Explorations of the Practical Issues of Learning Prediction-Control Tasks Using Temporal Difference Learning Methods

Charles L. Isbell

submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Science

at the
Massachusetts Institute of Technology

December 1992

Author ______________________________
Charles Isbell
Department of Electrical Engineering and Computer Science

Certified by ______________________________
Tomaso Poggio,
Thesis Supervisor

Accepted by ______________________________
Campbell L. Searle,
Chairman, EECS Committee on Graduate Students

Explorations of the Practical Issues of Learning Prediction-Control Tasks

Using Temporal Difference Learning Methods

by

Charles L. Isbell

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of Master of Science
in Computer Science

Abstract

There has been recent interest in using a class of incremental learning algorithms called temporal difference learning methods to attack problems of prediction. These algorithms have been brought to bear on various prediction problems in the past, but have remained poorly understood. It is the purpose of this thesis to further explore this class of algorithms, particularly the TD (λ) algorithm. A number of practical issues are raised and discussed from a general theoretical perspective and then explored in the context of several case studies. The thesis presents a framework for viewing these algorithms independent of the particular task at hand and uses this framework to explore not only tasks of prediction, but also prediction tasks that require control, whether complete or partial. This includes applying the TD (λ) algorithm to two tasks: 1) learning to play tic-tac-toe from the outcome of self-play and the outcome of play against a perfectly-playing opponent and 2) learning two simple one-dimensional image segmentation tasks.

Thesis Supervisor: Professor Tomaso Poggio
Uncas and Helen Whitaker Professor,
Department of Brain and Cognitive Sciences

Acknowledgments

I would like to thank my advisor, Tomaso Poggio, for his advice and direction while I worked on this thesis. His intensity and constructive comments helped me to move this project forward.

I would also like to thank Federico Girosi, Scott Hofmeister, Jim Hutchinson, Mike Jones and Paul Viola for their thoughts during the course of this work.

Finally, I must thank Charlton J. Coles, Jacqueline D. Isbell and Patience R. Singleton. Their support, while not being of a technical nature, proved to be invaluable.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. This research is sponsored by a grant from the Artificial Intelligence Center of Hughes Aircraft Corporation (LJ90-074) and by Office of Naval Research contract N00014--89--J--3139 under the DARPA Artificial Neural Network Technology Program. Support for the A.I. Laboratory's artificial intelligence research is provided by the Advanced Research Projects Agency of the Department of Defense under Army contract DACA76--85--C--0010, and in part by ONR contract N00014--91--J--4038.

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Chapter 1

1 Learning and Neural Networks

Learning is a question central to the study of Artificial Intelligence. The sub fields that make up AI are varied and sometimes quite divergent in their immediate goals and methods; however, the goal of "learning" in a way that mimics—and hopefully illuminates—the process employed by human beings is an omnipresent one.

There has been great interest in the study of neural networks as a method for attacking this problem of learning. This interest has led to the creation of many different structures which have been dubbed "networks." In this thesis, I will use the General Radial Basis Function (GRBF) and HyperBF networks (Poggio and Girosi, 1990), to discuss algorithms for training neural networks. In particular, I will discuss and propose evaluation criteria for the TD (λ) temporal difference learning algorithm (Sutton, 1988). As a training algorithm it is provably equivalent to the more widely-used supervised learning algorithms; however, questions remain about its usefulness and efficiency with more complex real-world problems. In this thesis I will identify a number of practical issues that this algorithm must address and use several case studies to provide an empirical context to study its strengths and limitations.

This thesis is divided into several parts. This first chapter introduces feedforward networks in some detail and broadly defines two main classes of algorithms for training them. The second chapter introduces and derives another class of training algorithms, the TD (λ) algorithms, and distinguishes them from the two aforementioned classes as a temporal difference learning method. Theoretical work is presented, relating TD (λ) to currently studied problems and to the prediction paradigm for which the algorithm should be ideal. In addition, this chapter develops a theoretical and algorithmic formalism for studying TD (λ), allowing one to encompass not only tasks of simple prediction, but more complicated prediction tasks that involve control—whether complete or partial—as well. Chapter three presents related work, describing many of the practical issues that the TD (λ) method raises and attempts to show where this research relates to the greater body of work. Chapters four and five describe the case studies used in this research to evaluate the TD (λ) algorithm, relating them to the practical issues discussed in the previous chapter. In particular, the case studies explore two problems involving prediction and control: determining an evaluation function for tic-tac-toe positions through both self-play and play against a superior opponent and simulating a restricted class of recurrent networks to learn to do two kinds of simple segmentation. Results are presented and discussed. Finally, chapter six concludes with a brief discussion of TD (λ) and a review of the thesis.

1.1 Why networks at all?

In practice, neural nets are difficult to train and often have trouble performing even what would seem to be simple tasks. As number-crunching mechanisms, networks are often unable to deal easily with the processing of symbols; are extremely sensitive to the representation used for the data; and oftentimes require an inordinate amount of time to train. Still, neural networks do enjoy several advantages:

1. They can learn to perform tasks for which computational algorithms do not exist or are poorly understood.

2. They learn on the fly, adapting their behavior to a changing environment.

3. As mathematical abstractions, they are not wedded to any specific algorithmic engine for training.

4. They inherit a wealth of theory and empirical data from approximation theory, particularly from the fields of regression and statistical inference.

In practice, the first property is probably the most important. For many problems of interest, the level of understanding of the problem is poor. For example, with a computer vision problem, we might want to perform some sort of object recognition but are unable to actually define what we mean by the "object." Despite this lack of a specification, we can usually describe how a correct algorithm should perform on particular examples. In this case, we can define learning as a process of associating particular inputs to particular outputs, that is as function approximation. This allows our analysis of neural networks to draw upon approximation theory, inheriting information from the statistical and regression communities.

If a self-training network is capable of discovering a function that performs correctly on a set of examples, it might be able to generalize to solve the problem for inputs that it has yet to see. If the network does well in generalizing the problem, this is an indication that it has discovered some important underlying structure. If this is the case, subsequent analysis of the network's "answer" might contribute to our understanding of the original problem. Unfortunately, this generalization problem is ill-posed: any finite set of examples for a function is consistent with an infinite number of functions, many of which may have nothing to do with the original problem.

The second property of networks—their built-in adaptability—is also important. If a neural network's behavior is learned in the first place, then re-learning based on some change in the environment should be easy to implement. This is, of course, a property found in all high-level biological organisms and one computer scientists seek to emulate.

As mathematical abstractions, neural network structures should be independent of changes in a particular training algorithm. The purpose of a training algorithm is to adjust the parameters of the network to move its outputs closer to the desired outputs. Of course, some training algorithms perform better than others and some algorithms might learn more quickly with certain networks than with others. Still, the choice of a training algorithm should not change the inherent

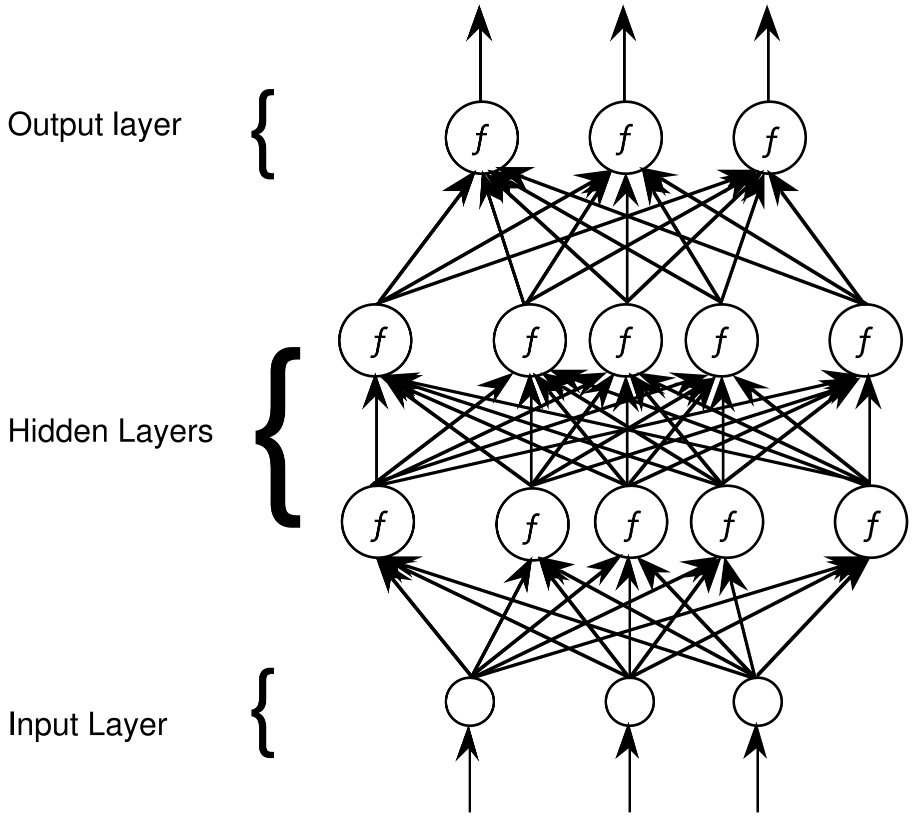


Figure 1.1: A perceptron-like feedforward network. Input values are "clamped" at the input layer. A unit at a higher level receives a weighted sum of the values of units below as input. It then passes this input through a function, usually non-linear, to produce its own value.

power of the network itself. This allows researchers to experiment with various algorithmic engines without affecting the universality of whatever neural network they choose to use for a base.

1.2 Radial Basis Function Networks

Neural networks consist of a set of interconnected computational units. The connections are directed and usually weighted. For most kinds of networks, it is these weights that training algorithms adjust.

One of the most common types of networks is the feedforward network. A feedforward network is any network that can be divided into distinct layers, such that there are no connections from a unit in an upper layer to any unit in a lower layer. The lowest layer consists of input units onto which input values are clamped. These values are then passed through a set of weights to produce the inputs to the next layer of so-called "hidden" units. These units take their input and modify them using some function, passing their values to the next layer. This process is continued, with the output of the network being the outputs of the final layer of units. A typical feedforward network is shown in figure 1.1. For simplicity, we have assumed that the function at each unit is the same.

If x_j denotes the output of unit j and w_{ij} denotes the weight on the connection from unit i to j (where w_{ij} can be zero), we can express the output of unit j simply:

$$x_j = f(\sum_i w_{ij} x_i). \tag{1.1}$$

The functions at each unit do not have to be nonlinear but they usually are. In sigmoidal networks, for example, each unit employs a sigmoidal function, such as the logistic function ($f(x) = \frac{1}{1+e^x}$).

One particular kind of feedforward network is the radial basis function network (Broomhead and Lowe, 1988; Poggio and Girosi, 1989). This type of network always contains three layers: a layer of input units, a hidden layer of radial basis function (RBF) units and a layer of output units. Each of the RBF units has a vector of parameters, $\vec{t}_i$, called a center and is connected to the output units by a

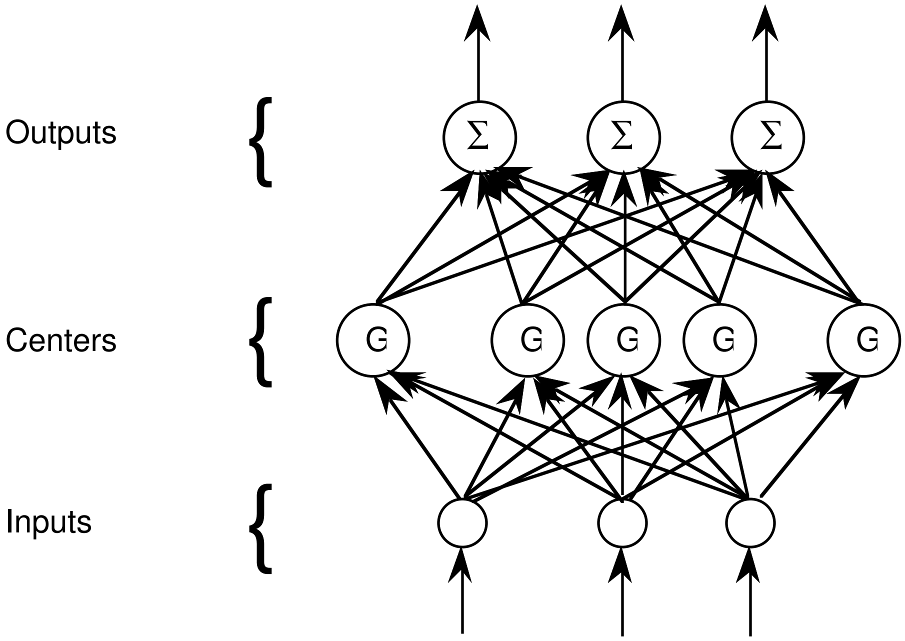


Figure 1.2: An RBF network. All RBF networks have three layers: inputs, centers and outputs.

weighted coefficient vector, $\check{c}_i$. We can express the value of an output unit, j, as:

$$y_j = \sum_{i=1}^{n} c_{ij} G(\|\check{x} - \check{t}_i\|^2). \tag{1.2}$$

$G()$ is a radial basis function, usually a gaussian ($G(x) = e^{-\sigma x^2}$) or multiquadratic ($G(x) = \sqrt{\gamma^2 + x^2}$), and $\|\check{x}\|$ represents the L_2 norm.

In an RBF network, the number of RBF units is equal to the number of training examples with each center, $\check{t}_i$, set equal to one of the training examples. Only the coefficients, $\check{c}_i$, must be learned in this case. This reduces the process of learning to a simple linear problem solvable by matrix inversion (Broomhead and Lowe, 1988).

The RBF network can be generalized by allowing fewer centers than training examples. The centers of this generalized radial basis function network (GRBF) are usually initialized to some subset of the training examples. The centers can then remain fixed or be allowed to change. In general, if the centers remain fixed, the system of linear equations is over constrained; however, the pseudo-inverse can be used to find a mapping with the smallest possible error on the training examples (Poggio and Girosi, 1989).

Poggio and Girosi (1990) have proposed a further generalization: weighting the connections between the input units and the RBF units, effectively replacing the L_2 norm with a weighted norm:

$$y_j = \sum_{i=1}^{n} c_{ij} G(\|\vec{x} - \vec{t}_i\|_W^2) \tag{1.3}$$

where

$$\|\vec{x} - \vec{t}_i\|_W^2 = (\vec{x} - \vec{t}_i)^T W^T W (\vec{x} - \vec{t}_i). \tag{1.4}$$

If the weighting matrix, W, is diagonal (i.e. may only have non-zero values along its diagonal), then for some simple tasks it is possible to interpret each diagonal component of W as indicating the importance or contribution of the corresponding component of the input vectors. A key component, x_i, is exaggerated by a large value for w_{ii} while an unimportant component, x_j, is minimized by a small w_{jj}.

This kind of RBF network, referred to as a HyperBF network, is derived from regularization theory. By imposing smoothness constraints, the ill-posed problem of generalizing a function from input-output example pairs is changed into a well-posed one. Like both the RBF and GRBF networks, the HyperBF network can approximate any continuous function arbitrarily well on a compact, finite set (Poggio and Girosi, 1989).

For the case studies in this thesis, we have chosen to use a GRBF network using the gaussian as the radial basis function:

$$y_j = \sum_{i=1}^{n} c_{ij} e^{-\sigma_i \|\vec{x}-\vec{t}_i\|^2} \tag{1.5}$$

with a distinct, adjustable σ_i for each adjustable center. Results should not be limited to just this type of network.

1.3 Supervised versus Unsupervised Learning

Whatever the kind of network used, training algorithms have been traditionally divided into two major categories: supervised and unsupervised (Hinton, 1987; Lippman, 1987). Generally speaking, a supervised learning algorithm is any algorithm that involves a knowledgeable teacher who provides the correct answer for every input example presented to the network. With an unsupervised learning algorithm there is no teacher and the network is left to discover some useful structure on its own. Of course there is an implicit mapping that the network must learn in any unsupervised algorithm and, therefore, an implicit teacher. In fact, it is sometimes possible to simulate one type of algorithm with an algorithm of the other type.

As such, it may be most practical to describe the differences between supervised and unsupervised algorithms as differences in goals, as opposed to technique:

the object of supervised learning is to approximate a particular input-output mapping while the object of unsupervised learning is to find a mapping which possesses some specific underlying properties that have been deemed important. Both methods have their strengths and weaknesses.

Supervised learning algorithms have met with considerable success in solving some difficult tasks, fairing somewhat better in this regard than unsupervised learning algorithms. This makes some sense. While it is not too difficult to imagine that there might exist a (complex) function that maps, say, bit-image representations of hand-drawn digits to the numbers they represent, it seems a bit harder to imagine an important “underlying principle” that would straight-forwardly accomplish the same.

On the other hand, supervised learning algorithms have been limited by their poor scaling behavior and tend to produce problem-specific representations that do not carry over well to new tasks. Unsupervised learning algorithms seem to do better in this regard. Further, they appeal to the goal of emulating the human learning process, which at least seems to be unsupervised.

Of course, these two categories do not exhaust the possibilities. Clearly there is a continuum of learning types between these two extremes. For example, we could combine some sort of “underlying principle” that generalizes well to many problems with the power of an external teacher. In this way we can guide a network to a final solution which not only performs complex tasks, but chooses functions that capture important underlying structures.

One class of algorithms that approaches the problem of learning in a way that is different than both unsupervised and supervised learning algorithms is the class of temporal difference learning methods. Instead of changing network parameters by means of the difference between predicted and actual outputs, these methods update parameter values by means of the difference between temporally successive predictions (Sutton, 1988). Feedback is usually provided by a teacher at the end of a series of predictions. This combines the principle of temporal (and spatial) coherence—the notion that the environment is stable and smooth and any function that predicts behavior should reflect that notion—with the power of an external teacher.

In the following chapters, we will explore the TD (λ) algorithm, a member of this class, evaluating its usefulness with a number of test cases and exploring whether real-world problems can be better thought of as prediction problems and, perhaps, better attacked by this class of learning algorithms.

Chapter 2

2 Temporal Difference Learning

In this chapter we discuss a class of learning algorithms called temporal difference learning (TD) methods. This is a class of incremental learning procedures specialized for prediction problems. As noted earlier, more traditional learning procedures update parameters by means of the error between the neural net's predicted or proposed output and the actual or desired output. TD learning methods are driven instead by the error between temporally successive predictions. In this way learning actually occurs whenever there is a change in a prediction over time.

The earliest use of a TD method was Samuel's (1959) checker-playing program. For each pair of successive game positions, the program would use the difference between the evaluations of the two positions to modify the earlier position's evaluation. Similar methods have been used in Holland's (1986) bucket brigade, Sutton's (1984) Adaptive Heuristic Critic and Tesauro's (1991)

Backgammon program. Unfortunately, TD algorithms have remained poorly understood. Sutton (1988) has provided a theoretical foundation for their use, proving convergence and optimality for special cases; Dayan (1991) has extended Sutton's proofs and Tesauro (1991) has provided an empirical study of the superiority of TD algorithms in at least one domain; however, it is still unclear how well these algorithms can perform in general with complex, real-world domains or with structures other than linear and sigmoidal networks. To explore these issues, we will first discuss in some detail the different approaches used by temporal difference and more traditional learning methods to solve prediction problems. Then, we will explore the difference between problems of simple prediction and problems of both prediction and control, proposing a general framework for discussing both.

2.1 Temporal Difference versus Traditional Approaches to Prediction

Suppose that we attempt to predict on each day of the week whether it will rain the following Monday. A traditional, supervised, approach would compare the prediction of each day to the actual outcome, while a TD approach would compare each day's prediction to the following day's prediction. Finally, the network's last prediction would be compared to the actual outcome. This forces two constraints upon the neural net: 1) it must learn a prediction function that is consistent or smooth from day-to-day and 2) that function must eventually agree with the actual outcome. The first is accomplished by forcing each prediction to be similar to the prediction following it, while the second is accomplished by forcing the last prediction to be consistent with the actual outcome. The correct answer is propagated from the final prediction to the first.

This approach assumes that the state of the environment is somewhat continuous and does not radically change from one point in time to the next. In other words, the environment is *predictable* and *stable*. If we accept this assumption, the TD approach has three immediate advantages:

1. It is incremental and, presumably, easier to compute.

2. It is able to make better use of its experience.

3. It is closer to the actual learning behavior of humans.

The first point is a practical as well as theoretical one. In the weather prediction example, the TD algorithms can update each day's prediction on the following day while traditional algorithms would wait until Monday and make all the changes at once. These algorithms would have to do more computing at once and require more storage during the week. This is an important consideration in more complex and data-intensive tasks.

The second and third advantages are related to the notion of single-step versus multi-step problems. Any prediction problem can be cast into the supervised-learning paradigm by forming input-output pairs made up of the data upon which the prediction is to be made and the final outcome. For the weather example, we could form a pair with the data at each day of the week and the actual outcome on Monday. This pairwise approach, though widely used, ignores the sequential nature of the task. It makes the simplifying assumption that its tasks are single-step problems: all information about the correctness of each prediction is available all at once. On the other hand, a multi-step problem is one where the correctness of a prediction is not available for several steps after the prediction is made, but *partial* information about a prediction's correctness is revealed at each step. The weather prediction problem is a multi-step problem; new information becomes available on each day that is relevant to the previous prediction. A supervised-learning approach cannot take advantage of this new information in an incremental way.

This is a serious drawback. Not only are many, perhaps most, real-world problems actually multi-step problems, but it is clear that humans use a multi-step approach to learn. In the course of moving to grasp an object, for example, humans constantly update their prediction of where their hands will come to rest. Even in simple pattern-recognition tasks, such as speech recognition—a traditional domain of supervised learning methods—humans are not faced with simple pattern-classification pairs, but a series of patterns that all contribute to the same classification.

2.2 Derivation of the TD (λ) learning algorithm

In the following two subsections we derive the TD (λ) learning algorithm (Sutton, 1988). First we introduce a temporal difference learning procedure that is directly derived from the classical general delta learning rule and induces the same weight changes. With this basic learning procedure defined, we expand it to encompass the much larger and more general TD (λ) learning algorithm which produces weight changes that are different than any supervised-learning algorithm. In the next section, we explore exactly what the TD (λ) procedures compute and how this differs from the more traditional approaches.

2.2.1 The General Learning Rule

We consider the multi-step prediction problem to consist of a series of observation-outcome sequences of the form $\vec{x}_1, \vec{x}_2, \ldots \vec{x}_n, z$. Each $\vec{x}_t$ is a vector representing an observation at time t while z is the actual outcome of the sequence. Although z is often assumed to be a real-valued scalar, z is not prevented from being a vector. For each observation in the sequence, $\vec{x}_t$, the network produces a corresponding output or prediction, P_t. These predictions are estimates of z.

As noted in the first chapter, learning algorithms update adjustable parameters of a network. We will refer to these parameters as the vector $\vec{w}$. For each observation, a change to the parameters, $\Delta\vec{w}_t$, is determined. At the end of each observation-outcome sequence, $\vec{w}$ is changed by the sum of the observation increments:

$$\vec{w} = \vec{w} + \sum_{t=1}^{n} \Delta\vec{w}_t \,. \tag{2.1}$$

This leaves us with the question of how to determine $\Delta\vec{w}_t$. One way to treat the problem is as a series of observation-outcome pairs, $(\vec{x}_1, z), (\vec{x}_2, z) \ldots (\vec{x}_n, z)$, and use the backpropagation learning rule:

$$\Delta\vec{w}_t = \alpha(z - P_t)\nabla_w P_t \tag{2.2}$$

where α is a positive value affecting the rate of learning; $\nabla_w P_t$ is the vector of partial derivatives of P_t with respect to $\vec{w}$; and $(z - P_t)$ represents a measure of the error or difference between the predicted outcome and the actual outcome. This learning rule is a generalization of the delta or Widrow-Hoff rule (Rumelhart et al, 1986).

This is a clear supervised learning algorithm with each $\Delta\vec{w}_t$ depending directly on z. This falls prey to the disadvantages noted earlier. To convert this to a temporal difference algorithm, we must represent the error $(z - P_t)$ in a different way. We can use the "telescoping rule" to note that:

$$(z - P_t) \equiv \sum_{i=t}^{n} (P_{i+1} - P_i) \tag{2.3}$$

if $P_{n+1} \equiv z$. Using this, we can combine equations (2.1) and (2.2) to produce a temporal difference update rule:

$$\begin{aligned}
\vec{w} = \vec{w} + \sum_{t=1}^{n} \Delta\vec{w}_t \quad &= \vec{w} + \sum_{t=1}^{n} \alpha(z - P_t)\nabla_w P_t \\
&= \vec{w} + \sum_{t=1}^{n} \alpha \sum_{k=t}^{n} (P_{k+1} - P_k)\nabla_w P_t \\
&= \vec{w} + \sum_{k=1}^{n} \alpha \sum_{t=1}^{k} (P_{k+1} - P_k)\nabla_w P_t \\
&= \vec{w} + \sum_{t=1}^{n} \alpha(P_{t+1} - P_t) \sum_{k=1}^{t} \nabla_w P_k .
\end{aligned}$$

In other words:

$$\Delta\vec{w}_t = \alpha(P_{t+1} - P_t) \sum_{k=1}^{t} \nabla_w P_k . \tag{2.4}$$

Note the incremental nature of this rule: each $\Delta \check{w}_t$ depends only on a pair of successively-determined predictions and the sum of past values of $\nabla_w P_t$, which can be accumulated with each observation.

2.2.2 The TD (λ) learning algorithm

With equation (2.4), $\Delta \check{w}_t$ is updated in such a way that any difference between P_{t+1} and P_t affects all of the previous predictions, $P_1, P_2, \ldots, P_t$, to the same extent. For some problems, however, it may be preferable to provide some way to weight the gradients, $\nabla_w P_i$, so that more recent predictions are affected the most. To this end, we will consider an exponential weighting with recency, in which the predictions of observation vectors occurring k steps in the past are weighted according to new parameter λ^k where $0 \leq \lambda \leq 1$:

$$\Delta \overset{\nabla}{w}_t = \alpha(P_{t+1} - P_t)\sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k . \tag{2.5}$$

Equations (2.4) and (2.5) are equivalent for $\lambda = 1$. We can therefore refer to (2.4) as a TD (1) algorithm and as a member of the more general TD (λ) family of algorithms.

When $\lambda < 1$, TD (λ) produces weight changes that are different than the more traditional (2.4). This is particularly true with TD (0), when $\Delta \check{w}_t$ is determined solely by the difference between the two most recent observations:

$$\Delta \check{w}_t = \alpha(P_{t+1} - P_t)\nabla_w P_t . \tag{2.6}$$

2.3 Maximum-Likelihood Estimation

It is known that TD (λ), for $0 \le \lambda \le 1$, converges asymptotically to the ideal predictions—at least for absorbing Markov processes and linearly separable data—after an infinite amount of experience (Sutton, 1988; Dayan 1991); however, it is instructive to explore exactly what the TD (λ) procedures compute after a finite amount of experience and to contrast this with the more traditional supervised learning procedures. Following Sutton (1988), we will concentrate on the differences between linear TD (1)—the Widrow-Hoff procedure—and linear TD (0) on linearly separable data, since this is where differences are most clear.

After a finite number of repeated presentations, it is well-known that TD (1) converges in such a way so as to minimize the root squared error between its predictions and the actual outcomes in the training set (Widrow and Stearns, 1985). But what does TD (0) compute after a finite number of repeated presentations? Suppose that one knows that the training data to be used is generated by some Markov process. What might be the best predictions on such a training set?

Probability and statistical theory tell us that if the *a priori* distribution of possible Markov processes is known, the optimal predictions on a training set can be calculated through Bayes' rule; however, it is difficult to justify any *a priori* assumptions about this distribution. In this case, mathematicians use what is known as the *maximum-likelihood estimate*. In general, the maximum-likelihood estimate of the process that produced a set of data is that process whose probability of producing that data is largest. For example, assume we flip a coin 50 times and see a head 41 of those times. We can then ask for the best estimate of the probability of getting a head on the next coin flip. The real answer depends, of course, upon the probability of having a fair coin; however, absent this *a priori* knowledge, the best answer in a maximum-likelihood estimate sense is simply 82% (or $\frac{41}{50}$).

In the sort of prediction problems addressed by TD (λ), the maximum-likelihood estimate can be defined simply. If each terminal observation has associated with it an outcome value, then the best prediction for some state i, in the maximum-likelihood estimate sense, is the expected value of the outcome assuming that the observed fraction of transitions from observation state i to each of the terminal observation states is the correct characterization of the underlying process. In other words, if seeing a particular observation state, i, always leads us to a particular termination observation state, j, then the best prediction for i is the outcome value associated with j. The TD (0) procedure moves toward this maximum-likelihood estimate with repeated presentations of the training data (Sutton, 1988). For other values of $\lambda : 0 < \lambda < 1$, it is harder to characterize exactly what is happening; however, there is some interpolation between the maximum-likelihood estimate and the minimial root squared error calculated by the traditional supervised learning procedure (Dayan, 1991).

2.4 Prediction versus Control

From Samuel's checkers-playing program to Tesauro's backgammon player, temporal difference learning algorithms have been used with some success in the domain of games. Learning in this domain, however, is not just a matter of prediction. If a network is learning to play a multi-step game by predicting the “goodness” of a position, a neural net can actually exercise control over the next position that it will see when its own output is used to pick the next “good” position. In other words, we use the evaluation function that the network is learning to chose its moves in the game.

We will avoid becoming bogged down in the details of any particular task or game at this point by describing problems of learning to predict and control as the problem of finding a good heuristic function for searching on a graph. We shall see that this formalism not only applies to domains such as game playing, but also to domains that do not at first seem to fit well into this description, such as the weather prediction problem.

2.4.1 Searching through a Graph

Any game can be described in the following way: a directed graph made up of nodes or states, $s_1, s_2, \ldots s_n$, and rules for moving from one state to another (the vertices). Some of the states are initial states and some are terminating states. If we assume that a neural net has no built-in knowledge of the rules and is only interested in learning to predict the “goodness” of a given position, we can use the following algorithm to teach it:

$s = s_{initial}$
P = net_prediction(s)

while s is not a final state
  next_list = generate_next_states(s)
  s = best_state(next_list)
  determine ∇P and accumulate the sum of the gradients.
  accumulate the difference between the last prediction and the
    prediction of the new state.
  P = net_prediction(s)

accumulate the difference between the final prediction and the actual
  value for the final state.

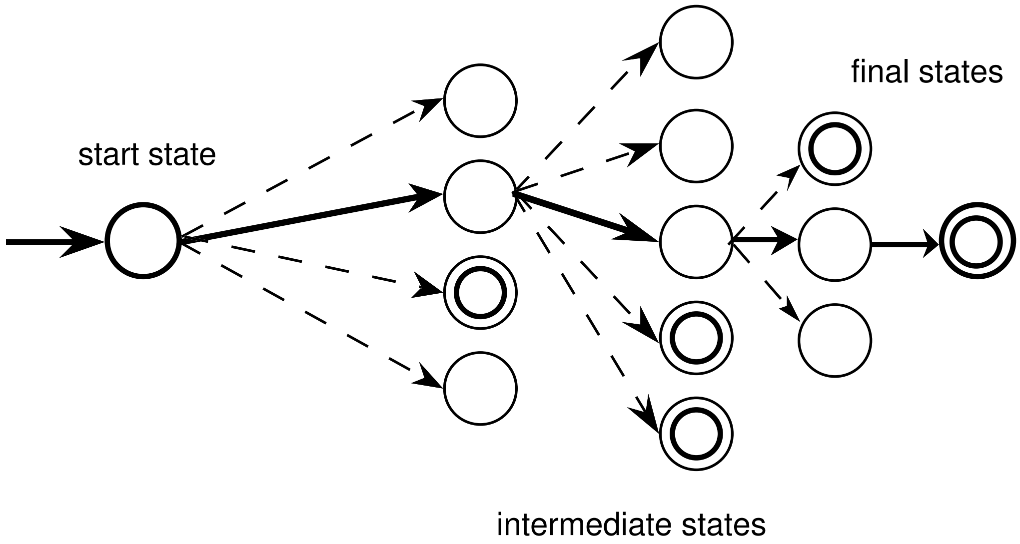


Figure 2.1 : Using ordered depth-first search to choose a path in state space.

In short, we move from observation to observation, using the neural net's current idea of "good" to decide among the next possible observation states. The function *generate_next_states()* embodies the "rules" of the game while the function *best_state()* simply uses the network's current prediction function to provide a value for each possible next state, returning the "best" one (i.e. the state with the highest or lowest prediction value). This general algorithm has been implemented and used for the purposes of studying the test cases in this thesis.

Recasting prediction and control problems in this manner allows us to view these problems as a variation on depth-first search, using a heuristic function to order the nodes. The goal of the neural net then is to learn this heuristic function.

2.4.2 Non-controlling prediction tasks

This approach is not restricted to prediction-control tasks. Our first example, the daily prediction of the likelihood of rain sometime in the future, also fits into this model. In this case, the *generate_next_states()* function simply returns the observation for the next day, making the job of *best_state()* somewhat easy. Similarly, if we wish for our system to passively observe a game, as opposed to both observe and control its direction, *generate_next_states()* can simply return the next position. In fact, as we will see later in this thesis, the approach even works for restricted cases of recurrent networks where the next observation state is actually the prediction from the network itself.

Although the theoretical underpinning for TD (λ) presented by authors such as Sutton prove the equivalence of these methods to their supervised-learning counterparts and provide a strong argument for their superiority in some specific cases, there are several practical issues left unaddressed. In the next chapter, we will use the approach above to identify and explore some of these practical issues.

Chapter 3

3 Practical Issues in TD (λ)

In this chapter we forego our previously theoretical treatment to concentrate on the more practical questions that must be addressed in order for temporal difference procedures to be used effectively. Although some of the issues we discuss here have been explored in some detail by both Tesauro (1991) and Sutton (1984, 1988), they have not been completely addressed. In some cases, their explorations has raised even more questions that have remained largely unanswered.

We categorize these issues into two broad groups: algorithmic and task-dependent. We begin with algorithmic considerations but concentrate mostly on the task-dependent issues as these issues better define the kinds of problems that are likely to be encountered in the real world.

3.1 Algorithmic Considerations

3.1.1 Credit Assignment

In multi-step problems, the sequence of states, $s_1, s_2, \ldots s_n$, are dependent. That is, each state, s_t, bears some relationship to the state, s_{t-1}, that preceded it. In a game like chess, for example, a particular position constrains the positions that follow it. If no pawns are present on the board in some position, for example, no positions that follow can have pawns present. Because of this interdependence, it may be difficult to determine which states have had the most affect on the outcome. Nevertheless, after all the states have been seen by a net and an outcome signal has been presented, the training algorithm must apportion credit to each state, determining in some way which states are most responsible for the final outcome. In our chess example, we might want to know which move was our worst one and actually contributed most to a loss. This is known as the *temporal credit assignment problem*.

There is a similar *structural credit assignment problem*. Each weight parameter in $\check{w}$ contributes in some way to the network's prediction and there must be some way to determine which parameters are most responsible for a correct or incorrect prediction. There are various schemes for determining this fairly well, the most commonly used being gradient descent.

By contrast, temporal credit is often impossible to determine. TD (λ) uses the λ parameter in Equation (2.4) to address this directly. As an exponential weighting scheme, it assumes that the later states contribute the most to the final outcome or, conversely, that the final outcome should affect the prediction function's view of the last states the most. The effect on earlier states will "bubble" back after continued iterations. In a prediction-control task, this corresponds approximately to a depth first search: the values of the states that are furthest from the root are changed first. In other words, the same path in the search space is followed with only the last state in the sequence changing. When all the last states have been tried, the next-to-last state is changed and all the paths from that state are explored. This continues, "bubbling" up the search tree until an optimal path, or heuristic function, is found. Of course, this is only approximate. The states may be related in arbitrary ways and may seem similar enough to the untrained network that a change to the predicted value of the last state in a sequence produces a similar change to the predicted value of one of the earlier states. This analogy is more suited to the searching pattern of the network late in the learning process.

In theory, this depth-first search could be changed to correspond more closely to some sort of breadth-first search. In this way, states *furthest* from the final states are changed first. One possible way to accomplish this is by inverting λ in Equation (2.5) and further restricting its range:

$$\Delta w_t = \alpha(P_{t+1} - P_t)\sum_{i=1}^{t}\left(\frac{1}{\lambda}\right)^{t-i}\nabla_w P_i . \tag{3.1}$$

Another possibility would involve reversing the order of the gradients:

$$\Delta w_t = \alpha(P_{t+1} - P_t)\sum_{i=1}^{t}\lambda^{t-i}\nabla_w P_{t-i+1} . \tag{3.2}$$

It appears that this breadth-first search possibility has not been fully explored. For some non-Markovian problems, including the kinds of recurrent net learning tasks presented in chapter five, this kind of search might be more appropriate than the depth-first approach.

3.1.2 Tweaking α and λ

As with most training algorithms, TD (λ) has a number of its own parameters that can be changed from task to task, most notably λ and α. In practice, it may be best that α changes over time. Initially, large values might help bootstrap the network, but as the net approaches a more stable function, this parameter can be reduced to allow the net to fine-tune itself. Intuitively, the research in this area from the supervised learning community would seem to apply.

On the other hand, it is more difficult to characterize the way in which we should choose λ. Watkins (1989) has pointed out that in choosing the value of λ, there is a trade-off between the bias caused by the error in P at the current stage of learning and the variance of the real terminal values, z. The higher the value we choose for λ, the more significant are the values of P_t for higher values of t, and the more effect the unbiased terminal values will have, leading to higher variance and lower bias. On the other hand, P_t for larger t will have less significance and the unbiased terminal values will have less effect if we lower λ. This leads to smaller variance and greater bias.

3.1.3 Convergence of TD (λ)

Sutton (1988) and Dayan (1991) have proved that TD (λ) converges in the case of a linear network trained with linearly independent data sets. Unfortunately, linear networks are of limited use. More practical applications require the use of non-linearities. In this case, the TD (λ) algorithm may not converge to a locally—let alone globally—optimal solution.

3.1.4 Completion of TD (λ)

Even in cases where TD (λ) will converge, it is unclear how long this convergence might take. The state space upon which the algorithm searches may be infinite or incredibly large (as is the case with chess). Further, in prediction and control tasks, the paths explored by the algorithm may be circular. For example, when controlling a simulated car, the network may end up at some state where it has been before within the current sequence of states. Since the prediction function is not updated until an entire sequence is generated and concluded, this will lead to infinite repetition. In cases where this is possible, some outside agent must be employed to terminate an infinitely repeating sequence.

3.1.5 Sequence Length and the Curse of Dimensionality

The fact that learning time on networks increases exponentially as the dimension of the input increases is known as the *curse of dimensionality*. In the case of the multi-step problems that temporal difference algorithms attack, this problem may extend as well to the length of the observation or state sequences. It is still unclear exactly how well the TD (λ) algorithm scales with the length of state sequences. There is no reason to believe that the algorithm's performance will not degrade exponentially.

3.2 Task Dependent Considerations

Many of the practical issues that we have described above are best understood in the context of specific kinds of tasks. It is the type of task presented to the neural network that drives many of our algorithmic decisions. Good values of λ and α will differ radically from problem to problem, for example. In order to better understand the algorithmic considerations, we present a few types of tasks and discuss their effects upon the particulars of the TD (λ) algorithm.

3.2.1 Prediction and Partial Control

Throughout this chapter and in chapter two, we noted that many problems of prediction are actually problems of both prediction and control. It is worth noting that in some problem domains, such as game theory, we are interested in problems of prediction and only partial control. In a game like chess for example, a network might learn by playing opponents, some of which are considerably more skilled, as opposed to just playing itself.

In this case, the network has only partial control of its environment. It makes a prediction which is used to choose the next state, but then another agent, perhaps an adversarial one, chooses the state that follows. What should the form of the sequence that it sees be?

One possibility is that it should only see the states for which it has predicted values. This would not allow it to take advantage of the information of the states chosen by the other agents. On the other hand, the network might also be presented these states along with the other agent's predictions.

In the first case, the sequence seen by the network is roughly halved, meaning that the importance of each state in changing the parameters of the network is roughly squared. How would this affect the learning rate and the ability to converge? Can this be overcome by simply using a value for λ that is the square root of the value we would normally use? More importantly, does skipping every other state affect the validity of the assumption that there is some temporal continuity?

In the second case, it is unclear how the interaction of these other prediction values with the net's predictions will affect learning. They could simply act as a straight-forward supervised learning signal, as if the network were presented with input-output pairs, or they might have much more complex effects.

In either case, the impact would seem to depend greatly upon both the absolute accuracy of the external agent and its accuracy relative to the network's. In the case of a game, the relative accuracy takes three forms: a vastly inferior opponent, a vastly superior opponent and an opponent of about the same skill. With a vastly inferior opponent and a constant stream of positive feedback, the net might find a solution that only learns to win against poor play while with a vastly superior opponent and continual negative feedback, the net might simply learn to lose, predicting all states to be equally bad.

Further, a huge disparity in ability could lead to an unstable search. Since successive states would probably differ widely from one to another, the series of predictions would also. This would likely prolong learning and increase the possibility of poor behavior. With an opponent of about the same skill, it seems more likely that the network would slowly improve its performance; however, this requires that the opponent's skill level changes to keep pace with the network's ability. Otherwise, the network would end up in one of the situations described above. For the purposes of our examples, this is achieved through self-play.

With the net playing both sides, the probability of each side remaining at equal skill levels is increased.

An interesting question involves the ability of TD (λ) to generalize well in a task involving only partial control. For example, if the network only predicts values for, say, odd-numbered states while an adversary predicts values for even-numbered states, the network may learn a prediction function that only applies to the odd-numbered states to which it as been exposed. It is possible, for instance, that the odd-numbered states have their own substructure and a different function optimizes the predictions for the even-numbered states. If the net then "switches sides" with the adversary and attempts to predict even-numbered states, it may perform poorly.

3.2.2 Prediction and Control Revisited

As touched on above, even the case of complete control by the network is different from the problem of pure prediction. By controlling its own actions, the net runs the risk of finding a self-consistent but sub-optimal predictor-controller. This problem has not been addressed theoretically and may be beyond the scope of TD (λ).

3.2.3 Relative and Absolute Accuracy

One potential problem with the TD (λ) learning rule is that it is designed to teach a network to accurately predict a final outcome, z. Many times in prediction-control problems, however, we are really more interested in the network's ability to choose among several alternatives. For this, it does not need to provide an accurate estimate of the actual "goodness" of a state so much as provide an

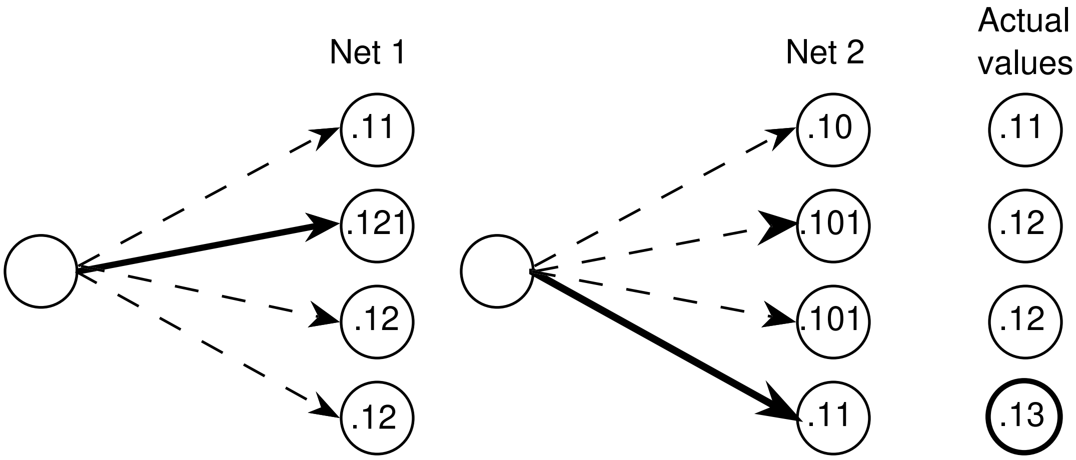


Figure 3.1 : Network 1 predicts values that are very close to the actual values while network 2 has a sum of square error that is more than twelve times as large; however, because network 2 has done a better job of *ordering* the states, it choose the best state.

estimation that accurately *orders* each state; however, the method by which feedback is provided to the network is not always conducive to this goal. In fact, this method may be counter-productive because small errors in absolute accuracy can lead to very large errors in relative accuracy.

Similarly, since the error signal is measuring the absolute accuracy of the network's prediction instead of its relative accuracy, it is difficult to analyze and determine how well the algorithm is really doing. For an example of these problems, see figure 3.1.

3.2.4 Random Tasks and Noise

It is worth noting that Sutton (1988) and others have performed analysis with noise-free, deterministic tasks. Although, Tesauro (1991) has explored teaching a net to play backgammon—a game which involves randomness or noise in the form of dice—it is not yet know how the introduction of other kinds of noise affects TD (λ).

3.2.5 Representation

As with all neural network training algorithms, temporal difference procedures are sensitive to the representations chosen for both the data and the output. Representations can be designed so that they explicitly contain a wealth of relevant information or can be designed plainly, so that a neural net attempting to generalize must somehow learn to represent a great deal of some underlying structure. Without enough information in the representation it may be extremely difficult to generalize.

A representation issue that is of particular importance to TD (λ) is the linear dependence of the observation vectors. Dayan (1991) shows that if the observation vectors presented to the network are not linearly independent, then TD (λ) for $\lambda \neq 1$ converges to a solution that is different than the least means squares algorithm, at least for linear networks. In this case, using the inaccurate estimates from the next state, $P(\check{x}_{t+1})$, to provide an error signal for the estimate of the current state, $P(\check{x}_t)$, may not be harmless. With linearly dependent observation vectors, these successive estimates become biased on account of what Dayan has deemed their "shared" representation. The amount of the extra bias between the estimates is related to the amount of their sharing and the frequency with which the transitions occur from one state to the next. So, while TD (λ) for $\lambda \neq 1$ will still converge, it will be away from the "best" value to a degree determined by the matrix:

$$\left(\mathbf{I} - (1-\lambda)\mathbf{Q}[\mathbf{I} - \lambda\mathbf{Q}]^{-1}\right),$$

where $\mathbf{Q}$ is the square matrix of transition probabilities. It remains unclear exactly how this affects the usefulness of TD (λ) for typical problems.

3.2.6 Lookup Tables

If there are enough parameters available for a task, a network can act as a lookup table by explicitly storing the values of the training data. In the case of RBF networks, this requires one center or RBF node for every member of the training data.

Sutton's proof for convergence relies on a lookup-table approach and therefore requires that every possible state be visited an infinite number of times. This is impractical with real world problems.

3.2.7 Maximum-Likelihood Estimates and Non-Markovian Tasks

As seen in section 2.3, Sutton's convergence and optimality proofs rely on the assumption that the tasks are absorbing Markov processes. For these kinds of Markovian processes, TD (λ) computes a maximum-likelihood estimate, arguably a desirable feature. On the other hand, it is unclear how useful these procedures can be with non-Markovian processes. For example, we may have a task where the observation state c in the sequence $(\ldots, a, c, \ldots)$ should be viewed differently than when the same observation c is proceeded by a different state, $(\ldots, b, c, \ldots)$.

If there no straightforward and computationally tractable way of encoding this in the data or some way for TD (λ) to discover it—and the latter is probably not the case—then TD (λ) may produce very inaccurate predictions and have no way of correcting them.

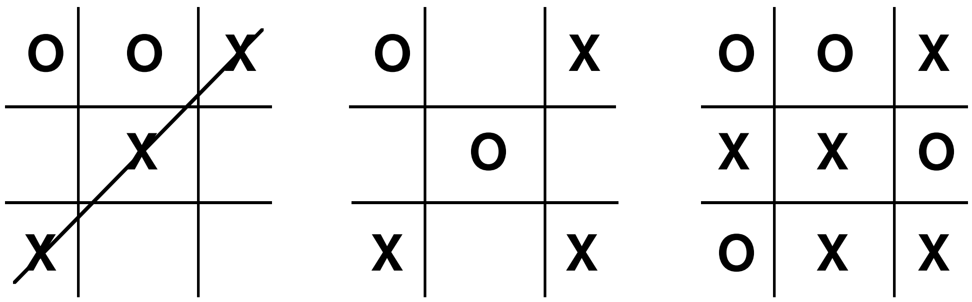
Chapter 4

4 Example: Tic-Tac-Toe

In this chapter and the next, we explore several case studies that ground some of the issues that have been raised in the previous chapters in particular contexts. Each experiment in this chapter used the approach that was outlined in section 2.4.1.

4.1 Tic-tac-toe

Our first case study involves the game tic-tac-toe. Tic-tac-toe is a two-player game played on a three-by-three grid. Each player is represented by a token, usually **X** and **O**. For our purposes, we will assume that **X** always goes first. Players take turns placing a token in an empty spot on the grid. A player has won the game when she has placed three of her tokens in a row, either vertically, horizontally or diagonally. The game is a draw if all nine spots are filled and



a b c

Figure 4.1 : a) shows a win for **X**, b) shows a “fork” for **X**, meaning that **X** has two ways to win on the next move and figure c) shows a draw.

neither player has placed three of her tokens in a row. Some game positions are shown in figures 4.1 and 4.2.

Tic-tac-toe is a completely deterministic game with a finite number of states. In fact, the size of the state space can be reduced to only a few hundred by taking advantage of the game’s symmetrical nature. Best play by both sides will always result in a draw. The following simple set of rules describes best play in any position:

> if you can place a token immediately and win, do so.
> if your opponent can win on her next move, block her.
> if you can “fork” (i.e. place a token such that you have two ways to win on your next move), do so.
> if you can place a token on a square so as to force a block on the next move by your opponent *and* that by making that block, she will not “fork” you, place your token on that square. Prefer corner squares to non-corner squares.
> if your opponent can fork, block her.
> if the center position is free, take it.
> if a corner is free, take it.
> if none of the other rules apply, place your token randomly.

For each of these rules, it is possible that more than one state will satisfy the condition. In this case, a player can simply pick one randomly.

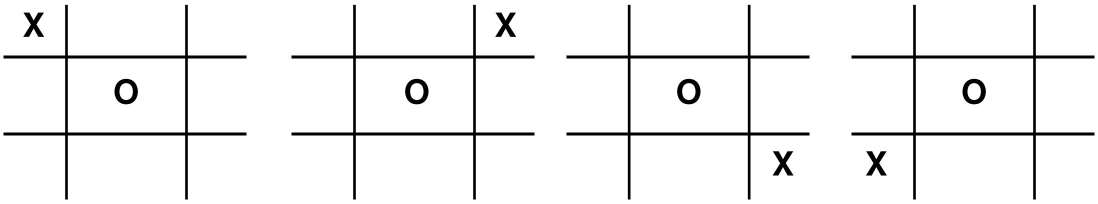


Figure 4.2: Each of these positions is equivalent as they can all be rotated or flipped into the same prototypical positions. The rightmost position, for example, can be rotated clockwise 90˚ and become the leftmost position.

4.2 Practical Issues in Learning Tic-tac-toe

As a two-player adversarial game, tic-tac-toe provides an opportunity to explore many of the issues that have been described in earlier chapters. In particular, it allows us not only to test the ability of TD (λ) to solve prediction-control problems but to compare its ability to solve prediction tasks that allow complete control with prediction tasks allowing only partial control. To this end, we shall see how well a network can learn by self-play as well as by playing against an opponent who, in our experiments, always follows an optimal strategy. We will also explore the ability of a these latter nets to generalize, by testing their ability to "switch sides" and provide accurate predictions for states that they have not seen.

As a game with a relatively small number of states, the network *should* be able to visit almost every state in the course of learning. Further, since this game has a relatively short number of states in any sequence—at most nine with self-play and five against an opponent—we should not have to concern ourselves with this particular version of the curse of dimensionality.

Although tic-tac-toe is a relatively simple game to learn to play well, there are some key positions that are very bad. For example, on the first two moves, a

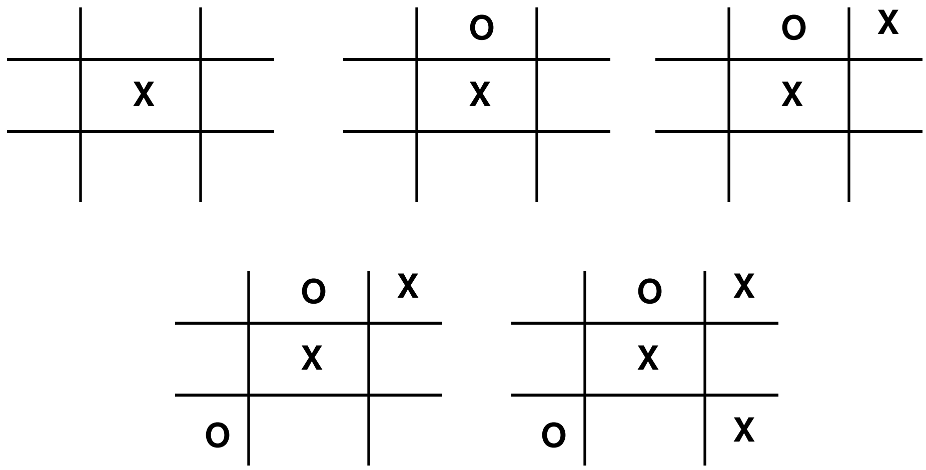


Figure 4.3: If **X** places her token in the center square on the first move, **O** cannot place hers in a non-corner square. **X** is able to force a "fork" no matter which non-corner square **O** chooses because of the symmetrical nature of the game.

player facing an expert opponent is guaranteed a loss by placing an **O** in a non-corner square in response to an **X** being placed in the center square (see figure 4.3). Because the "good" and "bad" moves are somewhat clear-cut and always deterministic, it is easier to determine the quality of the ordering capability of an evaluation function, independent of its absolute "error."

Finally, it is worth noting that tic-tac-toe positions contain a great deal of structure. Thousands of positions are collapsible into several hundred by symmetry alone. A network learning the most compact function for this problem would find some way to represent this information.

4.3 Experiments with Tic-tac-toe

Four experiments were conducted. One experiment involved a network learning to play tic-tac-toe through self-play while the other networks learned by playing against an opponent employing the strategy described in section 4.1, randomly choosing among all the available best moves in any given position. Against this "perfect" opponent, one network always played **X**; another always played **O** and the last alternated between **X** and **O**.

Each experiment used a GRBF network with 200 centers. Each of the experiments began with the same initial parameter values. The first 425 iterations were treated as a bootstrapping phase. In this phase all but one of the initial board positions had eight of their slots filled. The exception was the blank board. The next 575 iterations added all the board positions with seven slots filled. For the final 1000 iterations, we chose one fifth of all the possible board configurations for starting positions.

For all of these experiments, the value of λ was set to 0.6. The value of α was decreased after each phase, with the assumption that accuracy would suffer for large values of a during the last phases of learning. This assumption was verified by some initial results.

Each board position was represented by a ten-dimensional vector. The first nine components represented the tokens placed in each of the nine squares on the tic-tac-toe board, numbered from left to right and top to bottom while the last component determined which player's turn it was to move. **X** was always represented by -1, **O** by 1 and an empty square by the value 0.

The output was a vector in three dimensions, representing the probability of **X** winning, the probability of **O** winning and the probability of a draw, respectively.

The *best_state()* function computed a scalar value from these three probabilities to determine the ordering of possible states:

$$E = \Pr(i) - \Pr(j) + \frac{\Pr(draw)}{2} \qquad (4.1)$$

where i represents the player with the current turn and j her opponent. In the case of a certain win for i, $E = 1$; when i is certain to lose, $E = -1$; and when a draw is certain, $E = \frac{1}{2}$. This tends to make drawing much more like winning than losing. In the case of tic-tac-toe, where best play by both sides always leads to a draw, this seems like a desirable trait.

4.4 Tic-tac-toe Results

There are several general results: the neural network trained through self-play performs the best against the optimal opponent; each network learned to accurately predict most drawing positions; all networks perform best when playing **X**; none of the networks learned the underlying symmetrical structure of the tic-tac-toe positions; and while the networks may play well against a rational opponent, an irrational opponent can defeat them.

It is not too surprising that the networks learned to play better when playing **X**. The design of the first bootstrap phase only allowed for one move, and that move was always by player **X.** Therefore, there was a great deal of experience—even for the network that played **O**—with the positions for the first player.

The design of the bootstrap phase also provided early experience for positions leading to draws since many of those bootstrap positions necessarily lead to a draw. The next phase—which only allowed for a maximum of two moves—often forced draws as well. Given this initial bias and the fact that most lines of play in tic-tac-toe lead to draws, it is also unsurprising that all the nets learned to predict draws fairly well.

The choice of input representation was a deliberate one, designed to provide the minimum of information. As such it may not be surprising that the networks did not learn some way to transform the data and recognize symmetries. It is possible that allowing a network to change its **W** matrix (thus moving from a GRBF to a HyperBF network) might facilitate this somewhat.

Even when a network played well against a good opponent, it would fail against a bad opponent. When playing against these TD (λ) networks, an opponent had only to ignore the network when it had a pending win. Instead of blocking that win, an opponent would be better served to set up her own win. In some positions, the network would rather block her thrust than actually win while in others it would simply make some other move. One way to explain this is to assert that the networks learned to play defensively, preferring “not losing” to “winning.”

It may seem that some of this is due to the form of equation (4.1); however, a choice function that makes the network a more “aggressive” player by placing the value of drawing exactly between the values for winning and losing:

$$E = 2\Pr(i) - \Pr(j) + \frac{\Pr(draw)}{2} \tag{4.2}$$

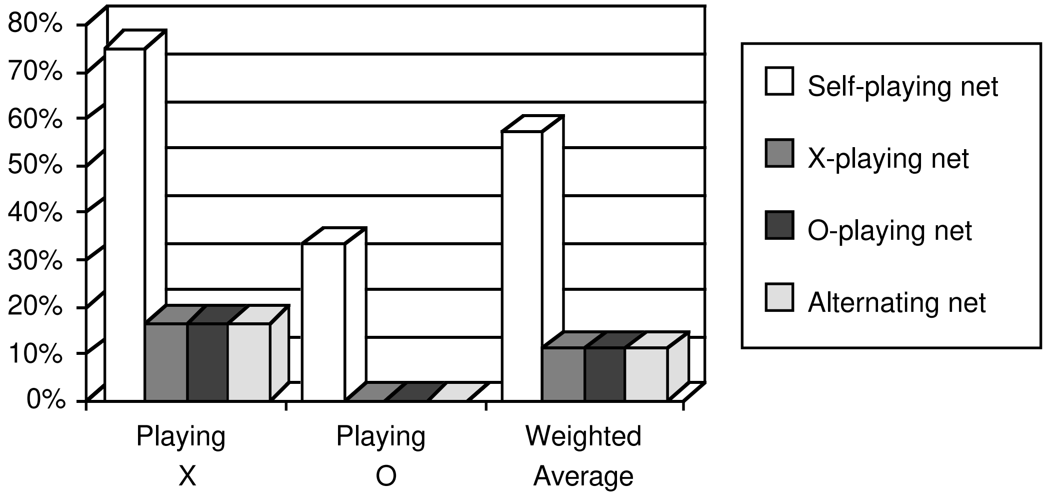


Figure 4.4: The percentage of times each net drew against the optimal player. Since the networks were deterministic there are only a small number of possible games each net can play against the optimal strategy. For example, the self-playing net as **O** always responded to an **X** in the center of the board by placing an **O** in the bottom left-hand corner. From here, given the network's level of play and the strategy used by our optimal player, there were only three possible lines of play that could follow.

seemed to have no effect on the lines of play chosen by any of the networks, at least after many training epochs.

4.4.1 Self-playing networks

As a matter of strategy, the self-playing network learned to place its token in the center position when playing **X**. The four possible next "best" moves are all symmetric (see figure 4.2); however, the network only learned to draw against three of them. In fact, the network learned non-symmetrical strategies for each of the positions it learned to draw against.

As **O**, the network played its token in the bottom left-hand corner, one of the four best moves against an **X** in the center; however, it could only manage to draw one third of the time. It is worth noting that one of the losses is due to its inability

O / X	Self-playing net	X-playing net	O-playing net	Alternating net
Self-playing net	draw	O	O	O
X-playing net	draw	draw	draw	draw
O-playing net	draw	draw	draw	draw
Alternating net	draw	draw	draw	draw

Figure 4.5: Results of each of the networks playing against each other and themselves. Although the self-playing net seems to be the best player against an optimal opponent, it manages to lose against the sub-optimal networks.

to prevent a fork. It seems that it has not learned to prefer corner squares to non-corner squares when forcing a block.

In short, it appears that the network has learned the importance of the center and the corner squares fairly well in the most common lines of play, mastering some of the opening. While it seems to have learned much less about middle game strategies such as forking, its choices for second moves tend to lead to a series of forced moves. When it follows the forced moves, it tends to draw.

The network taught by self-play learned to play against the optimal player better than any of the other networks. It is possible that the network performs best because it has had the most "stable" opponent and the most stable set of inputs. Each state was similar to the ones surrounding it. On the other hand, the networks facing an opponent saw vastly different states from time point to time point and almost always had large temporal errors. There is evidence that suggests that it is difficult to learn well under these circumstances (Poggio, personal correspondence).

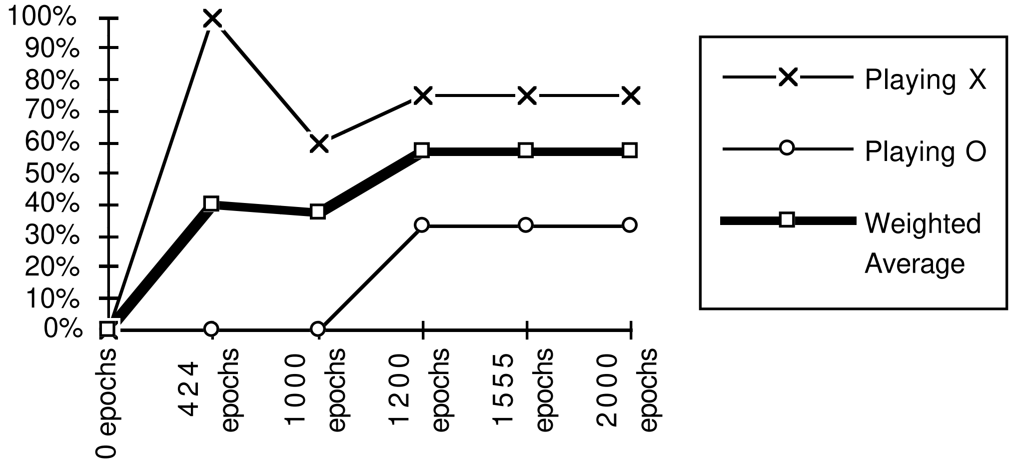


Figure 4.6: The percentage of times the self-playing network drew against the optimal player as a function of the number of epochs the network has experienced. Although the percentage of games drawn seemed to level off, the prediction function used by the network continued to evolve.

As noted before, these networks often had difficulty playing against an opponent who was irrational. In fact, even though the self-playing network played best against the optimal opponent (see figure 4.4), it would sometimes lose against the networks which played "worse" (see figure 4.5).

It is worth noting that the self-playing network seemed to be the only one to choose moves from among the most positive possibilities instead of from the least negative possibilities. For example, when choosing among the best first move for **X**, this type of network was the only one whose prediction function generated numbers with positive values for equation (4.1). Each of the others generated only negative values. This is due in part because the self-playing network learned to predict values near zero for the probabilities of either **X** or **O** winning and values nearer to one for the probability of a draw. It is also due to this network coming closest to representing its output as probabilities. The other networks produced outputs with components as high as 2.9 and values for equation (4.1) as low as -3.1.

4.4.2 Opponent-playing networks

None of the networks trained against the perfect opponent “learned to lose” in the sense that they never drew or learned a function that drove all states to “bad” values. Analysis suggests that the depth-first analogy fits well in this case. A game sequence was repeated several times with the last states in the sequence given lower and lower scores. Eventually, the last state received a score so low that the sequence changed at that point. If none of these changes led to good play, the states earlier in the sequence had accumulated enough changes that the sequence changed at those points instead. Eventually, the network could not help but to stumble upon a good—or rather a non-disastrous—line of play.

Even though each of these networks learned different prediction functions, each network learned to order states in more or less the same way. Thus, each net played the same games against each other and the optimal opponent. This is somewhat counter-intuitive since the X-playing network and the O-playing network were exposed to different lines of play.

Like the self-playing network, each of these nets learned to play **X** in the center as a first move. Their prediction functions and the optimal strategy allowed for six possible lines of play. The opponent-playing networks could only draw in one of these lines. Of the five losing lines of play, the networks lost to a fork only once. In general, the networks failed to block immediate wins. This was as true for the X-playing network as it was for the other networks.

This same problem surfaced with these networks when they played **O**. Even though the networks played a corner square as a first move against an **X** in the center, they simply never blocked immediate wins. Therefore, each network lost every game playing **O**, including the O-playing network.

Many of these problems seem to be traceable to a simple phenomenon. The probability predictions for the final positions are much more accurate than the early predictions. It appears that the sequence length of the states for these networks was short enough that the negative feedback for the last states had a strong effect on the middle states. Useful signals about the final states in a sequence leading to an evaluation function that would choose appropriate final moves was confounded by these signals being associated with middle states too strongly. Instead of minute changes in the evaluation function and incremental improvement, the network experienced large changes and played sometimes vastly different games from training epoch to training epoch. As noted before learning is often difficult in these circumstances. This is especially noticeable with the network that alternated play between **X** and **O**. This network saw very different lines of play over a short time and developed correspondingly different evaluation functions, sometimes oscillating between evaluations.

In theory, these opponent-playing networks could eventually play as well as the self-playing network against expert play. The depth-first search nature of the training still keeps the network away from learning to lose. Unfortunately, even if the networks can improve their play against the optimal opponent, it appears that they will take much longer than the self-playing network and follow too many fruitless paths in the interim.

For at least the first several series of training phases, self-play seems superior. Training a network by having it play against an optimal or near-optimal opponent might not be useful until a self-trained net has already learned a fairly good evaluation function on its own. At this point, with a sufficiently small learning rate, occasional negative feedback might help to jostle it from a self-consistent but suboptimal evaluation function.

Chapter 5

5 Example: Recurrent Networks

In this chapter, we use the problem of simple segmentation to explore the use of TD (λ) with neural nets that are not structured as feedforward networks—namely a restricted subclass of so-called recurrent networks—in order to learn complex functions that can be described as the iteration of simpler functions. Each experiment in this chapter uses the approach outlined in section 2.4.1.

5.1 Function Iteration

It is sometimes possible to describe a complex function as the iteration of a simpler function:

$$\check{F}(\overset{\vee}{x}) = \lim_{n \to \infty} \check{f}^{n}(\overset{\vee}{x}). \tag{5.1}$$

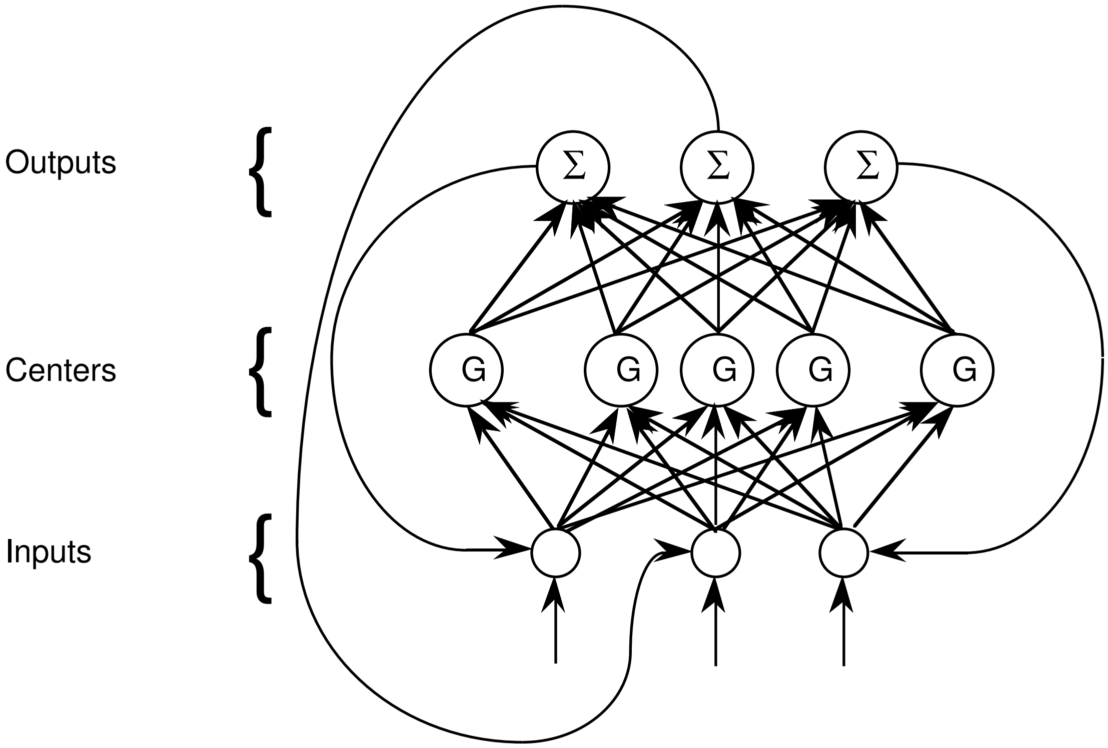


Figure 5.1 : A HyperBF network with recurrent connections.

Phrased in the language of a neural network, the outputs of the network are fed back into the inputs. Instead of each unit in a network calculating a value once, a unit updates its value each time its input changes. There is a class of networks that uses this principle called *recurrent networks* (see figure 5.1).

These networks are equivalent to feedforward networks in that they can approximate any function arbitrarily well. Beyond this, they have been used for various tasks, including dimensionality reduction, with some success (Jones, 1992).

This formulation of a function can be useful for other reasons as well. For some problems, $\check{f}(\overset{\vee}{x})$ may be an easier function to learn than $\check{F}(\overset{\vee}{x})$ and this ease of learning is worth the trade-off in computation time. In other words, while $\check{F}(\overset{\vee}{x})$ is learnable, it may be extremely difficult for backpropogation or other methods to tune the parameters of the network in order to duplicate the function. On the other hand, $\check{f}(\overset{\vee}{x})$ may be a simpler function to emulate.

These recurrent networks raise serious questions of their own. For example, it is not guaranteed that the outputs of a particular recurrent network will become stable and cease changing. However, these problems are beyond the scope of this thesis and so we will only concern ourselves with the case where the output of the network achieves a fix point.

5.2 Segmentation

Segmentation is the problem of taking a vector or matrix of values and returning a vector or matrix of equal size that groups similar values. In segmenting an image, for example, an algorithm would take as input a matrix of pixel values and output a transformed image where each pixel in an object has the same value. It is assumed that adjacent objects in the original image are colored differently enough that we can distinguish between them. If this is not the case, the algorithm may blend adjacent objects and produce an output image that looks like a single large object.

In this chapter, we will explore two methods of segmentation. One very simple approach is to decide the range of pixels values that are possible and to drive all values below some threshold to the minimum value while driving all values above that threshold to the maximum value (see figure 5.2). This technique turns a "color" image into a black and white image. Determining the periphery or edges

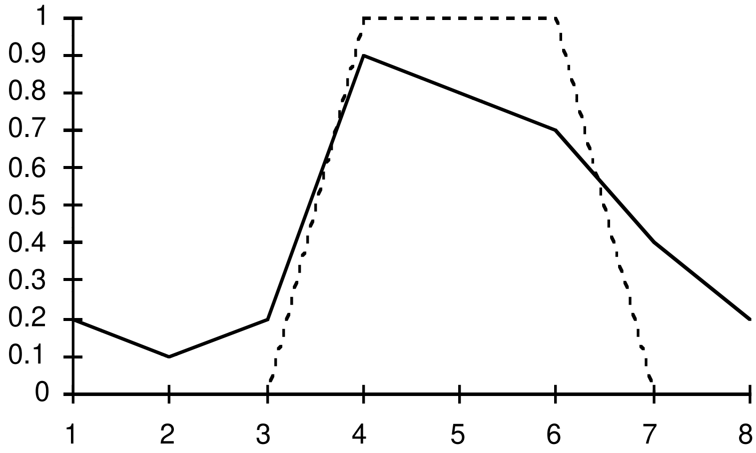


Figure 5.2: Each value below 0.5 along the solid line is driven to zero while each value above 0.5 is driven to one, yielding the segmented dashed line with one-dimensional "objects" between points 1 and 3, points 3 and 7 and points 7 and 8.

of objects is then rather straightforward: simply find the points where "black" borders "white."

A more complicated approach to this problem is to continually average the values of pixels that are spatially close whenever their values are sufficiently "near" one another. The averaging continues until it produces no new changes (Hurlbert, 1989). The idea is to bring the values of the pixels within each object closer together while widening the gap between pixel values across objects (see figure 5.3).

Both of these segmentation algorithms can be formulated as recurrent problems. Hurlbert's algorithm is perfectly suited to this. In fact, it is described as a recurrent problem. In practice, the network only has to learn to perform an approximation towards local averaging at each time step. If this approximation is more or less accurate, the recurrent nature of the network will drive it to the correct final answer.

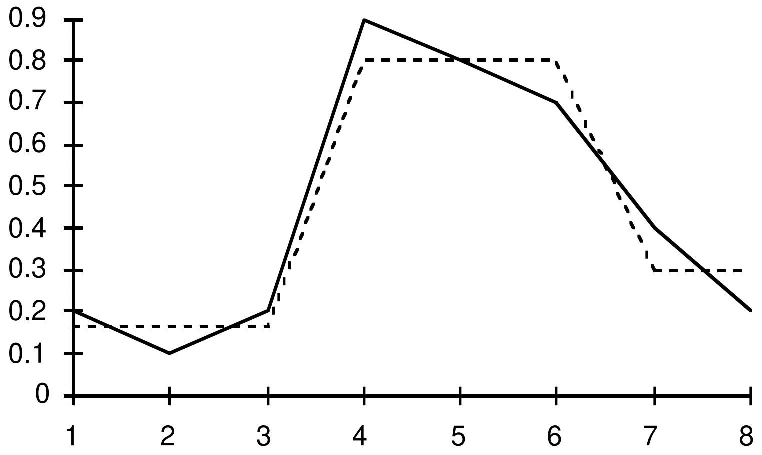


Figure 5.3: Using Hurlbert's algorithm, the segmentation of the solid line from figure 5.2 yields the same three "objects." However the pixel values found for each object are different and more representative of the original "image."

Similarly, the black-and-white algorithm can be successfully implemented so long as pixel values less than the threshold always move towards the minimum and pixel values above the threshold always move towards the maximum on each time step. Given enough recurrent iterations, the network should arrive at the correct answer.

5.3 Practical Issues in Learning Recurrent Segmentation

The recurrent network poses an interesting problem of prediction and control. The *get_next_states()* function simply returns the prediction of the network on the current state. So while in some sense the network is producing a control signal, it is more accurate to say that it is creating its own states as it goes along. Because of this, the state space is infinite. In addition, the sequence of states generated by the network can be infinitely long. As we have noted before, both of these possibilities can present difficulties for the TD (λ) algorithm.

The iterative approach of recurrent networks, particularly in the case of segmentation, would suggest that each successive state should be very similar. This would indicate that a small value for λ, perhaps even zero, would be most appropriate. On the other hand, this assumption may be invalid. The task is not described well by an absorbing Markov process and so may prove difficult for values of λ close to zero, as seen in section 3.2.7.

5.4 Experiments with Simple Recurrent Segmentation

The performance of the TD (λ) algorithm was tested with the one-dimensional versions of both segmentation algorithms. For both problems, experiments were conducted for values of λ beginning with zero and incremented by 0.1 until λ reached a value of one.

5.4.1 Black-and-White Segmentation

The networks trained on the black-and-white segmentation problem were GRBF networks containing sixteen centers. Since the function for learning this task need only work on one pixel at a time, the input to this network was simply a scalar. The output was redirected into the network four times. All input values ranged between zero and one with a threshold value of 0.5. All values less than the threshold were associated with zero and all values greater than or equal to the threshold were associated with one.

5.4.2 Hurlbert's Segmentation

The networks trained on the Hurlbert segmentation problem were GRBF networks containing twenty eight centers. The input to this network was an eight-dimensional vector. The output was redirected into the network eight times.

The correct output was determined by repeatedly averaging a pixel with its neighbor using a simple function:

$$x_i' = \begin{cases} \dfrac{x_{i-1} + x_i + x_{i+1}}{3}, & |x_i - x_{i-1}| \le \sigma \text{ and } |x_i - x_{i+1}| \le \sigma \\ \dfrac{x_{i-1} + x_i}{2}, & |x_i - x_{i-1}| \le \sigma \text{ and } |x_i - x_{i+1}| > \sigma \\ \dfrac{x_{i+1} + x_i}{2}, & |x_i - x_{i-1}| > \sigma \text{ and } |x_i - x_{i+1}| \le \sigma \\ x_i, & |x_i - x_{i-1}| > \sigma \text{ and } |x_i - x_{i+1}| > \sigma \end{cases} . \tag{5.1}$$

with $1 \le i \le 8$. For these experiments, σ was set to 0.2. Pixel values just off the edges of the input vector $(i = 0 \text{ and } i = 9)$ were assumed to be equal to ∞ in order to deal correctly with the boundary conditions. Equation (5.1) was applied in a recurrent manner eight times, after which most outputs ceased changing in any significant way.

5.5 Segmentation Results

Graphs of the root squared error for each experiment can be seen in figure 5.4. For both kinds of problems, the networks with lower values for λ exhibited the worst performance. In fact, once λ fell below a certain value, the root squared error remained more or less constant. This is because each of these networks learned approximately the same function. For all input values, the network

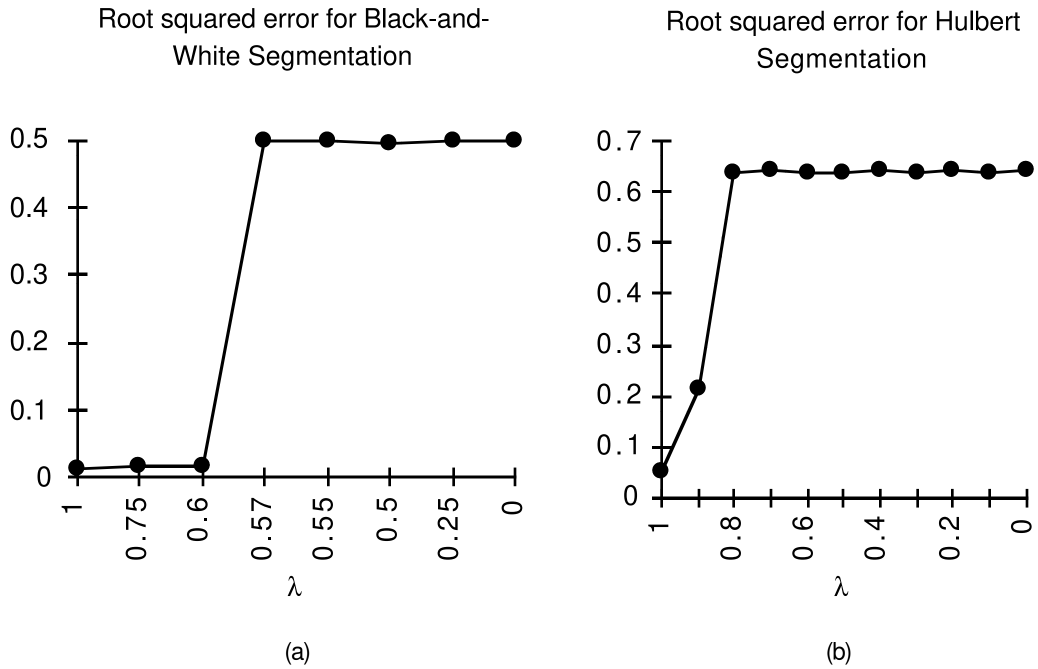


Figure 5.4: In a) each network was trained on ten values between one and one such that half of the correct outputs were equal to zero and the other half equal one. Networks trained with values for $\lambda < 0.6$ learned a constant function, driving all inputs to the value 0.5. In b) each network trained with values for $\lambda < 0.9$ also learned a constant function, driving all inputs to the same output vector.

produced the same constant value at each stage of the iteration. The value of this constant depended only on the distribution of outputs in the training set.

In the case of the black-and-white segmentation problem, the constant function learned by these networks was simply the percentage of times that the value 1.0 occurred in the output of the training set (i.e. the average of the outputs). For a training set with half the outputs equal to zero and the other half equal to one, the function learned was: $f(x) = 0.5, 0 \le x \le 1$; for a training set with zero appearing only 30% of the time, the function learned was: $f(x) = 0.7, 0 \le x \le 1$ and so on.

Similarly, the networks training on the Hurlbert segmentation task learned a constant function that depended on the distribution of the outputs of the training set. In general, the constant function learned was simply the average of the outputs presented to the network.

To understand this result, assume that our training data for the black-and-white segmentation problem consists of two values, 0.4 and 0.8. Let us posit these two sequences seen by the network early in the learning procedure:

0.4:	0.2	0.5	0.8	0.7	0.9
0.8:	0.3	0.9	0.6	0.4	0.2

Naturally, we want our training values, 0.4 and 0.8 to be associated with zero and one, respectively. Note, however, that 0.8 also appears in the first sequence and 0.4 appears in the second. This means that zero will be associated with both 0.4 *and* 0.8. Similarly, one will be associated with 0.4 as well as 0.8.

Since the network sees that one and zero are associated with 0.4 and 0.8 equally, a maximum-likelihood estimator naturally associates these values with their average, 0.5. Similarly, other values in the sequences above will be associated with 0.5 as well. In other words, for values of λ approaching zero, the network does exactly what we would expect it to do. A similar problem occurs with the Hurlbert segmentation algorithm. So the question becomes: how can we overcome this so that the network will learn the function that we want?

The answer lies in the non-Markovian nature of the recurrent task. There is no way to distinguish between the 0.4 in the beginning of the first sequence and the 0.4 that occurs in the middle of the second sequence; however, in this task they **are** distinctly different entities. The problem is that our "state" information is too impoverished.

This problem occurs even in the weather prediction problem discussed in chapter two. There we were using weather data to predict whether it would rain on a particular Monday. Let us suppose that in one sequence we have a rainy day on the previous Tuesday and in another sequence we have a rainy day on the previous Sunday. If the Monday in the first sequence is sunny and the Monday in the second is rainy, we will associated a 50% probability of sunny Mondays with rainy days. In actuality, a rainy day six days before the day we wish to predict isn't as useful a predictor as a rainy day one day before the day we wish to predict. Unfortunately, we have not distinguished Tuesdays from Sundays.

One way to address this issue is to find some way to "tag" the states so as to distinguish initial values from values that occur later in a sequence. This possibility is explored in the next sections.

5.6 Experiments with "Tagged" Recurrent Segmentation

As before, the performance of the TD (λ) algorithm was tested with the one-dimensional versions of both segmentation algorithms. Again, experiments were conducted for values of λ beginning with zero and incremented by 0.1 until λ reached a value of one.

5.6.1 Black-and-White Segmentation

In this experiment, the scalar input to the network was supplemented by an additional component. The two components of the initial vector were both set to the same scalar value. No matter what the two-dimensional output from the

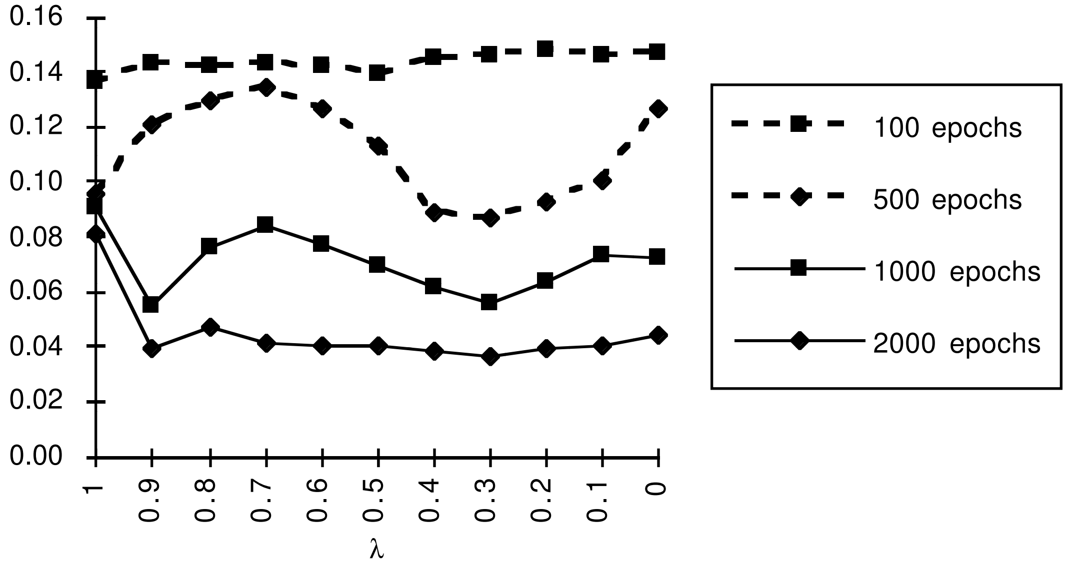


Figure 5.5: The root squared error for eleven values of λ averaged over several training trails on the black-and-white segmentation task. As training continued, the improvement rate of the supervised network slowed.

network on a given input, the second component was always transformed into the initial scalar value before being redirected into the network as input. In this way, initial states were distinguished from intermediate states.

As before, the output of the network was redirected into the network four times. All component input values ranged between zero and one with a threshold value of 0.5. For the first component, all values less than the threshold were associated eventually with zero and all values greater than or equal to the threshold were associated with one. The second component of the final output vector presented to the network was always the initial value given to network.

5.6.2 Hurlbert's Segmentation

The input to this network was doubled from an eight-dimensional vector into a sixteen-dimensional vector and the output from the network transformed in a way

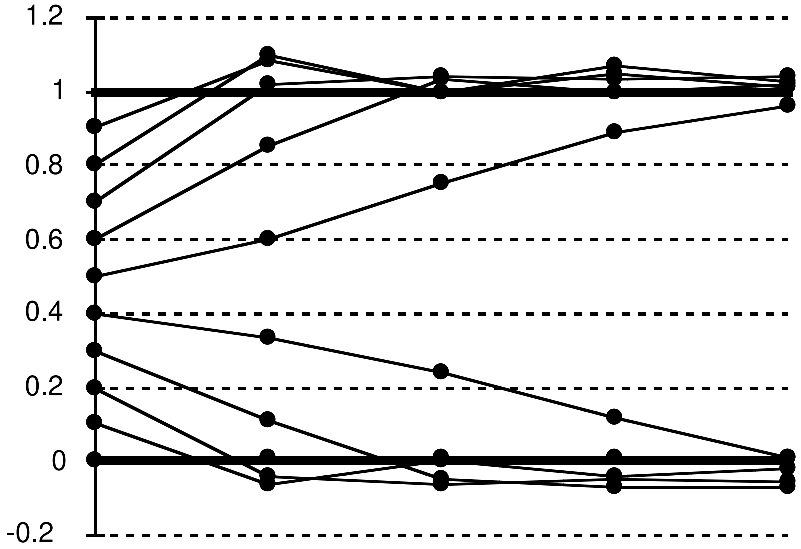


Figure 5.6: A typical series of sequences generated by the network. This is from one of the networks trained with λ=0.2 after 1000 epochs.

similar to that done with the black-and-white segmentation problem. As before, the output was redirected into the network eight times.

5.7 Segmentation Results

5.7.1 Black-and-White Segmentation Results

Graphs of the root squared error can be seen in figure 5.5. Unlike before, the networks with values of λ near zero did not learn a constant function. In fact, the networks trained with values of $\lambda \neq 1$ became increasingly better with practice, outperforming the traditional "supervised" learning procedure.

Figure 5.6 shows the sequences generated by one of the networks. Typically, values were moved towards the correct final value in fairly uniform increments. Once at the desired value, subsequent outputs remained at or near this value.

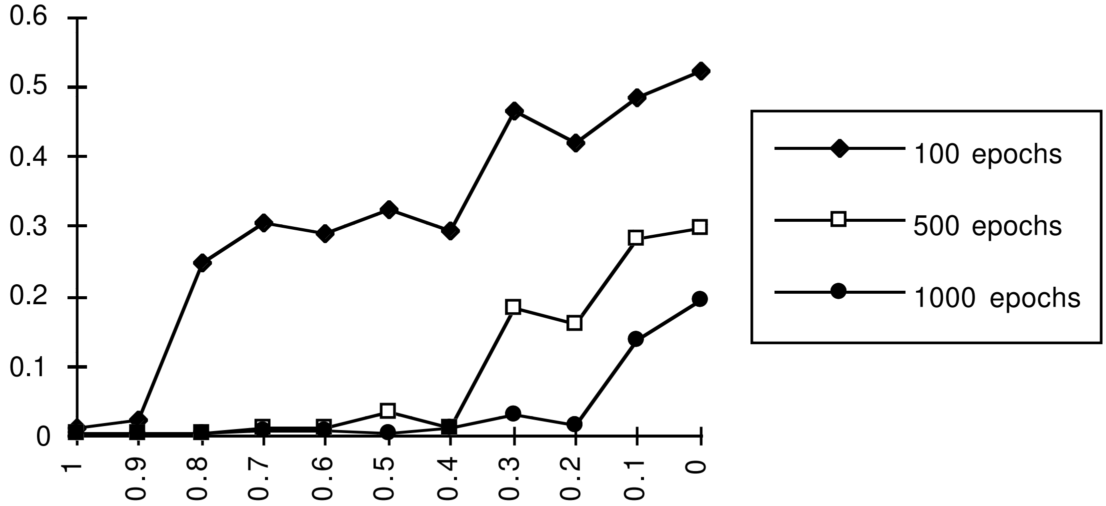


Figure 5.7: The root squared error for eleven values of λ averaged over several training trails on the Hurlbert segmentation task. The supervised network performed best initially; however, the other networks improved with repeated practice.

Finding a way to "tag" the states seemed to return the necessary Markovian to the task. At the very least it allowed the network to distinguish between initial states and intermediate ones. Presumably, other methods of tagging could be used and produce the same effect, at least for these simple sorts of tasks.

It is worth noting that the errors discussed in this section only take into account the error generated on the one component of interest; however, the networks tended to learn an identity function for the second component and so the errors including this component did not add significantly to the overall error.

5.7.2 Hurlbert's Segmentation Results

Graphs of the root squared error can be seen in figure 5.7. All of the networks avoided the trap of finding a constant function. As with the black-and-white

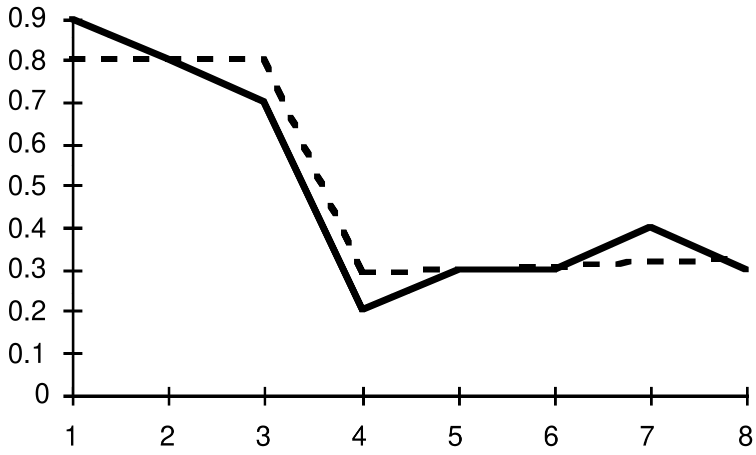


Figure 5.8: The dashed line represents the application of the Hurlbert algorithm on the solid line.

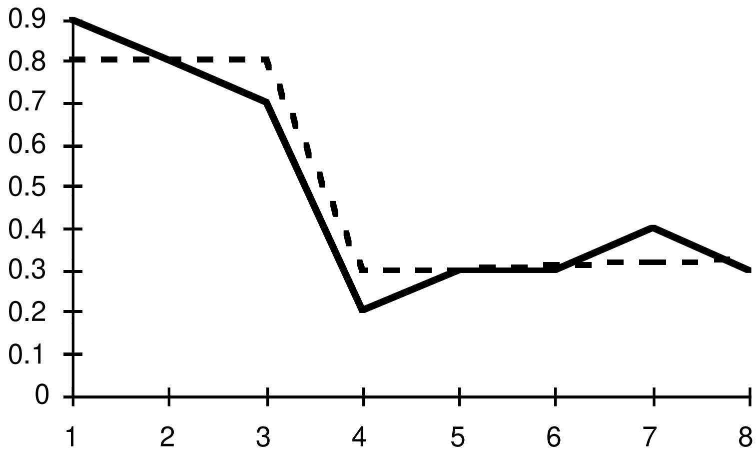


Figure 5.9: The application of one of the neural networks trained with λ=0.5 to the same line as in figure 5.8.

segmentation task, values were moved towards the correct final value with each step.

This segmentation task is much more difficult than the previous task. Even though the networks performed well on the training data, all of the networks generalized poorly and often were unable to discover a useful function when trained on very large datasets, indicating that perhaps an eight-dimensional

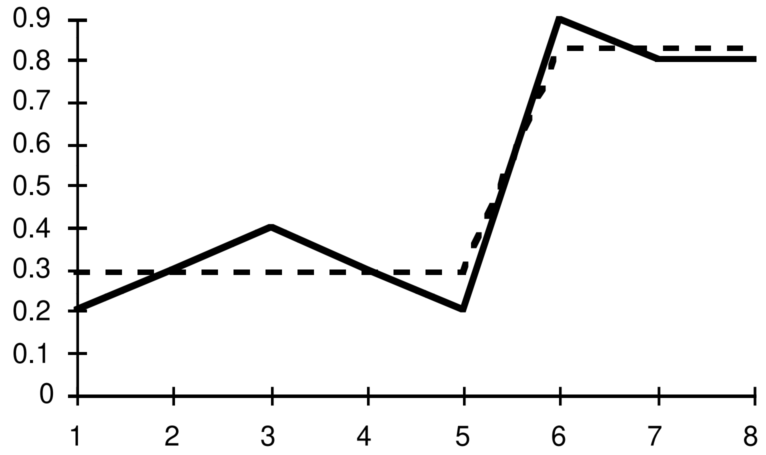


Figure 5.10 The application of the Hurlbert algorithm on another line.

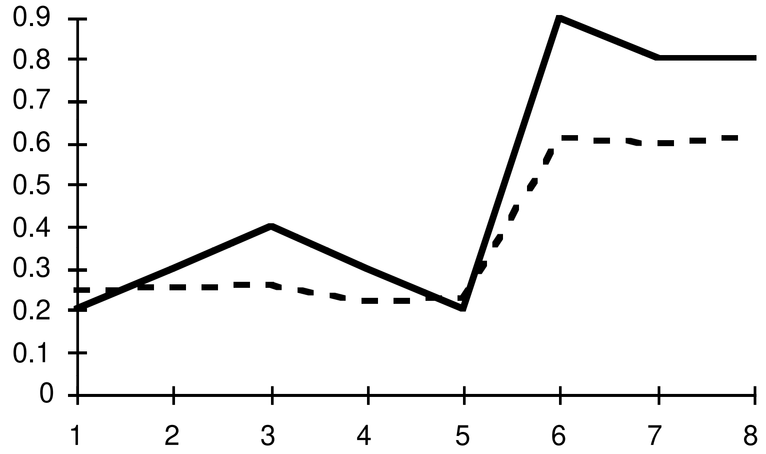


Figure 5.11: The application of the network from figure 5.9 to the same line as in figure 5.10. Although the network does not generate the same values as the Hurlbert algorithm, it does segment the line into the same two pieces.

version of this task is too difficult for a network. A more reasonable version of the task might use only three dimensions, the actual size of the Hurlbert "window," and combine copies of the network to perform segmentation on larger images. An approach similar to this has been explored in Jones (1992).

Still, despite the problems encountered by the networks with this particular task, the results from this and the simpler black-and-white segmentation problem are still useful. At first glance, TD (λ) networks would seem ill-equipped to deal with recurrent problems of the type described in this chapter. After all, there is usually no Markovian process underlying these tasks and the "control" signal generated by the networks is not really constrained to generate states that are meaningful in the context of the problem. Indeed, these very difficulties arise in the original formulation of the segmentation problems.

However, with an appropriate mechanism for "tagging" the states appropriately, the networks trained with $\lambda \neq 1$ may not only perform well, but they may sometimes perform better on training sets and generalize better than networks trained with the traditional supervised learning algorithm. If this is indeed the case, the increase in accuracy and ability to generalize may well be worth the trade-off in the increased number of dimensions and the corresponding possible increase in learning time (and in this case, there was no large increase in learning time).

In particular, the training time for the tagged black-and-white task did not increase significantly and the networks trained with $\lambda \neq 1$ performed better than the network trained in a supervised fashion on both the training set and a broader test set. On the Hurlbert segmentation task, the TD networks took longer to perform as well as the supervised network; however after some time some of the TD networks appeared to be slowly outperforming it. This proves nothing, but does provide some hope that these networks can often outperform their supervised counterparts.

Before leaving this problem, it is worth noting that both of these problems, particularly the black-and-white segmentation task, are defined so as to create the most important distinctions between initial and intermediate states. Where an intermediate state appears relative to other intermediate states is not very important. By contrast, in the weather prediction problem, the *distance* between a state and the final prediction is much more relevant.

For tasks where the first state is most important, it may be useful to attempt some way to change the depth-first nature of the TD (λ) procedure into a breadth-first search as described in section 3.1.1. For tasks like the weather prediction problem, this would not seem as useful.

Chapter 6

6 Conclusion

There has been some theoretical study of using temporal difference algorithms to address issues of prediction, outlining some distinct advantages of these methods over more traditional supervised learning paradigms. Nevertheless, there has been little evidence, either theoretical or empirical, to outline the limits of these methods in more complex real-world domains using multilayer networks.

Even as researchers such as Sutton (1988) have shown that many problems that have been attacked by traditional supervised learning algorithms are better understood as predictions tasks, we have sought to show that many of these problems are even more complicated, involving not only making accurate predictions about the environment, but also about using those intermediate predictions to actually control the environment.

We have attempted to develop a general formalism for looking at these problems that is robust enough to describe tasks involving prediction and control—whether complete or partial—as well as tasks that require only prediction. We have used this formalism with several case studies to explore several practical issues that arise when using the TD (λ) algorithm.

In the realm of games, training through self-play seems to be a powerful tool for learning robust evaluation functions. This tool seems to provide the best opportunity for a network to develop the skills necessary to play well. By contrast, networks that play against opponents are much less reliable and seem unable to move towards a stable evaluation function, either discovering local minima or oscillating between suboptimal solutions.

Further, the formalism described in this thesis highlights how well suited the training structure of the TD (λ) procedure is to these kinds of game-playing tasks. Indeed, one of this procedure's strengths is the way in which it deals with these sorts of naturally sequential problems.

Beyond this kind of domain, TD (λ) algorithms seem capable of dealing with tasks that appear ill-defined in a Markovian sense. Most of the TD networks performed as well as or better than their supervised counterparts. Furthermore, repeated presentations continue to improve the performance of these network even when the increased performance of the supervised networks would slow down.

Although there are still unanswered questions, these results suggest various ways to best take advantage of TD (λ) networks on various kinds of tasks. If nothing else, these results suggest that TD (λ) can be used in complex domains without completely obscuring our ability to analyze the algorithm and its limitations. We propose to use the formalism developed in this paper to continue developing these case studies in order to better understand the practical questions that still remain unanswered about the power and usability of TD (λ).

Appendix A

A: TD (λ) and GRBF networks

In section 2.2, we introduced the temporal difference update rule proposed by Sutton (1988):

$$\Delta \overset{\nabla}{w}_t = \alpha(P_{t+1} - P_t)\sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k \,, \tag{2.5}$$

where $\check{w}$ represents a vector of the updatable parameters of the network. This update rule is generally applicable; however, it uses a notation that is usually associated with perceptron-like feedforward networks (see section 1.1). With these kinds of networks, not only is the same function usually associated with each computational unit, but each component of $\check{w}$ serves the same purpose. Each of parameter is used as a transforming agent between two computational units or "neurons", weighting the output of one of these units before passing it to the other. In particular, if x_j denotes the output of the j^{th} unit of the network and

w_{ij} denotes the weight on the connection from unit i to j (where w_{ij} is allowed to be zero), the output of unit j can be expressed as:

$$x_j = f(\sum_i w_{ij} x_i). \tag{1.1}$$

For the case studies explored in this thesis, however, we use a Guassian GRBF network as described by Poggio and Girosi (1990):

$$y_j = \sum_{i=1}^{n} c_{ij} e^{-\sigma_i \|\check{x} - \check{t}_i\|^2}. \tag{1.5}$$

With this kind of network, the purpose of each kind of adjustable parameter is different. It is useful to keep these differences in mind and explore the way in which equation (2.5) must be applied to update these parameters. For equation (1.5), we divide the update rule into three rules, where each rule reflects the details of each type of parameter: the coefficients, $\check{c}$; the centers, $\check{t}$; and the $\check{\sigma}$ parameters, which are particular to the gaussian radial basis function and define the extent of each gaussian. Since the important difference between each update rule is the form of its gradient, we will focus on how it changes for each type of parameter.

For the i^{th} coefficient, $\check{c}_i$, the gradient is:

$$\nabla_{\check{c}_i} \check{P} = e^{-\sigma_i \|\check{x} - \check{t}_i\|^2}. \tag{A.1}$$

Notice that each coefficient can be a vector, allowing the output of the network to be a vector of the same dimensions. Since $\nabla_{\vec{c}_i} \vec{P}$ is a scalar, substituting the gradient into equation (2.5) yields a vector of the appropriate size, the dimension of the output of the network:

$$\Delta \vec{c}_i^{\,t} = \alpha(\vec{P}_{t+1} - \vec{P}_t)\sum_{k=1}^{t} \lambda^{t-k} e^{-\sigma_i \|\vec{x}_k - \vec{t}_i\|^2} \,. \tag{A.2}$$

For the i^{th} center, $\vec{t}_i$, the gradient is:

$$\nabla_{\vec{t}_i} \vec{P} = 2\sigma_i \vec{c}_i e^{-\sigma_i \|\vec{x} - \vec{t}_i\|^2} \left(\vec{x} - \vec{t}_i\right). \tag{A.3}$$

This gradient is a matrix (with rank the dimension of the output of the network and order the dimension of the input to the network). Substituting in equation (2.5) yields a vector of the appropriate size, the dimension of the input to the network:

$$\Delta \vec{t}_i^{\,t} = \alpha(\vec{P}_{t+1} - \vec{P}_t)\sum_{k=1}^{t} 2\sigma_i \lambda^{t-k} \vec{c}_i e^{-\sigma_i \|\vec{x}_k - \vec{t}_i\|^2} \left(\vec{x} - \vec{t}_i\right). \tag{A.4}$$

For the i^{th} σ, the gradient is:

$$\nabla_{\sigma_i} \vec{P} = -\vec{c}_i e^{-\sigma_i \|\vec{x} - \vec{t}_i\|^2} \left\|\vec{x} - \vec{t}_i\right\|^2. \tag{A.5}$$

This gradient is a vector in the dimension of the output. Substituting in equation (2.5) yields a scalar:

$$\Delta\sigma_i^t = \alpha(\vec{P}_{t+1} - \vec{P}_t)\sum_{k=1}^{t} -\lambda^{t-k}\vec{c}_i e^{-\sigma_i\|\vec{x}-\vec{t}_i\|^2}\|\vec{x}-\vec{t}_i\|^2 . \tag{A.6}$$

The intuitive discussion in chapter two, and a similar discussion from Sutton (1988) and Dayan (1991), applies for each of these update rules. Formally, these update rules differ from Sutton's examples in that these networks are non-linear and the proofs that show convergence for TD (λ) do not apply directly; however, this is also true of the non-linear multi-layer perceptrons used by Tesauro (1992).

Bibliography

D. H. Ackley, G. E. Hinton and T. J. Sejnowski. A Learning Algorithm for Boltzman Machines. *Cognitive Systems*, (9):147-169, 1985.

A. G. Barto. Learning by Statistical Cooperation of Self-Interested Neuron-Like Computing Elements. *Human Neurobiology*, (4):229-256, 1985.

A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike Elements That Can Solve Difficult Learning Control Problems. *IEEE Transactions on Systems, Man, and Cybernetics*, (13):834-846, 1983.

D. S. Broomhead and David Lowe. Multivariate Functional Interpolation and Adaptive Networks. *Complex Systems*, (2):321-355, 1988.

J. Christensen. Learning Static Evaluation Functions by Linear Regression. In T. M. Mitchell, J. G. Carbonell and R. S. Michalski (Eds.), *Machine Learning: A Guide to Current Research*, Boston: Kluwer Academic, 1986.

J. Christensen and R. E. Korf. A Unified Theory of Heuristic Evaluation Functions and its Application to Learning. *Proceedings of the Fifth National Conference on Artificial Intelligence*, 148-152. Philadelphia: Morgan Kaufmann, 1986.

P. Dayan. Temporal Differences: TD (λ) for General λ. *Machine Learning*, in press, 1991.

E. V. Denardo. *Dynamic Programming: Models and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1982.

T. G. Dietterich and R. S. Michalski. Learning to Predict Sequences. In R. S. Michalski, J. G. Carbonell and T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach Vol II*. Los Altos, CA: Morgan Kaufmann, 1986.

G. E. Hinton. Connectionist Learning Procedures. Technical Report CS-87-115, CMU, 1987.

J. H. Holland. Escaping Brittleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems. In R. S. Michalski, J. G. Carbonell and T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach Vol II*. Los Altos, CA: Morgan Kaufmann, 1986.

Anya C. Hurlbert. The Computation of Color. A. I. Technical report 1154, MIT, 1989.

M. Jones. *Using Recurrent Networks for Dimensionality Reduction*. Master's Thesis. Department of Electrical Engineering and Computer Science, MIT, 1992.

J. G. Kemeny and J. L. Snell. *Finite Markov Chains*. New York: Springer-Verlag, 1976.

R. P. Lippmann. An Introduction to Computing with Neural Nets. *IEEE ASSP Magazine*, 4-22, 1987.

Tomaso Poggio and Federico Girosi. A Theory of Networks for Approximation and Learning. A. I. Memo 1140, MIT, 1989.

Tomaso Poggio and Federico Girosi. Extensions of a Theory of Networks for Approximation and Learning: Dimensionality Reduction and Clustering. A. I. Memo 1167, MIT, 1990.

Tomaso Poggio and Federico Girosi. Regularization Algorithms for Learning that Are Equivalent to Multilayer Networks: Dimensionality Reduction and Clustering. A. I. Memo 1167, MIT, 1990.

D. E. Rumelhart, G. E. Hinton and R. J. Williams. Learning Internal Representations by Error Propagation. In D. Rumelhart and J. McClelland (Eds.), *Parallel Distributed Processing*, *Vol I*. MIT Press, 1986

A. L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal on research and Development* , (3):210-229, 1959. Reprinted in E. A. Feigenbaum and J. Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill.

R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. Doctoral Dissertation. Department of Computer and Information Science, University of Massachusetts, Amherst, 1984.

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

R. S. Sutton and A. G. Barto. Toward a Modern Theory of Adaptive Networks: Expectation and Prediction. *Psychological Review*, (88):135-171, 1981(a).

R. S. Sutton and A. G. Barto. An Adaptive Network that Constructs and Uses an Internal Model of its Environment. *Cognitive and Brain Theory*, (4):217-246, 1981(b).

R. S. Sutton and A. G. Barto. A Temporal Difference Model of Classical Conditioning. *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, 355-378. Seattle, WA: Lawrence Erlbaum, 1987.

Gerald Tesauro. Practical Issues in Temporal Difference Learning. To appear in *Machine Learning, 1992*.

C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD Thesis. University of Cambridge, England, 1989.

B. Widrow and M. E. Hoff. Adaptive Switching Circuits. *1960 WESCON Convention Record, Part IV*, 96-104, 1960.

B. Widrow and S. D. Stearns. *Adaptive Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1985.

R. J. Williams. *Reinforcement Learning in Connectionist Networks: A Mathematical analysis* . Technical Report 8605, University of California, San Diego, Institute for Cognitive Science, 1986.

I. H. Witten. An Adaptive Optimal Controller for Discrete-Time Markov Environments. *Information ad Control*, (34):286-295, 1977.