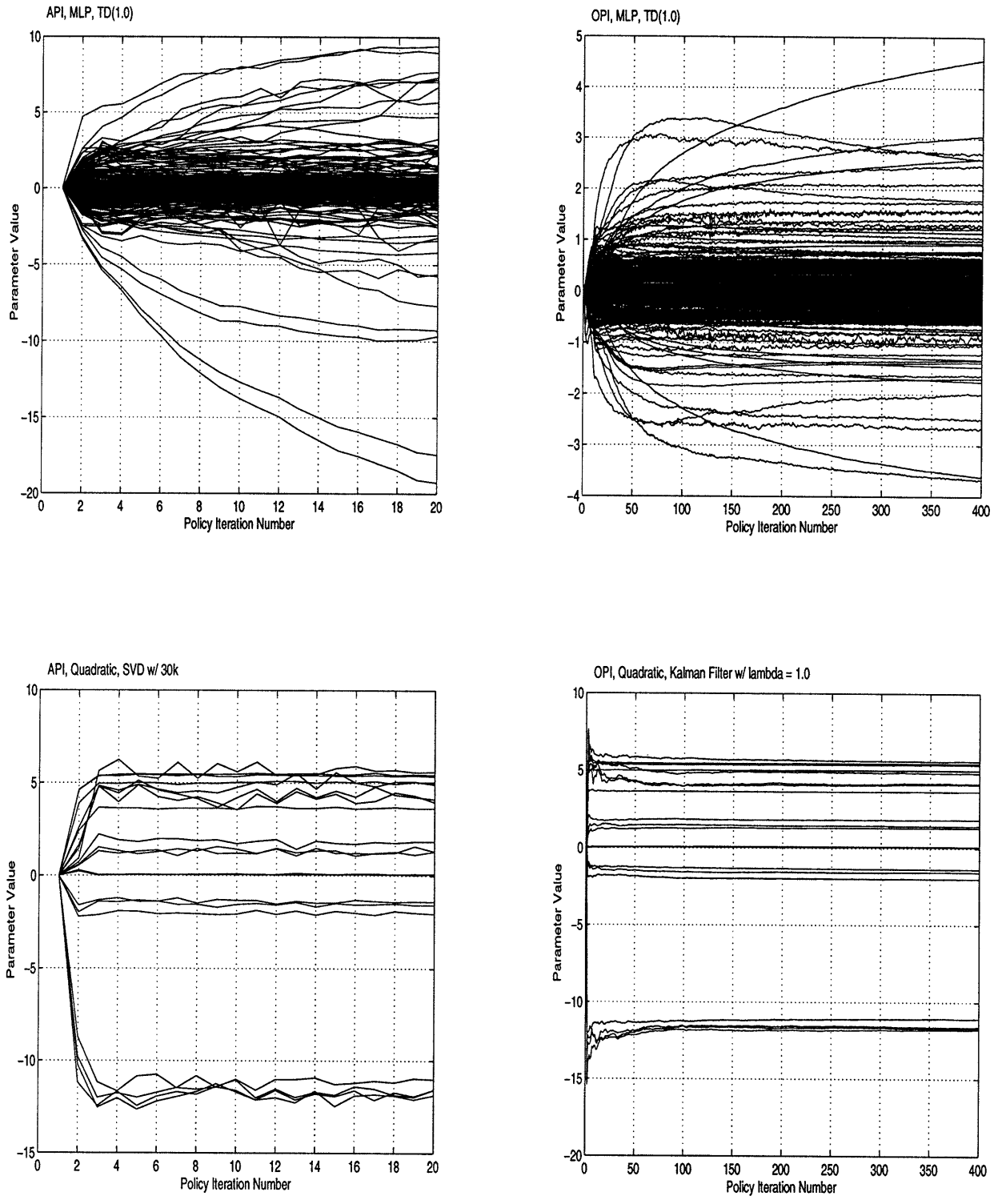

Figure 13: Evolution of approximation weights for selected trials from the case study.

6 Discussion and Conclusions

Our observations from the case study are as follows.

1. Regarding API:
   (a) This methodology (along with the MLP architecture) has yielded the best results.
   (b) For the best API runs, the first-down approximations of the reward-to-go are fairly close to the optimal reward-to-go function. *Exact* evaluations of the best suboptimal policies are extremely close to the optimal reward-to-go function.
   (c) In general, the more complex the architecture, the better the results (at the expense of longer computation times.) The existence of local minima in training the MLP architecture does not seem to have impacted the results.
   (d) When using TD(λ) to train the approximations, we found that $\lambda = 1$ gave the best results (compared to $\lambda = 0$ and $\lambda = .5$), although not by a very great margin. The Bellman Error method gave the worst results.
   (e) API seems to be very sensitive to the method used to pick initial conditions for the simulation. (More experimentation along these lines is necessary.)
   (f) The "oscillatory" limiting behavior of API can be seen in the means plots of Figures (4) through (6), as well as in the weights plots Figure 13.
   (g) As an imitation of exact Policy Iteration, API is not totally convincing. In particular, the means plots are not always monotonically increasing. On the other hand, usually only 10 iterations are required to obtain "peak" performance. (This compares favorably with the 6 policy iterations that are required to solve the problem exactly.)
2. Regarding OPI:
   (a) Despite the methodology's lack of theoretical guarantees, OPI *can* find policies whose sample evaluations from i^* are competitive.
   (b) In the end, even for the "best" OPI runs, the approximations of the reward-to-go function at first down are not very close to the optimal reward-to-go function. The exact evaluations of the reward-to-go function are also not very close to optimal. In general, these exact evaluations have strange properties: (i) while there are prominent exceptions, it is often the case that for states where $x < 20$ the exact evaluation is fairly accurate, but extremely inaccurate for $x > 20$, and (ii) in some cases there is a precipitous drop in performance for values of x just greater than 80.
   (c) In general, the more complex the architecture, the better the results (again, at the expense of longer computation times.)
   (d) Regarding the incremental training algorithms used in conjunction OPI, there is no clear winner. For the MLP architecture, the best results were obtained with TD(1). For the Quadratic architecture, Kalman filtering with $\lambda = .95$ worked best (although the resulting approximations look very strange). For the Recursive architecture, the Bellman Error method was best.
   (e) The "oscillations" in policy space predicted by the "greedy subsets" heuristic understanding of OPI is exhibited in the figures. (We observe corresponding convergence of the parameters for the reward-to-go approximations.)
   (f) OPI will very quickly find a policy which is significantly better than the initial policy. On the other hand, to come up with policies that are close to optimal, it is necessary to let the algorithm run for a very long time. (There is one exception: OPI with the Quadratic architecture using $\lambda = .95$.)
   (g) In light of the strange nature of the evaluations of reward-to-go, it is not completely clear that the peak performances obtained in OPI are not just statistical flukes. The difficulty of evaluating the many policies that go by is significant.
3. Football is a nice testbed for NDP:

(a) It is a truly medium-scale example. Because we can compute the optimal solution to the problem, which have a yardstick for comparisons of the alternative methods.

(b) Football is not totally trivial, as evidenced by (i) the relatively poor performance of the approximate methods (API and OPI) and (ii) the poor performance of the heuristic policy discussed in Section 3.2.

(c) Finally football is fairly intuitive. This aids in the implementation and debugging of the methodologies and also provides a means for interpreting the results.

A few words about computation times are in order. Our emphasis in this case study was to determine the limits of performance for the competing forms of NDP. As a result, we were not careful to keep records of run-times. Nonetheless, the following comments should be useful. As a rule, the trials which gave the best results took the longest to complete. In general, the amount of time required for a particular algorithm depends on the complexity of the approximation architecture. Holding everything else fixed, API and OPI take roughly the same amount of time to complete. (One is not clearly always faster than the other.) Except for the experiments with the Recursive architecture, the "good" API and OPI runs took significantly *less* time than the exhaustive search through the heuristic class of policies discussed in Section 3.2. On the other hand, for this model of football, the exact computation of the optimal policy required considerably less time than the fastest of the approximate methods (by more than an order of magnitude.)

As for the future, football can provide a vehicle for many more interesting case studies. In particular, by adding new features to the model and enhancing its realism, the dynamic optimization problem can easily become intractable, providing a truly large-scale testbed for NDP. One significant change to the model would involve allowing the defense to make play selections in addition to the offense. The case of an intelligent opponent would be very interesting to explore computationally. Other case studies may involve alternative methods of NDP, most notably Q-learning.

Appendix: The Rules of the Game

Our model describes a simplified version of football. We consider one offensive drive in the middle of an infinitely long game from the perspective of the offense. The objective is to maximize the score earned during "our" team's current offensive drive *offset* by the opponent's expected score from the position at which he receives the ball. The state of "our" team is characterized by three quantities: x = number of yards to goal (field position), y = yards to go until next first down, and d = down number. The offensive drive ends (terminates) whenever:

1. A new first down fails to be earned after four plays. (For the benefit of those who are not familiar with American football, the offense always receives the ball at first down, i.e. $d = 1$. At first down, a value for y is set which is the number of yards the offense must move forward in four plays (called downs). If the offense is successful in this endeavor, then they receive a *new* first down, i.e. $d = 1$, along with a new value for y. Otherwise, each play results in d being incremented by one, along with with y being decremented by the number of yards earned during the previous play. New y-values at first down are always computed according to $\min\{10, x\}$.)

2. A touchdown is scored. (This occurs whenever the offense is able to achieve $x \leq 0$.)

3. A run attempt is fumbled, a pass is intercepted, or whenever a punt or field-goal attempt is made.

The playing field here is discretized, with x and y taking on integer values. It is important to note that states in which $y > x$ are impossible and are thus not considered to be part of the state space. Moreover, since there are no penalties in this game, there can be only one value of y associated with each value of x at first down. All totaled, there are 15,100 states for which the quarterback (the team's decision maker) must have some control action in mind.

The outcome of a given drive is random, dependent on the quarterback's strategy and the transition probabilities associated with the various play options and points in state-space. The quarterback always has four play options to choose from: 0- Run, 1- Pass (attempt), 2- Punt, and 3- Field-goal (attempt). Transition probabilities and further details of the rules are spelled out in the following paragraphs. These data should make the results in this paper completely repeatable.

Play option 0: Run attempts The number of yards which the ball moves (toward the goal) during a run attempt is given as the outcome of a Poisson random variable (with mean 6.0) *minus* two. Negative gain during the run is entirely possible, although not probable. The run attempt may (with probability .05) result in a fumble, in which the opposing team recovers the ball at the new ball position.

If the run attempt is not fumbled and results in $x \leq 0$, then the current drive ends with a touchdown. If the run attempt is fumbled in the opponent's end zone (i.e. $x \leq 0$), then the opponent recovers the ball at $x = 20$. (When the "opponent recovers the ball at $x = 20$", the opponent has 80 yards to go to reach his goal.) If the run attempt is not fumbled and results in $x > 100$, then the opposing team scores a safety and "recovers the ball at $x = 20$". Even worse, if the run attempt is fumbled in the offense's own endzone (i.e. $x > 100$), then the drive ends with the opposing team scoring a touchdown.

Play option 1: Pass attempts Pass attempts can result in one of four possibilities: pass intercept (with probability .05), pass incomplete (with probability .45), quarterback sack (with probability .05), or pass complete. If the pass is either completed or intercepted, the amount by which the ball moves is given as the outcome of a Poisson random variable (with mean 12) *minus* two. Incomplete passes result in no movement of the ball. If the quarterback is sacked, the sack occurs a number of yards behind the initial position which is the outcome of a Poisson random variable with mean 6.0.

If the pass attempt is completed and results in $x \leq 0$, then the current drive ends with a touchdown. If the pass attempt is intercepted in the opponent's end zone, then the opponent recovers the ball at $x = 20$. If the pass attempt is completed and results in $x > 100$, then the opposing team scores a safety and recovers the ball at $x = 20$. Even worse, if the pass attempt is intercepted in the offense's own endzone, then the drive ends with the opposing team scoring a touchdown.

Play option 2: Punt attempts Punts always result in the ball being turned over to the other team. Unless punt distance exceeds the distance to the goal, the amount by which the ball moves after a punt attempt is given as six times the outcome of a Poisson random variable (with mean 10.0) *plus* six. If this distance is greater than the distance to the goal, then the punt is considered out-of-bounds, and the opposing team receives the ball at $x = 20$.

Play option 3: Field goal attempts The probability of a successful field-goal attempt is given as $\max\{0, (.95 - .95x/60)\}$. If the field goal attempt is successful, the opponent receives the ball at $x = 20$. However, if the field-goal attempt fails, the opponent picks up the ball wherever the field-goal attempt was made.

Drive score and Expected net score If either team scores a touchdown, then the immediate reward for that team is 6.8 points. (Note that with the above setup it is possible for either team to score a touchdown. If the opposing team scores a touchdown, then the offense receives a net score of -6.8.) The ".8" reflects the fact that the probability of a successful extra-point is taken to be roughly .8. If a successful field-goal attempt is made, then the immediate reward is 3 points. If a safety is scored against the offense, then the immediate reward received (by the offense) is -2.0. When the offense's current drive is over, an amount equal to the opposing team's *expected* score (for their drive) is subtracted from the immediate reward received by the offense. The opposing team's expected score is a function of where they receive the ball at the beginning of their drive: $6.8x/100$.

References

[1] D. P. Bertsekas. A counterexample to temporal differences learning. *Neural Computation*, 7:270–279, 1995.

[2] D. P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 2. Athena Scientific, Belmont, MA, 1995.

[3] D. P. Bertsekas. *Nonlinear Programming.* Athena Scientific, Belmont, MA, 1995.

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

[5] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, pages 303–314, 1989.

[6] Hertz J A, Krogh A, Palmer R G. *Introduction to the Theory of Neural Computation.* Addison-Wesley, Reading, MA, 1991.

[7] B. V. Roy J. N. Tsitsiklis. An analysis of temporal-differences learning with function approximation. *LIDS Technical Paper, LIDS-P-2322,* 1996.

[8] Haykin S. *Neural Networks, a comprehensive foundation.* Macmillan, New York, 1996.

[9] G. J. Tesauro. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, 38:58–68, 1995.