# Learning to Play Games From Experience: An Application of Artificial Neural Networks and Temporal Difference Learning

Daniel Kenneth Olson

December, 1993

**Abstract**

Computer programs exist for playing many sorts of games, often at and beyond expert human levels. Still, most are static in nature, unable to learn from their mistakes. In this thesis, the beginnings of a somewhat general learning architecture (i.e. algorithms and guidelines) are pieced together from ideas developed in the fields of game theory and artificial neural networks. By including Richard Sutton's Temporal Difference (TD) method, the resulting architecture is essentially a game tree search algorithm with an adaptive static evaluator that improves its performance through playing experience alone. The architecture is explored by building programs which learn to play three different games—Maze Mapping, Blackjack, and Tic-Tac-Toe. In each instance, decent playing ability is achieved. In addition, a number of fundamental issues are investigated including a problem known as state space exploration.

# Contents


**1 Introduction** **5**
- 1.1 Project Goal . . . 5
- 1.2 Objectives . . . 6
- 1.3 Approach . . . 7
- 1.4 Why Study Game Learning? . . . 8
- 1.5 Sequential Decision Tasks . . . 9
- 1.6 Reinforcement Learning vs. Supervised Learning . . . 11
- 1.7 Overview . . . 12

**2 Background** **13**
- 2.1 Artificial Neural Networks . . . 14
- 2.2 Temporal Difference Methods . . . 23
- 2.3 Game Playing . . . 26
- 2.4 Programs That Learn to Play Games . . . 30

**3 Learning System Architecture** **37**
- 3.1 Playing . . . 38
- 3.2 Learning . . . 40
- 3.3 State Space Exploration . . . 41
- 3.4 Implementation . . . 41

**4 Maze Mapping** **42**
- 4.1 The Problem . . . 42
- 4.2 Architecture Setup . . . 43
- 4.3 Performance Metrics . . . 45
- 4.4 TD Mode Comparisons . . . 46
- 4.5 Exploration Though Noise . . . 47

4.6 Operating in a Random Environment . . . . . 49

**5 Blackjack** **50**
5.1 Blackjack Rules . . . . . 50
5.2 Architecture Setup . . . . . 51
5.3 Performance Metrics . . . . . 52
5.4 Table Lookup Encoding . . . . . 53
5.5 Thermometer Encoding . . . . . 57
5.6 Measuring Learning Speed . . . . . 60
5.7 Measuring Stability . . . . . 61
5.8 Increasing State Information . . . . . 61

**6 Tic-Tac-Toe** **65**
6.1 Tic-Tac-Toe Rules . . . . . 65
6.2 Performance Metrics . . . . . 66
6.3 Architecture Setup . . . . . 67
6.4 Self Play . . . . . 69
6.5 Mimicking The Opponent . . . . . 70
6.6 Training Against Different Skill Levels . . . . . 71
6.7 Noise . . . . . 72
6.8 Raw State Information . . . . . 72

**7 Conclusions** **75**
7.1 Results . . . . . 75
7.2 Future . . . . . 77

# List of Tables


4.1 Comparison of TD modes on maze trials. . . . . 46
4.2 Comparison of TD modes on maze trials when noise is added. 48
4.3 Comparison of TD modes in maze with random transports. . . 49

5.1 Capabilities of four different fixed Blackjack strategies. . . . . 53
5.2 Results of various Blackjack learning trials using the Table state encoding. . . . . 55
5.3 A policy which was learned using the Table state encoding. . . 56
5.4 Results of various Blackjack learning trials using the Thermometer state encoding. . . . . 58
5.5 A policy which was learned using the Thermometer state encoding. . . . . 59
5.6 Effect of increased Blackjack state information. . . . . 63

6.1 The capabilities of three fixed policy Tic-Tac-Toe players against each other: Randy(1), Huey(1), and Randy(9). . . . . 67
6.2 Encoding of line condition for Tic-Tac-Toe. . . . . 69
6.3 Effect of mimicking a Tic-Tac-Toe opponent. . . . . 70
6.4 Neural players trained against opponents with different skill levels. . . . . 71
6.5 Comparison of Tic-Tac-Toe scores with and without noise. . . 72

# List of Figures


1.1 An example of a dynamic system. . . . . 9
1.2 An example state diagram. . . . . 10

2.1 A typical artificial neural unit. . . . . 16
2.2 Graphs of popular unit activation functions. . . . . 17
2.3 An 8-3-8 encoder neural network. . . . . 18
2.4 Example of why TD(0) is better than TD(1). . . . . 25
2.5 An example tree for a game of length three. . . . . 27

3.1 Overall framework for the architecture. . . . . 37
3.2 The three major functions computed to select an action. . . . 38

4.1 An example maze. . . . . 43
4.2 Maze neural predictor. . . . . 44
4.3 A maze map which was learned. . . . . 47

5.1 Blackjack neural evaluator using the Table state encoding. . . 54
5.2 Blackjack neural evaluator using the Thermometer state encoding. . . . . 57
5.3 Graph comparing learning speeds for different encodings. . . . 60
5.4 Graph comparing policy change rates for different encodings. . 62
5.5 Example Blackjack decision policy using dealer face up card. . 64

6.1 Graph of Tic-Tac-Toe ability change versus experience. . . . . 73

# Chapter 1

# Introduction

The idea of playing games against a machine intrigues us. The popularity of home video games is at an all time high. Looking back, evidence of this fascination can be traced to the 1890's. It was during this period that an electro-mechanical device created by the inventor Torres y Quevedo entertained World Fair goers by playing the chess endgame King and Rook against King (Bramer, 1983). From artificial intelligence's (AI) beginnings, researchers have been using games as a testbed for search and planning algorithms, and continue to do so. In many cases, programs based upon these algorithms compete at and beyond human playing abilities. Unfortunately though, they tend to rely on carefully hand-tuned heuristics and brute force search. They do not learn from their mistakes!

## 1.1 Project Goal

The overall goal is to develop an architecture (i.e. algorithms and guidelines) for designing *game playing* programs which *learn* their skills from *experience*. Three words are emphasized because they indicate key problems in building such an architecture. Fortunately, solutions have been devised for each that can be readily built upon.

**Game Playing** techniques for computers are well established, having a history that dates back to at least 1950 (Shannon, 1950). As such, there is a ready library of approaches including game trees and various search algorithms (min-max, alpha-beta pruning, etc.).

**Machine learning** has been one of the more difficult aspects of AI (are any easy?), especially as applied to games. Pioneering ideas were put forth in the 1950's by A. L. Samuel (Samuel, 1959) with a Checkers program that could improve just by playing itself, but until recently there have been few advances. Now during the 1980's, learning techniques based upon statistics, genetics, and neural networks have become popular and are showing a capacity for solving real world problems. It is the latter of these—artificial neural networks[1] (ANN)—that provides the foundation for the architecture being developed.

**Using experience** acquired from actual play means that learning is directed by successes and failures. This form of adaptation is known as *reinforcement learning* since moves which bring about success are reinforced, while those leading to failures are weakened. When the outcome is not discovered until some time after a move, it becomes a difficult *temporal reinforcement* problem. A promising solution which will be adopted is the method of *Temporal Difference* (TD) (Sutton, 1988). Its power has been demonstrated by the recent success of Gerald Tesauro's backgammon program (Tesauro, 1993) which is perhaps the best computer player to date.

## 1.2 Objectives

The somewhat general architecture that will be proposed might be best described as a pattern classification system which learns to associate a given game situation with its likelihood of being on a path to success or failure. If such a mapping can be learned, good play should result by just selecting those moves which yield situations classified as successful. The program would be learning to react as opposed to using sophisticated plans (building habits as opposed to reasoning, shallow knowledge as opposed to deep knowledge).

There are several things that need to be determined about such an architecture. To start with:

- Is it applicable to more than one kind of game. For example, will it

[1]Even Samuel in his pioneering paper mentions the possibility of a general "Neural-Net Approach" for game learning but discards it due the technology limitations of the time.

work well with and without opponents? How about when the game involves an element of chance?

- Can it be used to perform "knowledge-free"[2] learning? This means that only raw game state information is explicitly provided as opposed to special features measured from the board (e.g. the piece advantage in Checkers). Determining good features for a particular game requires some amount of expertise. By working with only raw information little prior knowledge about a task would be required; instead the learning system could discover the necessary features.

  In the same vein, when opponents are involved, can self play be substituted for an expert opponent?

- How should the learning parameters be controlled and how sensitive is the learning process to their settings? Both ANNs and the TD method have several parameters that determine learning speed and stability. Finding appropriate settings is generally a trial and error process; ultimately they might be controlled automatically.

- How well is the game learned? In most cases this can be measured as a win percentage over a series of games. In addition, there is a potential problem that rarely occurring situations will not be handled well. To continually improve, there must be some way of ensuring that the entire game is explored.

## 1.3 Approach

Initially it is important to keep the scope somewhat simple. If experimental tasks are too complex it might turn out that fundamental issues pertaining to the architecture would be overlooked. For this reason the following three tasks are used:

- *Maze Mapping* — Learning a map to a maze, meaning that from any random starting point an efficient path can be found to the exit. This is an ideal problem for studying temporal reinforcement learning without

[2] This is a term used by Gerald Tesauro in his papers.

any extra fluff. It involves sequences of up to 20 moves and can be used to study both random and non-random situations.

- *Blackjack* — Learning to play a hand of blackjack, a game which is highly controlled by chance. The rules of play are simple and hand length is short, thus reducing computational complexities. In addition, suggested guidelines for playing are available which can be used to benchmark the learned skills.

- *Tic-Tac-Toe* — Learning to play Tic-Tac-Toe against opponents using traditional game playing techniques. Objectives include comparing adaptive players without lookahead against an exhaustive tree searching opponent and looking at "knowledge-free" learning.

## 1.4 Why Study Game Learning?

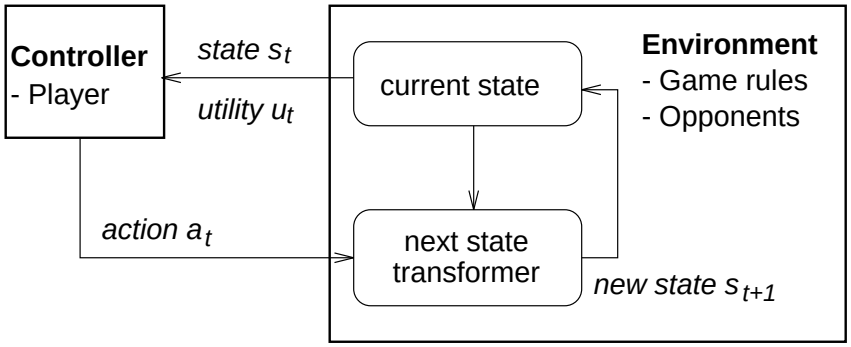
Basically everything that animals, corporations, and computer programs do involves reaching a goal through a series of actions. These kinds of processes can be referred to collectively as *sequential decision tasks* (Barto et al., 1989). Whether driving a car to work, or buttering a piece of toast, there is a goal in mind and a sequence of actions for attempting to reach it. In many instances, the temporal credit assignment problem rears its ugly head; trying to improve a company's long term profits is such an example.

An adaptive control system that can learn to perform sequential decision tasks might find application in many domains such as controlling a manufacturing process, autonomous vehicles, avionics, and aids for the disabled. As noted by Paul Werbos (Werbos, 1989), work towards solving these problems involves a large share of the U. S. Gross National Product (GNP).

Games, which are sequential decision tasks, provide convenient testbeds for the study of such learning. Their goals and rules[3] are well defined simplifying the modeling and simulation process. At the same time, the problems they present are both challenging and interesting.

[3]Learning tasks can also include the discovery of rules.

Figure 1.1: A dynamic system in which a controller (player) interacts with its environment (game and opponents).




## 1.5 Sequential Decision Tasks

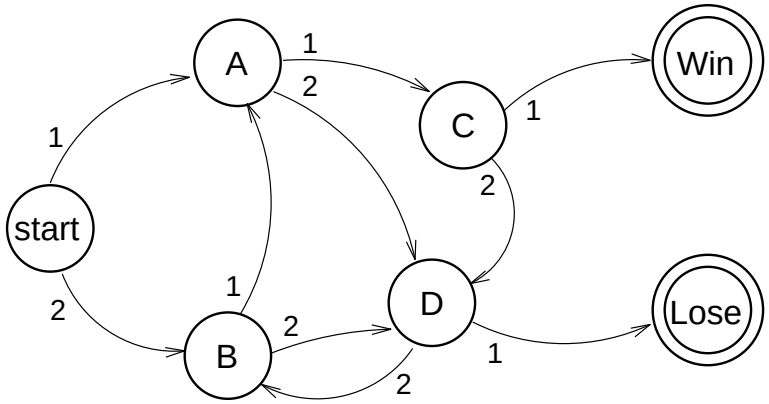
Sequential decision task are typically modeled as the interaction between two components, a controller (player) and its environment (game state, rules, and opponents) (see Figure 1.1). The controller's job is to manipulate the environment to bring about desirable situations (win). At discrete time steps $t$, the controller receives sensory input from the environment in the form of a state $s_t$. In response, the controller decides upon an action[4] $a_t$. This action in turn influences the environment as it produces the next state $s_{t+1}$. The environment also generates a *utility* signal $u_t$ which is considered the reward or reinforcement; the goal of the controller is to maximize the cumulative utility over the lifetime of the task.

In a game like Tic-Tac-Toe, the current state is represented by the placement of X's and O's on a grid. From a given player's perspective, it is the controller and the combination of board/opponent/rules is the environment. Beginning at some initial state, the player decides upon an action. The opponent within the environment then makes another action resulting in a new state. The player choses yet another action resulting in additional state transitions, and the process repeats on and on. The cycle typically ends when certain halting conditions occur (e.g. "Three-in-a-row" in Tic-Tac-Toe) although some tasks repeat indefinitely.

The set of states that can occur during a task is called the *state space*. It

[4]The terms *action* and *move* are used interchangeably throughout this document.

Figure 1.2: A state diagram which describes a very simple game with two actions (1 or 2) and two final outcomes (Win or Lose). Note that one play of this game could go on forever since there are feedback connections.




can be represented as a directed graph in which nodes represent unique states and arcs represent possible transitions (see Figure 1.2). Traveling enough of the paths through the state space to ensure that the task is well learned is referred to as *state space exploration*. State space exploration turns out to be very important when using reinforcement learning since it is necessary to prevent surprise situations ("Oh no, what do I do now!") and performance ruts (consistent yet bad play).

Typically, the state offered to the controller does not provide all the information used by the environment to evolve to the next state. In these cases, state transitions appear to be guided by an element of chance, such as a deck of shuffled cards. The deck—a part of the environment —"knows" what order the cards are in but to the player they are random.

How a controller selects actions is referred to as its *decision policy*. A decision policy is a set of rules which associates an action with each possible state. For example, a simple decision policy for the card game Blackjack might be, "Hit me if my card total is 16 or less, else stand". A policy is adaptive if its rules change over time. Changing a policy so that it maximizes the total utility received is essentially what learning is all about.

Initially it might seem that a policy should select the action at each step that yields the largest reward. Unfortunately, maximizing your profits in the short term may lower the long term gains. For one, it is possible that

the action with the greatest reward will move the system into a part of the state space in which the remaining rewards are few. For most games this is taken to an extreme since the reward is zero for all time steps up until the very end when you know if you won, lost, or tied. In these cases no explicit information is provided from the environment to help discriminate between the possible actions.

## 1.6 Reinforcement Learning vs. Supervised Learning

It is probably worthwhile to compare reinforcement learning to one of its popular counterparts—*supervised learning*. A supervised learning method would need a training set which contains, for example, particular game situations and ratings for each possible move. For "good" play to be obtained a "good" training set is required. This implies an expert teacher. In addition, the supervised program can only be as good as its training set. There are other problems related to supervised learning, such as ensuring there are enough examples to cover all situations that might come up during a game.

A reinforcement learning method instead relies only upon a single reward/punishment signal. When rewarded, it must find the responsible moves and make them more likely to occur; when punished, the influential moves must be prevented from occurring. Because there is no explicit feedback specifying which actions are best, a large number of potential strategies might have to be experimented with in order to determine a good style of play. For this reason, using a reinforcement method to teach a program tends to be more time consuming than using an explicit teacher.

In general, reinforcement learning is considered to be the more difficult of the two, but it has several advantages. Reinforcement learning allows a system to adapt *online* while it is performing the task, while supervised learning usually requires adaptation to take place *offline*. Online learning is desirable in many instances since it permits a control system to improve "on the job". In addition, since supervised learning requires a teacher to provide training examples, reinforcement learning is appropriate when none is available. Finally, the ability to improve is not bounded by a bad training set.

## 1.7 Overview

The rest of the chapters cover the following topics:

**Background** (Chapter 2) describes the technologies—artificial neural networks, temporal difference methods, and game theory—that are the foundation of the learning system. This is followed by a review of past and recent ideas used to build game learning programs.

**Learning System Architecture** (Chapter 3) outlines the elements of the learning system architecture and its implementation.

**Maze Mapping** (Chapter 4) is the first problem and involves learning how to traverse a maze. This problem was selected because it illustrates well the capabilities of a temporal difference method.

**Blackjack** (Chapter 5) is the second task to be learned. This chapter will show how the learning system works with simple games of chance.

**Tic-Tac-Toe** (Chapter 6) is the third game to be worked with, and the only one in which there is an active opponent (Blackjack's opponent—the dealer—makes no decisions until all players are done). The two main concepts dealt with are state space exploration and knowledge free learning.

**Conclusions** (Chapter 7) summarizes the results and discusses issues that surfaced during the three example problems.

# Chapter 2

# Background

In this chapter, the key concepts and technologies for building the game playing/learning architecture are described.

- *Artificial Neural Networks* and an accompanying training procedure known as *backpropagation* are introduced as a general purpose learning mechanism.
- *Temporal Difference* methods are described next as a technique for applying artificial neural networks to temporal reinforcement learning problems.
- *Game Playing* ideas, specifically the use of game trees, min-max theorem, and static evaluators, are described as a framework in which to construct game playing programs.

To wrap up, several game learning attempts are presented, including some which use artificial neural networks and temporal difference methods.

## 2.1 Artificial Neural Networks

*Artificial Neural Networks* (ANN) or simply *neural networks*[1] are a family of biologically inspired computation systems consisting of many simple processing units operating in parallel. These processing units, which are only loosely based on real neurons, influence each other through weighted (variable strength) connections. These weights define the network's behavior, and by changing them many different functions can be approximated. This has allowed ANNs to be applied to a range of problems such as pattern recognition, system control, and parameter optimization problems.

- *They learn:* One of the most notable features of the neural network is that procedures exist for automatically finding connection weights; functions are taught as opposed to being programmed. This allows a quite different approach to finding solutions to problems since explicit knowledge of an algorithm is no longer required. Instead, only examples of inputs and corresponding outputs of a desired function need be specified.

- *They generalize:* Some classes of neural networks are capable of generalizing patterns learned to inputs not encountered during training. This becomes very important for complex games and sequential decision tasks since the number of unique states can be tremendous. If instead a table-lookup style of learning—direct memorization of input/output pairs—was used, every state would have to be visited at least once to fully learn the task. Memory would also be required for each of these states (a simple game like Tic-Tac-Toe has 6045 states; imagine the number in a game like Chess or Go). Generalization helps reduce the need to visit each state, and most importantly, greatly reduces the amount of information stored.

- *They're fast and robust:* Neural networks lend themselves well to speedy and reliable hardware implementations. The massive parallelism inherent in these networks has lead to silicon chips like Intel's 80170NX

[1]The adjective *artificial* is typically dropped when artificial neural networks are discussed in the company of computer scientists and electrical engineers, but care must be taken. Researchers and doctors working with *real* neural networks—brains—are dealing with something much more complex and sophisticated.

which is capable of running at 5 billion connections per second[2]. Much faster optical hardware is also becoming available. Once cast in hardware, the neural network can show a high degree of fault tolerance. Experiments have shown that as random units and connections are destroyed, the behavior of a system tends to degrade gracefully. Additional training after a loss can even return a network to its previous performance level.

The field is rich with interesting neural architectures and training procedures, each designed to tackle specific kinds of problems. Unfortunately, it is also rich in vocabulary that refers to similar concepts. For example, the terms *connectionist networks* (Feldman and Ballard, 1982), *parallel distributed processors* (Rumelhart and McClelland, 1986), and *neurocomputing* (Hecht-Nielsen, 1989) all refer to artificial neural networks. A very good summary can be found in (Simpson, 1988).

## General Neural Network Anatomy

Even though neural networks come in many flavors, they are all built from two basic elements: units and weighted connections.

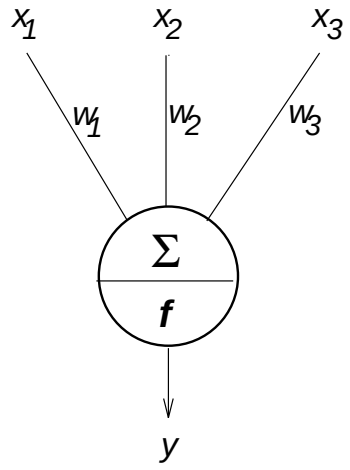
- *Units*, also known as neurons, or nodes, are simple processors which receive input signals from several (often many) other units and produce a single output. The simplest is the threshold unit which collects inputs and only generates an output when their sum exceeds a threshold.
- *Connections* pass signals between two units and amplify or attenuate the signals depending on a weight. Connections with positive weights are *excitatory* and those with negative weights are *inhibitory*.

The operation of a typical unit as shown in Figure 2.1 is to compute a weighted sum of its inputs which is then passed through an *activation function*, also known as a squashing or transfer function. If a vector $\mathbf{x}$ represents the input signals and $\mathbf{w}$ contains their corresponding weights, then this operation can be modeled as

$$y = f\left(\sum_{i=1}^{n} w_i x_i - T\right) \tag{2.1}$$

[2] *Connections per second* is a typical speed benchmark which measures the rate at which signals are passed between the processing units.

Figure 2.1: Typical artificial unit which performs a weighted sum of its inputs $\mathbf{x}$, passes the results through an activation function $f$, and produces a single output $y$.



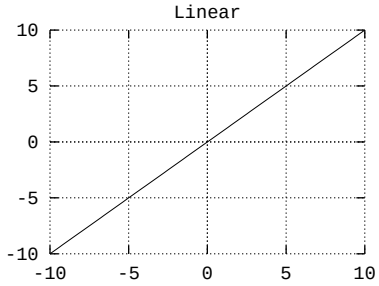

where $y$ is the unit's output, $f()$ is the activation function, and $n$ is the number of inputs. $T$, the threshold or bias, is usually defined as a special connection weight $w_0$ with a constant input $x_0$ set to -1. This simplifies neural network algorithm implementations since now the threshold is just another weight.

Each neural architecture can be defined in terms of its activation function, topology, and learning rule. The choice of activation function often depends on the kind of outputs desired, and the requirements of the learning rule used. For example, some learning rules use gradient descent and require a continuous and differentiable function. Some of the more popular activation functions are displayed in Figure 2.2.

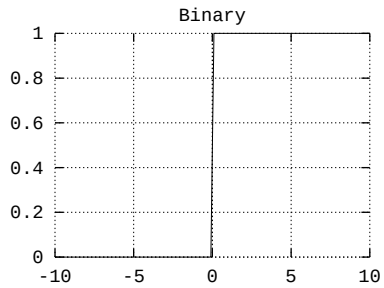
The *topology* of the network specifies how the units are interconnected. A popular method is to arrange units into *layers* allowing only connections between units in adjacent layers. A multi-layer feedforward network is defined such that units from layer 1 can only connect to units in layer 2, and layer 2 can only connect to units in layer 3, and so on. An example of such a topology is shown in Figure 2.3. A *recurrent* network is one containing feedback connections. A fully connected network has every unit connected to each other.

Some units, usually all in the same layer, receive input from the outside world. Likewise, a set of one or more units produce the network's outputs.

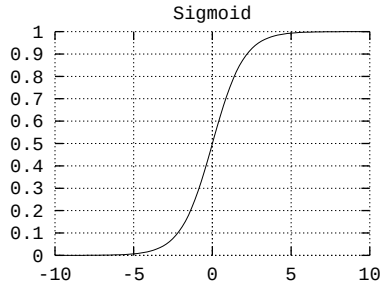
Figure 2.2: Popular unit activation functions. The sigmoid shape is used extensively in cases were a bound output range is desired, but the learning rule requires a continuous function.




(a) Linear $f(s) = s$

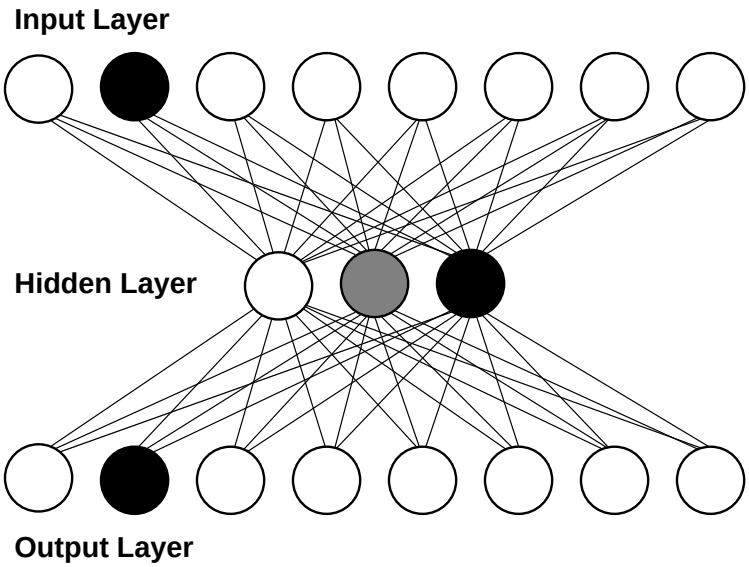



(b) Binary




(c) Sigmoid $f(s) = 1/(1 + e^{-s})$

Figure 2.3: A 3-layer feedforward network (8-3-8 encoder) which can learn to compress/decompress 8 bit patterns (of which only one bit is turned on) to 3 units. Units in layer 2 and 3 also receive a bias (threshold) input which is not shown.




Layers that aren't directly connected outside are called *hidden layers*.

Network connection weights could be setup manually for problems with known solutions, but the more common and fun approach is to let a special algorithm—the *learning rule*—find the weights. Learning rules, of which there are many to chose from, "smartly" change a network's weights to realize some goal. There are three main styles of learning, each distinct in the kind of feedback that is provided to guide the rule to its solution:

1. Supervised — a teacher specifies the correct output for a given input.
2. Reinforcement — the network must discover the correct output based only a reward/punishment signal.
3. Self organizing (unsupervised) — the network must form groups of related patterns from a data set (clustering) without assistance.

Today, there are many different architectures to chose from, an artifact of the attitude in late 80's in which neural network research was equated with inventing new paradigms. Examples of of the most common paradigms

are Adaptive Resonance Theory (ART) (Carpenter and Grossberg, 1987), Hopfield Net (Hopfield, 1982), and the Adaptive Vector Quantizer (Kohonen, 1981). Another popular one, known as *Backpropagation*, is used for most of the work in this thesis. The next section provides an overview for this popular multi-layer feedforward network[3] and its accompanying training procedure.

## Backpropagation

The Backpropagation architecture gets its name from its supervised learning rule which stands for "Backward Error Propagation" and often shortened to just backprop. The network's topology is usually multi-layered and feed-forward, but modifications exist for using feedback connections (recurrent backpropagation).

A training set of example inputs and outputs for a desired function drives the learning process. It operates iteratively over the examples trying to find a set of weights which will let the network's outputs closely match the desired ones. The error $E$ is measured as a sum of squared differences

$$E = \frac{1}{2}\sum_{p,j}(t_{jp} - y_{jp})^2 \tag{2.2}$$

where $t_{jp}$ is the desired *target* output for unit $j$ given input pattern $p$ and $y_{jp}$ is unit $j$'s actual response (the $1/2$ exists only to make an eventual derivative look nicer).

### Delta Rule

The delta rule, or Least Mean Squares (LMS) training procedure (Widrow and Hoff, 1960), uses gradient descent to minimize the error $E$ in Equation 2.2. By taking the derivative of $E$ with respect to a given weight $w_{ij}$, a function—the delta rule—is derived that tells which way to adjust that weight to reduce the error.

$$\Delta w_{ij} = \alpha \sum_{p}(t_{jp} - y_{jp})\frac{\partial y_{jp}}{\partial w_{ij}} \tag{2.3}$$

[3]Multi-layer feedforward neural networks are also erroneously referred to a multi-layer perceptrons. The term "perceptron" refers to the network and learning rule used by Rosenblatt in the late 1950's which has never been extended to handle multiple layers.

The parameter $\alpha$ controls the magnitude of weight changes and is known as the learning rate.

The delta rule can be used to learn many kinds of mappings, and is used for most of the problems investigated in this thesis. However, its use is limited to networks with only two layers since there is no corresponding target $t_{jp}$ for the units within hidden layers. This little annoyance is known as the *structural credit assignment problem*, a hurdle that helped stopped neural network research dead in its tracks in the 1960's. Without its solution, only linearly separable[4] functions could be learned.

### Generalized Delta Rule

A welcome enhancement known as the generalized delta rule (Rumelhart et al., 1986) allows gradient descent to be applied to weights in hidden layers. First of all, equation 2.3 is expanded into three pieces.

$$\text{net}_j = \sum_i w_{ij} x_i \tag{2.4}$$

$$\frac{\partial E_j}{\partial \text{net}_j} = (t_{jp} - y_{jp}) \frac{\partial y_{jp}}{\partial \text{net}_j} \tag{2.5}$$

$$\Delta w_{ij} = \alpha \sum_p \frac{\partial E_j}{\partial \text{net}_j} x_i \tag{2.6}$$

Equation 2.6 can now be used for weights feeding a hidden layer unit by substituting the following equation for 2.5:

$$\frac{\partial E_j}{\partial \text{net}_j} = \sum_k \frac{\partial E_k}{\partial \text{net}_k} w_{jk} \frac{\partial y_{jp}}{\partial \text{net}_j} \tag{2.7}$$

Here, $k$ indexes the units that use hidden unit $j$'s output. By starting with the output units, weights changes are computed one layer at a time backwards through as many hidden layers as desired.

[4]Neural networks employing a two layer feedforward topology are only able to represent linear separable functions. The term *linear separable* describes pattern classification problems in which the data points for two classes A and B can be separated by a line (or hyperplane for dimensions greater than 2). A common example of a simple function that is not linearly separable is the Boolean "exclusive or" (XOR).

### Enhancements

Although backprop is a great leap above its predecessors, it is typically very slow at finding weights for a solution. In response, many enhancements have been suggested that help speed up learning and are more likely to converge to good solutions. Some examples are:

- *Momentum* which attempts to dampen radical weight changes.
- *QuickProp* (Fahlman, 1988) which uses second-order derivatives to greatly decrease training time.
- and many others ...

Since there are so many variations, confusion can arise when someone reports "I did such and such with backprop". To be more specific, the original backprop as presented above from now on will be referred to as "vanilla" backprop, and any variations used will be explicitly specified.

## Brief History

Although many believe artificial neural networks to be a recent development, their roots extend at least 50 years back. The earliest work is generally considered to have occurred in the 1940's with McCulloch and Pitts' (McCulloch and Pitts, 1943) model for neurons. Their work introduced the concept of the binary threshold activation unit and showed that any logic function could be configured with networks of such processors. During the same period, Donald Hebb introduced key ideas (Hebb, 1949) that later would be incorporated into an associative learning rule known today as Hebbian Learning.

The late 1950's began the first wave of neural network popularity with such systems as Frank Rosenblatt's *Perceptron* (Rosenblatt, 1962) and Bernard Widrow's *Adaline* (which included the delta rule) (Widrow and Hoff, 1960). These two systems were quite similar in that they used a layered feedforward topology along with a supervised learning rule for adapting connection weights. They also shared a limitation: their learning rules only worked for two layer networks. This problem was later amplified when Minksy and Papert produced the book "Perceptrons" (Minsky and Papert, 1969) which

pointed out that two layer networks could only implement linearly separable classification functions and that there were no obvious extensions of the perceptron training rule which allowed it to work with hidden layers.

Although funding for research dwindled, the concepts where kept alive by such people as Stephen Grossberg (Adaptive Resonance Theory) and Tuevo Kohonen (Adaptive Vector Quantizer). Then as the 1980's rolled around, two significant events occurred, rekindling interest: advances such as backpropagation were made well known[5], and computers became cheaper and faster allowing just about anyone with a desire to join in. This has lead to a steady increase of new concepts and applications, still growing today.

[5]Backpropagation was actually discovered much earlier than the 80's, but wasn't well known until the PDP group rediscovered it.

## 2.2 Temporal Difference Methods

The Temporal Difference (TD) procedure of Richard Sutton (Sutton, 1988) provides a method for solving the temporal credit assignment problem. Its goal is to teach a controller to accurately predict future events (rewards) by just observing a sequential decision task over and over. It is not a learning engine like a neural network, but instead is an approach that is wrapped around supervised learning algorithms.

A sequence of states $(s_1, s_2, \ldots, s_f)$ are visited during a task that ends at some final state $s_f$. At this point, a reward may or may not be received as signified by a utility $u_f$ set to 1 or 0 respectively. An *obvious approach* would pair each state with this final utility, creating a training set $\{(s_1, u_f), (s_2, u_f), \ldots, (s_f, u_f)\}$. In contrast, a temporal difference method might use training pairs as $\{(s_1, P_2), (s_2, P_3), \ldots, (s_{f-1}, P_f), (s_f, u_f)\}$ where $P_t$ designates the prediction of the controller at time step t. This special pairing is referred to as TD(0) for a reason described later. Using the delta rule as the supervised learning algorithm, the weight updates become

$$\Delta w_t = \alpha(P_{t+1} - P_t)\frac{\partial P_t}{\partial w} \tag{2.8}$$

where the summation and indices have been dropped for clarity. Instead of adjusting predictions for each state directly toward the final outcome, they are instead adjusted toward the predictions produced at the next time step. This is why it's called a *temporal difference* method since it attempts to minimize the *difference* between *temporally* successive predictions.

The ideas that Sutton describes are not entirely new, but mostly a refinement and formalization of temporal difference methods used by previous learning systems. Examples are Samuel's checker player (Samuel, 1959) (yes, he had it from the beginning), Holland's Bucket Brigade (Holland, 1983) for Genetic Algorithms, and Sutton's earlier work with the Adaptive Heuristic Critic network (Sutton, 1984).

### Incremental TD($\lambda$)

Sutton developed an incremental real-time learning procedure for neural networks known as TD($\lambda$). It is essentially a modification of the delta rule which

allows it to work in a forward only fashion. This forward only feature removes the need to save a history of states visited and reduces the number of computations needed.

Sutton begins with the delta rule (Equation 2.3) where the target is $u_f$ and $P_t$ is the network's prediction for the state visited at time $t$.

$$\Delta w_t = \alpha(u_f - P_t)\frac{\partial P_k}{\partial w} \tag{2.9}$$

This, of course, was the *obvious approach*, since each prediction is paired with the final outcome $u_f$. Through several steps, an equivalent equation is derived which is known as the TD(1) procedure.

$$\Delta w_t = \alpha(P_{t+1} - P_t)\sum_{k=1}^{t}\frac{\partial P_k}{\partial w} \tag{2.10}$$

Note that $P_{f+1} = u_f$. What is special about Equation 2.10 is that it only relies on saving the last prediction and a sum of past derivatives.

The TD(1) procedure is then generalized by adding an additional parameter $\lambda$ which controls the extent to which the final outcome trickles back to previous states. This new weight update rule is known as TD($\lambda$)

$$\Delta w_t = \alpha(P_{t+1} - P_t)\sum_{k=1}^{t}\lambda^{t-k}\frac{\partial P_k}{\partial w} \tag{2.11}$$

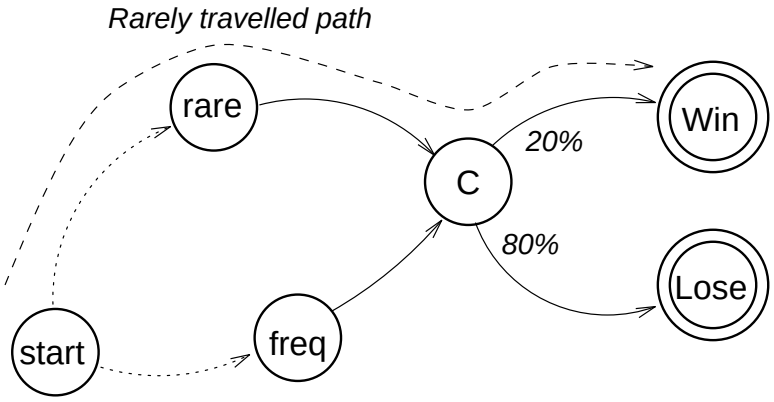
where $\lambda \in [0, 1]$. The setting of lambda interpolates between two extremes where TD(0) pairs each prediction with its successor (this is why the earlier method was called TD(0)) and TD(1) pairs each evaluation with just the final outcome (the *obvious approach*).

## TD(0) Advantage

For special classes of tasks governed by an absorbing Markov process, both TD(0) and TD(1) are guaranteed to converge asymptotically to predictions whose expected values match that of the system being learned. What this means is that given an infinite amount of time to train, either approach will do the job. What about when you can't wait that long?

Sutton proved that the TD(0) method using a network with linear activation functions converges to optimal estimates for matching future experiences

Figure 2.4: In the above state diagram, a state which is rarely encountered occurs. Using TD(1) learning, the prediction for state *rare* would be pushed toward win. The TD(0) method, on the other hand, would make use of the experience at node C, which knows that a loss is more likely.




when a finite amount of training has been performed. At the same time he shows that TD(1) produces suboptimal estimates. To understand why, consider a state space in which there are paths that are rarely taken. Since states along these paths are visited less frequently, predictions learned by a TD(1) approach will be based on fewer samples and less accurate than those for more frequently visited states. The TD(0) approach, on the other hand, will make use of experience gained along a frequent path. This is illustrated with an example in Figure 2.4.

## 2.3 Game Playing

Building programs to play competitive games was a central theme during the artificial intelligence movement of the 1950's and most programs today still rely on the same basic principles developed during that era. At their core are the data structures and mathematics of *Game Theory* (McKinsey, 1952) which provided the game tree, static evaluator, and min-max theorem to analyze game outcomes.

### Game Attributes

While all competitive games follow the sequential decision task model, it is useful to further characterized them using four additional attributes.

- *Length* — The number of moves typically required from start to finish. Tic-Tac-Toe, for example, never has more than 9 moves per game while Othello has up to 60 moves. A long game length implies a more difficult temporal reinforcement problem.

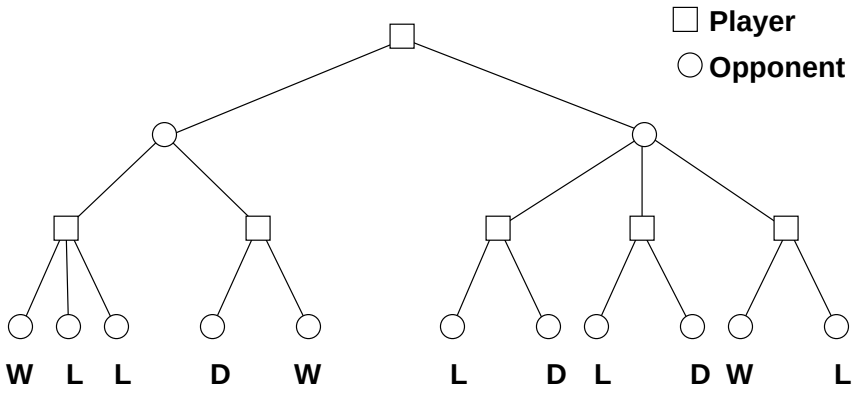
- *Players* — The number of players interacting in a game. When there is more than one player, each must consider how other's actions will influence the outcome. Typically, more players means a more difficult task, although they might provided clues on better playing style.

- *Information Visibility* — How much game state information is available to each player. A game of *perfect information* like Chess allows all moves to be visible to each player. In contrast, Poker is a game of *imperfect information*.

- *Chance Moves* — A game with chance moves involves random state transitions in which a single action can lead to several different next states. Games that require dice to be tossed or cards shuffled are examples of games of chance.

Both games with imperfect information and chance moves appear as a sequential decision task with random transitions; such transitions present difficulties for look-ahead schemes to deal with.

Figure 2.5: An example tree for a game of length three.




## Game Trees

A *play* of a game is a sequence of moves and states from start to finish. All possible plays of a game can be represented in a data structure called a *game tree*. Within this tree, each node represents a game state and the arcs represent transitions to each of the possible next states. All of the leaf nodes represent terminal states of the game and can be labeled as either a win, loss, or draw. A *ply* number is commonly used to identify a number of moves along a path in a tree; its greatest value is the length of the game (which is one less than the tree depth). The *branching* factor of a tree identifies the average number of arcs leading out of each node. Figure 2.5 is an example tree for a game of length three.

Game trees can be used to represent both games of perfect and imperfect information and games with and without chance moves. For those with hidden state information or random transitions, there may be multiple next states for each possible move. In this case each next state has an additional probability attached to it which identifies its likelihood of occurring. These probabilities, depending on the game, can be very difficult to determine.

To win a game of perfect information without chance moves it is only necessary to proceed along a path in the tree to a leaf node which is a winning state. The opponent, also wanting to win, is probably not going to cooperate. In an ideal situation, though, there might be a path in which the opponent has no ability to prevent a win. It may also be discovered that the best that either side can do is draw. In any case, the ultimate best outcome

can be determined with the *min-max* search procedure.

## Min-Max Search

The min-max search procedure proceeds as follows:

1. Given a game tree, mark all terminal nodes with their utility value, perhaps 1 for win, 0 for draw, and -1 for loss.

2. Calculate the utility for each non-terminal node starting on the next to last level in the tree and work back to the root, one level at a time. The utility is computed by examining the utility of all nodes below it and becomes

$$u_t = \begin{cases} \text{max of all } u_{t+1} & \text{for even (player's) levels} \\ \text{min of all } u_{t+1} & \text{for odd (opponent's) levels} \end{cases}$$

3. The final "backed-up" value at the root node is the utility for the game and the move which lead to the maximum value is selected.

Using this technique, it would be interesting to determine if white could always win in Chess. Unfortunately the number of nodes grows exponentially with the depth of the tree, so that except for modest games like Tic-Tac-Toe, exhaustive search becomes quickly infeasible. Thus, generating a complete game tree for Chess could consume an eon.

## Static Evaluator

If there was an infallible, quickly computable function for labeling board positions as to whether they are on the path to success or failure, there would be no reason to generate the tree. Such a perfect function is likely to be unobtainable but an imperfect *heuristic static evaluator* might be available instead.

Designing a heuristic for a particular game can be tricky and usually requires an expert player. A very common method is to form a linear combination of features measured from a state $s$ as

$$E(s) = c_0 F_0(s) + c_1 F_1(s) + \cdots + c_n F_n(s) \tag{2.12}$$

where $E(s)$ is the rating, $F_i$ is a feature which can be measured from a game state, and $c_i$ is a feature coefficient. Determining useful features for a static evaluator and the coefficient values involves much trial and error.

To improve the utility of the imperfect static evaluator, a limited game tree can be generated to some reasonable depth. Each leaf node is evaluated and the results backed-up using the min-max procedure. It should be noted, though, that according to Perl (Pearl, 1983) no theoretical model supports the fact that the backed-up evaluation is any betterfor decision making than that of the current node. Instead, it's based upon intuitive statements like

- the fate of the game is more visible near its end so that a later evaluation should be more informed, and
- the backed-up value is produced from the evaluation of many nodes as compared to just the single current node.

## Improvements

Improvements on the techniques presented so far have been in terms of improving the min-max search to reduce the number of nodes that need to be visited in the game tree while still reaching deep. Tree pruning techniques such as *alpha-beta* and move ordering are examples of such improvements. In other cases, game tree searching hardware has helped produce deeper searches, such as that in the chess machine "Deep Though" currently rated at the grandmaster level.

The work in this thesis instead focuses on improving play ability by improving the static evaluator. As will be seen, the benefit of a better static evaluator is that it requires less search. In the upcoming problem of Tic-Tac-Toe, static evaluators are learned by neural networks that perform about as well as a full game search.

## 2.4 Programs That Learn to Play Games

This section reviews past and current game playing systems which have an ability to learn. All operate in either a supervised or reinforcement learning mode through which some form of game state/move evaluator is improved. The learning mechanisms range from Bayesian pattern classifiers to neural networks. Of most interest are the applications using TD learning, especially Tesauro's ground breaking work with TD-Gammon. Since his talk at the *Internation Joint Conference on Neural Networks* in July 1991, the approach has been adopted by others; several such case will be covered.

### Samuel's checker player

Samuel's work[6] with a checker playing program (Samuel, 1959) has probably been reference more than any other game learning research. Using a standard game tree search architecture, he investigated several different ideas for teaching the static evaluator. One method he toyed with during the 50's was what he called *generalization* learning, an early temporal difference method. He also investigated other techniques such as rote learning and nonlinear signature tables (Samuel, 1967).

The static evaluator was the usual linear combination of features as in equation 2.12. The coefficients for the features were adjusted during play by comparing the evaluation of the current state against the backed-up value provided by a min-max search. Any difference between these two values was assumed to be caused by error in the current state's evaluation. The generalization learning algorithm proceeded to adjust the coefficients to reduce this error there by making the static evaluator more accurate.

Training experience was gained via self play, the program acting as two different players, Alpha and Beta. Alpha adjusted its coefficients after each move, while Beta used a fixed static evaluator throughout the entire game. When Alpha won, Beta copied its coefficients; if Alpha lost three times in a row, its largest coefficient was reset to zero in order to bump it out of what

[6]Samuel included many notes in his paper on efficient program implementation since the computers of his day were quite limited by todays (1990's) standards. He even added a feature so that multiple games could be played in parallel against human players, avoiding the waste of precious machine time. In contrast, here I am at home with a personal computer on my desk that is probably faster than his IBM 704. Ain't technology great :-)

was believed to be a local minima. Human intervention was also required every now and then to tinker with the parameters that controlled the learning procedure.

### Results

Overall, the generalization learning resulted in play that was unconventional with weak openings, yet performed well from middle to end game. In fact, if it had a piece advantage toward the end, it typically had no problem finishing off its opponent. In 1962, the program played and won a match against checkers champion Robert Nealey using a static evaluator that was taught using both rote and generalization learning.

This being probably the first temporal difference method programmed on a computer, it is interesting to highlight the differences between Samuel's and Sutton's algorithms. For one, Samuel's used the backed-up value from a min-max search as the target for adjusting the evaluator as opposed to the evaluation from the successive state. In addition, the evaluation for the final board position was not adjusted as in Sutton's TD method. This made it possible to learn useless evaluation functions (ex. a constant function) that actually minimized the temporal difference while providing no hint at how to win. To prevent such functions, Samuel added the piece advantage to the static evaluator's output.

## Kolen and Goel's Neural Tic-Tac-Toe

In a study by John F. Kolen and Ashok K. Goel (Kolen and Goel, 1991) into the limitations of backpropagation learning, they used Tic-Tac-Toe as an example problem. Their system used a network with 18 raw inputs, 9 for the X positions and 9 for the O's, a single hidden layer in which the number of units was a parameter, and 9 outputs (one for each possible move). When presented with a board position, all moves selected (as indicated by an output greater than 0.5) were collected into a list and the illegal ones were weeded out. A move was then selected at random from the remaining list. If none of the outputs was greater than 0.5, a legal move was selected at random.

To train the network, they played it against a program that used an imperfect heuristic evaluator without lookahead. As such, it could be beaten. During the game, the sequence of board positions that occurred were re-

membered; at the end of each game, the network was taught in a supervised fashion using backpropagation to learn the move selections of the winner.

### Results

As a test, 10 sets of 1000 games were played between the neural net and its opponent. At the beginning of each set the neural net was given random weights. Usually the network learned to draw its opponent by the 100th game, and in 6 of the sets, less than 85 games were lost. Yet out of all 10,000 games, only one game was ever won by the network. They believe the reason that the network never learned how to win was that backprop failed to discover the needed abstractions (proper features and generalization).

There is likely another cause: their network failed to learn how to win because it wasn't given the opportunity! By training against the heuristic opponent in a supervised mode the best that could be hoped for was a clone. Only if it won by accident during the first few games could it adopt a better playing style. What was needed was a network that experimented, trying various moves. What was needed was state space exploration.

## Tesauro's NeuroGammon

Gerald Tesauro, creator of the successful TD-Gammon, previously tried a supervised learning approach for neural based backgammon players. This early network which he named NeuroGammon (Tesauro and Sejnowski, 1989) used a layered feedforward network as a static evaluator. The game state could be encoded as input for the network in two possible ways: raw ("Knowledge Free") or as a collection of features measured about the board (*pip count, degree of contact, number of points occupied in inner board, etc.*). When a decision was required for a give dice roll, each possible move was evaluated by updating the board as if the move was made; both this potential game state and the previous state were input to the network; the move which lead to the largest network output was then selected.

The network was trained using backprop with a database of 3202 board positions and their associated expert evaluations. Each entry consisted of a board position, a roll, a suggested move and the teacher's rating. In contained both bad and good moves so that a balance of "what and what not to do" could be learned.

### Results

The NeuroGammon configuration which used features was tested by playing against Sun Micro System's GammonTool, considered by Tesauro to be a reasonably good computer player. Although NeuroGammon was able to win 60% of the games, its ability was considered to be well below human expert-level. Still, in 1989 it proved to be an expert for computer opponents by winning the International Computer Game Olympiad.

Tesauro states that while it was possible to achieve descent playing ability, it was not easy, and required his intervention during the learning process (Samuel also had this problem). It was also found that NeuroGammon faired poorly when only given raw board inputs. It had been hoped originally that the learning process and feature discovery might have been more automatic, not requiring a domain expert.

## Tesauro's TD-Gammon

Tesauro decided to try the TD($\lambda$) method to determine whether it could be successfully applied to a more complex learning task than those outlined by Sutton. The architecture was similar to that of NeuroGammon except that instead of learning from a database of moves, it now learned while playing itself.

To begin with, several problem areas were addressed in (Tesauro, 1991) which could present road blocks to TD success.

- Will learning during self play turn into self consistent yet non-optimal play? Answer: No, if trained long enough.
- Will a TD method work when learning to predict and control simultaneously? Answer: Yes.
- Sutton's convergence proof is for linear activation functions. Will it converge with a nonlinear sigmoid? Answer: Yes.
- How does TD scale with the length of the temporal sequence (game length)? Answer: Still open question, but worked with Backgammon's typical game length of around 60 moves.

- Will TD work when the game exhibits *volatility*. Volatility occurs when the probable reward at state $s_t$ varies significantly from that at state $s_{t+1}$. Answer: So far, yes.

### Results

To train the network, it needed to play many games, up to 200,000 when initially reported on in 1991. Resulting capability was again benchmarked against GammonTool; other players are included for comparison.

| Player | Win Percentage Against GammonTool |
|---|---|
| Random | 1.0% |
| Beginning Human | 40.0% |
| NeuroGammon | 59.4% |
| Intermediate Human | 60.0% |
| TD-Gammon (raw input) | 62.2% |
| TD-Gammon (features) | 71.0% |
| Expert Human | 75.0% |

As can be seen, the TD based network with features performs nearly at an expert level. In addition, a network was able to learn only using the raw board as input and performed as well as NeuroGammon (which used features). Even better, Tesauro reported in June 1993 (Tesauro, 1993) that with additional training (1,500,000 games total) and a 2-ply search, TD-Gammon has actually achieved master-level play[7].

## Other TD Game Players

Papers have been written by at least two other individuals on using Sutton's TD method for game learning. Steven Walker's undergraduate thesis (Walker, 1992) describes learning experiments using the game Othello. His work mirrors Tesauro's approach, trying both supervised and TD method trained networks. Likewise, he also discovered that TD produced a superior player. After 30,000 training games of self play, the TD network beat the best supervised network about 70% of the time. Walker predicts that its

[7]Sorry that I haven't described TD-Gammon's latest achievement to any greater length, but its news just came out at the same time this paper was being finalized.

average capability is somewhere between the second and third levels of the Othello (Reversi) player that comes with MicroSoft Windows.

Justin Boyan (Boyan, 1992) extended Tesauro's ideas for his Master's thesis by creating a modular TD network which also played backgammon. His resulting program was entered into the 1992 Computer Games Olympiad under the name MAESTRO 1.0 and proceeded to take second place.

## A Bayesian Othello Player (alternative to ANN)

Researchers Kai-Fu Lee and Sanjoy Mahajan have been evolving a series of highly successful Othello playing programs known as BILL. The latest and greatest version, BILL 3.0 (Lee and Mahajan, 1988), was taught using supervised learning much like Tesauro's NeuroGammon. Instead of using a neural network, though, they relied on a classic pattern recognition technique, a Bayesian classifier. In addition, a human expert was not required to generate the training database; BILL 3.0's predecessor, BILL 2.0, was available to perform that tedious task.

BILL 2.0 evaluated board positions using game tree search and a linear static evaluator as in equation 2.12. Based on trial and error, coefficients belonging to four features were carefully hand tuned until expert play was achieved. BILL 3.0 replaced the linear evaluator with a Bayesian classifier, a nonlinear function capable of discriminating between classes of normally distributed patterns. This classifier is adapted automatically via a statistical training procedure that operates with a database of labeled patterns, in this case Othello board positions known to develop into a win or loss. By arranging for BILL 2.0 to play itself for 3000 games, a database of both good and bad board positions was generated.

### Results

BILL 3.0 gained additional capability which could be equated to 2 additional plys of search over BILL 2.0. Given Othello's game tree branching factor, the authors computed that BILL 2.0 would take 13 times longer to look ahead far enough to be evenly matched with BILL 3.0. In a 200 game tournament in which each program was allowed 25 minutes to complete all its moves, BILL 3.0 won 139 and tied 6.

It appears that Lee and Mahajan may have found a possible solution to the supervised learning problem—the student unable to surpass the abilities of its teacher. Is it possible, by continued leap frogging of programs each trained by a predecessor using deeper search, that the static evaluator will continue to improve? Of course, this is only useful for games in which lookahead is feasible.

# Chapter 3

# Learning System Architecture

This chapter defines a learning system architecture which contains a framework and set of guidelines for implementing programs to perform and learn sequential decision tasks such as games. The approach is simplistic; just play the game and adapt a static evaluator based on the final outcome. It does not rely on fancy game tree search or complex planning although such features are not ruled out. In fact, game tree search is included in the architecture although it is never used beyond a 1-ply search for the tasks in this thesis.

At the highest level, the framework consists of two elements: the *Player* (controller) and its *Environment* (Figure 3.1). The Player is essentially a classical game tree program with an adaptive static evaluator built from a feedforward neural network. This is combined with a temporal difference method to support reinforcement learning. The Environment must keep tabs on the game making sure that the rules are properly followed and that the state is correctly updated according to the Player's actions. When the game

Figure 3.1: The framework for the learning system consists of an Environment that ensures that a proper game is played, and the Player which must learn how to win.

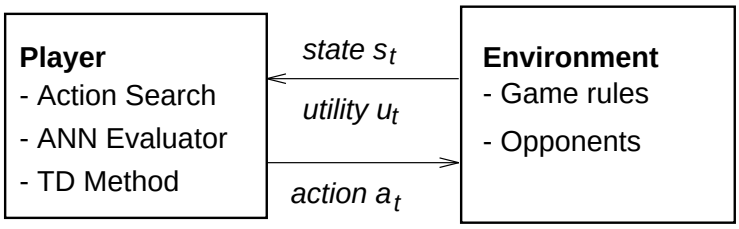
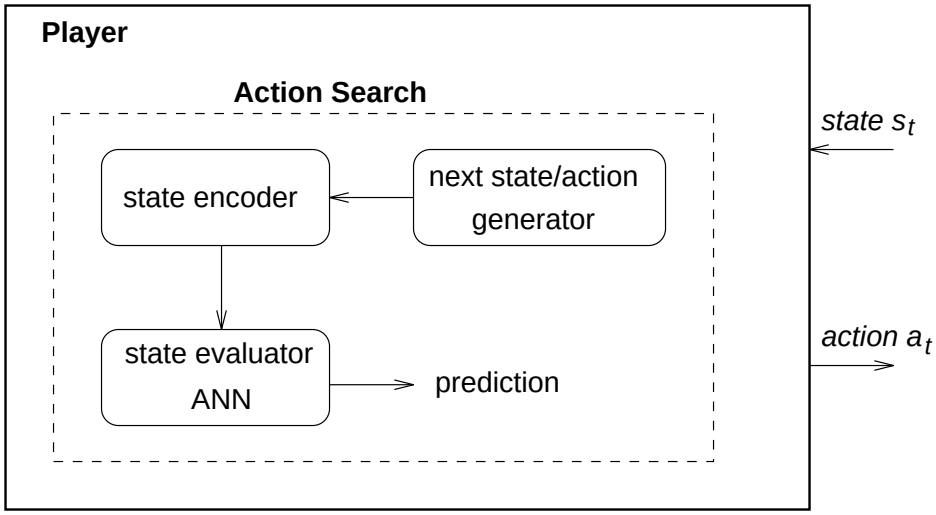


Figure 3.2: The three major functions computed to select an action.




reaches a terminal point (win, loss, or tie), the Player is notified, who can then adjust its state predictions. Opponents are considered to be a part of the Environment from the Player's viewpoint.

## 3.1 Playing

During a game, the Player receives a state vector from the Environment to which it must respond with an action. The process of selecting an action consists of three functions: action/next state generation, state encoding, and state evaluation (see Figure 3.2). Two distinct methods for selecting an action are *direct* and *indirect* (Thrun and Möller, 1991).

**Direct** — A direct selection scheme uses a neural network to map a state to an action without any search.

*Advantage:* It selects an action quickly and and in constant time.

*Disadvantages:* It doesn't handle many-to-many mappings of states to actions (there might be several equally good actions) and can be cumbersome when the the set of possible actions is large.

**Indirect** — An indirect selection scheme searches for appropriate actions by generating and evaluating each with a model of the Environment that predicts its resulting outcome (the neural network).

There are several ways to design the network, each with different advantages. Essentially, the prediction can either be a function of the state and potential action $P(s_t, a_t)$ or just the state $P(s_t)$.

- Function of State and Action, $P(s_t, a_t)$ —
  Actions are selected by evaluating each legal action $a_t$ and selecting the one with the highest evaluation.

  *Advantage:* Good for when state transitions have an element of chance or when a model that can generate next states is not available.

  *Disadvantage:* Becomes complex when number of possible actions is large since predictor needs to consider a larger input space. States may not be evaluated same way when reached through different actions.

- Function of State, $P(s_t)$ —
  This is essentially a static evaluator used in a game tree search. Using this method, an action is selected for a given game state $s_t$ by generating all legal actions $a_t$ their next states $s_{t+1}$. These are then evaluated with the network and the action receiving the largest evaluation is selected. When there are multiple resulting states from a single action, the predictions for each can be combined using something like a weighted average based on the probability of its associated state occurring.

  *Advantage:* Can look ahead using game tree search. Allows states to be evaluated the same regardless of the path that got to them.

  *Disadvantage:* A full model of the game must be available to determine each possible $s_{t+1}$, and search time can grows excessive when the set of possible $s_{t+1}$'s is large.

Asking questions about a tasks such as, "How many possible actions are there?", or "Is there a model for predicting the next state?" help determine the best approach to use. For example, Blackjack has just two legal actions (hit and stand) but is difficult to generate the resulting next states due

to dealing from a deck of shuffled cards. Here, indirect selection with the prediction as a function of the current state and an action is the preferred choice. On the other hand, it's easy to determine the resulting state from an action in Tic-Tac-Toe so the $P(s_t)$ form can be used. For robot controllers, direct selection is usually the best choice such as the Adaptive Heuristic Critic.

## 3.2 Learning

Reinforcement learning is provided by a TD method similar to that of Sutton's but with a slight difference. As previously noted, TD(0) has been shown by Sutton to be an optimal choice for $\lambda$ since it converges to the best estimates for matching future experience. With long sequences of actions, though, it may take longer than the TD(1) approach to find a solution. This results from the fact that the utility being learned is only propagated back one state at a time; a game with 100 moves would take 100 games to trickle reinforcement to the first move.

A solution is to use a non-incremental version of the procedure, which will be referred to as TDX (for TD eXperimental); it includes the benefits of TD(0)'s optimality while learning faster. It operates by saving all the states visited during a sequence and then learning predictions backwards in time. The weights are updated after $\Delta w$ is computed for each state/prediction pair so that the final utility will trickle throughout the whole sequence, gradually diminishing in proportion to the learning rate $\alpha$.

The neural networks can be configured with optional features such as momentum, skip-layer connections, and variable rates for each unit. In addition, the Cascade Correlation Architecture (Fahlman and Lebiere, 1990) which "grows" a network topology is being modified to work with the TD method, but at the time of this writing is not yet ready. It would be interesting to compare different options but this is currently beyond the scope of this thesis. In general, vanilla backprop will be used on all problems, performing weight updates after every presentation (as opposed to summing up weights changes and applying them in one *batch*).

## 3.3 State Space Exploration

As stated earlier, for a TD method to work well, every state in the space must be visited multiple times. Although neural networks can generalize and reduce the number of states that need be encountered, it's still important to try alternatives. Games with chance state transitions inherently explore the space; it comes for free. Games with deterministic transitions, though, can lead to players being stuck with consistently bad play. This is especially a problem when trying to learn from self play.

A simple solution is to add a random offset (noise) in a fixed range to the predictions of the neural network. The magnitude of the range specifies the degree to which the network has control over its playing capability. It might even be useful to put the level of noise on a schedule such that it begins large and is gradually reduced over time, perhaps proportional to $1/t$. This reduction in the amount of noise added could be interpreted as adding confidence to the decision making process as more experience is gathered, where experience is measured by time.

## 3.4 Implementation

Libraries of neural network and temporal difference routines were developed in C to support many kinds of neural architectures, although only backpropagation is used at this point in time. The main elements of the framework—Players and Environment—operate as separate processes communicating states and actions via message passing. This allows different player programs (traditional game search or neural net based) and different Environment programs (some with graphical user interfaces, others designed for testing) to be easily combined in different configurations.

The software was developed and operated on an Amiga 1000. This is 9 year old computer with a Motorola 68000 running at a little under 8 MHz without a floating point coprocessor. Due to its lack of speed, the ability to try alternative network topologies and parameter settings was very limited. In fact, a learning trial of 13,600 games of Tic-Tac-Toe with raw state inputs (see Section 6.8) took about 5 days to complete. In contrast, Boyan (Boyan, 1992) reports training times of about 2 hours for a comparable learning trial on DEC 3100 workstation.

# Chapter 4

# Maze Mapping

In this example, capabilities of the TD learning procedure are tested by using a fairly simple sequential decision problem. Answers will be sought to questions such as

- How do TD(1), TD(0), and TDX modes compare?
- How do they function under influence of both deterministic and random state transitions?
- What is the "best" learning rate?
- How state space exploration via noise help?

## 4.1 The Problem

A maze is laid out as a matrix (see Figure 4.1) in which each cell is marked as a passage or wall, and one special cell is the exit. The main goal is to find a map for the maze that describes how to get to the exit from any starting position. A secondary goal is to find a map that minimizes the number of moves needed to get to the exit.

Just what is this map? At any given location you must decide whether to go up, down, left, or right. The map provides information for discriminating between these four choices by associating a number with each cell representing how close it is to the exit. For example, $Map(0,1) > Map(1,1)$

Figure 4.1: An 8 x 8 maze in which '###' denotes a wall and ' E ' marks the exit.

```
################################
### E ###   ###           ###
###               ###   ######
###   ###############   ######
###   ###         ###     ###
###         #########   ######
###   ###             ######
###   ######   ######     ###
###            #########  ###
################################
```

means that the coordinate (0,1) in the maze is likely further from the exit than (1,1).

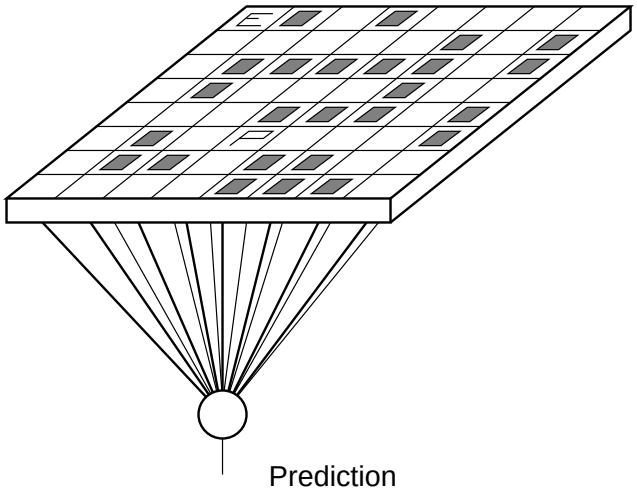
## 4.2 Architecture Setup

The system components for the learning system are:

- The *environment* is the matrix representing the maze
- The *State* is the coordinate of Player's current cell $(i, j)$
- Valid *actions* are {*up, down, right, left* }
- *Utility* is number of moves used to reach the exit
- *Player* uses an indirect action selection scheme $P(s_{t+1})$:

```
FOREACH possible action {up, down, right, left}
   DETERMINE the next state coordinates
   LOOKUP its prediction in the map
SELECT action which has lowest map value.
```

Figure 4.2: The map is represented by a single linear unit with inputs from each cell of the maze. P represents the player's position and is the only input turned on (1); all the rest are off (0). E signifies the exit.




There are more efficient methods for solving a maze such as depth-first search in which routes are exhaustively searched until the exit is found. At worst case, every cell in the maze is visited once. In contrast, the method used here is only allowed to search one step ahead at a time and has no capability for backtracking. As such, it may wander about aimlessly until it gains enough experience.

A single layer neural network with a linear output is used to represent the map (see Figure 4.2). Its inputs are arranged such that each corresponds to a unique location in the maze. In this way, each weight's value can be thought of as an estimate of the number of moves needed from its corresponding location in the maze to reach the exit. Although this representation does not allow generalizations to take place, it is easier to understand how and what is learned.

## Learning Cumulative Cost

A slight twist on the TD methods must be presented since this problem involves learning to predict a cumulative outcome (the number of moves) as opposed to a final outcome. The general TD learning rule can be changed so

that it accumulates a cost function (Sutton, 1988).

$$\Delta w_t = \alpha(c_{t+1} + P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \frac{\partial P_k}{\partial w} \tag{4.1}$$

The only difference is that a new term $c_{t+1}$—the cost of traversing from state $s_t$ to $s_{t+1}$—is added to the previous prediction. For the maze, the cost is will always be 1 move to get to the next cell. There are situations in which the cost might be different for each traversal, allowing it to be applied to problems such as the Traveling Salesman (though not very efficiently).

## 4.3 Performance Metrics

Initially the network's weights are set to random values. A trial begins by selecting a random starting position and letting the Player move about until it either reaches the exit or exceeds some maximum number of moves, in this case 20. If the exit is reached, the final utility $u_f$ is 0, otherwise it is 20 (meaning, cells visited along your last trial were not good choices to find the exit). The TD learning procedure is then applied using $u_f$ as the target. The values for maximum moves and the utility for exceeding it were chosen arbitrarily without testing the sensitivity of the learning procedure to their settings.

### Maze Runs

A maze run consists of a series of trials, each with a random starting position in the maze. The trials continue until the maze is fully mapped, which is determined by testing the map after every trial to see if the exit is reachable from every possible starting position. If the run needs more than 300 trials to find a map it is declared a failure.

Some neural networks will have "natural ability" for solving the maze, meaning that their initial random weights and the random starting positions will help them find a map quicker. To compensate, results are averaged over 20 runs for a given set of parameters.

Ability of the Players will be measured in two ways: quickness in finding a map and the efficiency of the resulting map. To measure quickness, the average number of trials needed to complete the map, plus the min and

Table 4.1: Comparison of TD modes on maze trials.

| Mode | Rate | Runs | Avg Trials | Min | Max | Std. Dev. | Avg. Path |
|---|---|---|---|---|---|---|---|
| TD(1) | 1.0 | 18 | 180.7 | 64 | 275 | 58.2 | 7.79 |
| TD(0) | 1.0 | 20 | 116.0 | 85 | 157 | 20.8 | 7.59 |
| TDX | 1.0 | 19 | 191.3 | 107 | 288 | 46.0 | 7.92 |
| TD(1) | 0.5 | 20 | 142.9 | 65 | 266 | 48.4 | 7.86 |
| TD(0) | 0.5 | 20 | 110.4 | 76 | 144 | 19.3 | 7.60 |
| TDX | 0.5 | 20 | 64.8 | 40 | 104 | 17.5 | 7.62 |

max are computed. The standard deviation is also provided to show consistency. To measure efficiency—the shortness of the paths found—the average path length from all possible starting positions is measured after the maze is mapped. For the maze which is used in all the following test runs (Figure 4.1), the minimum average path length is 7.5.

## 4.4 TD Mode Comparisons

For the first series of runs, the different modes, TD(1), TD(0), and TDX, are compared at different learning rates, 1.0 and 0.5 in Table 4.1 . The column *Runs* indicates the number of successful runs. The following observations can be made:

- Rates lower than 1.0 are quicker for learning a map.
- TD(0) tends to perform better than the other modes at a rate of 1.0 since TD(1) and TDX fully modify their weights based on the results of the last trial, forgetting any previous predictions.
- TD(0) finds more efficient maps (average path length closest to 7.5), and is more consistent at doing it (low standard deviation).
- At a rate of 1.0, TD(1) and TDX perform exactly the same[1]. As the learning rate lowers, TDX begins to match the behavior of TD(0) by finding more optimal maps.

Figure 4.3: Here is an example maze map which was learned. In most cases, the entries turned out to be an estimate of the number of moves needed to reach the exit. The 40's that show up, though, are that way since they were sufficient to lead the proper way to the exit but hadn't actually shown up during a learning trial which made it to the exit.

```
##############################
### E ###  4###  6  7  8  9###
###  1  2  3  4  5###  9######
###  2############### 10######
###  3###  7  8  9### 11 40###
###  4  5  6######### 12######
###  5###  7  8  9 10 11######
###  6######  9###### 12 40###
###  7  8  9 10######### 40###
##############################
```

- At rates less than 1, TDX is faster than TD(0) since TDX allows the reinforcement to go all the way back, though it is diminished in proportion to $1 - \alpha$.

The map that is learned is very easy to understand. One of the maps learned during these runs is displayed in Figure 4.3 where the predictions for the number of steps to the exit can be seen to form trails of decreasing values.

## 4.5 Exploration Though Noise

By watching the maze get mapped in the last runs, it appears that the Player initially puts too much faith in its inexperienced map. This is apparent in that the Player gets into holes which happen to be local minima and won't

[1]This can be explained as follows. Remember that TDX works like TD(0) in that the error is driven by $P_{t+1} - P_t$, except that the weights are updated before the previous prediction. Now when a linear output unit is used and each state is represent by exactly one input being set to 1, learning a target output at a rate of 1.0 will exactly learn that target. This means that $P_f = u_f, P_{f-1} = P_f, \ldots, P_{t+1} = P_{t+2}$ so error turns into TD(1)'s, $u_f - P_t$.

Table 4.2: Comparison of TD modes on maze trials when noise is added.

| Mode | Rate | Runs | Avg Trials | Min | Max | Std. Dev. | Avg. Path |
|---|---|---|---|---|---|---|---|
| TD(1) | 1.0 | 20 | 73.5 | 21 | 133 | 30.3 | 7.93 |
| TD(0) | 1.0 | 15 | 162.5 | 88 | 258 | 61.1 | 7.74 |
| TDX | 1.0 | 20 | 100.1 | 32 | 299 | 57.7 | 7.85 |
| TD(1) | 0.5 | 20 | 103.0 | 31 | 255 | 54.4 | 7.87 |
| TD(0) | 0.5 | 20 | 91.3 | 50 | 207 | 41.4 | 7.57 |
| TDX | 0.5 | 20 | 45.1 | 21 | 78 | 16.2 | 7.75 |

climb back out. People wouldn't do that. If you were dropped off into a maze that you had never visited before, your first reaction would be to explore and gain new knowledge. Later, as the maze became more familiar, you would use your experience to find the way out.

The Player can be forced to explore by adding a little noise to each map value so that any of the four directions might be selected. As experience is gained, the difference between map values will expand giving the noise less influence. For these runs, the random noise offset ranges from 0 to 1.5. This time we find in Table 4.2 that

- The amount of time to find a complete map for TD(1) and TDX drops a quite a bit with the introduction of noise.
- As the learning rate lowers, TDX improves even more in finding efficient paths and becomes more consistent (signified by a low standard deviation).
- On the other hand, TDX with a lower rate seem to be disturbed by the 1.5 random offset. This is likely caused by the use of a constant noise range throughout the run. With higher learning rates, TDX is able to quickly separate map values so they are less affected by the noise. For TDX at lower learning rates, this separation of predictions does not occur as fast.

Table 4.3: Comparison of TD modes in maze with random transports.

| Mode | Rate | Runs | Avg Trials | Min | Max | Std. Dev. | Avg. Path |
|---|---|---|---|---|---|---|---|
| TD(1) | 0.5 | 0 | **Never completed a run** | | | | |
| TD(0) | 0.5 | 5 | 205.2 | 108 | 274 | 68.7 | 7.68 |
| TDX | 0.5 | 2 | 133.0 | 82 | 184 | 72.1 | 7.65 |
| TD(1) | 0.25 | 0 | **Never completed a run** | | | | |
| TD(0) | 0.25 | 20 | 133.5 | 83 | 192 | 36.1 | 7.56 |
| TDX | 0.25 | 20 | 92.3 | 50 | 183 | 35.8 | 7.57 |

## 4.6 Operating in a Random Environment

So far the runs have been made in a maze that is quite predictable. If the Player decides to move to the left, it actually ended up to the left. For this series of runs, the maze environment will be revised so that this is no longer true. This is a good test since many decision problems do not allow you to know for sure what the resulting next state will be.

Suppose that there is some entity that likes to have fun with you and occasionally will transport you to a random position in the maze. This is precisely what will now happen with a 1/8th chance. Again runs are made with various learning rates, and the noise is still set at 1.5.

It can be seen that from Table 4.3 TD(1) is definitely not the right choice, as it puts too much emphasis in just the results of the previous trial. The problem is that the last trial may have had transitions that are unlikely to happen exactly that way again. As a result, it never mapped the maze in under 300 trials. In contrast, TD(0) and TDX both were capable at forming a map, especially when the rate was at 0.25. The maps learned were near optimal for both as indicated by the low average path lengths near 7.5. TDX appears to be slightly faster than TD(0), although with a sample size of 20 and large variance, it's hard to tell.

# Chapter 5

# Blackjack

In this section, the learning system architecture is used to implement a program that learns how to play Blackjack. It is felt that this game might be especially good for understanding learning issues related to random transitions. In addition, the rules are simple, the game length is short, and there are only two valid actions, *Hit* and *Stand*. The objectives are to

- Observe what happens when information is withheld and state transitions are random.
- Look at how different methods of encoding the game state as input to the neural net affect learning speed and stability.
- Compare the modes TD(0), TD(1), and TDX again. This time, though, the temporal sequence is typically quite short so only small differences are expected.

To show that the networks actually do learn something useful their policies will be compared with the abilities of both good and bad playing strategies.

## 5.1 Blackjack Rules

Blackjack is played with a regular deck of 52 cards and involves a single player against the dealer. The object is to hold cards which sum up to or almost 21, but without going over (busting). The cards are counted as follows: An

*Ace* can be a 1 or 11 as decided by the player; face cards count as 10; all other cards are their number.

Play goes as follows: Two cards are dealt to the player and the dealer, the first one face down—it can can only be viewed by its holder—and the second face up. The player may either stand or take additional face up cards by saying "Hit me". If the total happens to exceed 21, then the player immediately loses. Once the player has decided to stand, the dealer must take additional cards until his total is 17 or greater. If the dealer busts, the player then wins, otherwise the player's hand must be greater than the dealer's to win. In gambling-house rules, there are many other interesting aspects (betting, doubling) which are being ignored to keep the overall problem simple.

## 5.2 Architecture Setup

The components for the Blackjack learning system are:

- An *environment* which consists of a deck of cards shuffled before each hand and the decision policy governing the dealer's actions.
- A *state* comprised of the sum of cards being held and the dealer's "up card". When summing up the hand, an *Ace* is automatically valued as an 11 unless it would cause a bust. In this case it is used as a 1.
- Valid *actions* are {*hit, stand*}.
- A *utility* of 1 for win and 0 for loss.
- A neural *player* using an indirect action selection scheme in which two outputs, $P_{hit}$ and $P_{stand}$, are produced for predicting the utility of their respective actions. Using these values an action is selected as

$$\text{action} = \begin{cases} \text{hit} & P_{hit} \geq P_{stand} \\ \text{stand} & \text{otherwise} \end{cases}$$

To limit the outputs to the range $[0, 1]$, the typical sigmoid activation function[1]

$$f(x) = \frac{1}{1 + e^{-x}}$$

is used where $x$ is the weighted sum of the units inputs. Noise will not be used to force exploration of the state space; in this situation, the random nature of the game should provided the necessary prodding.

## 5.3 Performance Metrics

It is important that what is learned and how it is learned can be measured. For this reason, the following metrics will be used:

- A learning trial consists of 1000 games and begins with random network weights. *Overall capability* of learning while playing is the win percentage for the whole trial. Note that this score will include the poorly played games when the network is inexperienced.
- Final *test capability* is measured as the win percentage over a fixed set of 400 test games (same seed used to initialize the random number generator for card shuffles).
- The overall and final capabilities are benchmarked against several fixed policy Blackjack players. This helps decide if useful learning took place.

It turns out that the variance in win percentages for Blackjack is quite large. For this reason, it is necessary that many games are played—in this case at least 1000—before the results of a trial are meaningful. To further dampen this variance and that of the learning process, the results of 4 trials[2] are averaged.

### Benchmarks

Fixed policy players are used as absolute benchmarks for the policies learned by the neural networks. Four separate decision rules are used ranging from bad to good. None make use of card counting, as that would be useless with the deck being reshuffled before each hand.

[1] It is possible that a simple linear transfer function, $f(x) = x$ might also work even though its range extends above and below 1 and 0.

[2] This is probably not enough to make a statistically sound statement about capabilities, but given the limited processing power of the Amiga 1000 (ancient by today's standards), it was a tradeoff I could settle for.

Table 5.1: Capabilities of four different fixed Blackjack strategies.

| | Test | Overall Win% | | |
|---|---|---|---|---|
| Policy | Win% | Avg | Max | Min |
| Randy | 30.0 | 28.5 | 30.5 | 26.9 |
| Hold | 36.5 | 38.0 | 39.9 | 35.7 |
| Dealer | 39.8 | 39.6 | 42.0 | 37.4 |
| Hoyle | 41.0 | 42.6 | 45.6 | 41.3 |

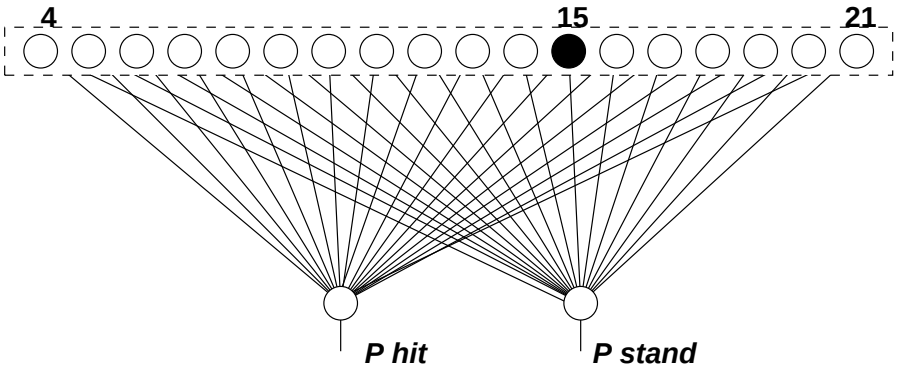
- *Randy* — Selects hit or stand at random.
- *Hold* — Always stands with the first two cards dealt.
- *Dealer* — Use the dealer's policy of standing when sum is 17 or greater.
- *Hoyle* — A policy suggested by *Hoyle's Rules of Games* (Morehead and Mott-Smith, 1963). Stand on 17 or greater, unless dealer's face up card is a 2 through 6, in which case stand on 13 or greater. If an ace is held and counted as 11, then stand on 18 or 14 if dealer's face up card is 2 through 6.

The capabilities of each are summarized in Table 5.1. The column labeled "Test Win%" contains results of playing the 400 fixed test games. The other columns are the results of four trials of 1000 games each. Note that the difference in maximum and minimum win percentages for each policy is about 4%. If the policies learned by the neural networks display this kind of variance, the process is probably consistent. At the same time, the difference between the "conservative" *Hold* policy and the better *Hoyle* policy is only 5%. This small improvement suggests that besides being a difficult game to win, it may be difficult to measure where a neural player's ability level really stands in relation to the others.

## 5.4 Table Lookup Encoding

The first test will be to compare the various TD modes and sensitivities to different learning rates. A simple encoding of the Blackjack state is used in

Figure 5.1: Blackjack neural evaluator using a Table lookup state encoding in which a single input unit is associated with each possible sum. Here a sum of 15 is being represented.




which each possible sum, 4 to 21, is encoded by turning on a single input unit. Each input is attached to both output units, $P_{hit}$ and $P_{stand}$, forming a single layer 18-2 neural net (see Figure 5.1). Since only one input will be on at a time, information about each state is stored by a single connection weight, making the network just a lookup table.

Each of the TD modes (TD(0), TD(1), and TDX) are tried using 3 different learning rates: 0.5, 0.1, and 0.05. The results are summarized in Table 5.2.

- Final play capability was on average near the level of the fixed Dealer policy of 39.8%. At worst, it still performed as well as the Hold policy. Since information about the dealer's up card is being withheld, this result seems reasonable.

- Useful learning did take place. The average capability before any learning took place was measured at 27.5% wins, which is actually worse than the Randy policy. This means that the overall improvement was about 11% to 12%.

- The learning process was somewhat consistent as the difference in final test percentages was about 3%. The overall min and max values differed by about 6%, which is slightly larger than the 4% variance exhibited by the fixed policy players.

Table 5.2: Results of various Blackjack learning trials using the Table state encoding.

| | | Test | Overall Win% | | |
|---|---|---|---|---|---|
| Mode | Rate | Win% | Avg | Max | Min |
| TD(0) | 0.5 | 39.6 | 39.3 | 39.9 | 38.5 |
| | 0.1 | 40.1 | 37.7 | 39.5 | 36.8 |
| | 0.05 | 40.6 | 40.5 | 41.8 | 39.6 |
| TD(1) | 0.5 | 38.9 | 38.9 | 42.9 | 36.9 |
| | 0.1 | 37.4 | 39.3 | 41.1 | 37.9 |
| | 0.05 | 39.2 | 40.0 | 40.5 | 39.7 |
| TDX | 0.5 | 38.7 | 39.3 | 42.1 | 38.3 |
| | 0.1 | 39.7 | 40.1 | 41.5 | 38.4 |
| | 0.05 | 38.1 | 39.4 | 40.4 | 39.6 |
| Overall | | 39.2 | 39.4 | 42.9 | 36.8 |

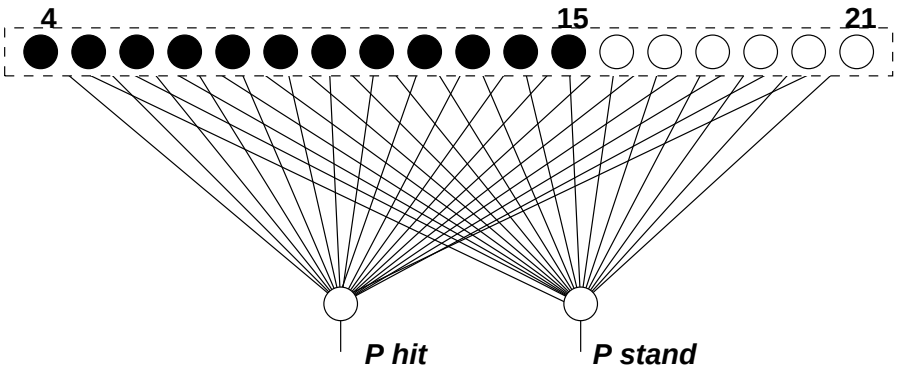
- The final capability appeared insensitive to the set of learning rates tried. It was expected that the larger learning rate might cause large oscillations in the capability.
- No major differences noted in results due to different TD modes. This was expected, since decision sequences are short.

It is interesting to observe the learned decision policy. Since information was limited to just the sum of the hand, there are only 18 possible states to examine. The policy in Table 5.3 resulted from one of the previous runs with a learning rate of 0.05 and tested at 39.3% wins. Even though the values of $P_{hit}$ and $P_{stand}$ have been discussed as if they represent the probability that their respective actions will lead to a win, they did not converge to the true expected values. Obviously all that really matters is that their outputs properly discriminate between good and bad actions. Note also that the differences between $P_{hit}$ and $P_{hold}$ tend to be very small. This is a result of the lower learning rate.

Table 5.3: A policy which was learned using the Table state encoding.

| Card Total | $P_{hit}$ | $P_{stand}$ | Action |
|---|---|---|---|
| 4 | 0.505 | 0.502 | hit |
| 5 | 0.505 | 0.500 | hit |
| 6 | 0.493 | 0.490 | hit |
| 7 | 0.504 | 0.491 | hit |
| 8 | 0.496 | 0.495 | hit |
| 9 | 0.507 | 0.500 | hit |
| 10 | 0.505 | 0.497 | hit |
| 11 | 0.504 | 0.502 | hit |
| 12 | 0.471 | 0.470 | hit |
| 13 | 0.450 | 0.453 | stand |
| 14 | 0.461 | 0.461 | hit |
| 15 | 0.456 | 0.454 | hit |
| 16 | 0.452 | 0.454 | stand |
| 17 | 0.460 | 0.461 | stand |
| 18 | 0.476 | 0.489 | stand |
| 19 | 0.486 | 0.544 | stand |
| 20 | 0.488 | 0.551 | stand |
| 21 | 0.492 | 0.607 | stand |

Figure 5.2: Blackjack neural evaluator using a Thermometer state encoding in which a sequence of input units are associated with each possible sum. The sum 15 is represented.




## 5.5 Thermometer Encoding

A problem with the Table lookup approach is that as learning progresses, predictions for less frequently visited states cannot take advantage of experience gathered from similar states. There is simply no ability to generalize. For example, if the neural network learned that standing at 5 often resulted in a loss, then that knowledge should somehow help imply that standing at 4 might also result in a loss. The evaluator needs a hint how states are related, namely the closer the sum of card values, the more similar the state.

One way to code this information is to use a thermometer scale (Figure 5.2). Again there are 18 input units to represent the sums of 4 to 21, but now a sequence of units is turned on to represent the value. Using this method to encode a sum of 15, units 4 through 15 are turned on instead of just unit 15.

Again, a series of trials are run as before with different learning rates and TD modes. The results are summarized in Table 5.4.

- Final playing capability was at worst (looking at mins) as good as Dealer policy and overall capability was slightly yet consistently better than the Table encoding.

- The learning process appears to be even more consistent than the Table Encoding as the difference in final test percentages was about 1% as compared to 3%. The overall min and max values differed by 6% as in

Table 5.4: Results of various Blackjack learning trials using the Thermometer state encoding.

| Mode | Rate | Test Win% | Overall Win% Avg | Max | Min |
|---|---|---|---|---|---|
| TD(0) | 0.5 | 39.3 | 39.2 | 41.8 | 37.6 |
| | 0.1 | 39.5 | 40.7 | 43.3 | 39.5 |
| | 0.05 | 40.2 | 41.1 | 41.8 | 40.6 |
| TD(1) | 0.5 | 39.8 | 40.7 | 41.5 | 39.9 |
| | 0.1 | 39.2 | 40.0 | 40.7 | 38.9 |
| | 0.05 | 40.1 | 41.0 | 42.5 | 39.9 |
| TDX | 0.5 | 39.0 | 40.7 | 41.4 | 39.8 |
| | 0.1 | 41.1 | 40.3 | 44.0 | 38.3 |
| | 0.05 | 41.0 | 40.4 | 41.5 | 39.0 |
| Overall | | 39.9 | 40.5 | 44.0 | 37.6 |

the Table Encoding case.

- This time average capability before any learning took place was measured at 19.1% wins, which is much worse than the initial Table encoded policies[3]. Average improvement from start to finish was 21%.

- Again, no major differences noted in results due to different learning rates or TD modes.

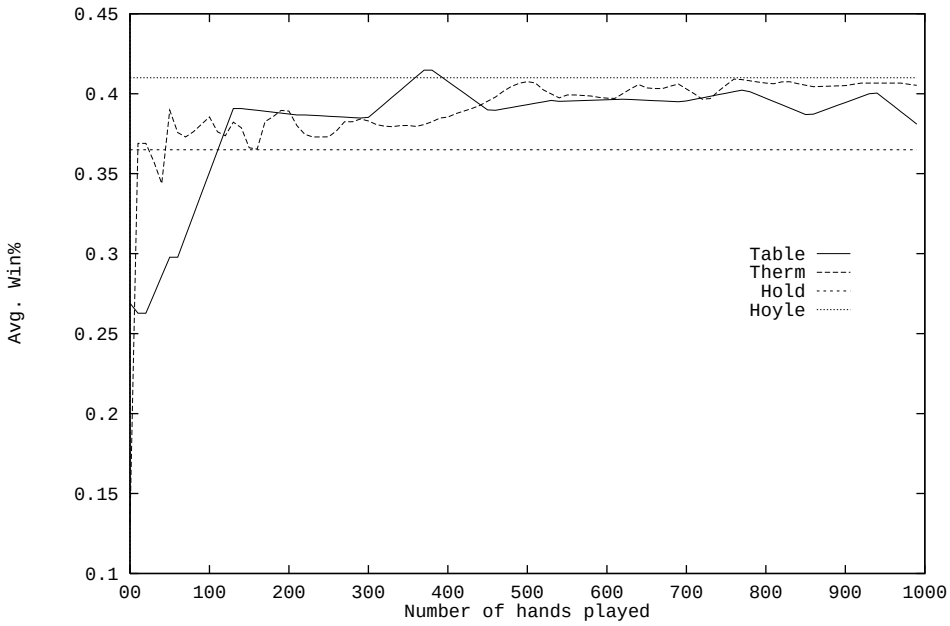
Table 5.5 shows the final decision policy from a run in which the learning rate was 0.05 and the final capability tested at 41.7% wins. This time the outputs of $P_{hit}$ and $P_{stand}$ appear closer to the real probabilities of win. This is likely due to each input unit being on more often, giving its weights additional chances to adapt. Note also that in the range of card totals from 12 to 15 the outputs of the two predictors are very close. This range also happens to

[3]The reason that the Thermometer style encoding resulted in much lower average starting abilities is an effect of multiple input units being turned on together. When more units are feeding the sigmoid activation function, it is more likely to saturate either to 1 or 0. This in turn has a good chance of generating an always "hit" policy which never wins. In any event, it appears that initial ability has little or no effect on the final ability.

Table 5.5: A policy which was learned using the Thermometer state encoding.

| Card Total | $P_{hit}$ | $P_{stand}$ | Action |
|---|---|---|---|
| 4 | 0.424 | 0.388 | hit |
| 5 | 0.413 | 0.366 | hit |
| 6 | 0.406 | 0.338 | hit |
| 7 | 0.398 | 0.318 | hit |
| 8 | 0.379 | 0.297 | hit |
| 9 | 0.371 | 0.289 | hit |
| 10 | 0.357 | 0.276 | hit |
| 11 | 0.337 | 0.257 | hit |
| 12 | 0.296 | 0.252 | hit |
| 13 | 0.269 | 0.254 | hit |
| 14 | 0.246 | 0.256 | stand |
| 15 | 0.239 | 0.281 | stand |
| 16 | 0.228 | 0.333 | stand |
| 17 | 0.227 | 0.420 | stand |
| 18 | 0.217 | 0.538 | stand |
| 19 | 0.211 | 0.666 | stand |
| 20 | 0.211 | 0.750 | stand |
| 21 | 0.210 | 0.803 | stand |

Figure 5.3: Tested capability of several neural network configurations versus experience (total of hands played). The two curves displayed resulted from using Table and Thermometer state encodings with TDX mode and learning rate 0.05.




correspond to the states in which it is important to consider the dealer's face up card when deciding to stand (recall Hoyle's strategy). In a later section we will see if the neural network can properly utilize this extra information.

## 5.6 Measuring Learning Speed

The previous runs seems to indicate that perhaps the Thermometer encoding performs better because it can learn faster. To find out if this is truly the case, the networks are each tested using the 400 hand sequence as usual, except now at 20 game increments.

Representative curves of capability versus games played are shown in Figure 5.3 for both Thermometer and Table encodings. The points are averages over four trials. For comparison, the straight lines represent capabilities of

two fixed policies, *Hold* and *Hoyle*.

First of all, the tiny zone of win percentages between the better Hoyle policy and the mindless Hold policy is obtained by the 100th hand played. This indicates that most important learning takes place rather quickly. The Thermometer based policies tend to have a little edge although smaller than expected. In addition, the up and down swing of the curves means that the policies are probably changing too much. Obviously the Table policy would have looked highly successful if learning was stopped around 380 games.

## 5.7 Measuring Stability

Since the state space has been kept purposefully small (only 18 states), it is feasible to measure policy stability as the average number of decisions which are changed each hand. For example, changing from "stand" on 16 to "hit" on 16 would be a single decision change. This turns out to be the hamming distance between the before and after policies.

The same neural network configurations used for measuring learning speed are again measured for policy changes each time the weights are updated. Figure 5.4 is a graph of the average policy changes over 1000 hands. The Thermometer based networks make many more changes in their strategies than the Table bases ones, though at a rate that decreases much more rapidly. It even appears by the 1000th hand that the Thermometer's rate of policy change is still decreasing.

It would seem that near the end with the Thermometer based networks averaging around one decision change per hand that they must be somewhat unstable. As seen in the Figure 5.3 though, the capability is relatively stable. This suggests that changes made are just repeated flip flops on hard to decided states, probably those sums in the range of 13 to 16.

## 5.8 Increasing State Information

The state is now expanded to include information about the dealer's face up card to see if it can be properly utilized. Both the Table and Thermometer encodings will be used as before except 10 additional input units are needed to represent the up card forming a 28-2 topology as follows:

Figure 5.4: Policy change rate of various neural network configurations versus experience (total of hands played). The two curves displayed resulted from using Table and Thermometer encodings with TDX mode and rate 0.05.

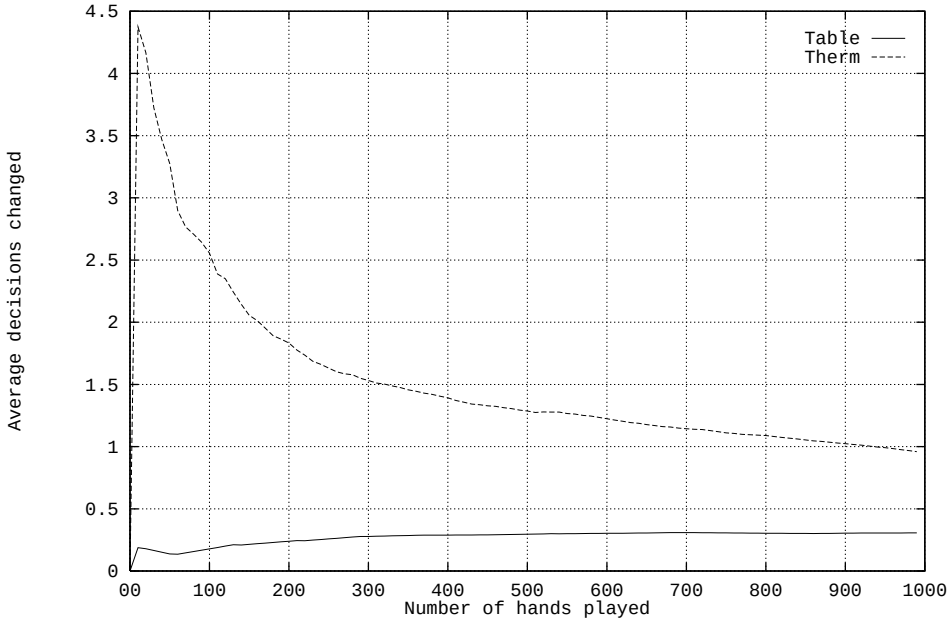


Table 5.6: Additional learning trials in which the dealer's face up card is considered. The Hoyle policy is included for comparison.

| Mode | Test Win% | Overall Win% Avg | Max | Min |
|---|---|---|---|---|
| Table | 40.2 | 41.0 | 42.6 | 38.5 |
| Thermometer | 40.4 | 41.4 | 43.2 | 39.8 |
| Hoyle | 41.0 | 42.6 | 45.6 | 41.3 |

- For the Table encoding, the card will be represented by turning on one unit that corresponds to its value. For example, if the card were a *Jack*, unit 10 would be turned on.

- For the Thermometer encoding, the card is represented again by turning on a sequence of input units. In this case the *Jack* would be represented by turning on units 2 through 10. Unit 1 is not turned on since an *Ace* is really closer to a 10 than a 2. In this way, the order in which units are turned on is $(2, 3, 4, 5, 6, 7, 8, 9, 10, 1)$.

Since the state space has been expanded from size 18 to 180, it's probably fair to increase the number of hands from 1000 to 2000 to allow more states to be visited. This time only TDX mode is used—all the TD modes did seem equivalent—and the best performing learning rates: 0.5 for Table, 0.05 for Thermometer. A total of 8 trials were run and the results, as usual, averaged.

Compared to earlier results, the win percentages summarized in Table 5.6 have gained at least one point in the overall average scores. They still are a bit more than a point behind the 42.6% ability of Hoyle's policy. Note though that one more piece of information—whether you are counting an *Ace* as 11—is still not being encoded. Perhaps this could be the difference? Figure 5.5 is a picture of a final policy that shows that the dealer's card gets used in a fashion similar to Hoyle's strategy; when the face up card is low in value, the threshold at which to stand also gets lower.

Figure 5.5: A policy which uses the dealer's face up card. Each row represents the card's value and the columns represent the possible hand totals. "H" stands for hold and "s" stands for stand.

```
       1         2
  456789012345678901
A HHHHHHHHHHHHHHsssss
2 HHHHHHHHHHsssssssss
3 HHHHHHHHHHsssssssss
4 HHHHHHHHHHsssssssss
5 HHHHHHHHHHssssssssss
6 HHHHHHHHHHsssssssss
7 HHHHHHHHHHHsssssss
8 HHHHHHHHHHHHssssss
9 HHHHHHHHHHHHssssss
10 HHHHHHHHHHHHHsssss
```

# Chapter 6

# Tic-Tac-Toe

Tic-Tac-Toe is a 2-player game of perfect information without chance moves. Its length is short—9 moves at most—and game tree small making it easily conquered by an exhaustive search program. Still, the number of unique board positions that can occur during play is 6045, meaning a table lookup scheme would require that many inputs. Here, a neural network's ability to generalize will help out immensely with the largest network topology in this section consisting of 600 weights (its likely that many of these weights are unnecessary). The set of objectives for this section are:

- Understand issues of state space exploration. In particular, can useful skills be learned from self play? Can skills be learn from success and failures of opponents? Does noise help in Tic-Tac-Toe?
- Observe differences when the network gets to see just the raw board state or computed features.

In all cases, the neural player will be using only a 1-ply search, while some opponents used for benchmarking will play perfectly with full 9-plies of search.

## 6.1 Tic-Tac-Toe Rules

There are two players, one is $X$ and the other is $O$. The board consists of a 3 by 3 grid (9 squares) in which the players alternate taking turns writing their markers, $X$ or $O$, into unoccupied squares. $X$ always takes the first

turn. The object is to get three markers in a line along either a row, column, or diagonal. The first player to succeed at this wins. If all the squares are filled without "three in a row" the *Cat* is declared the winner (a draw).

## 6.2 Performance Metrics

The prime metric will be capability as measured by win, loss, and draw percentages. An overall score is also computed as (wins - loses)/games which ranges from 1.000 (always wins) to -1.000 (always loses).

After a network has under gone a learning trial, it is pitted against opponents with different skill levels for 100 games—half played as $X$ and half played as $O$—to observe how it fairs against different styles of play. Unlike Blackjack, Tic-Tac-Toe's state transitions are deterministic so that the variance problem is greatly reduced. In fact, the only variance that occurs is due to any random actions on the part of an opponent. This means that 100 games should provide a fairly accurate measure of capability.

### Benchmarks

The fixed policy players used to benchmark the policies learned by the neural networks can perform an alpha-beta search to any tree depth thus providing bad, intermediate, and expert opponents. Two separate decision policies are used:

- *Randy* — Selects valid square at random.
- *Huey* — Uses a static evaluator based upon the following heuristic: Any move which results in "three in a row" is immediately evaluated as 1. Otherwise each potential line (row, column, and diagonals) is examined for friendly or enemy singlets and doublets. A singlet is a line with only one marker in it; a doublet is a line with only two markers of the same player in it. An evaluation is computed for the board position as

$$E = \begin{cases} -1 & D_e > 0 \\ \frac{2D_f + S_f - S_e}{6} & \text{otherwise} \end{cases}$$

  where $S_f, D_f, S_e$, and $D_e$ are the number of friendly singlets, friendly doublets, enemy singlets, and enemy doublets respectively. If several

Table 6.1: The capabilities of three fixed policy Tic-Tac-Toe players against each other: Randy(1), Huey(1), and Randy(9).

| | | Randy(1) | | | Huey(1) | | | Randy(9) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Score | W | L | C | W | L | C | W | L | C |
| Randy(1) | -.573 | 53 | 43 | 4 | | | | | | |
| Huey(1) | .257 | 92 | 3 | 5 | 0 | 0 | 100 | | | |
| Randy(9) | .350 | 93 | 0 | 7 | 12 | 0 | 88 | 0 | 0 | 100 |

actions happen to produce equivalent evaluations, one is selected at random. It should be noted that this heuristic is beatable if the ply is small enough.

Both Randy and Huey look ahead to any desired depth where the plys of search will be notated in parentheses. For example, Huey(3) means use the heuristic evaluator with 3 plys of search. Either policy with a 9-ply search plays perfectly by exhaustively searching the game tree.

To represent poor, intermediate, and expert players, Randy(1), Huey(1), and Randy(9) have been selected. The performance of each against the others over 100 games is summarized in Table 6.1. Note that the games in which Randy(1) plays itself tend to end in wins instead of draws. This is due to the fact that all players are provided with a function that recognizes winning and legal moves, including the upcoming neural players. While this prevents the neural nets from having to learn the rules, it also allows a random player to make a winning move if there is one.

## 6.3 Architecture Setup

The components for the Tic-Tac-Toe learning system are:

- An *environment* consisting of a Tic-Tic-Toe board and some opponent.
- A *state* listing the positions of each player's $X$'s and $O$'s on the board. Two different encodings of the state are tried, one using only the raw board and another which makes use of higher level features.

- The valid *actions* are to place a marker into any unoccupied square.
- A *utility* provided at the game's end is one of win (+1), lose (-1), or draw (0).
- A *neural player* using indirect action selection form $P(s_t)$ (resulting states from each legal action are evaluated). Two sigmoid outputs are produced, $P_{win}$ and $P_{lose}$ where each predicts the likelihood of the game ending in a win or a loss respectively[1]. These two values are then combined into a single score as $P_{win} - P_{lose}$ providing an evaluation from -1 to +1. This use of separate outputs to represent different game outcomes was inspired by Tesauro's backgammon player (Tesauro, 1991).

## Encoding the State

Most of the networks trained in this section encode the state into a set of 8 features related to the singlet and doublet idea used by Huey. Each feature is the condition of a line on the board—row, column, or diagonal—where the possible conditions are *empty, friendly singlet, friendly doublet, enemy singlet, enemy doublet,* and *mixed*. Empty simply means all squares on the line are open while mixed means that there are both $X$'s and $O$'s. These 6 states are then encoded by 5 input units as shown in Table 6.2. The feature is computed for each potential line (3 rows, 3 columns, and 2 diagonals) resulting in a grand total of 40 inputs.

The alternate raw encoding is quite simple. Each of the 9 squares has two inputs assigned to it: one for the presence of an $X$ the other for an $O$ yielding a total of 18 inputs. A bi-polar encoding may also work as used by Walker in (Walker, 1992) although more compact encodings tend to increase learning time.

[1]Earlier attempts with Tic-Tac-Toe used a network with a single output $P_{win}$. This caused problems when deciding what utility to assign a draw. For example, if a game ending in a draw was assigned a utility of 0.5, the network was learning that it had a 50% chance of winning; this is very different from 100% chance of resulting in a draw. This makes a world of difference when playing against a perfect player for which the best that can be achieved in a draw.

Table 6.2: Encoding of line condition for Tic-Tac-Toe.

| condition | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| empty | 1 | 0 | 0 | 0 | 0 |
| friendly singlet | 0 | 1 | 0 | 0 | 0 |
| friendly doublet | 0 | 0 | 1 | 0 | 0 |
| enemy singlet | 0 | 0 | 0 | 1 | 0 |
| enemy doublet | 0 | 0 | 0 | 0 | 1 |
| mixed | 0 | 0 | 0 | 0 | 0 |

### Board Symmetry

Many of the arrangements of $X$'s and $O$'s on the Tic-Tac-Toe board turn out to be identical (isomorphic) if you rotate and reflect the board. By using this bit of information, the state space becomes smaller, allowing fewer training games to be played. A very simple way to take advantage of these isomorphisms is to train the network on each of the 8 possible board reflections/rotations, although the total number of weight updates will be as if 8 times more games were played.

## 6.4 Self Play

The first experiment finds out how well the neural player can learn through self play. Self play is very appealing since it requires very little knowledge to be built into the learning system, i.e. an existing playing policy is not needed. It is especially important in this case to use a random noise offset on the outputs to assist exploration of the state space; otherwise self consistent play would surely develop. Each training trial uses a network using TD(0) mode, a learning rate $\alpha = 0.4$, and a random noise offset = +/-0.1. The state is encoded using the line features resulting in a 40-2 network topology.

After 5 trials each of 500 games (chosen arbitrarily) one of the players learned how to never be beat, especially by the perfect player Randy(9). Although the others learned to play about as well as Huey(1), they failed to learn how to prevent losing when having to defend (play $O$) against him.

Table 6.3: Effect of mimicking a Tic-Tac-Toe opponent.

| | Score | | |
|---|---|---|---|
| Mimics Rand(9) | Avg | Max | Min |
| No | -0.306 | -0.050 | -.730 |
| Yes | -0.008 | 0.000 | -.040 |

None learned how to beat Huey(1) (although Randy(9) can do this about 12% of the time). It may very well be that additional training games with sufficient state space exploration might always yield better players through self play.

## 6.5 Mimicking The Opponent

Since Tic-Tac-Toe is a reflexive, perfect information game, something might be learned from the successes and failures of an opponent. As a result, overall learning time could be greatly reduced, especially if the opponent is initially the better player. Mimicking is not supervised learning of the opponent's evaluation function; this information is not available. Instead, the neural player acts as if it had also made it's opponent's moves, learning them in addition to it's own.

As a test, a neural player is setup to defend against Randy(9); the best that can be hoped for is a draw every game. For one set of trials, the moves of Randy(9) are visible in the training sequence; in another they are removed. This time 800 training games are played per trial to allow the non-mimicking network to see enough game states.

Of 5 trials without mimicking moves, none of the neural players learned to successfully defend against all of Randy(9)'s attacks, although 3 were able to draw over 90% of the time. In contrast, four player's that learned from Randy(9)'s play were undefeatable by the 800th game, and even the one that wasn't only lost 4% of the time (see Table 6.3). Clearly, mimicking helps and will be used during the rest of the Tic-Tac-Toe experiments.

Table 6.4: When neural players are trained against fixed strategy players with different skill levels, it's probably no surprise that those that sparred with better opponents typically learned how to be better players. The best and worst neural player's scores after training are presented to give an idea of variance.

.

| Trained Against | Which | Score | Fixed Policy Players: Randy(1) W | Randy(1) L | Randy(1) C | Huey(1) W | Huey(1) L | Huey(1) C | Randy(9) W | Randy(9) L | Randy(9) C |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Randy(1) | BEST | .227 | 75 | 4 | 21 | 21 | 0 | 79 | 0 | 24 | 76 |
| | WORST | -.160 | 81 | 10 | 9 | 0 | 50 | 50 | 0 | 69 | 31 |
| Huey(1) | BEST | .310 | 96 | 0 | 4 | 0 | 0 | 100 | 0 | 3 | 97 |
| | WORST | .170 | 89 | 5 | 6 | 0 | 0 | 100 | 0 | 38 | 62 |
| Randy(9) | BEST | .327 | 98 | 0 | 2 | 0 | 0 | 100 | 0 | 0 | 100 |
| | WORST | .127 | 93 | 0 | 5 | 0 | 50 | 50 | 0 | 4 | 96 |

## 6.6 Training Against Different Skill Levels

The whole point of this section is to see how much effect the skill level of the opponent has on the speed at which the game is learned. Intuitively it would seem that a better opponent would assist with training. How much though? Training a player against the 3 benchmark players—Randy(1), Huey(1), and Randy(9)—will help reveal this.

The setup is similar to previous cases: 500 games/trial, but now the players will switch sides every 5 games. Again, 5 training trials are run against each of Randy(1), Huey(1), and Randy(9). To test the resulting players, they are pitted against each of the benchmarks. The results of the best and worst players out of 5 are summarized in Table 6.4. It's really no surprise that those trained against Randy(9) did best overall with a median score of +.306 compared to +.287 and -.026 for those trained against Huey(1) and Randy(1). What is worth noting is that the players trained against the weaker opponents were able to surpass their adversaries. Once again, its an open question on how far each player could improve with continued training.

Table 6.5: Comparison of Tic-Tac-Toe scores with and without noise.

.

| Noise | Median Score When Trained Against | | |
|---|---|---|---|
| Offset | Randy(1) | Huey(1) | Randy(9) |
| 0.1 | -.026 | .287 | .306 |
| 0.0 | .160 | .250 | .127 |

## 6.7 Noise

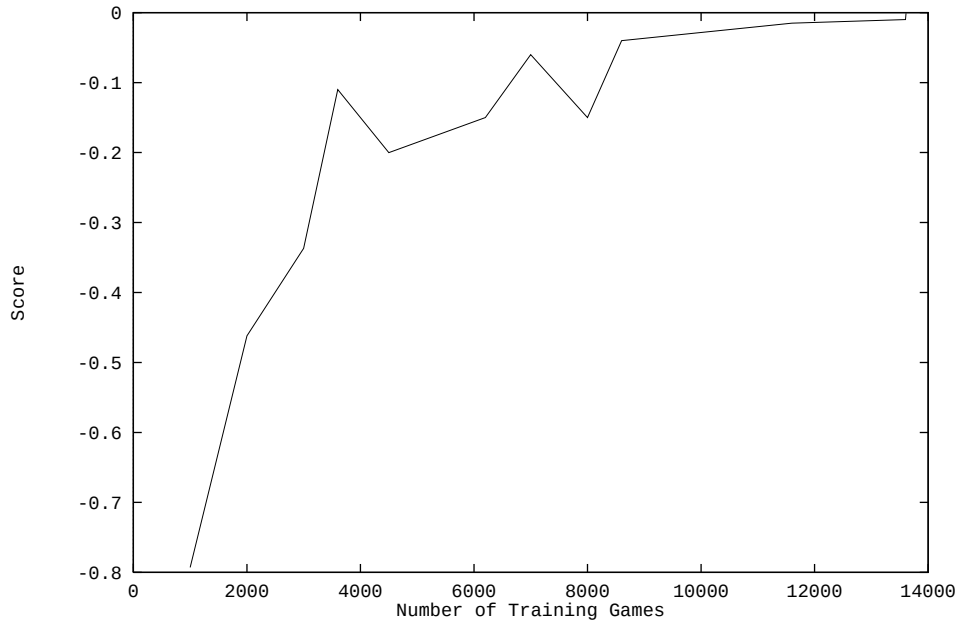
The next question is "How much does the random offset help with learning speed?" It has been assumed so far that learning to play Tic-Tac-Toe is more successful with state space exploration induced by noise. To find out, the learning trials of the previous section against Randy(1), Huey(1), and Randy(9)are rerun without a random disturbance of the network predictions. The median scores for each of 5 trials are compared in Table 6.5. In all cases, noise has a positive effect except for those nets trained against the random opponent. The reason may very well be that there is too much randomness, Randy(1)'s unpredictable moves plus the noise added to the prediction.

How noise helps has been observed in two ways. The first is that it separates predictions between good and bad moves. By occasionally being forced to revisit bad moves, their evaluations are punished multiple times, each bumping the prediction further from that of the correct move. Eventually the prediction gets far enough away that the random offset is no longer able to reach it. On the other hand, a move can get erroneously labeled as bad. By revisiting, it gets additional chances to show its worth.

## 6.8 Raw State Information

As a final experiment, the encoding of line features is removed leaving just raw board information. Initial trial and error has shown that a hidden layer is necessary for a network to perform this mapping, but how many hidden units is unknown. Thirty was chosen after initial attempts with more units seemed to take too long to converge; thus a 18-30-2 network topology was settled

Figure 6.1: A multi-layer neural Tic-Tac-Toe player learns from raw state information while competing against a perfect opponent, Randy(9). Score versus number of games played is plotted where the score was computed by playing 100 test games with learning off.




upon. Other parameters were set as follows: learning rate $\alpha = 0.1$, mode is TDX, random noise offset = +/-0.15, and the opponent was Randy(9).

This time the network is allowed to play a larger number of training games. Figure 6.1 is a graph of the neural player's score vs. the number of training games played. Again, since the opponent plays perfectly, the best score that can be achieved is zero. It should be noted that at about 3500 training games, the network reached a plateau in which it dipped above and below a score of -0.10 until game 8600. By dropping the learning rate to 0.05, noise offset to +/-0.1, and turning off the learning of isomorphic states, the score dropped smoothly for the next 5000 games to -0.01.

After a total of 13600 games, a state was discovered which was evaluated incorrectly leading to a loss while playing $O$. It appeared that because this state only showed up about once every 200 games, it wasn't getting the

attention it deserved. A remedy which worked quite well was to force the neural player to repeat the same bad game until it was "fixed". After 10 games, the problem was solved. Final testing over 1000 games resulted in just 1 loss and a score of -0.001.

Although Boyan's setup for Tic-Tac-Toe (Boyan, 1992) was slightly different, the number of training games to reach the best play took about the same time. In contrast, Kolen and Goel's (Kolen and Goel, 1991) Tic-Tac-Toe player learned to draw its opponent much more rapidly, usually by the 100th game. Apparently their heuristic player was deterministic, meaning, faced with the same board position, it selected the same move. This made the state space very small since once the network began mimicking the heuristic player, it probably began playing just two games over and over: one as $X$ and one as $O$.

# Chapter 7

# Conclusions

## 7.1 Results

Three different kinds of sequential decision tasks—Maze Mapping, Blackjack, and Tic-Tac-Toe—were learned by computer programs built upon a common architecture. In all cases, reasonable results were achieved with a training process that is mostly automatic process although configuring the neural network topology and selecting proper learning rates did required some trial and error.

While Blackjack appeared to be a simple game, its random nature presented certain difficulties. For one, it is hard to measure capability over anything but many, many hands. Fixed strategies used for comparison displayed variances of 4% when measured over 1000 hands. In addition, the amount of improvement that could be attained by the expert *Hoyle* strategy over the novice *Hold* strategy was just 5%. Still, the neural players were able to achieve capabilities that were consistently within 1% of the *Hoyle* strategy.

Neural players using just 1-ply of search also learned how to successfully compete in Tic-Tac-Toe against a program using exhaustive search. Even when beginning with zero builtin knowledge about the game, features needed to successfully defend against the perfect opponent were discovered (although taking somewhat more time).

## Temporal Difference Advantages

The temporal difference methods—TD(0) and its cousin TDX—clearly were superior to the TD(1) approach when many decisions had to be made before the final outcome was learned. For problems like Blackjack with only a few decisions per hand, the differences were minimal (kind of expected this). Except for a few limited cases in which TD(1) learning reached solutions quicker in the Maze Mapping experiment, TD(0) and TDX found shorter paths and were much more consistent at it. One of the more interesting outcomes was how well a temporal difference method performed when random transports took place in the maze; in contrast, the TD(1) approach never could complete the map in the allotted 20 trials.

TDX turned out to be a mode that can yield the results of TD(0) but in fewer trials. As the learning rate goes from 1.0 to 0.0, TDX's behavior smoothly transitions from that of TD(1) to TD(0)[1]. Because of this, rates somewhat less than 1.0 tend to work best.

## State Space Exploration

Adding a fixed random offset to predictions created by the networks helped to push the neural players into regions of the state space that would have otherwise been ignored; this was demonstrated in both the Maze Mapping problem and Tic-Tac-Toe. This simple approach worked by forcing the network to revisit moves which it already thought as bad, giving them a chance to redeem themselves. If the moves truly represented bad strategy they were gradually pushed beyond the range off the random offset, eventually creating a stable policy.

In a few cases though, noise hurt the process, showing how too much randomness can create a moving target. In other cases, not enough exploration seemed to be a problem. For example, in games against the fairly intelligent yet beatable opponent (*Huey*), the ability to take advantage of his weakness rarely evolved; perhaps more games would have helped.

The other technique which improved exploration was learning an opponent's actions in addition to own moves (mimicking). Even when the opponent made silly moves, the time to learn the task was greatly decreased. Of course this is of little for tasks without opponents such as solitaire.

[1]This is not entirely true when nonlinear activation functions are used.

## 7.2 Future

There is still much more to do, as usual. Probably the biggest question is "How well will it scale up?". The three problems tried so far seem quite trivial when compared to tasks like playing Chess or Go. Stepping up to such problems will likely present numerous other challenges. Since the time it takes for a neural network to evaluate a state may take much longer than simple feature computations, it is important that the worth of the network greatly outweights a simpler evaluator that can be used in deeper searches.

### Cascade Correlation Architecture

One of the problems with backpropagation is determining the correct topology. With small problems, trial and error can suffice; with complex mappings (larger state space, bizarre nonlinearities), trial and error leads to despair. Larger state spaces typically require more units which tends to paralyze the vanilla backpropagation learning rule.

The Cascade Correlation Architecture (CCA) (Fahlman and Lebiere, 1990) automatically finds "good" topologies by incrementally adding hidden units. In addition, since CCA adds one unit at a time and freezes its weights, there are less units adapting which greatly increases the ability to learning difficult mappings. The work that was begun to adapt the CCA for use in a temporal reinforcement should be continued to see if will help out in game learning as well.

### Automatic Parameter Control

Selecting and properly controlling learning parameters such as the noise offset and learning rate is another area deserving attention. Again, it would be nice to get away from guesswork. Ideally these parameters might be adjusted depending on the state of the player. As was seen, higher levels help when first learning a task, but lower levels are necessary later on to prevent the loss of good weights. For example, in Blackjack it turned out that most of the key decisions were rapidly learned in the first 100 hands. After this point, the player should have been allowed to evolved at a slower pace since policy changes can't accurately be measured over short periods. A lower or gradually decreasing learning rate should give this behavior.

## Improved State Space Exploration

Better ideas should be developed for directing state space exploration. If a novel situation can be detected (a rarely occurring state which leads to a loss, for example), information about it could be saved allowing the learning algorithm to spend more time on it (this was done manually for Tic-Tac-Toe). Also the use of a fixed random offset could probably be improved upon, possibly adjusted with some form of "boredom" factor (a detection of when the neural player's abilities have peaked).

## Alternative Learning Mechanisms

Backpropagation is just one of many mechanisms that could learn how to map states to predictions. Backprop variations, other neural network architectures, and statistical methods are all potential approaches that could be incorporated into the learning system architecture.

## Modularity

Another methods for improving playing capability is modularity. Instead of trying to pack all the the knowledge for a static evaluator into one monolithic network perhaps several neural player's policies could be combined in a voting system (which could be another neural network). Boyan (Boyan, 1992) also demonstrated that using separate neural networks for different phases of the game can be a tremendous help.

Even without augmenting the basic approach, it does appear that more complicated problems might be solved; Tesauro's success at Backgammon seems to indicate this. Initial work with an Othello playing program based upon the architecture is showing some early signs of success. After playing 1000 games against an opponent which uses a heuristic static evaluator, the neural player employing a 2-ply search has develop skills which equals that of the opponent at 5-plies.

# Bibliography

Barto, A. G., Sutton, R. S., and Watkins, C. J. C. H. (1989). Learning and sequential decision making. Technical Report COINS 89-95, Dept. of Computer and Information Science, University of Massachusetts, Amherst MA 01003. Available via FTP from archive.ohio-state.edu:/pub/neuroprose.

Boyan, J. A. (1992). Modular neural networks for learning. Master's thesis, University of Cambridge. Available via FTP from archive.ohio-state.edu:/pub/neuroprose.

Bramer, M. A., editor (1983). *Computer game-playing: Theory and practice.* Ellis Horwood Limited, Market Cross House, Cooper Street, Chichester, West Sussex, PO19 1EB, England.

Carpenter, G. and Grossberg, S. (1987). Invariant pattern recognition and recall by an attentive self-organizing ART architecture in a nonstationary world. In *Proceedings of the IEEE First International Conference on Neural Networks*, volume II, p 737.

Fahlman, S. E. (1988). An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

Fahlman, S. E. and Lebiere, C. (1990). The Cascade-Correlation learning architecture. Technical Report CMU-CS-90-100, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

Feldman, J. and Ballard, D. (1982). Connectionist models and their properties. *Cognitive Science*, *6*,205.

Hebb, D. (1949). *Organization of behavior*. Wiley, New York.

Hecht-Nielsen, R. (1989). *Neurocomputing*. Addison-Wesley Publishing Company, Inc.

Holland, J. H. (1983). Escaping brittleness. In *Proceedings of the International Machine Learning Workshop*, pp 92–95.

Hopfield, J. (1982). Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Academy of Sciences USA*, volume 79, pp 2554–2558.

Kohonen, T. (1981). Automatic formation of topological maps in a self-organizing system. In *Proceedings of the 2nd Scandinavian Conference on Image Analysis*, pp 214–220.

Kolen, J. F. and Goel, A. K. (1991). Learning in parallel distributed processing networks: Computational complexity and information content. *IEEE Transactions on Systems, Man, and Cybernetics*, *21*(2),359–367.

Lee, K.-F. and Mahajan, S. (1988). A pattern classification approach to evaluation function learning. *Artificial Intelligence*, *36*,1–25.

McCulloch, W. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 7,115–133.

McKinsey, J. C. (1952). *Introduction to the theory of games*. The RAND Series. McGraw-Hill Book Company, Inc.

Minsky, M. and Papert, S. (1969). *Perceptrons*. MIT Press, Cambridge.

Morehead, A. H. and Mott-Smith, G., editors (1963). *Hoyle's rules of games*. The New American Library, Inc., 1633 Broadway, New York, New York 10019.

Pearl, J. (1983). Game-searching theory: Survey of recent results. In Bramer, M. A., editor, *Computer Game-Playing: Theory and Practice*, chapter 20. Ellis Horwood Limited, Market Cross House, Cooper Street, Chichester, West Sussex, PO19 1EB, England.

Rosenblatt, F. (1962). *Principals of neurodynamics.* Spartan Books, Washington.

Rumelhart and McClelland, editors (1986). *Parallel distributed processing.* MIT Press.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representation by error propagation. In *Parallel Distributed Processing*, volume 1, chapter 8. MIT Press.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, *3*,210–229. Reprinted in E. A. Feigenbaum & J. Feldman (Eds.), Computers and thought. New York: McGraw-Hill.

Samuel, A. L. (1967). Some studies in machine learning using the game of checkers, II. *IBM Journal on Research and Development*, *11*,601–617.

Shannon, C. E. (1950). Programming a computer for playing chess. *Philosophy Magazine*, *41*,256–275.

Simpson, P. K. (1988). A review of artificial neural systems II: Paradigms, applications, and implementations. Technical Report MZ 7202-K, General Dynamics, Electronics Division, P.O. Box 85310, San Diego, CA 92138.

Sutton, R. S. (1984). *Temporal credit assignment in reinforcement learning.* PhD thesis, University of Massachusetts, Amherst.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*,9–44.

Tesauro, G. J. (1991). Practical issues in temporal difference learning. Technical Report RC 17223 (#76307), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598.

Tesauro, G. J. (1993). TD-gammon, a self-teaching backgammon program, achieves master-level play. Available via FTP from archive.ohio-state.edu:/pub/neuroprose.

Tesauro, G. J. and Sejnowski, T. J. (1989). A parallel network that learns to play backgammon. *Artificial Intelligence*, *39*(3).

Thrun, S. B. and Möller, K. (1991). On planning and exploration in non-discrete environments. Available via FTP from archive.ohio-state.edu:/pub/neuroprose.

Walker, S. (1992). Untitled undergraduate thesis. Copy received via direct communication with the author.

Werbos, P. J. (1989). Backpropagation and neurocontrol: A review and prospectus. In *Proceedings of the International Joint Conference on Neural Networks*, volume I, pp 209–216.

Widrow, B. and Hoff, M. (1960). Adaptive switching circuits. In *1960 IRE WESCON Convention Record*, volume part 4, pp 96–104.