# 1 PLAY SELECTION IN AMERICAN FOOTBALL: A CASE STUDY IN NEURO-DYNAMIC PROGRAMMING

Stephen D. Patek[1] and Dimitri P. Bertsekas[2]

[1]Laboratory for Information and Decision Systems
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
sdpatek@mit.edu

[2]Laboratory for Information and Decision Systems
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
dimitrib@mit.edu

**Abstract:** We present a computational case study of neuro-dynamic programming, a recent class of reinforcement learning methods. We cast the problem of play selection in American football as a stochastic shortest path Markov Decision Problem (MDP). In particular, we consider the problem faced by a quarterback in attempting to maximize the net score of an offensive drive. The resulting optimization problem serves as a medium-scale testbed for numerical algorithms based on policy iteration.

The algorithms we consider evolve as a sequence of approximate policy evaluations and policy updates. An (exact) evaluation amounts to the computation of the reward-to-go function associated with the policy in question. Approximations of reward-to-go are obtained either as the solution or as a step toward the solution of a training problem involving simulated state/reward data pairs. Within this methodological framework there is a great deal of flexibility. In specifying a particular algorithm, one must select a parametric form for esti-

mating the reward-to-go function as well as a training algorithm for tuning the approximation. One example we consider, among many others, is the use of a multilayer perceptron (i.e. neural network) which is trained by backpropagation.

The objective of this paper is to illustrate the application of neuro-dynamic programming methods in solving a well-defined optimization problem. We will contrast and compare various algorithms mainly in terms of performance, although we will also consider complexity of implementation. Because our version of football leads to a medium-scale Markov decision problem, it is possible to compute the optimal solution numerically, providing a yardstick for meaningful comparison of the approximate methods.

## INTRODUCTION

In this paper, we present a case study of practical algorithms for solving large-scale dynamic optimization problems. In the class of problems we consider, rewards accumulate in stages as an underlying system transitions from state to state. What is desired is a controller which, at every stage, implements a control action, determining probability distributions for

1. the transition to a successor state and
2. the amount of reward to be earned that stage.

The objective is to synthesize a policy for the controller, i.e. a mapping from states to control actions, which maximizes the expected reward accumulated over time.

Dynamic programming is the classical framework for solving problems of this type. Included in this framework are the classical algorithms: value iteration and policy iteration (see [Bertsekas, 1995b], [Puterman, 1994], and [Ross, 1983]). In this paper we are primarily concerned with methods that relate to policy iteration, whereby an optimal solution is computed through a sequence of policy evaluations and updates. 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 an 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, classical policy iteration is primarily a conceptual algorithm, not of practical interest.

In recent years, Approximate Policy Iteration (API) has been suggested as a practical approach to solving large-scale dynamic optimization problems. In this framework, approximations of reward-to-go are trained through simulation and least squares optimization, and policy updates are computed based upon

these approximations. The approximations take on a fixed parametric form, called an approximation architecture.

API is one out of several classes of algorithms that comprise the methods of Neuro-Dynamic Programming (NDP) [Barto et al., 1995, Bertsekas and Tsitsiklis, 1996]. We view these methods as part of the broader field of Reinforcement Learning (RL), a long-standing field in artificial intelligence. In general, the methods of NDP are analogous to the classical algorithms of dynamic 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 reward-to-go approximations. For a sampling of other approximate dynamic programming algorithms we refer the reader to [Bertsekas, 1995b], [White, 1969], [Whitt, 1978], [Whitt, 1979], [Schweitzer and Seidmann, 1985], [Barto et al., 1995], and [Werbos, 1992].

As an alternative to API (but still within the framework of NDP), we consider a related class of algorithms known collectively as Optimistic Policy Iteration (OPI). In defining OPI it is useful to note first that in API an honest attempt is made at every stage to approximate the reward-to-go function. The goal there is to have an approximation which is accurate everywhere in the state space, requiring that a large number of sample trajectories be generated. The sample data is often stored in memory and is then presented many times to the training algorithms. In this way each state/reward data pair can have a significant impact on the outcome of training. OPI, on the other hand, can be defined to be approximate policy iteration where

1. a relatively small number of sample trajectories are generated per policy and

2. the data is allowed to impact the 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 of the current policy.) OPI has become a very popular method, with a number of important success stories appearing in the literature (see especially [Tesauro, 1995]).

For our case study, we have applied both API and OPI to a simplified version of American football. We consider the problem of play selection for one offensive drive in the middle of an infinitely long game. (End-game effects are ignored.) In contrast to real American football, we ignore the fact that there is an intelligent opponent which necessitates the use of randomized strategies and causes the probabilities of successful plays to be dependent on field position. State transitions in our framework are determined through a fixed probabilistic

model in which only offensive play decisions can influence the trajectory of the offensive drive. 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. That is, we want to determine a stationary policy which achieves

$$J^*(i) \stackrel{\triangle}{=} \max_{\text{policies } \mu} E \left\{ \begin{array}{l} \left( \begin{array}{l} \text{Points received at the end of our drive} \\ \text{from initial field-position, } i, \text{ under policy } \mu. \end{array} \right) \\ - \left( \begin{array}{l} \text{anticipated points gained by the opposing} \\ \text{team from our final field position.} \end{array} \right) \end{array} \right\}. \tag{1.1}$$

The probabilistic model we use is detailed in the Appendix. Despite the simplicity of our model, we obtain a moderately large problem, having 15250 states.

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 temporal differences learning [Sutton, 1988] (i.e. TD($\lambda$) with different values of $\lambda$) 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 $\lambda$ 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 [Bertsekas, 1995a]. 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 solution can be computed numerically, we have a useful basis for comparisons of the approximate methods.

The rest of this paper is organized as follows. First, we give a quick introduction to approximate and optimistic policy iteration. Our description there is largely qualitative. We refer the reader to [Bertsekas and Tsitsiklis, 1996] for a more complete description of the algorithms. Next, we formulate the problem of optimal play selection as a stochastic shortest path problem. After describing the optimal solution to the problem (obtained numerically), we describe a heuristic (suboptimal) solution, which, while consistent with conventional wisdom about football, is significantly worse than the optimal solution. Next, we discuss the technical issues we encountered in applying API and OPI to football. After describing approximation architectures, we specify our training algorithms and our technique for choosing initial conditions for sample trajectories. Next, we give the experimental results of our case study. The best

policies obtained by the approximate methods are compared to the exact solution obtained earlier. The main body of the paper ends with a discussion of the results and a few brief conclusions.

## APPROXIMATE AND OPTIMISTIC POLICY ITERATION

The main idea behind 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 vector $r \in \Re^d$ is chosen on the basis of simulation data and least squares regression. Let $i^*$ be a typical state of interest. In applying our numerical algorithms, we will use Monte Carlo estimates of the reward-to-go from $i^*$ to decide which policies are best and sometimes when to terminate algorithms. (More generally, we could choose a subset of interesting states.) API and OPI can be described by the following algorithm. The integer parameters $N_p$, $N_e$, $N_s$, and $N_t$ are set by the user in advance, determining the general behavior of the algorithm.

1. Start with an initial policy $\mu_0$.

2. Given $\mu_k$,

   (a) If $k \in \{j \cdot N_p \mid j = 0, 1, 2, \ldots\}$, then generate $N_e$ sample trajectories, each starting from $i^*$, to obtain an estimate of $J_{\mu_k}(i^*)$.

   (b) Given a probabilistic rule for picking initial conditions, generate $N_s$ sample trajectories, recording

   $$\mathcal{D}_k = \left\{ (i_t^l, g_t^l) \mid \begin{array}{l} l = 1, \ldots, N_s \\ t = 1, \ldots, T^l \end{array} \right\} \tag{1.2}$$

   where $i_t^l$ is the $t$-th state encountered in the $l$-th sample trajectory, $g_t^l$ is the corresponding sample reward-to-go, and $T^l$ is the length of the $l$-th trajectory. Store the data in memory for future use.

   (c) Tune the parameter vector $r^k$ based on $\mathcal{D}_k$ using a prespecified training algorithm. For some training algorithms, the training can be done in real-time (i.e. as the sample data is being generated). Let the training algorithms cycle through the data $N_t$ times.

   (d) Compute a new policy $\mu_{k+1} := G(r^k)$, where $G$ is the "greedy" operator which chooses the actions at each state that are best with respect to the reward-to-go approximation given by $r^k$.

In API, an honest attempt is made to approximate the entire reward-to-go function $J_{\mu_k}$ associated with each policy $\mu_k$. Generally, this requires $N_s$ and $N_t$ to be large. In this way a great deal of sample data is generated, and this data is heavily exploited by the training algorithms. In addition, $N_p$ will generally be set to one, and $N_e$ will be set to be very large, so that we obtain an accurate estimate of $J_{\mu_k}(i^*)$ for every policy. Assuming that, in the training phase of each iteration, a limiting parameter value $r^{\mu_k}$ is 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 $\mu_k$ and $r^{\mu_k}$. (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.

In contrast, policy updates in OPI are computed on the basis of very rough (optimistic) approximations of the reward-to-go function. Generally, very little sample data is generated for each policy, with $N_s$ set to one or a very small number. Also, $N_t$ is generally set to be a very small number, 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. 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 (i.e. we have to keep $N_e$ large). Generating lots of "extra" sample trajectories is contrary to the spirit of OPI, so evaluation of the successive policies is an inherent difficulty of the method. One way to circumvent this is to evaluate policies only periodically, setting $N_p$ to be some large positive integer. This technique will unfortunately ignore many of the policies that are produced, some of which may be very close to optimal. On the other hand, this technique allows most of the computational effort to be directed toward the underlying OPI method. Assuming that the parameter vector $r^k$ converges (as $k \to \infty$), there is no guarantee that it won't converge to a point where small perturbations can result in substantially different greedy policies. If this is the case, then it is possible that the true rewards associated with the successive policies will not converge. This observation provides a mechanism for the "oscillatory" behavior often exhibited by OPI. In general OPI is a poorly understood algorithm. In particular, we are unaware of any theoretical results which guarantee convergence.

## THE FOOTBALL MODEL

*Problem Formulation*

Here we present a simplified version of American football which we cast 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 the goal, $y$ = the number of yards to go until the next first down, and $d$ = down number. We discretize $x$ and $y$ by yards, resulting in 15250 states. (The rules of our model are such that at first down there is a unique value of $y$ associated with each value of $x$. Also, it is impossible to have $y > x$.) Each individual state is identified by $i \in S$, where $S$ is a finite set. 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. Transitions from state to state are governed by the probabilistic model described in the Appendix. At each state, the quarterback must choose one out of four play options: run, pass, punt, and kick (field goal). The quarterback's *policy* is a function $\mu : S \to U$, where $U$ denotes the set of control options.

Our team transitions to an absorbing, zero-reward termination state $T$ whenever it loses possession of the ball. Rewards in this framework are earned only upon first transitioning to $T$. The amount of reward 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 their drive. The latter is a function of where they receive the ball. As is the case in real football, termination of the current offensive drive is inevitable under all policies. Thus, the problem of maximizing expected total reward can be viewed as a stochastic shortest path problem (see [Bertsekas and Tsitsiklis, 1991]).

As we will discuss shortly, our model for football is numerically tractable. However, simple enhancements to the model can be implemented that make the problem computationally infeasible. For example, by more finely discretizing the playing field, say to half-yard units, we would have 60500 states, an increase by a factor of four. Alternatively, if we wanted to examine end-game effects, we could factor in time as a state variable. By discretizing time in 10 second intervals and then playing for the last two minutes of the game, we would have $15250 \cdot 120/10 = 183000$ states. For these larger problems, it becomes impractical or even impossible to use numerical implementations of the classical methods; the memory and processing requirements would be too great.

*The Optimal Solution*

Fortunately, our original 15250 state model leads to a problem that can be solved exactly through numerical implementation of policy iteration. On a 120 MHz Pentium machine running the Linux operating system, it took approximately 2.5 minutes to obtain the optimal solution. In evaluating the reward-to-go function for each policy of the iteration, we applied successive approximation applied until the sup-norm difference between the iterates was less than $10^{-6}$ football points. Starting from the initial policy "always run", six policy iterations were required to determine the optimal policy. The code was written in the C programming language.

The optimal policy and corresponding reward-to-go functions are shown in Figure 1.1. First down is distinctive because 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. Thus, if our team were always to receive the ball at this point, we could expect ultimately to lose the game. This is strictly a function of the parameters of our mathematical model.

The results for the remaining downs are presented as surface plots. In theory, $y$ can be as large as $x$. However, in practical terms it is unlikely to have $y > 20$. While the possibility of $x \approx y$ appears 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 the 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 team 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.

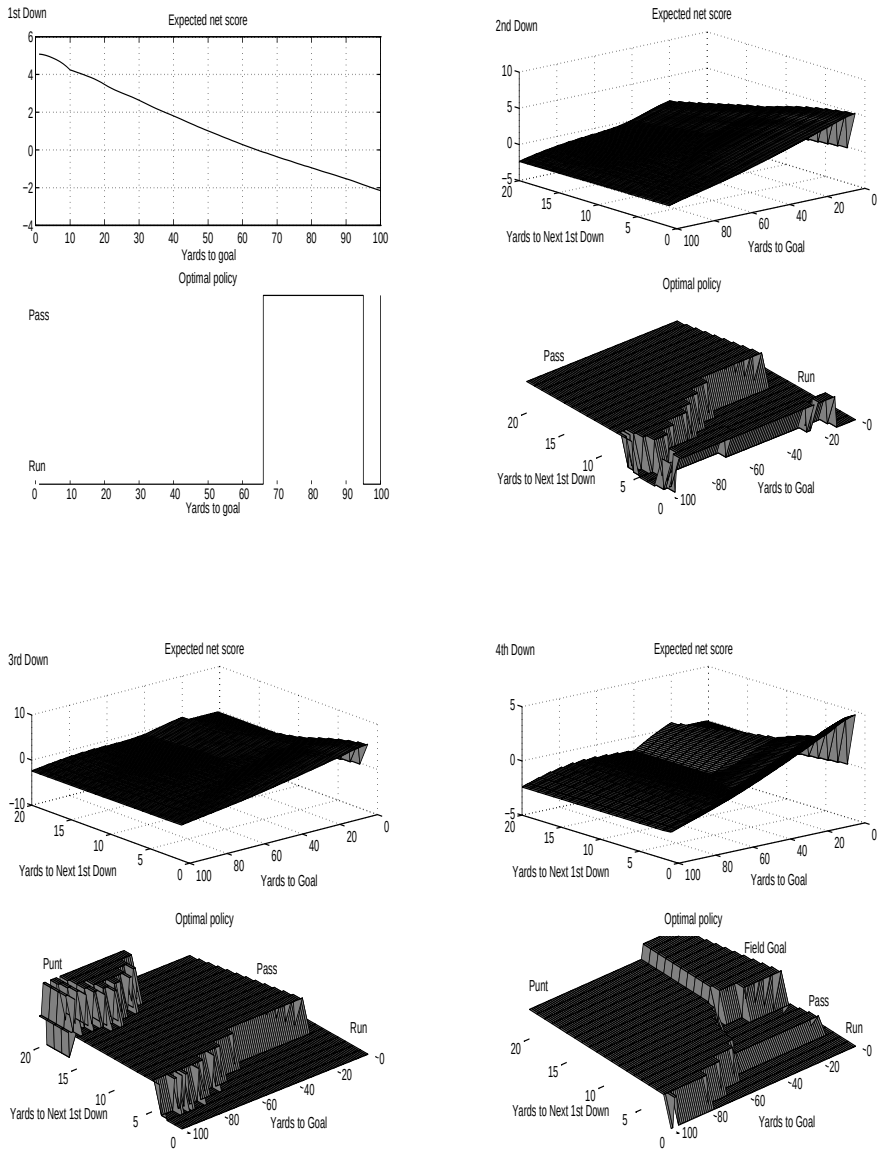


**Figure 1.1** Complete characterization of the optimal expected score and the optimal policy.

*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,
      - if the number of yards to the next first down is less than 3, then [either RUN or PASS],
      - otherwise, [either RUN or PASS]

   (b) if the number of yards to the endzone is greater than 40,
      - if the number of yards to the next first down is less than 3, then [either RUN or PASS],
      - otherwise, [either RUN or PASS]
4. At fourth down,
   (a) if the number of yards to the endzone is less than 41,
      - if the number of yards to the next first down is less than 3, then [either RUN, PASS, or KICK],
      - otherwise, [either RUN, PASS, or KICK]

   (b) if the number of yards to the endzone is greater than 40,
      - if the number of yards to the next first down is less than 3, then [either RUN, PASS, or PUNT],
      - otherwise, [either RUN, PASS, or PUNT]

The options chosen for each region of the state space collectively define a stationary policy which may be evaluated exactly (by numerical methods). Each such 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.

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

$$i^* \quad \leftrightarrow \quad (x_{i^*} = 80,\ y_{i^*} = 10,\ d_{i^*} = 1) \tag{1.3}$$

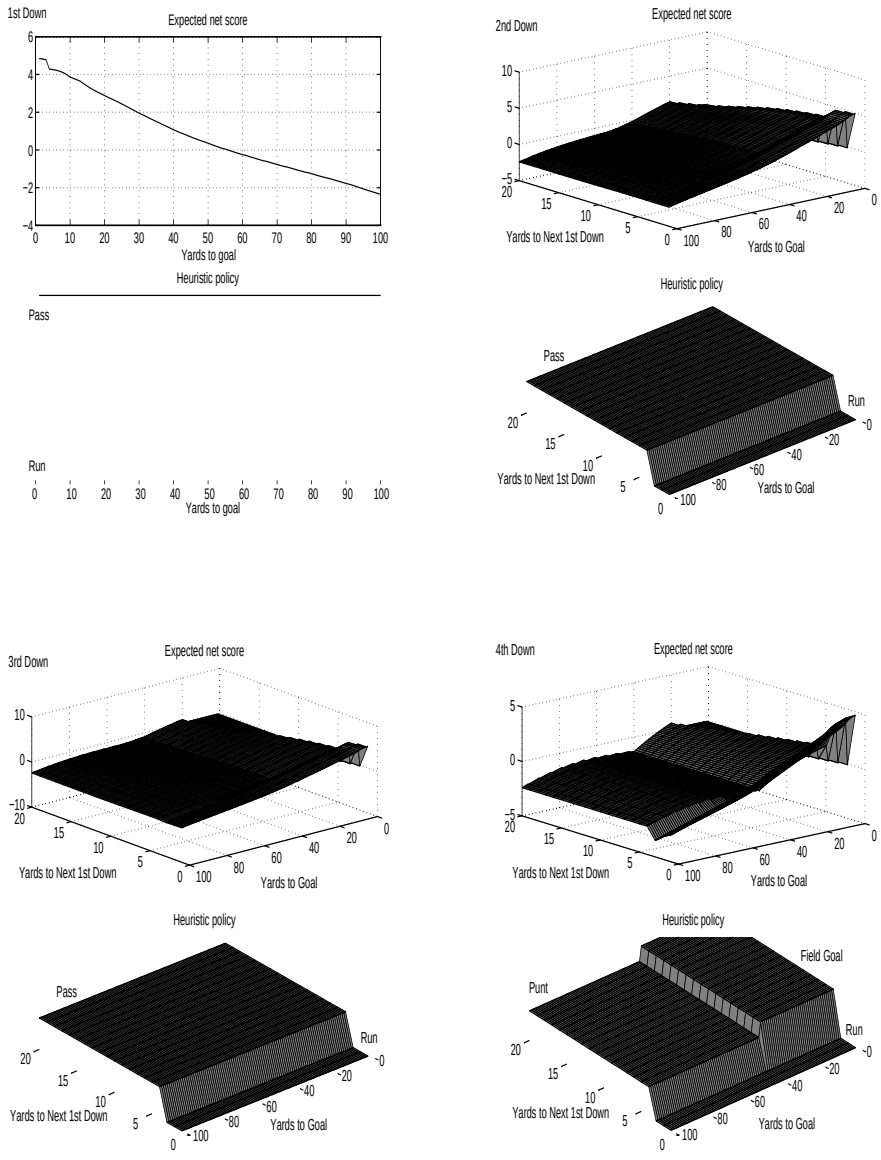


**Figure 1.2** Characterization of the best heuristic policy.

This is the "typical" state at which our team will receive the ball. The best policy in the heuristic class is defined to be the one which has the highest expected reward-to-go from $i^*$. Figure 1.2 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 our 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 just over seven days!

## ON APPLYING API AND OPI TO FOOTBALL

In this section we provide details about our application of NDP to football. We first discuss the approximation architectures we used and then we discuss training algorithms and our rule for picking simulation initial conditions.

*The Approximation Architectures*

In describing approximation architectures for football, it is useful to recall that each state $i \in S$ can be uniquely associated with the triple $(x_i,\ y_i,\ d_i)$. As functions of $i$, the quantities $x_i$, $y_i$, and $d_i$ can be viewed as features which characterize the state $i$.

We chose to use a piecewise continuous architecture, comprised of four independent subarchitectures. Each down number has its own subarchitecture. In mathematical notation, the approximation architectures are all of the form

$$\tilde{J}(i,r) = H(\ \xi^{\Sigma}(i),\ \zeta^{\Sigma}(i),\ r_{d_i}\ ) \tag{1.4}$$

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 $y_i$, $\zeta^{\Sigma}(i)$ 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 parameter vectors for the respective subarchitectures. The feature vectors $\xi^{\Sigma}(i)$ and $\zeta^{\Sigma}(i)$ are given by

$$\xi^{\Sigma}(i) \quad = \quad (\sigma^x_{d_i} \cdot x_i,\ \ \sigma^y_{d_i} \cdot y_i) \ \in \ \Re^2 \tag{1.5}$$

$$\zeta^{\Sigma}(i) \quad = \quad (\sigma^{f_1}_{d_i} \cdot f_1(i), \ldots, \sigma^{f_{n_f}}_{d_i} \cdot f_{n_f}(i)\ ) \ \in \ \Re^{n_f} \tag{1.6}$$

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. Rather, they are set in advance based on engineering judgment.

In the following subsections we discuss the three main parametric forms we used in football: MLP, Quadratic, and Recursive.

**Multilayer Perceptron (MLP).** A multilayer perceptron can be viewed as a parametric form for function approximation. MLP's 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 is fed by the output of the adjacent layer closer to the input 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 final affine transformation. In training, the coefficients of the affine layers are tuned according to a least squares criterion. For a comprehensive discussion of neural networks, including the multilayer perceptron, we refer the reader to [Haykin, 1996] and [Hertz et al., 1991].

For our case study, we used multilayer perceptrons with only a single hidden layer of activation units. The only input to each MLP subarchitecture is the scaled standard feature vector $\xi^\Sigma$. (This allows us to drop the feature vector $\zeta^\Sigma$ from our notation.) To make the definition explicit, let $R$ be a positive integer equal to the number of hidden nonlinear elements for each multilayer perceptron. Let $\xi^\Sigma$ represent the value of the scaled standard feature vector evaluated at some state $i$. Let $\rho = (W_1, b_1, W_2, b_2)$ be the parameter data structure which applies on the subset $d_i$, where $W_1 \in \Re^{R\times 2}$, $b_1 \in \Re^R$, $W_2 \in \Re^{1\times R}$, and $b_2 \in \Re$ are collectively the weights of the multilayer perceptron. The output of the architecture is computed as

$$H(\xi^\Sigma, \rho) = W_2\phi(\xi^\Sigma) + b_2, \tag{1.7}$$

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

$$\phi_l(\xi^\Sigma) = \tanh(\psi_l(\xi^\Sigma)), \tag{1.8}$$

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

$$\psi(\xi^\Sigma) = W_1\xi^\Sigma + b_1. \tag{1.9}$$

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

**Quadratic.** Here we describe an architecture which is quadratic in the feature vector $(x, y, d)$. The most appealing aspect of this architecture is its simplicity. Computer subroutines that implement this architecture are easy to develop and execute quickly.

To give a mathematical description, it is useful to define the *quadratic expansion* of a vector. Let $\theta = (\theta_1, \ldots, \theta_{\bar{n}})' \in \Re^{\bar{n}}$. Then,

$$Q(\theta) \stackrel{\triangle}{=} (1, \ \theta_1, \ldots, \theta_n, \ (\theta_1)^2, \theta_1\theta_2, \ldots, \theta_1\theta_n, \ (\theta_2)^2, \theta_2\theta_3, \ldots, (\theta_n)^2), \quad (1.10)$$

denotes the quadratic expansion of $\theta$. As before, let $\xi^\Sigma$ represent the value of the scaled standard feature vector evaluated at some state $i$. Let $\rho$ be the parameter vector which applies on the subset $d_i$. The quadratic architecture for reward-to-go approximation is given by $H(\xi^\Sigma, \rho) = \rho' Q(\xi^\Sigma)$. The scale factors in $\Sigma$ for the case study were chosen as: $\sigma_d^x = .01$ for all $d$; $\sigma_d^y = .05$ for $d = 2, 3, 4$; and $\sigma_1^y = 0$. (We use $\sigma_1^y = 0$ because, at first down, the number of yards to go until the next first down is uniquely determined by the number of yards to the end-zone.)

**Quadratic with Feature Recursion (Recursive).** Because the quadratic architecture uses relatively few parameters and is a form which admits an exact solution in training, approximations using this architecture can be evaluated and trained very quickly. Unfortunately, for the same reasons, the quadratic architecture has a limited ability to approximate very irregular functions. Intuitively, the richness of the architecture is limited by the number of features used. The recursive architecture that we describe here is essentially the quadratic architecture of the preceding subsection with the additional twist that every once in a while a new feature function is added. In this paper, the new features are themselves the past approximations of the reward-to-go function.

We first describe the recursive architecture in the context of API. Let $\xi^\Sigma$ represent the value of the scaled standard feature vector evaluated at some state $i$. Let $\rho$ be the parameter vector which applies on the subset $d_i$. Suppose that $\mu_k$ is the current policy and that we are trying to approximate $J_{\mu_k}$. 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 $\zeta^\Sigma$:

$$\zeta^\Sigma(i) = (\sigma_{d_i}^{f_1} \cdot \tilde{J}(i, r^{k-1}), \ \ldots \ , \ \sigma_{d_i}^{f_{n_p}} \cdot \tilde{J}(i, r^{k-n_p}) \ ). \quad (1.11)$$

The recursive architecture 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 $\zeta^\Sigma$ with zeros, so that the earliest iterations tend to imitate the quadratic architecture.

Although this architecture is basically "quadratic", there are significant complexities involved in its implementation. The architecture is inherently computationally intense because *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.

Because relatively few policies are ever generated in API, it is practical to implement the recursive architecture as described above. With OPI, since so many distinct policies are produced, this architecture would be impractical. A simple modification is to compute recursions infrequently (periodically), keeping track of the "good" policies in between. The scale factors in $\Sigma$ 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 $k = 1, \ldots, n_p$.

*Training Algorithms*

Here we briefly describe the training algorithms we used to tune approximations of reward-to-go. For more details we refer the reader to [Bertsekas and Tsitsiklis, 1996] and the references contained therein. We focused on the most commonly used algorithms: temporal differences learning TD($\lambda$), the Bellman error method, and linear least-squares regression (referred to as SVD). TD($\lambda$), with $\lambda$ a real number in the range $[0, 1]$, is actually a class of iterative algorithms which can be implemented in real-time. TD(1) reduces to the backpropagation algorithm commonly used in training neural networks. The Bellman error method is a related recursion which can be viewed as an incremental gradient method for minimizing the error in solving Bellman's equation. For architectures which are linear in the parameters (such as the quadratic architecture), the SVD method can be used to compute the least squares solution in a single step. The method is called "SVD" because the singular value decomposition is used to compute the pseudo-inverse of the covariance matrix of the sample data.

*Simulation: sampling initial conditions*

Here we describe the random mechanism by which we chose initial conditions for the simulated football trajectories. The most important aspect of this rule is that it selects initial conditions corresponding to the states our team is most likely to encounter.

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 $y = 10$.

## EXPERIMENTAL RESULTS

The table in Figure 1.3 describes the experimental runs for our 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($\lambda$), Bellman Error, or SVD.

In each entry of the column labeled "Training method," we specify several parameter values corresponding to separate runs. For example, by entering TD(0,.5,1), we mean to say we tried the algorithm with $\lambda$ set to 0, then with $\lambda$ set to .5, and finally with $\lambda$ set to 1. We show in **bold** the parameter settings which are best with respect to *sample* expected reward-to-go from the typical initial condition $i^* \leftrightarrow (80, 10, 1)$. (The sample evaluation is based on $N_e = 8000$ 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 the experiments in each row. For the OPI runs, sample evaluations from $i^*$ are computed every $N_p = 200$ policy updates.

The algorithmic parameters shown in the table represent the best settings we could find based on a considerable amount of tinkering. We tried to be objective in this process, not wanting to "sabotage" any particular algorithm. Our goal was to be both comprehensive in scope and honest in evaluation. Results for the case study are shown in Figures 1.4 through 1.9. 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 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 1.4 through 1.6, while Figures 1.7 through 1.9 are devoted to OPI.

## DISCUSSION AND CONCLUSIONS

Our observations from the case study are as follows.

1. Regarding API:
   (a) This algorithm along with the MLP architecture has yielded the best results.
   (b) For the best API runs, the first-down approximations of reward-to-go are close to optimal. *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 the MLP architecture does not seem to have effected the results.
   (d) When using TD($\lambda$) 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) The "oscillatory" limiting behavior of API can be seen in the means plots of Figures (1.4) through (1.6).
   (f) As an imitation of exact policy iteration, API is not totally convincing. In particular, the means plots are not 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 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 same is true for the exact evaluations of the suboptimal policies.
   (c) In general, the more complex the architecture, the better the results (again, at the expense of longer computation times).
   (d) Regarding algorithms for training the approximations, there is no clear winner. For the MLP architecture, the best results were obtained with TD(1). For the quadratic architecture the Bellman error method worked best; whereas, for the recursive architecture TD(.5) worked best.
   (e) We see that OPI can be oscillatory, despite the convergence of the parameter vectors (not shown in the figures).
   (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.
3. Football is an effective testbed for NDP:
   (a) It represents a challenge for the approximate methods and seems to have characteristics of truly large scale examples. However, because we can compute the optimal solution to the problem, we have a yardstick for comparisons of the alternative methods.
   (b) Our model for football is not totally trivial as evidenced by the poor performance of the best heuristic policy.
   (c) Finally, football is intuitive. This aids in the implementation and debugging of the algorithms and also provides a means for interpreting the results.

The main purpose of this case study was to determine the limits of performance for 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 required the most time to complete.

The amount of time required for a particular algorithm usually 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 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 described earlier. 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. 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.

| Scheme | Architecture | Training Method | Exact RTG of best | Fig |
|---|---|---|---|---|
| API | MLP | TD(0, .5, **1**) and Bellman Error | -.954 | 1.4 |
| API | Quadratic | SVD: 4k and **30k** sample trajectories per policy | -1.172 | 1.5 |
| API | Recursive | SVD: 30k traj/policy & $n_p = 2$, 30k traj/policy & $n_p = 5$, **45k traj/policy & $n_p = 7$** | -.957 | 1.6 |
| OPI | MLP | TD(0, .5, **1**) and Bellman Error | -1.022 | 1.7 |
| OPI | Quadratic | TD(0, .5, 1) and **Bellman Error** | -1.161 | 1.8 |
| OPI | Recursive | TD(0, **.5**, 1) and Bellman Error | -1.006 | 1.9 |

**Figure 1.3** Table of experimental runs for the football case study. The best run for each experiment is shown in bold. Note that the optimal reward to go from $i^*$ is -.9449.

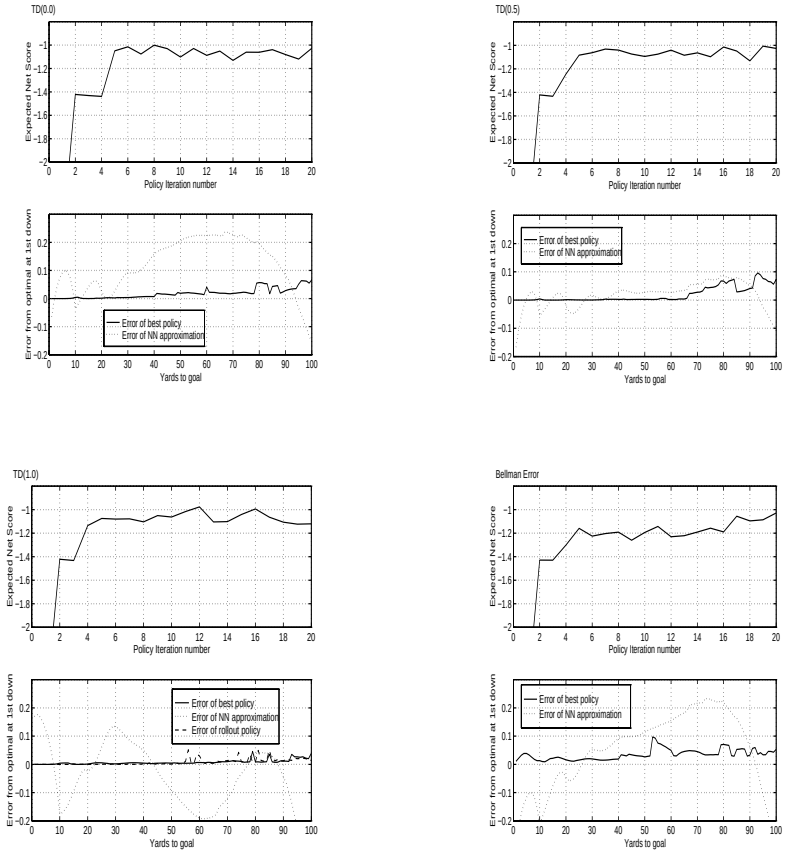


**Figure 1.4** API with the MLP architecture using the TD($\lambda$) ($\lambda = 0, .5, 1$) and the Bellman error methods with 100 cycles through 10000 sample trajectories in training per policy.

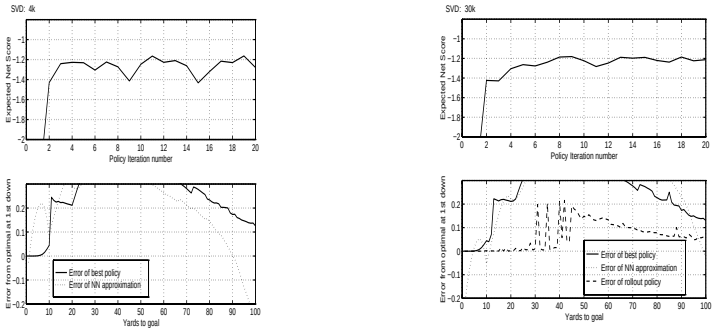


**Figure 1.5** API with the Quadratic architecture using the SVD method of training: 4k and 30k sample trajectories per policy.

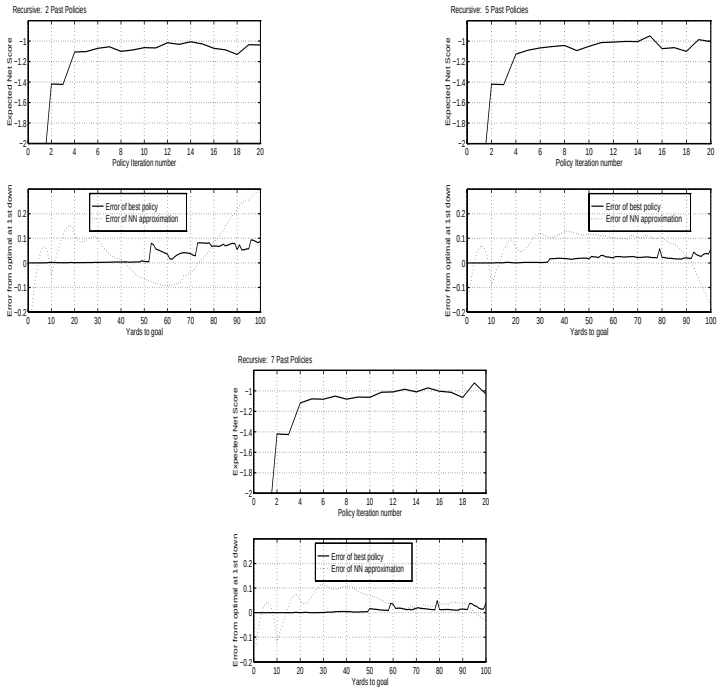


**Figure 1.6** API with the Recursive architecture using the SVD method of training: 2 past policies (30k traj./policy), 5 past policies (30k traj./policy), 7 past policies (45k traj./policy).

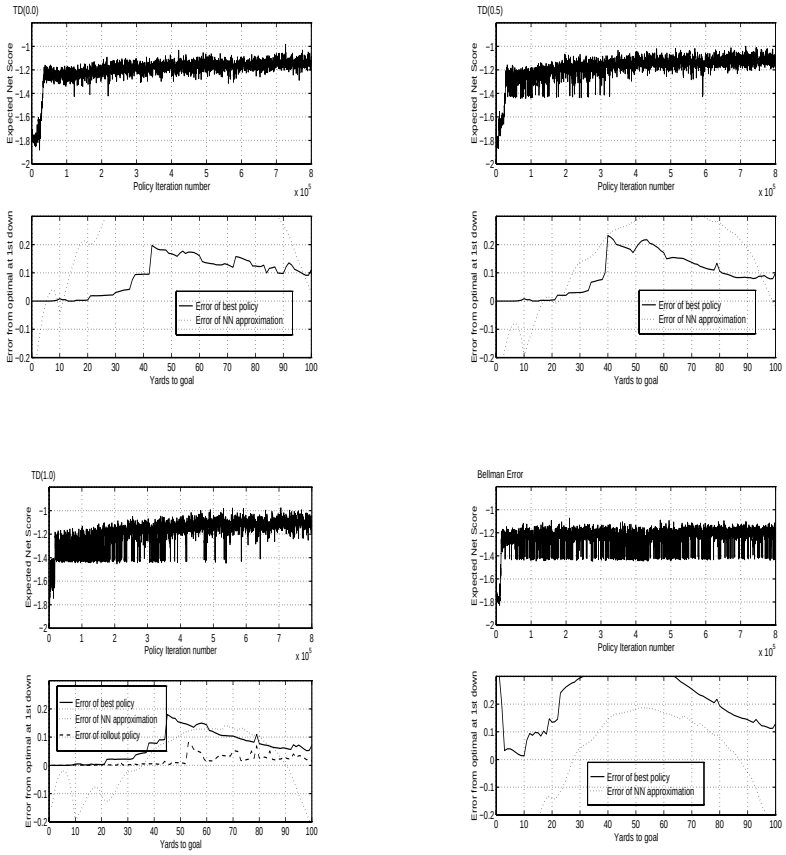


**Figure 1.7** OPI with the MLP architecture using the TD($\lambda$) ($\lambda = 0, .5, 1$) and Bellman error methods of training with one sample trajectory per policy.

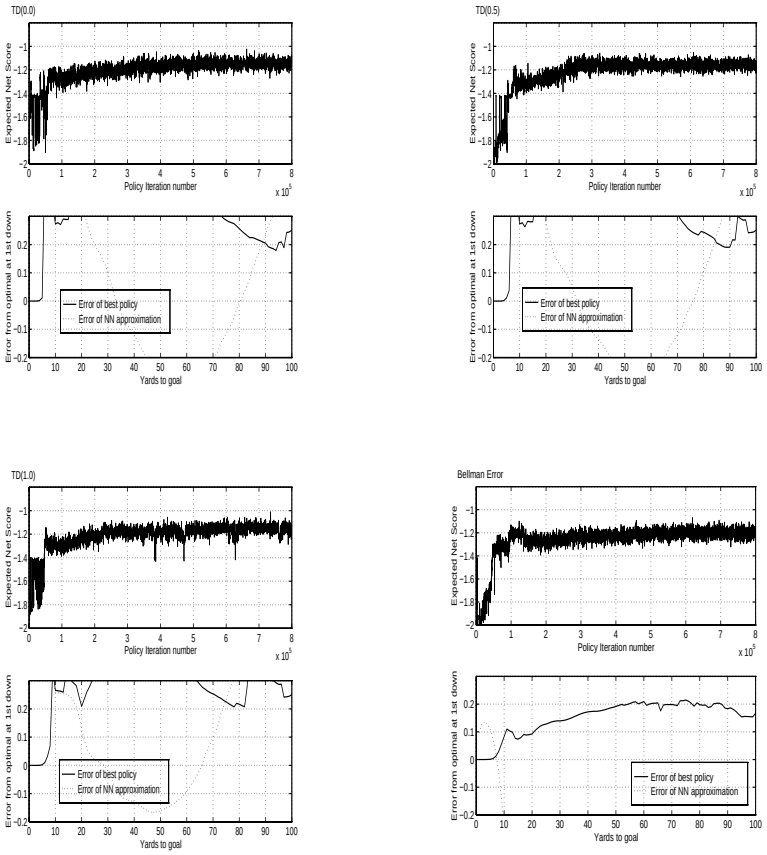


**Figure 1.8** OPI with the Quadratic architecture using the TD($\lambda$) ($\lambda = 0, .5, 1$) and Bellman error methods of training with one sample trajectory per policy.

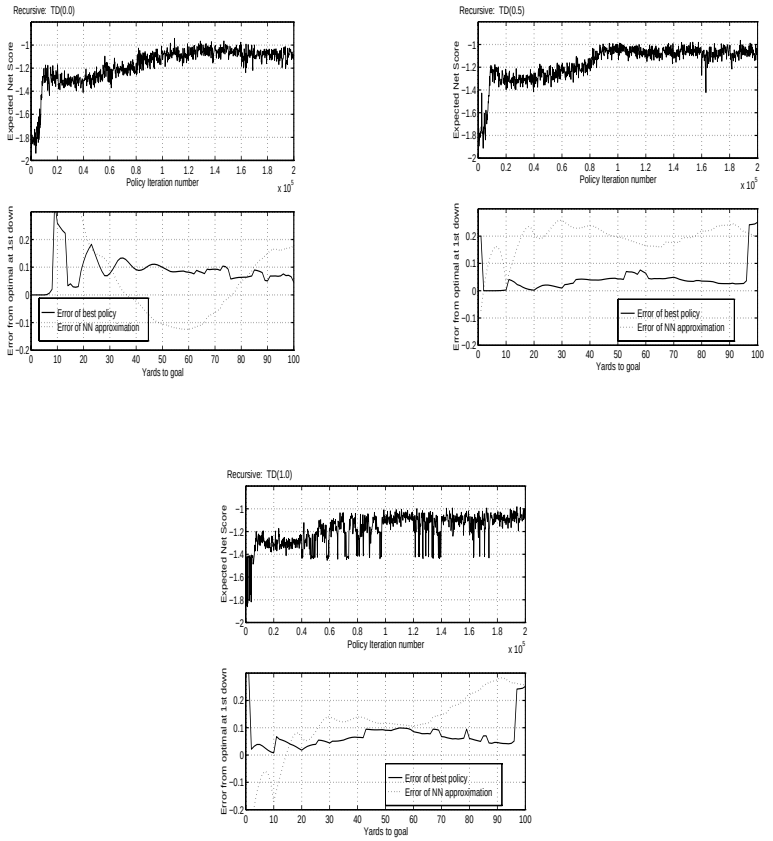


**Figure 1.9** OPI with the Recursive architecture using the TD($\lambda$) method of training: $\lambda = 0, .5, 1$, with one sample trajectory per policy.

## Appendix: The Football Model

We consider one offensive drive from the perspective of the quarterback. 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, as in (1.1). The state of the game is characterized by three quantities: $x$ = number of yards to goal (field position), $y$ = yards to go until the next first down, and $d$ = down number. The offensive drive ends whenever any of the following events occur.

1. A new first down fails to be earned after four plays.

   (The offense always receives the ball "at first down", i.e. with $d$ set equal to 1. At this time values for $x$ and $y$ are set, reflecting (respectively) our team's field position and the number of yards they must move forward in (nominally) four stages. Generally, each stage results in $d$ being incremented by one, along with with $x$ and $y$ being decremented by the number of yards earned. If our team manages to achieve $y \leq 0$ with $d \leq 4$, then they receive a new first down, i.e. $d := 1$ , along with a new value for $y$. New values of $y$ at first down are 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 a punt or field-goal attempt is made.

The playing field is discretized, with $x$ and $y$ taking on integer values. States for which $y > x$ are impossible. Since we do not include penalties in this model, there can be only one value of $y$ associated with each value of $x$ at first down. All totaled, there are $15250 = 3 * (100) * (101)/2 + 100$ states for which the quarterback must have some control action in mind.

The outcome of a given drive is random, depending on the quarterback's strategy and the transition probabilities associated with the various play options. There are always four play options from which to choose: run, pass, punt, and kick (field-goal). Transition probabilities and further details are spelled out in the following paragraphs. The last paragraph below describes how our team's net score is computed.

**Run attempts.** With probability .05, a run attempt will result in a fumble. If the ball is not fumbled, then the ball moves forward $D_r - 2$ yards, where $D_r$ is a Poisson random variable with a mean of 6. Negative gain is entirely possible, although not probable.

If the run attempt 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 but 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 with $x > 100$, then the drive ends with the opposing team scoring a touchdown.

**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, then the ball moves forward $D_p - 2$ yards, where $D_p$ is a Poisson random variable with a mean of 12. Incomplete passes result in no movement of the ball. If the quarterback is sacked, then the ball moves back $D_s$ yards, where $D_s$ is a Poisson random variable with mean 6.

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 (i.e. $x \leq 0$), 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 with $x > 100$, then the drive ends with the opposing team scoring a touchdown.

**Punt attempts.** A punt always results in the ball being turned over to the other team. The distance the ball moves forward is nominally $6 \cdot D_p + 6$, where $D_p$ is a Poisson random variable with a mean of 10. If this exceeds the distance to the goal, then the opposing team simply receives the ball at $x = 20$.

**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 our team scores a touchdown, then it immediately receives 6.8 points. If the other team scores a touchdown, then we immediately receive -6.8 points. (The opposing team can score a touchdown, for example, if our team fumbles the ball in its own end zone.) If a successful field-goal attempt is made, then the immediate reward is 3 points. If a safety is scored, then the immediate reward is -2.0 points. When the drive is over, an amount equal to the opposing team's *expected* score (for their drive)

is subtracted. The opposing team's expected score is a function of where they receive the ball: $6.8x/100$.

## References


Barto, A. G., Bradtke, S. J., and Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138.

Bertsekas, D. P. (1995a). A counterexample to temporal differences learning. *Neural Computation*, 7:270–279.

Bertsekas, D. P. (1995b). *Dynamic Programming and Optimal Control*, volume I and II. Athena Scientific, Belmont, MA.

Bertsekas, D. P. and Tsitsiklis, J. N. (1991). Analysis of Stochastic Shortest Path Problems. *Mathematics of Operations Research*, 16:580–595.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA.

Haykin, S. (1996). *Neural Networks, a comprehensive foundation*. Macmillan, New York.

Hertz, J. A., Krogh, A., and Palmer, R. G. (1991). *Introduction to the Theory of Neural Computation*. Addison-Wesley, Reading, MA.

Puterman, M. L. (1994). *Markovian Decision Problems*. Wiley, New York.

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

Schweitzer, P. J. and Seidmann, A. (1985). Generalized polynommial approximations in markov decision processes. *Journal of Mathematical Analysis and Applications*, 110:568–582.

Sutton, R. S. (1988). Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3:9–44.

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

Werbos, P. J. (1992). *Handbook of Intelligent Control*, chapter Approximate Dynamic Programming for Real-Time Control and Neural Modeling. Van Nostrand, New York. (eds. D. A. White and D. A. Sofge).

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

Whitt, W. (1978). Approximations of dynamic programs i. *Mathematics of Operations Research*, 3:231–243.

Whitt, W. (1979). Approximations of dynamic programs ii. *Mathematics of Operations Research*, 4:179–185.