REINFORCEMENT LEARNING IN GAME PLAY

by

Brian Powell

B.S., University of Colorado at Boulder, 1993

A thesis submitted to the

University of Colorado at Denver

in partial fulfillment

of the requirements for the degree of

Master of Science

Computer Science

2000

This thesis for the Master of Science

degree by

Brian Powell

has been approved

by

Ross McConnell

Christopher Smith

William J. Wolfe

Date

Powell, Brian (M.S., Computer Science)

"Reinforcement Learning in Game Play"

Thesis directed by Professor William J. Wolfe

ABSTRACT

Machine Learning has always paralleled biologic learning research, and, more recently, that of reinforcement learning. Reinforcement learning breaks from standard supervised learning by allowing the agent to explore its environment to better understand the surrounding. This exploration of the environment and exploitation of previous knowledge is particularly well suited for stochastic game environments in which a supervisor could not possibly teach all achievable states.

This abstract accurately represents the content of the candidate's thesis. I recommend its publication.

Signed ________________________________
William J. Wolfe

DEDICATION

To Lisa

ACKNOWLEDGEMENT

My gracious thanks to the dedicated people at the University of Colorado system for inspiring my interest in learning methods for machine learning. I also would like to thank the dogs and cat not only for all of their sleeping during the writing of this document, but also for showing me that basic intelligence is much more complex than we give credit for.

CONTENTS

Appendix

FIGURES

Figure

TABLES

Table

CODE LISTINGS

Code

1. Introduction: Concept of Learning

The paradox of understanding learning begins with the seemingly indefinable concept itself. The dictionary reveals that learning is to “1. gain knowledge or skill 2. acquire as a habit or attitude” [Websters, 1982]. These definitions and concepts are rather vague as is the current understanding of learning. This has led many to argue that the search for the laws of learning is a never-ending task because they simply do not exist. Exploring the philosophical implications within the definition of learning–though a fascinating Platonic discourse–is not the author’s intent.

For the sake of this discussion, learning shall be defined as merely an adaptive change in behavior. Using this simple definition, intelligence can easily be assigned. Consider that “behavior is usually considered intelligent when it can be seen as adaptive” [Balkenius, 1994]. Superior intelligence can be assigned when the adaptive behavior improves our domain knowledge. Inferior intelligence can be assigned when the adaptive behavior worsens our domain knowledge.

Balkenius has an excellent example of perceived intelligence [Balkenius, 1994]. A squirrel gathering, hiding, and storing food for the winter would appear to be an intelligent activity. Upon further inspection we find that the squirrel forgets where 80% of his food is hidden, stores far more food than would ever be needed, and most oftentimes takes food hidden by another squirrel. No longer does this activity seem intelligent, but, simply random. Such counter-examples exemplify the notion that “intelligence is in the eye of the beholder” [Brooks, 1991].

2. Biologic and Artificial Intelligence

Artificial Intelligence, throughout its history, has followed the philosophical and theoretical trends in the biological learning fields rather closely. The fundamental difference between the studies of animal intelligence and that of artificial intelligence is that the field of artificial intelligence works to build new intelligent entities utilizing the knowledge learned. These entities are built for research as well as applications, such as assisting in the diagnosis of medical patients, controlling robotics within factories, guiding robotic rovers on other planets, and even playing a challenging game against a human opponent.

Alan Turing defined intelligent behavior as the machine having the ability to perform all human cognitive functions. This, rather homo-centric, view of intelligence is not the goal of much of artificial intelligence because most believe that intelligence does not have to achieve the level of human thought to be considered truly intelligent.

2.1. Biological Learning

The roots in the understanding of biological learning followed the belief that all behaviors could be simplified into stimulus and response pairs. However, over one hundred years of research has yet to form the general rule of learning based upon these stimulus-response pairs. Other researchers, such as MacFarlane in 1930, set out to prove that learning was more abstract [MacFarlane, 1930]. He taught rats to swim their way through a maze in order to find the goal of a food treat. After the rats were adept at swimming the maze and discovering their food every time, the maze was drained of water and the rats were allowed to again run the maze in search of food. Every rat was able to recall the proper path through the maze even though it was no longer swimming but running. He had proven that the knowledge learned by the rats was not merely a response to the stimulus of swimming. Through this research, MacFarlane and others showed that learning is not simply a complex memorization of stimulus-response pairs.

Other biological responses or actions are not learned at all, but are instictual: handed down from generation to generation by genetic code. For this discussion, instinct is defined as any *fixed* reaction to a given stimulus (e.g. a "tick will bite everything that has a temperature above +37°C and smells of butyric acid" [Lorenz, 1977]). Using this definition of instinct gives rise to the belief that learning may be nothing more than slight parameter adjustment of pre-programmed motor reactions. An example of this theory can be displayed with a newborn child. The child will smile at anything resembling a human face (from true faces to masks and cardboard cutouts). As the child develops, however, these visual cues must begin to become more and more resembling that of a true human face to elicit the child's response.

Researchers from each of these two theories worked for years to prove their beliefs and attempt to disprove the other side; however, as research continued, it led to the realization that there is more than one type of learning and that neither is mutually exclusive of the other. This has led to the now more widely held belief that there is only one general learning system comprised of several (possibly many) learning methods.

As attention turned from attempting to fit all learning into stimulus-response or adaptive instinct, researchers began to focus on the role of reinforcement learning in the 1950's. During this time, it was found that reinforcement learning is based upon causing adaptive behavior with stimulus reinforcement: a rat that learns the maze to be rewarded with food is simply being given a stimulus to learn. Some argue that this stimulus reinforcement (or punishment) forces the actual learning (as opposed to adaptation). Premack showed, however, that oftentimes, this stimulus is not so much as forced learning, but that the stimulus serves only to "reinforce a less probable activity." For example, the rats that swam the maze, though, not necessarily hungry learned to swim the maze not out of need or of the capacity to learn, they swam the maze simply to get their treat.

The reinforcement learning theory is quite different from the mechanical view of stimulus-response and is more generalized. Some argue that reinforcement learning is, in fact, the oft-sought single general learning system. Most researchers, however, would argue that reinforcement learning "plays a role in some but not all learning" [Balkenius, 1994].

To understand the forms of learning is an important step in the eventual understanding of how learning as a whole occurs in biological organisms. Though many theories are (or have been) presented separately by researchers, taken as a whole they can help us to understand the rather abstract learning process that occurs.

2.2. Artificial Intelligence

It is interesting how development of general artificial intelligence theories tend to closely follow the their respective theories and development of researchers in biological learning. All primary threads of AI research can be mapped into one of the three learning methods understood by biologic researchers.

Stimulus-Response actions are very much common place in rule based systems. Simple look-up tables or expert system rules dictate a response or chain of responses for a given stimulus or stimuli-chain respectively. Neural Networks, on the other hand, theoretically attempt to model biological brains (utilizing a collection of synaptic nodes) such as a Hopfield neural network (once it has been suitably trained) tend to follow the instinctual model of parameter adjustment to a pre-programmed set of motor-controls and responses. Genetic Algorithms—in general—also fit into this parameter adjustment of instinctual responses.

Thorndike defines *law of effect* to mean, “learning of a response is governed by the effects of that response.” The first to follow this belief in an Artificial Intelligence representation and implementation was Arthur Samuel with his Checkers Program. Sutton and Barto [Sutton, 1990] have explicitly made mention of the parallel between the behavioral research in reinforcement learning and the temporal difference algorithm they have developed and deployed in adaptive control.

2.3. Machine Learning

Machine Learning is the concept of adapting the theories of Artificial Intelligence to that of a physical (typically discrete) machine. A broad definition for machine learning can be given as, “a machine learns whenever it changes it structure, program, or data (based on its inputs or in response to external information) in such a manner that its expected future performance increases” [Nilsson, 1996]. In addition to the research of animal learning, other fields have contributed philosophy and research to machine learning (as shown in table 2.1).

Subject	Contribution
Statistics	Common statistical problems attempt to resolve the value of an unknown function at some point given some sampling of previous points. This is considered because "decisions" are made only with data from the problem domain.
Brain Models	One important discipline of machine learning is that of Neural Networks which attempt to model or mimic the basic structure and workings of biological brains.
Control Theory	Adaptive controls attempt to control some process in an environment with unknown parameters that must be estimated (typically in real time) during the span of the process.
Psychological Models	Much of the work performed by Sutton and Barto in the realm of Reinforcement Learning is based upon trying to model how rewards encourage learning. "Reinforcement Learning is an important theme in machine learning research" [Nilsson, 1996].
Evolutionary Models	Genetic Algorithms attempt to model the evolutionary change as a species tries to "adapt" to the problem domain(s).

Table 2.1, Disciplines Contributing to Machine Learning

Within the application of Machine Learning, there are three classifications of the type of learning taking place (as shown in table 2.2).

Classification	Description
Supervised Learning	The expected values, f, are known over n samples for the training set, S. The learning model can then be "taught" to adjust itself until it produces a result consistent with f. This is typically applied to Hopfield Neural Networks.
Unsupervised Learning	For this case, only the training set, S, is known for which, the expected values, f are not known. In this case, the learning model must learn to typically maximize f for the set S.
Hybrid (or intermediate) Learning	The learning model is shown subsets of the training set, S, in both supervised and unsupervised learning.

Table 2.2, Learning Type Classifications

Utilizing the philosophies of learning, incorporating the mathematics of statistics, the process of control-theory and utilizing a choice of learning method, what can the machine actually learn? Though there are many applications in use of such ideas and technologies, there are a typical variety of structures that machines can learn (as listed, [Nilsson, 1996]):

- Functions
- Logic Problems and Rule Sets
- Finite State Machines
- Grammars
- Problem Solving Systems

It is the problem of teaching a machine to best fit (or approximate) a non-linear function that most of our discussion will center.

2.4. Role of Game Playing in Machine Learning

Game playing has been a popular research topic in Artificial Intelligence and Machine Learning because the problem typically has well-defined goals, is easily represented, and has strict-rules. This is not to say that game play is easy. The computer must deal with the uncertainty of its opponent while also managing a search space, which is oftentimes infinite (in the realm of computer time and memory). Finally, once a computer has excelled at a challenging game against strong human components, others typically perceive it as exuding intelligence.

The first groundbreaking machine playable game was Samuel’s Checkers Program. His program “learned” a value-function (as represented by his linear function approximator) and applied a training function that is very similar to what became temporal difference within reinforcement learning. Arguably, the most successful game-playing machine is

Tesauro’s TD-Gammon, which fully employs the methods of temporal difference to achieve a master’s level skill at the game of backgammon.

Samuel chose the game of checkers because he argued that checkers allows an emphasis on learning techniques without complex rules to steer the concentration of the learning method. The characteristics with which he believed checkers displayed intelligence by a machine were:

- Non-deterministic

  No known algorithm guarantees a win or a draw because the state space is too large to exhaustively search.
- Definite Goal

  Winning the game, with intermediate goals along the way.
- Activity rules are definite and known

  All responses must be allowed by the rules, thus keeping the possible space much smaller.
- A background of knowledge exists

  This is important for supervised learning. If you can teach the program what is good and bad (with known inputs and outcomes), your learning will be more accurate.
- Activity is understood by many people

  By playing the game of checkers, even a child would be able to understand exactly what the computer has learned and achieved.

To this day in research, games continue to attract tremendous attention because the ability to plan, reason, and choose an option is perceived as intelligent. Additionally, these exact skills are extremely well suited for translation to develop “real-world” applications.

3. Samuel's Checkers Program

The game of Checkers (also referred to as Draughts) was originally developed in ancient Egypt. It is a basic game played between two players and does not involve an element of chance. The game is played using only the black tiles of a standard chessboard as shown in figure 3.1. Each opponent begins with twelve pieces. Each piece may move only forward to its diagonally bordering, unoccupied tile. If an opponent's piece borders a player's piece and the opposite tile is unoccupied, a player may "jump" the opponent's piece and therefore remove it from the game. If a player's piece reaches the opposite side of the board, it is crowned a "king" and is allowed to traverse in either forward or backward direction. The player that takes the entirety of the opponent's pieces wins.

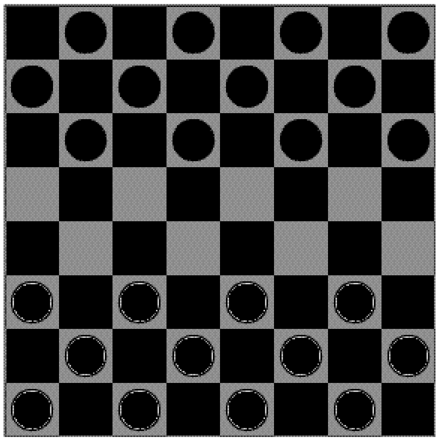

Figure 3.1, Checkers Board

All popular computer checkers programs employ the use of a search tree. In each of these trees, nodes represent possible positions within the game and branches represent possible moves from each position. Each level of the tree represents a position with possible moves based upon the opponent's move. For the game of checkers, to construct the entire game tree would require 10^{40} non-terminal nodes. Assuming one had a rather fast computer, capable of solving $3x10^9\ {}^{nodes}\!/\!{}_{\mathrm{sec}}$, it would require 10^{21} centuries to find all possible positions.

3.1. Samuel and Checkers

Arthur L. Samuel, a researcher at IBM began work in 1952 on developing a program which could play checkers at the level of a human opponent. By 1955, Samuel had developed a program employing the first heuristic search method to play the game of checkers. Samuel's Checkers Program was developed to run upon an IBM 700. He would oftentimes sneak into the production facility after-hours to run his program, as IBM would not give him computer time.

Samuel laid out the checkers board representing each possible location with a single bit within the 36-bit word of the IBM 700. Starting at the lower, right position numbered as 1, the next position left 2, and so forth for all possible 32 positions on a checkers board. Each player was then assigned two words containing that player's current position. The first word contained the location of all of the player's standard pieces. The second word contained the location of all of the player's "kings." To make a move, a simple bit-shift could be performed upon the bit representing the piece to move. To move forward and to the right, 4 right bit shifts would move the proper checker. To move forward and left, 5 right bit shifts. Samuel further created one additional word for all occupied spaces on the board. These five words sufficiently describe the state-space of the checker's game.

An evaluation function to judge where the computer (and opponent) stand within the game had to be created. The linear polynomial model that Samuel employed is based upon the summation of weighted parameters. Such parameters include: piece advantage (number of pieces vs. opponent's), simple boolean tests (do I have a "king"?), and others. The selections of these parameters were initially set *a priori*. Later, the computer was allowed to use those parameters with the most influence on the outcome of the game.

3.2. Searching the Space

With a representation and an evaluation function, a search tree is setup for each move and its possible scores. Employing a simple Depth-First Search (DFS) when searching for the best move, the machine cannot simply choose the move with the maximum score (our assumed goal), instead, from that position, it must traverse back up the tree to ensure that the goal chosen is attainable. The uncertainty in the attainability of this goal is due to the opponent's moves (which are presumed to maximize his own position while minimizing our position). This procedure, developed by Shannon came to be known as the *MINIMAX* search [Shannon, 1950].

The *MINIMAX* search examines each of our possible moves. For each of those moves, it again applies the evaluation function to each of the possible moves made by our opponent after our potential move. Choosing the node with the minimum score, we traverse back up to the previous node and assign it the minimum value. Thus, traversing back to our root node, we choose the branch with the maximum score. This is the maximum score we choose that would minimize our opponent's score.

For example, in figure 3.2, Player 1 may choose to move piece 1 forward left, forward right, or piece 2 forward right.

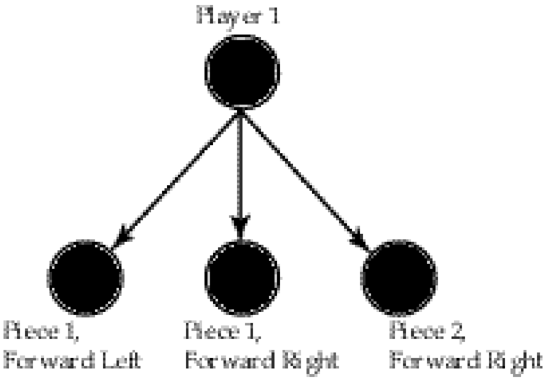

Figure 3.2, Player 1 Possible Moves

For each move Player 1 makes, there are a resultant number of moves that are available to Player 2. For every possible move available to Player 2, we employ our evaluation function to determine the "score" of Player 2 at each of these potential points in the game as shown in figure 3.3.

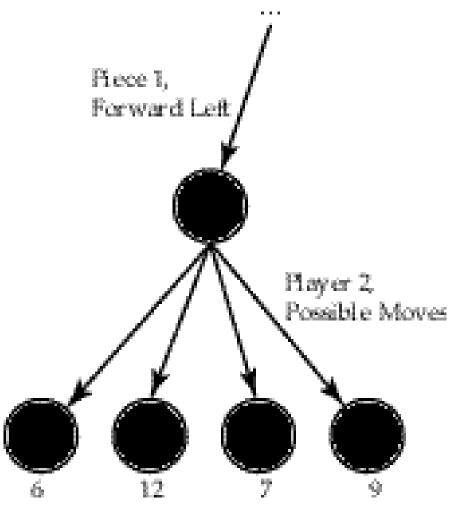

Figure 3.3, Player 2 Moves and Scores

Thusly, we choose the minimum score on each branch for Player 2 and assign the parent's node that minimum value as shown in figure 3.4.

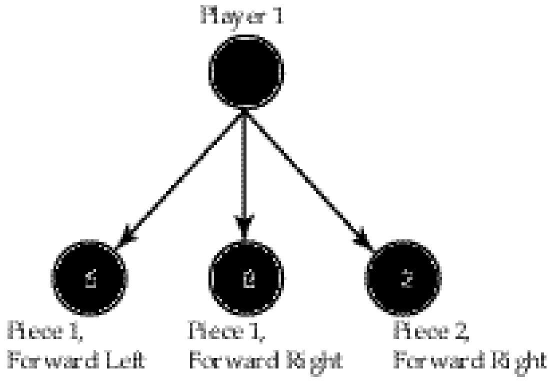

Figure 3.4, Potential scores of each move

We can then evaluate our potential moves based upon the minimum score that is achievable by our opponent. We maximize this score and make our move. In this case, Player 1 chooses to move piece 1 forward right because it will maximize his position while minimizing his opponent's maximum position. This example proceeds only two levels (or ply's) deep; however, the process is repeated as the search continues descending the tree.

Employing *MINIMAX* search (which utilizes DFS) would require a tremendous amount of processing capability, in fact, an upper bound of $O\left(B^d\right)$ (where *B* represents branches and *d* the depth of the tree) nodes must be searched. Samuel applied what came to be known as *Alpha-Beta* pruning of the *MINIMAX* search tree.

In the earlier example of *MINIMAX*, Player 1 first examined the Piece 1, forward left move. The best score (which minimized his opponent's move) achieved by his move was six. Examining the Piece 1, forward right move returned a result of eight, which was better. In the final case of Piece 2, forward right, the best score is two, worse than eight; therefore, this branch may be "pruned" and eliminated from searching further. This is the concept of α–β pruning.

As we descend the search tree, we set α to be the best value for our score and β to be the best (minimum value) that we can force upon our opponent. While searching the tree, if any branch is found to be worse than the current α or β value, we can back up to our parent and move onto the next branch thus eliminating an entire branch from our search tree.

This pruning of our branches eliminates much of the unneeded time involved in searching and gives us $O\left(B^{\frac{d}{2}}\right)$ nodes that will be searched. This allows us to double the amount of look-ahead in the same time allotment as the standard *MINIMAX* procedure.

Another modification was made to the search method to assist in the time that is required by DFS. The problem with DFS is that the order in which the offspring are examined can prove to be crucial. Unfortunately, α–β pruning will not assist the search in the case in which the best result lies upon the last branch searched. By searching a child with a better score first, the results will be found quicker, allowing α–β pruning to trim out the results. Samuel ordered each offspring in the search employing what he titled the "plausibility analysis." The search scored each of the children nodes with the evaluation function and then ordered each of these nodes. The search would then return to the parent node and use the α–β pruning with respect to these ordered, calculated child nodes.

Because of the search space and time restrictions placed upon Samuel with such an early computer, the search tree is further limited in the number of ply to search forward. The ply limitation algorithm he developed followed six basic rules:

1) Always search a minimum of 3 ply.
2) The search stops at ply 3 unless any of the following occur:
   a) The next move is a jump.
   b) The last move was a jump.

c) An exchange of pieces is possible.

3) The search stops at ply 4 unless 2a or 2c are then met.

4) Continue searching unless at any ply no jump is offered.

5) The search stops at ply 11 if one side is ahead by more than two “kings.”

6) The search stops at ply 20 regardless of conditions

By limiting the ply levels available for searching and employing the α–β pruning, the program is required to search a path that follows a more winning (direct) route rather than allowing the machine to explore paths which are known to lead away from the goal of winning.

By utilizing this search method, the computer would be capable of calculating a game of checkers; however, it would not “learn” about strategy, moves, etc. It would simply try to maximize its score while minimizing its opponent’s. Any average human opponent could develop a basic strategy around this simple, fixed game play.

3.3. Methods of Learning

Samuel began work on developing a system to allow the machine to adapt to mistakes and strategies presented to it during game play. The first component of this method, *rote-learning*, was a very basic system to store results of previous games. After failing to get substantial results, Samuel developed a second algorithm he called, *learning by generalization*.

In *rote learning*, the objective is to allow the computer the ability to search much further into its search tree. This is accomplished by storing the best move found every time the DFS α–β pruning search is employed. During play, if the same move is encountered again, the result is already computed, allowing the computer to begin searching deeper into the tree. If a particular move was found during a 3 ply search, the second time that node is encountered, the computer can begin its search at the third ply, thus enabling it to search much deeper into the tree. Every time the same move was encountered, it was "rewarded" by increasing its value slightly to encourage further exploration.

The problem Samuel found with this method is that no sense of direction was given to the search routine. The machine did not recognize that other paths (though equal or lower in ply-value) may lead to a quicker victory. In order to try to compensate for this problem, each calculated score was weighted based upon the ply-value required to achieve the score. Thus, if the machine is losing, it will take the path that maximizes the number of moves, and if winning, it will take the path with fewest moves.

Samuel played this algorithm against himself, human players of varying abilities, itself, and book games. During all of these games, the computer stored 53,000 positions. Samuel found that the machine had very strong opening and closing games (where the number of possible positions and moves is much lower) and that the middle game was extremely weak. He estimates that over one million positions would have to be stored in order for the machine to have a strong middle game.

Samuel began work on a more adaptive system by focusing not on the mechanics of searching but on the evaluation function itself. The evaluation function is the most important component of the game play and it is in combination with the *rote-learning* that Samuel achieved his legendary results. The evaluation function must quickly and accurately measure the current state of the game while also giving a reasonable interpretation toward the possibility for the position to "win the game." Samuel developed a system using a first-order linear polynomial.

Samuel's *learning by generalization* "learns" by adjusting the weights of various chosen parameters, which are used to calculate the current position's score. This evaluation function was a basic linear polynomial of the form shown in equation 3.1.

$$v = \omega_1 p_1 + \omega_2 p_2 + \ldots + \omega_n p_n \qquad (3.1)$$

These parameters are selected for the game of checkers and are set up *a priori* to the execution of the game. Samuel deemed thirty-eight different parameters to be important to the outcome of the game. At any single calculation, sixteen were used by the linear polynomial while twenty-two were in reserve. The machine arbitrarily chose these sixteen parameters along with their weights at the initial stage. After each cycle, if a particular parameter's weighting coefficient fell below the threshold, its counter would be increased. Once a parameter's counter reached a set maximum (Samuel chose eight), that parameter would be taken from the polynomial and placed at the end of the reserve queue. The top parameter in the reserve queue was then removed and added into the polynomial.

A sampling of the parameters that Samuel applied to p_i include:

- ADV

  Advancement of the piece
- EXCH

  Exchange of pieces
- MOB

  Mobility of the piece
- THRET

  Currently a threat to an opponent's piece or pieces. (If p_i is a THRET for some i, assume Q leads to position R; therefore, p_i is equal to the value of THRET for R minus the value of THRET for Q.)

Although the use of an *a priori* list of parameters is somewhat simplistic, at the time, no other method for computer-generated terms had been found. Additionally, due to the underlying complexity of the strategy of checkers, no one knows which parameters create the minimum group required to play a winning game. Finally, a fraction of the parameters created for representations by Samuel were combined into a single non-linear parameter used within the linear evaluation function.

To enforce the learning, Samuel played the computer against itself. The first player, called α, would adjust the weights during each turn. The second player, β, would leave the weights untouched; however, it used the same evaluation function that was being modified by α. During play, α would perform its standard look-ahead calculations as performed by the α–β pruning *MINIMAX*. This value would be stored and the move made. On the next turn, α would use the evaluation function on its current position and compare this with the stored value. If the difference were positive, the weights of the polynomial would be adjusted.

Samuel found that these calculations and adjustments had to be made after each move; otherwise, instability would become prevalent. This instability was due to the fact that the determination of the opponent's move is not made with the scoring polynomial; however, the anticipation of the opponent's move is calculated with the scoring polynomial during *MINIMAX*. Samuel including another means to help stabilize the polynomial by only updating the parameter weights if the delta between the stored value and the new value was above a minimum threshold. This minimum was recomputed at each cycle and was set to the average value of the coefficients for the terms of the utilized parameters.

The final stabilizing feature (and one which becomes key in reinforcement learning) was to scale the weight based upon the number of cycles that the parameter has been used. This aided in keeping stability when a parameter term is first used (and thus, possibly a large delta) and after it has been in used for many iterations. The correction value is computed as shown in equation 3.2.

$$C_n = C_{n-1} - \frac{C_{n-1} \pm 1}{n} \tag{3.2}$$

This correction value forces those weights that have been in use longer to be worth more than those that have recently been added.

Samuel's game as described became a good starting point in solving checkers. When the machine would compete against itself (using the α - β pairing as described), α changed the top of its parameter list 14 times and lost 4 of 7 matches. Throughout the trials, α continued this pattern and repeatedly played quite poorly. After approximately twenty-five games, the machine would begin to play at a "better-than-average" level; however, the learning at that point became "erratic and none too stable" [Samuel, 1959].

The program was also fooled by terrible play of the opponent. It seemed to be learning to play like its opponent as opposed to learning to beat it. Samuel found that changing the weighting coefficients less drastically when the delta was positive as compared to when it was negative helped to stabilize the program. In a positive delta case, only parameters that contributed negative terms in the positive scoring polynomial were updated. In the negative scoring polynomial case, only parameters that contributed positively were updated.

The program was found to be adding and removing parameters too frequently and it caused very unstable evaluations. Samuel increased the removal count threshold from eight to thirty-two to try and keep parameters more stable.

The program did not assign proper credit to the moves it had made. If a spectacular, high scoring move was performed, credit was given to the previous move as opposed to the series of moves that laid the groundwork for the spectacular move. Samuel never found a precise method for correcting the incorrectly assigned credit; however, the fix applied to the program allowed the remembered moves to increase until the delta exceeded its arbitrary, minimum value and then apply the update to that historical position. This, however, may assign credit falsely to a move along the chain.

The final instability found lied within the way the α–β pairing was played. If α played poorly and worsened its parameters, but, simply by chance, scored a victory over β, then in the next game, β would utilize these poor parameters and α would never improve its game play. Simply by only updating the parameters used by β if a minimum number of victories were achieved by α solved this problem.

It was with this system that Samuel performed further testing, playing book games against the computer, many games against itself, and games against humans. It eventually achieved a master level beating all but the expert level players. Samuel showed that the generalization scheme could be an effective learning method. The memory requirements for such as system were modest (even by 1955 standards) and operating times remained fixed and constant. Finally, even with an incomplete set of parameters, the computer was able to learn to play a very high level game of checkers.

Samuel's method of *learning by generalization* was one of the first applications of what is now referred to as Reinforcement Learning. Specifically, the method he developed is quite similar to the temporal difference learning employed within the field of Reinforcement Learning. Samuel assigned "rewards" for various parameters deemed as important to the game (by increasing their weight). Samuel's algorithm always attempted to increase its piece advantage; however, because the function was never given a reward to the generalization function, it could (and, in fact, did) get worse with time.

4. Reinforcement Learning

The entire concept of reinforcement learning is built upon the principal of learning through our interaction with a given environment. This idea utilizes the concepts observed in the study of biological learning. A rat that runs through a maze, learning its way around each turn until it finally reaches the destination, is rewarded with a treat, and freed from the maze is a primary example of the basic tenets of reinforcement learning. The two most important features of reinforcement learning are:

- Trial-and-Error Search:

  Discovery of actions within the environment that yield results.
- Delayed Reward:

  An action may affect the subsequent action or any subsequent rewards.

It is the trial-and-error search, which provides us with another key to reinforcement learning. The learner must "trade-off between exploration and exploitation" [Sutton, 1998]. The learner needs to explore new territories that it has never seen, but, also exploit the knowledge that it has already gained. To help with exploration, the learner should (at randomly selected intervals) choose a non-greedy (or non-exploitative) path to seek new answers. We call this an exploratory move "because they cause us to experience states we might otherwise never see" [Sutton, 1998].

The standard reinforcement model (shown in figure 4.1) is based upon an agent (A) with an environment (E). The agent at some point (t) takes an action (L). Upon the agent's completion of action L, the state of the environment (S) may change. This change in the state is given to the agent via a reward (R). Poor decisions are given smaller rewards (or possibly a correction) than wise decisions.

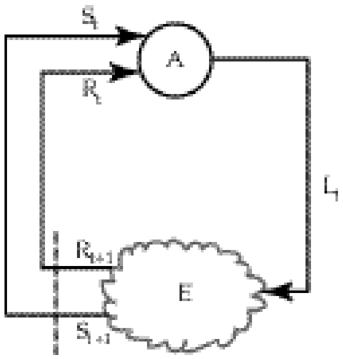

Figure 4.1, Standard Reinforcement Model

Sutton and Barto define four primary components of the reinforcement learning system [Sutton, 1998]:

- Policy

  Defines how the agent is to react to a given situation.

- Reward Function

  Ranks possible actions from the policy on their reward to the agent. The reward function essentially maps our state with a given policy and scores it upon its ability to reach the goal.

- Value Function

  Defines the total number of rewards the agent can accumulate between the current state and the goal. Without rewards, we would not have a value; however, it is the values with which the agent is most concerned simply to achieve more reward.

- Environment Model

  An estimator that can mimic the environment the agent is acting upon. Given a state and policy, the environment model should give us a reasonably accurate representation of the new environment.

Therefore, the job of the agent within its environment is to create a policy which maps its actions to the environmental states that will achieve (typically maximize) the desired goal (or long-term) state. What the rat running through a maze example shows us is an agent interacting with its environment (the maze) making decisions to reach its goal in the face of uncertainty on what lies around the next corner. This uncertainty is much more akin to a true environment as opposed to a supervised learning model.

In supervised learning techniques, the agent is presented with known examples and outcomes until the agent sets its policy to match the cases it was given; unfortunately, the agent tends to learn only what is has been shown or "taught" by the supervisor. The reinforcement learning agent learns through its interaction with the environment. It is this interaction which distinctly distinguishes reinforcement learning from supervised learning in two ways:

- There is no feeding of known input and expected output.
- The evaluation of each policy action is "concurrent" with learning in the supervised method.

Many research and industrial applications have begun to be implemented utilizing reinforcement learning techniques. In fact, reinforcement learning is becoming quite popular with researchers because it "serves as a theoretical tool for studying the principles of agents learning to act" [Kaelbling, 1996]. Some of the applications which have proven reinforcement learning to be a practical computational tool:

- Robotics
- Industrial Manufacturing
- Job Scheduling
- Combinatorial Search Problems
- Computer Game Research

4.1. History

Although the concept of reinforcement learning has been researched in both biologic and artificial intelligence for years, it has been primarily in the last ten years that reinforcement learning has gained more acceptance in machine learning. Samuel was the first to employ what would now be called temporal difference learning; however, no one carried on with his thoughts and methods until the resurgence of reinforcement learning in the late remaining years of the 1980's.

The subject of reinforcement learning as it is now researched is a combination of three differing research studies:

- Trial and Error Learning

  The focus of trial and error learning is quite common in psychology where it was presented by Thorndike that actions followed by a reward (or punishment) cause the action to be remembered by the agent for proper selection in the future. This is more precisely Thorndike's *law of effect* discussed earlier. The first most important characteristic of the *law of effect* is that it is "selectional, meaning that it involves trying alternatives and selecting among them by comparing their consequences" [Sutton, 1998]. The second important characteristic is that it is "associative, meaning that the alternatives found by selection are associated with particular situations" [Sutton, 1998].
- Optimal Control & Dynamic Programming

  Optimal Control systems began in the late 1950's as a means to solve the problem of controlling a dynamic system to minimize the dynamics over time. Most of this work continued upon the foundations laid by Hamilton and Jacobi in the previous century. Richard Bellman furthered this approach to use the system's state and a function valuator to create an equation for the system. The methods that Bellman invented came to be known as Dynamic Programming. Bellman furthered his research in optimal control and developed a discrete stochastic version that came to be known as the Markovian Decision Processes. All of these methods and theories have provided a pivotal theme to the theory of reinforcement learning.

- Temporal-Difference Methods

  In temporal-difference, learning is "driven" by repeated estimates of the same valuation. Temporal-Difference parallels the concept in biological learning of *secondary reinforcers*, which are paired with primary reinforcers and exhibit the same qualities. Arthur Samuel was one of the first to include ideas of temporal-difference by incorporating these concepts into his checkers program. During the 1960's, most work began shifting toward supervised learning until 1972, when Klopf continued with some of the earlier works. In the last ten years, interest and work in temporal-difference has increased. Temporal-Difference plays a smaller role in reinforcement learning than the other two threads; however, its inclusion is important and unique to the concept of reinforcement learning.

4.2. Classes of Reinforcement Learning Methods

Sutton and Barto define three "fundamental classes of methods" which can be employed for the solution of Reinforcement Learning problems. The first, dynamic programming (DP), is well understood and developed from a mathematical viewpoint; however, it requires an accurate and complete model of the environment. Because of the stochastic nature of game play, there is not a complete model of the environment. The second is based upon Monte-Carlo (MC) methods. These algorithms are conceptually simple and, as opposed to dynamic programming, do not require a complete model of the environment; unfortunately, Monte-Carlo methods are not well suited to solving incremental (step-by-step) computations. This leads us to the third, and the focal point of our discussion:

the temporal difference method. Temporal Difference (TD) does not suffer from the requirement of a complete, accurate environment and it is rather well suited for incremental computation; however, temporal difference methods are far more difficult to analyze and solve.

Every form of reinforcement learning discussed satisfies the Markov property. A Markov property is simply a state that retains all relevant information. A problem that is bound by the Markov property requires only the previous state and an action to achieve a subsequent state. At any state, *s*, and any action, *a*, the probability of each next state, *s'*, in a finite Markov Decision Process is given by equation 4.1. The expected value of the next reward is shown in equation 4.2.

$$P_{ss'}^{a} = \Pr\left\{s_{t+1} = s' \middle| s_t = s, a_t = a\right\} \tag{4.1}$$

$$R_{ss'}^{a} = E\left\{r_{t+1} \middle| s_t = s, a_t = a, s_{t+1} = s'\right\} \tag{4.2}$$

These two quantities detail the important facts of the Markov Decision Process for reinforcement learning.

4.3. Temporal Difference

Temporal Difference learning is the concept that is unique and key to the current concept of reinforcement learning. Temporal difference methods take ideas from both Dynamic Programming and Monte Carlo methods to form the framework of its method. From DP, TD methods can utilize previously learned estimates for updating their current estimate without the requirement of a final solution. From MC, TD methods do not require the complete model of the environment for making their decisions–environmental stimulus is sufficient.

The most interesting characteristic with TD is the fact that TD uses past experience to predict its next estimation. The states are represented with V^{π}, a state at some time interval, *t*, to be s_t, and the estimate at a given time of our states to be $V(s_t)$. We begin with the simple, constant-α MC method for changing environments as in equation 4.3.

$$V(s_t) = V(s_t) + \alpha\delta_t \tag{4.3}$$

where,

$$\delta_t = R_t - V(s_t) \tag{4.4}$$

We define R_t to be the actual value found after time, *t*, and α as a constant step-size parameter. The problem with this Monte-Carlo method is that we must wait until the end of the entire episode to find our R_t value that would allow us to update our estimation: $V(s_t)$. Temporal Difference uses a separate goal from MC, replacing the R_t value with a more immediate goal. The basic TD algorithm, called TD(0), uses the basic form of 4.3 with δ_t as defined in equation 4.5.

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{4.5}$$

So, the target for temporal difference is for the next time step, as opposed to the completion of the goal. We represent our next time step target with r_{t+1} as our immediate reward, and γ as the discount parameter (discount for estimation). This concept of using the previous estimation and results to generate the next estimation is similar to DP and is referred to as the *bootstrapping* method.

This combination of the MC sampling technique with the DP bootstrapping is the fundamental theme throughout the temporal difference method. The expected value from our experience of π is utilized with the current estimation at time *t*. To illustrate, let us look at the estimated value from the MC method as shown in equation 4.6. Next, we inspect the estimated value from the DP method as shown in equation 4.7.

$$V^{\pi}(s) = E_{\pi}\left\{R_t \middle| s_t = s\right\} \tag{4.6}$$

$$V^{\pi}(s) = E_{\pi}\left\{r_{t+1} + \gamma V^{\pi}(s_t + 1) \middle| s_t = s\right\} \tag{4.7}$$

What the TD method does is to employ the estimated value from the DP method; however, we do not know the current estimation in our experience because it has not been performed yet. Thus, we substitute V^{π} with V_t. This is the key to utilizing our expected value with our current estimation and what makes temporal difference a powerful tool to utilize.

To demonstrate the advantage of temporal difference over the Monte Carlo method, we shall examine a sample Markov Decision Process. This is an example modified from Sutton [Sutton, 1998]. Suppose we have a robot exploring the floor of a building looking for a particular room. It comes to a section of the floor in which it can make a series of decisions. See figure 4.2 for a visual layout of the environment. At any point, the robot may randomly (with equal probability) select to go to the next or previous point until it reaches one of the two rooms.

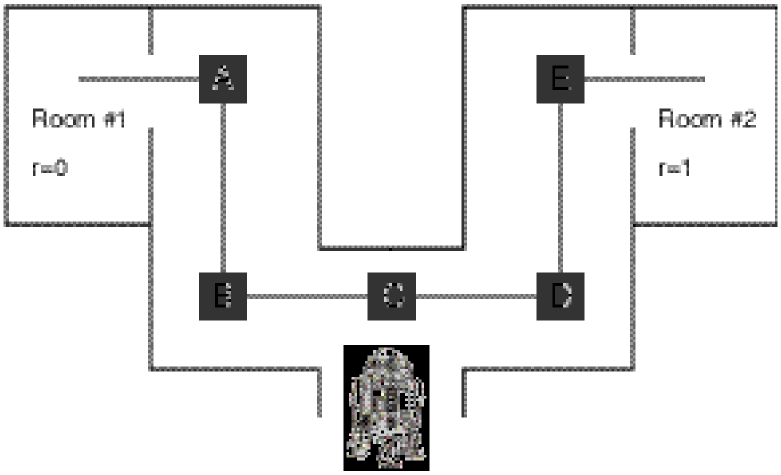

Figure 4.2, Robot Exploration as a MDP

If it reaches room 2, it will be rewarded. If it reaches room 1, it will receive no reward. In this simple example, we wish to properly find the estimated value of return for each step along the way. There are 6 potential positions (5 stepping points and a final point). Therefore, the proper estimate of reward for each point A through E is $1/6, 1/3, 1/2, 2/3, \textit{and}\ 5/6$.

We initialize every step to have an estimated return value of $1/2$.

```
initialize V[s] to 0.5
current := C
Repeat until current = a room
    next := random(next, previous)
    if next = room 2 then r := 1
    else if next = room 1 then r := 0
    V[current] := V[current]+α*(r+γ*V[next]-V[current])
    current := next
end
```

Code 4.1, Pseudo-Code for TD(0) MDP

We then program the robot to attempt the problem by updating the estimates with both temporal difference (See code 4.1) and constant-α MC (see code 4.2).

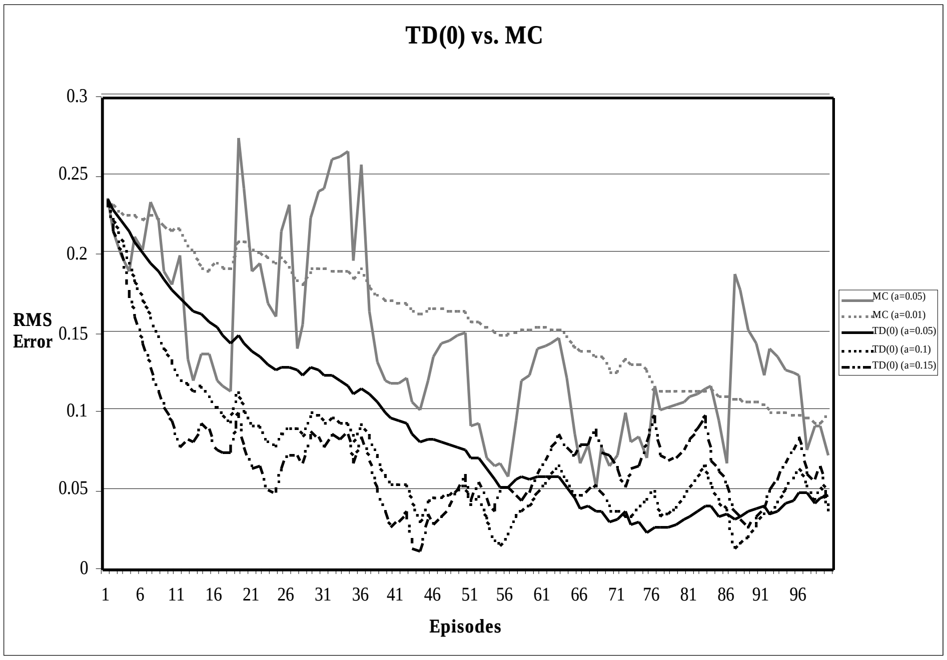

Figure 4.3, Comparison of TD and MC MDP results

As shown in figure 4.3, over several different α values, TD(0) is consistently better than MC. It is also apparent that the constant-α values for the MC method must be very small as they become very unstable as it grows larger. Though both refine their estimations to reasonably close values of the proper estimate, TD(0) has a smaller error. Furthermore, TD(0) converges much quicker due to the fact that it is allowed to update its values during each episode as opposed to waiting for the series of episodes to terminate before computing the changes as is required with MC.

```
initialize V[s] to 0.5
current := C
repeat until current = a room
    next := random(next, previous)
    if next = room 2 then r := 1
    else if next = room 1 then r := 0
    push current onto S
    current := next
end
for all n in S
    V[n] := V[n]+α*(r-V[n])
end
```

Code 4.2, Pseudo-Code for MC MDP

TD gives us several distinct advantages:

- Does not require a model or *a priori* knowledge of the surrounding environment: key to game play.
- Need only to wait until the end of each time step rather than until the end of the entire trial as is dictated by Monte-Carlo methods.
- Has been proven to converge for any fixed policy π. In fact, in constant-α, stochastic tasks, TD has been shown to converge even faster than MC methods [Sutton, 1988].

4.4. Temporal Difference with Eligibility: TD(λ)

Eligibility traces allow us to utilize the entire spectrum of learning methods from the Monte-Carlo (which stores all actions, then updates after completion) to TD(0) which stores none of the actions, but, simply updates after each action. Eligibility traces assign credit (or blame) for each action taken within a particular state allowing us to bridge between information and actions taken. We can apply eligibility traces to any of the reinforcement methods; however, it is when TD(0) is combined with eligibility traces to form TD(λ) that we are concerned.

Eligibility traces are employed to provide the benefits of both TD(0) and MC within one algorithm. With TD(0), we can only update based upon the immediate reward of the action we have just taken. What if there is a delayed reward? It will not be recognized until that reward is achieved. On the other hand, with MC, we can only update once we have achieved the goal state and received the reward (if any). What if the reward was achieved before reaching the goal state (or minor rewards were given along the path)? We will update every step we took along the path giving it the same reward regardless of whether it contributed or not. Eligibility traces will allow us combine the best of each algorithm without falling into the traps of each respective method.

To combine the two, we would simply credit rewards back *n* steps: greater than one, but less than every step taken until termination. The process of updating only one step backward is equivalent to TD(0) and similarly, updating only once we have every step is equivalent to MC. In MC, the only return we use is the reward plus the estimated value of the next state as shown by equation 4.8. Equation 4.8 also includes the discount value for discounting the estimates of next states (the discount is set to 1 for our examples, i.e. no discounting).

$$R_t = r_{t+1} + \gamma V_t(s_{t+1}) \tag{4.8}$$

Now, because MC is based upon a single step (the final reward step) to update its previous steps, we must expand this return function to include *n* steps. Equation 4.9 shows the return for an *n*-step algorithm.

$$R_t^{(n)} = \delta_t = r_{t+1} + \gamma^1 r_{t+2} + \gamma^2 r_{t+3} + \mathbf{K} + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n}) \qquad (4.9)$$

What if the episode was to hit a goal before the *nth* step? In that case, the standard MC return, equation 4.8, is employed. However, this situation helps us to understand why such an *n*-step method is not practical. The *n*-step method suffers the same restriction that is placed upon the MC methods: it must wait a certain (*n*) number of episodes before calculating the return. For real-time applications (such as control), this latency is can become problematic. Therefore, we must apply the benefits of the *n*-step method without the requirement of waiting for the steps to occur.

We define the eligibility trace to properly decay a state's contribution to the current state as it moves further into the past (i.e. a state's eligibility is the "degree to which it has been visited in the recent past" [Kaelbling, 1996]). Within each iteration, every state decays by $\gamma\lambda$, and if we are updating the current state, its eligibility is incremented by one. Updating the standard TD(0), equation 4.3, to include the eligibility trace, results in equation 4.10.

$$V(s_t) = V(s_t) + \alpha\delta_t e_t(s_t) \qquad (4.10)$$

Where δ_t is defined in equation 4.5 and $e_t(s_t)$ is the eligibility value for state s_t. Equation 4.11 defines the eligibility trace:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) + 1 & \textit{if s is our current state } (s_t) \\ \gamma\lambda e_{t-1}(s) & \textit{otherwise} \end{cases} \qquad (4.11)$$

The new parameter, λ, is defined as the *trace-decay parameter.* This parameter defines how quickly the weighting of each state as it contributes to the current state will fall off in time. For larger λ, states are heavily weighted even as they have drifted further back into the history. In fact, though it is computationally more expensive to utilize the generic TD(λ), it has been shown to converge considerably faster for systems using a larger λ [Dayan, 1992]. Now we have included a way to historically weight states without waiting for the reward to be given (as is the limitation of the MC and *n*-step algorithms). We can continually update as with the standard, TD(0); however, we may properly credit previous states with any rewards achieved.

In fact, through simple inspection, we can see that the TD(λ) algorithm is simply a generic form that combines both TD(0) and MC at its extremes. First, allow us to examine the case in which λ is 0 and γ is 1. In this case, our eligibility trace is reduced to equation 4.12. Substituting the eligibility back into equation 4.10, we arrive at equation 4.13 which is simply the TD(0) case as shown in equation 4.3.

$$e_t(s) = \begin{cases} 1 & \textit{if } s \textit{ is our current state } (s_t) \\ 0 & \textit{otherwise} \end{cases} \tag{4.12}$$

$$V(s_t) = V(s_t) + \alpha\delta_t \begin{cases} 1 & \textit{if } s \textit{ is our current state } (s_t) \\ 0 & \textit{otherwise} \end{cases} \tag{4.13}$$

Conversely, if we examine the case in which λ is 1 and γ is 1, we see that the eligibility never decays and every state will eventually be rewarded by the final reward. Thus, we may consider the TD(1) case to be the Monte-Carlo method as described in equation 4.3 and 4.4 with one major caveat: TD(1) is performed on-line, that is, it occurs during the episodes without waiting until a goal has been reached. Thus, with the TD(1) method, you can utilize MC methods without waiting until the final goal to update the states.

Revisiting the robot-walk example from the last section, we can now implement the TD(λ) algorithm (as shown in code 4.3), and test it in various cases of λ. At λ=0, we should expect to achieve the same results as the TD(0) case, and at λ=1, we should achieve very similar results to the MC method; although, because it is still performed on-line it will converge quicker than the standard MC method. It is at various values of λ that we should achieve faster convergence and better results. It will also introduce us to the problem of choosing the proper *trace-decay parameter* value for the problem at hand. Some general domain knowledge is required for proper selection of the decay value. For the simple robot walk, we would suspect that a higher λ (though less than 1) would lead to a faster convergence because every step is important to our walk. In a game of chess, however, a typical game that may last through 40 moves per player, a faster decay may be more valuable as the initial pawn moves may not have contributed much to the current state.

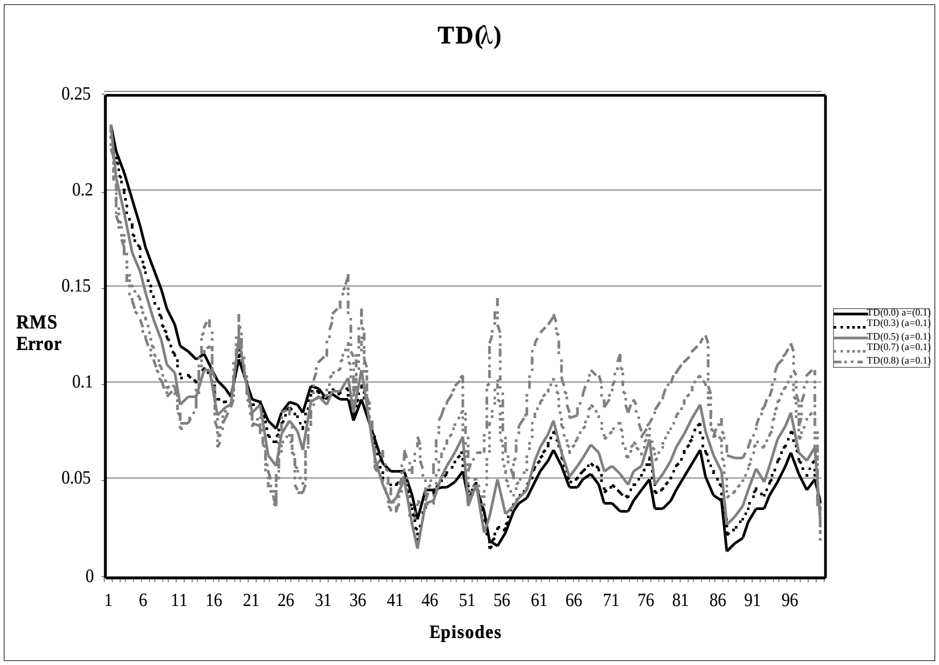

Figure 4.4, Comparison of Different λ values for TD(λ) and MDP Robot Walk

As shown in figure 4.4, the results of TD(λ) with a constant α=0.1, are consistently better than either TD(0) or MC as shown in figure 4.3. In fact, one can see that the λ=0 case is identical to the TD(0) case for α=0.1. Not only does the TD(λ) case converge quicker, but, it is less susceptible to worsening its estimates over time as we saw in the previous example with the MC method.

```
initialize V[s] to 0.5
initialize e[s] to 0.0
current := C
repeat until current = a room
    next := random(next, previous)
    if next = room 2 then r := 1
    else If next = room 1 then r := 0
    δ := r + γ*V[next] – V[current]
    e[current] := e[current] + 1
    push current onto S
    for all i in S
       V[i] := V[i] + α*δ*e[i]
       e[i] := γ*λ*e[i]
    end for
    current := next
end
```

Code 4.3, Pseudo-Code for TD(λ) implementation

4.5. Samuel’s Checkers and Temporal Difference

Samuel’s Checkers Program—as described in Chapter 3—was one of the first computer programs developed with the key aspect of reinforcement learning: the value of a state is dependent upon the value of its subsequent states. In Samuel’s simple *rote-learning* method, he initially stored the value of the Depth First Search at each node and attached with it a ply-value. Therefore, if the score were achieved with more ply levels (or moves) it was discounted more than a move with the same score that required fewer moves. So, the *rote-learning* score was based somewhat upon the fact that the state was dependent upon the subsequent states of the game. It was with his second method, *learning by generalization* that Samuel came closest to the modern notion of reinforcement learning.

In his *learning by generalization* method, Samuel adjusted the weights of each game characteristic that contributed to the current state. So, after a move by both players, the program would search for the best possible state to achieve. It would then backup the score of this move to the last state and adjust the weights to guide it. This would happen after each move so that at any state, the linear function would more accurately predict the following state. Samuel's backup method is the same in concept with Temporal Difference Learning; however, it lacked an important characteristic: rewards.

Samuel's *learning by generalization* method pushed the linear value function to be consistent and accurate; however, he had no way to bind the function to the actual value of the states. This is enforced in the standard Temporal Difference methods by tying rewards and discounting from these terminal reward states. Samuel tried to introduce consistency to his method by giving states with strong piece advantages a rather large, fixed weight. However, by simply introducing a large weight within the evaluation function itself, the function tried to fit the other weights in line with the single large term. This led to the program actually becoming worse with age. Samuel would "kick-start" the program by resetting some of the larger adjusted weights back to zero.

4.6. Gradient Descent with TD(λ)

Oftentimes, the values being adjusted are not state values, but weights (or parameters) of a non-linear function which attempts to estimate or predict our state. These functions allow us to generate a fairly good approximation of the larger state as opposed to a table type state (as was employed by the MDP robot-walking example previously). However, the problem becomes, how does the TD(λ) method allow us to alter these weights as opposed to table entries?

Firstly, we must now think of V_t as representing a parameterized function as opposed to a table. We declare that the vector $\vec{\theta}_t$ is our parameter list. Therefore the non-linear function, V_t, is completely dependent upon $\vec{\theta}_t$, and changes from state to state as $\vec{\theta}_t$ and the inputs change. This allows us to approximate many different functions, which are suitable for many applications. The most popular of these functions in approximation are those that fall into the gradient-descent methodology. Fortunately, TD(λ) lends itself very well to solving the gradient descent problem.

Quite simply, gradient-descent methods attempt to reduce the error between the estimated value and the actual returned value. So, on each iteration, we choose an action, *a*, and estimate our resultant state as computed by our differentiable function V_t. Upon arriving at our new state, we find that the estimate, V_t, is different than the actual value, V^{π}. We must, therefore, adjust our parameters, $\vec{\theta}_t$, in the direction that would most reduce the error. The generalized gradient descent method for adjusting $\vec{\theta}_t$ is shown in equation 4.14.

$$\vec{\theta}_{t+1} = \alpha\left[V^{\pi}(s_t) - V_t(s_t)\right]\nabla_{\vec{\theta}_t} V_t(s_t) \tag{4.14}$$

The gradient-descent method therefore adjusts $\vec{\theta}_t$ by a value that is "proportional to the negative gradient of the squared error" [Sutton, 1998]. So, by applying equation 4.14 to the general TD(λ) form as seen in equation 4.10, we come up with equation 4.15. No longer are we adjusting the actual values of V_t, but, now we are adjusting the parameters, $\vec{\theta}_t$, so that the function will return a more accurate estimate.

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t\vec{e}_t \tag{4.15}$$

Where δ_t is defined by equation 4.5 and $\vec{e}_t$ is the eligibility trace for each component of $\vec{\theta}_t$, defined by equation 4.16.

$$\begin{aligned} &\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} V_t(s_t) \\ &\textit{where } \vec{e}_0 = \vec{0} \end{aligned} \qquad (4.16)$$

The most immediate application of equations 4.15 and 4.16 is, of course, the multi-layer neural networks employing backpropagation. We can directly relate $\vec{\theta}_t$ to the weights in Back-Propagation [Hertz, 1991] and V_t to our neural activation. In fact, as we shall see in section 6.4, these two equations map directly to the updating of the weights neural nets using the back-propagation to compute the gradient descent of the eligibility traces.

5. Tesauro's TD-Gammon

Backgammon is an ancient game believed to have been developed more than one thousand years before Chess. It is played with blots ("checkers") on a board as shown in figure 5.1. Each player alternates rolling two dice and moving their blots in opposing directions. The first player to move his blots around and off of the board is the winner. A "gammon" is awarded if an opponent can move his blots off of the board without having lost any blots to his opponent. A "backgammon" is awarded if a player moves his blots off of the board without having lost any blots to his opponent and is able to keep all of his opponent's blots in the far quadrant of the board.

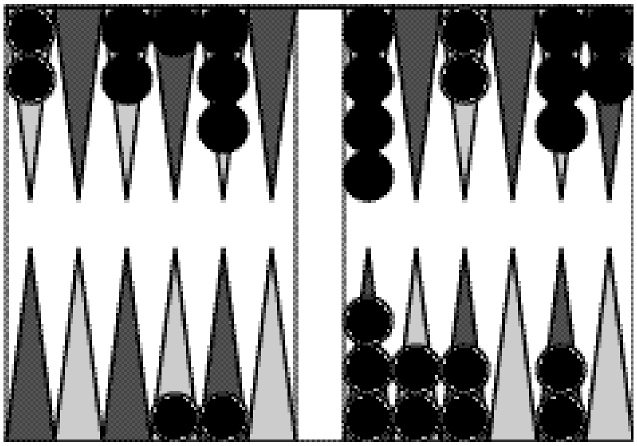

Figure 5.1, Backgammon Board

If a player lands on (or "hits") an opponent's single blot, the blot is sent to the bar in the center of the board. The person who has their blot "sent" must then re-enter the board before making any other moves. Additionally, by placing more than one blot on a point, the opponent is blocked from placing any of his blots onto that point. Finally, an opponent may offer to double the current stakes of the game. If the opponent accepts this "doubling cube," he is given the right to make the next double offer; whereas, if he resigns the double, he forfeits the current stakes. The points at the end of the game are calculated using the current value of the doubling cube multiplied by 1 for a standard win, 2 for gammon win, and 3 for a backgammon win.

The strategy complexity added to the game in hitting, blocking, and doubling make Backgammon a very challenging game for humans to play, let alone for a computer to master. In fact, due to the stochastic nature that comes from the dice roll, there are over 10^{20} possible states in the game of backgammon. This proves much too complex for a table lookup. Deep tree searching algorithms fail for Backgammon as well because of the random dice rolls. There are 21 possible dice rolls and an average of 20 moves per dice roll. This would mean for every single ply there is an average branching ratio of around 400. This would not be feasible on even the largest of computers.

5.1. Tesauro's Neurogammon

Neurogammon was Tesauro's first major work on the game of Backgammon. Tesauro employed a standard Hopfield Neural Network with back-propagation (as described in section 6.3) to attempt to give a generalized sense of direction for playing Backgammon. The initial setup contained no hidden layers, 459 input nodes, and 1 single output node. Each board location (of which there are 26) is represented by a number of input nodes. Other input nodes represent varying board situations. This initial setup tended to favor piling up as many pieces onto one point as the computer could. Tesauro and Sejnowski experimented with varying sizes of the neural network including adding hidden layers of varying node sizes.

Training the network was performed with supervised learning of approximately 3,200 positions and moves. These moves were generated by backgammon experts and repeatedly shown to the network. Learning was accomplished with approximately 50 cycles through the training set. Of course, the largest disadvantage to this was that the computer could not learn the game as it played. It learned the game based upon what it was shown (or taught). Tesauro (who is regarded as an expert player) continued to train the computer with his own moves and later served as the evaluation on how well the computer could play against a quality human opponent.

After training, the network was shown to have developed a generalized sense based upon the moves with which it was taught as opposed to simply memorizing those moves. The game was tested against other computer backgammon programs as well as against human opponents. It was able to best the other computer programs around 60% of the time, but, a human opponent only around 30% of the time. It made obviously stunning errors as well as some rather surprisingly “thoughtful” strategic moves; however, overall, it was an average to novice backgammon player. Even though (against human opponents) the Neurogammon was not very successful, it did win the “backgammon championship at the 1989 International Computer Olympiad” [Tesauro, 1989].

5.2. Tesauro’s TD-Gammon

Tesauro later began work to build his backgammon game based upon the new learning paradigms built around reinforcement learning. He focused his attention upon the temporal method of learning: the agent observes the input parameters and decides upon the output. It is then rewarded or scolded by the consequence of its actions as opposed to Neurogammon that required a teacher to correct it. Although Tesauro had been interested in temporal methods, the primary problem had been the delay in reinforcement proved to be difficult for proper credit assignment and that evaluations in previous models had tended to be linear. Two advancements allowed Tesauro to test the capabilities of TD(λ) and to compare to the previous Neurogammon work:

- Many non-linear approximation functions had been developed
- Advancement of temporal methods (particularly TD(λ) by Sutton [Sutton, 1998])

Tesauro constructed a standard Hopfield neural network with 198 input units, 40 (later 80) hidden units, and a single output unit. The output unit represented merely an estimation of the probability of winning based upon the input. For the input layer, four units represent the number of blots white possesses at a single point and four units represent black at the same point. For the 24 points, there are 192 input units used. Two more input units represent the number of blots white and black has on the bar. Two additional units represent the number of blots that white and black has removed from the board. Finally, two units specify whether it is white or black’s turn.

The input was fed into the network which then used a standard feed forward to compute the output: a value from 0 to 1 representing the probability that the specified player would win. TD-Gammon then utilized the gradient descent method of TD(λ)—as shown in Section 4.6 and Section 6.4—to update the network after each move and observed state. The initial weights were set to randomly small values, so the network had no *a priori* knowledge of backgammon or the strategy. TD-Gammon was then played against itself allowing the neural network to observe and predict for each player of the game.

This first version of TD-Gammon did not perform a multi-ply look-ahead search to determine the best possible move to make. Rather, it evaluated each available position due to its die roll and chose the position that would lead to the greatest probability of victory. In later versions, Tesauro implemented a 2 ply look ahead to improve the results to which the computer could use to choose the better move. Each version of TD-Gammon was tested against world class backgammon players and the immediate results were somewhat astonishing.

The initial version was trained for 300,000 games and played against Neurogammon. This version consistently bested Neurogammon. The most amazing result of this was that after only tens of thousands of games, Tesauro found a great deal of learning had taken place. This initial version 1.0, played at a high intermediate level without any assistance. Later, modifications were made to subsequent version to give the network more input as to the progress of the game. These versions achieved a grand-champion level playing very competitive games to the top human competitors in the world. Table 5.2 shows the results for each version of TD-Gammon.

	Training Games	Opponents	Result
TD 1.0	300,000	Robertie, Davis, Magriel	Lost by 13 points in 51 games (-.25 ppg)
TD 2.0	800,000	Goulding, Woolsey, Snellings, Russell, Sylvester	Lost by 7 points in 38 games (-.18 ppg)
TD 2.1	1,500,000	Robertie	Lost by 1 point in 40 games (-.02 ppg)

Table 5.2, Results of TD-Gammon

Version 2.0 of TD-Gammon was shown at the 1992 World Cup of Backgammon Tournament playing against some of the best players in the world. In fact, Wilcox Snellings and Joe Russell were both former World Champions. Bill Robertie played the most number of games against TD-Gammon, and, in fact, played a tournament against version 2.1 of TD-Gammon. Robertie trailed TD-Gammon for the entire tournament until the very last game, in which he was able to beat the computer and narrowly win the tournament by a single point. Robertie later stated that TD-Gammon 2.1 plays at a "strong master level that is extremely close…to equaling the world's best human players." [Tesauro, 1995] He went on to state that due to the tireless computation of the computer, it would be favored against the best players during a long, grueling match play.

Kit Woolsey, who (as of 1995) was rated the number 3 player in the world, wrote a favorable write-up of TD-Gammon. Woolsey analyzed the game play of TD-Gammon and offered the following personal comments to Tesauro [Tesuaro, 1995]:

> *"There is no question in my mind that its positional judgment is better than mine. Only on small technical areas can I claim a definite advantage… [TD-Gammon's] strength is in the vague positional battles where judgment, not calculation, is the key. There, it has a definite edge over humans… [Its] judgment on bold vs. safe play decisions, which is what backgammon really is all about, is nothing short of phenomenal."*

One extremely powerful example of the game play of TD-Gammon is in opening positional play. For 30 years, the standard human opening move (called "slotting") was to setup your piece for possible hitting, but, allowing for an attack position. TD-Gammon introduced a different strategy (now called "splitting") which avoids the "slotting" maneuver and is now considered a superior strategy. In fact, most world-class players now employ the "splitting" strategy for the opening move.

5.3. Tesauro's Conclusions

The results of utilizing the standard neural network to compute the gradient for non-linear function approximation as applied to the game of backgammon was an astonishing success that even surprised Tesauro. It was the first major display of what is possible using reinforcement learning combined with the non-linear computation available via the neural network. Due to his work, Tesauro realized three important insights in TD(λ) based game play.

The absolute error of each calculation made by the function estimator (neural network) was often around a tenth of one point. This is a rather large and substantial error in the estimation of the game; however, because every calculation has a similar error, the relative accuracy of the neural network was very high. In a task in which estimations are weighed and compared to each other, each estimation having very little (if any) relative error meant that results were very accurate within each other. This would suggest that the absolute error observed is simply a systematic error which therefore "cancels out in move-making decisions" [Tesauro, 1992].

The game of backgammon lent itself very well to the temporal difference method. Due to the stochastic nature of the game, during training against itself, the evaluation function would see a very wide array of possible states and outcomes. This forces the learner to more dramatically fit itself to a non-linear function describing the game. In more deterministic games (such as chess), the learner may focus on one particular strategy and never explore the environment; therefore, for deterministic applications of temporal difference, the learner must be forced to explore. However, simply forcing random exploration can not guarantee success by the learner. In the game of backgammon, the game always progresses forward as determined by the dice roll until a player reaches the end. In chess, a random player could, conceivably, consistently choose moves to avoid the other player forcing the game to last almost infinitely. Finally, non-deterministic games tend to have a much smoother continuous evaluation function; whereas, deterministic games typically have a more discontinuous evaluation function.

The final interesting lesson learned is that the neural network's backpropagation tends to learn more highly linear concepts during the initial training. Once these concepts tend to be properly weighted, the network begins adjustment of values that focus more on the non-linear aspects of that which it is learning. In the case of backgammon, the network tended to learn within the first learning iterations topics such as hitting, playing safe, and building up new points. It was later during training that the computer began to develop strategy and more non-linear concepts. In fact, it was found that having only learned the basic linear concepts of the evaluation, TD-Gammon was better than a typical beginning human player.

Tesauro found that theoretically, there are a number of limitations that could inhibit temporal difference learning. The algorithm may not necessarily converge for a predication and control task and even when it does converge, it could get stuck in a rather poor local minimum. Theoretically, as the problem size increases, the learning time required may become too great that the solution may be too resource intensive. However, with these initial theoretical concerns, Tesauro found that the method always at least converges to a local minimum with the output of the solution typically very good. He found that as you increase the number of hidden units, the results tended to improve. The network, he discovered did not suffer greatly in training time as the size and complexity of the problem was increased. In fact, after the work of TD-Gammon, he does not believe that this could be a problem. Finally, the most exciting outcome of the work was the indisputable fact that the network—trained against itself with zero knowledge about the game of backgammon—was able to learn quicker and achieve a vastly superior game knowledge and playing ability than Neurogammon (which was trained with a huge number of expert moves). This fact alone would suggest that a number of applications and research could be conducted on the utilization of learning non-linear functions with a neural network and temporal difference reinforcement.

6. Tic-Tac-Toe and Reinforcement Learning

Tic-Tac-Toe is a basic, children's game thought to date back to ancient Egypt. In fact, the Romans played a similar game called *Terni Lapilli* that was extremely popular; although, there is some discussion as to whether it was identical to modern Tic-Tac-Toe. It is believed that *Terni Lapilli* used three pieces (as opposed to the two in modern Tic-Tac-Toe) and was more complicated than Tic-Tac-Toe. For *Terni Lapilli* to be as popular as it was, it would almost certainly be more complicated that Tic-Tac-Toe.

Tic-Tac-Toe is played on a board with 9 positions. There are three possible states for each position: blank (no player has laid their piece), X (player X has laid their piece), or O (player O has laid their piece). A typical game would appear as in figure 6.1. The player with the X pieces goes first, putting a piece in any unoccupied space. The player with the O pieces makes their move in the same fashion. The first player to lay their pieces in a line (across, down, or diagonally) wins the game. If all spaces are occupied and no player has completed a line, then the game ends in a draw.

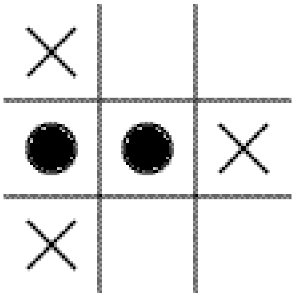

Figure 6.1, Tic-Tac-Toe Board

The strategy of Tic-Tac-Toe is extremely simple. If X begins the game by placing his piece on any of the corners, O must place his piece in the center to prevent X from winning the game (as long as O continues to block X). In fact, the player in the O position is at a disadvantage because he is almost always playing defense, reacting to the move that the X player has made.

Because there are three pieces (blank, X, O) and nine possible positions, the number of states is bounded by $O\left(3^9\right) = O(19,683)$. In actuality, there are far fewer states because many of the states are winning (or terminal) states. Additionally, many of the states within the upper bound are unachievable due to the basic rules (e.g. O could never have more pieces on the board than X). Needless to say, a table-based implementation will require storage space to hold several thousand states.

6.1. TD(λ) Implementation

The first attempt to solve the Tic-Tac-Toe problem used the standard TD(λ) method without gradient descent. In fact, it is the same algorithm as used to test the Markov Decision Problem in section 4.4. The game was set up so that one of three types of player could be assigned to each position: human, random, TD(λ). All states were initially set to zero, thus the initial TD(λ) player had no *a priori* knowledge about any state. If a player won, it was granted a reward of 1.0. If the player lost, it was corrected with a reward of –1.0. Various experiments were performed with varying rewards for a draw game before eventually settling upon a reward of 0 to both players in a draw game.

TD(λ) is a table-based implementation storing the value estimations for every state encountered. As discussed previously (and we shall see), this is not the best solution for a game domain type problem; however, it provides us with a good comparison (and even possible teacher) to the neural network based temporal difference estimator in section 6.5. The TD(λ) player implementation updates its estimates as shown in code 4.3.

The program controlled game play by giving the current player an opportunity to move based upon its policy. If the move resulted in a game winning move, that player was rewarded and the other player given a correction. The board was reset and a new game would occur repeating until the number of games specified had been completed. The random player would randomly select from the available board positions and place its piece. TD(λ) employed a more cohesive selection process. TD(λ) would select a random number between 1 and 20. If it chose the number 7, it would randomly choose from available board positions (just as the random player). This helped to force the TD(λ) player into exploration mode so that it would occasionally try a non-optimal move. This randomness was tested with an increased and decreased interval and was found to have little effect either way over the 1 out of 20 chance. The random element was required though to increase the number of states for which it had an estimate. For the majority of the choices, TD(λ) would examine each available board position and compare the state estimate given by the TD(λ) estimation with each of the others. The best estimate resulted in that move being made (see code 6.1 for the pseudo-code representation of the policy employed). If no move was better than the other, then a random move would be made. After the move was made and the opponent made their move, TD(λ) would make another move and compare the results of its move with the expected results of the previous move and apply the process as shown in code 4.3 to update the last state.

```
sub Estimate-Policy
    if random() = 7 then
        move := random open board position
    else
        foreach possible in open board positions
            if estimate(possible) > max then
                max := estimate(possible)
                move := possible
            endif
        end
    endif

    if move is empty then
        move := random open board position
    endif
end
```

Code 6.1, The Policy for a State-Estimation Player

Learning took place by running one player as random and the other player as the TD(λ) player. The TD(λ) player was typically used as the O player because O is a more difficult position. Some tests were also performed allowing TD(λ) to play as X against a random player. Finally, TD(λ) was allowed to play against itself as both X and O to develop strategies against another learning player.

6.2. Results of TD(λ)

With TD(λ) playing as the O position, the random player was selected to play the X position. The program was run for one and one-half million iterations to (hopefully) explore every state and calculate an accurate estimate for that state. The first comparison was to find the optimal α value with which to perform our learning (using λ=0.5 and rewarding a draw with 0). Three values were tested for α: 0.05, 0.10, and 0.15. The program was set up to run for 500,000 iterations, save the states to disk, and run again for three iterations. This would allow the pseudo-random number generator to reset and the game to avoid continual repetition of the same moves. In fact, it usually gave a large “boost” in the win percentage because it was no longer playing catchup to the initial default estimation states.

Figure 6.2 shows the case for α=0.5 through the first 750,000 iterations. As shown, after approximately 85,000 iterations, the TD(λ) player had finally surpassed the random player in the percentage of wins per 1,000 games. The TD(λ) player continued to improve until approximately 550,000 iterations at which point, it never bettered its percentage. By the 500,000 mark, it was losing fewer games than draw games. This meant that the TD(λ) player did a good job of trying to win the game, and failing that, would try to prevent a win by the random player. In fact, after one and one-half million iterations, TD(λ) had stored 2,217 states and was winning the game 51.8% of the time, would tie the game 29.5%, and lost 18.7% of the games played. The α=0.5 case was a very continuous learning function with little chatter as shown in figure 6.2.

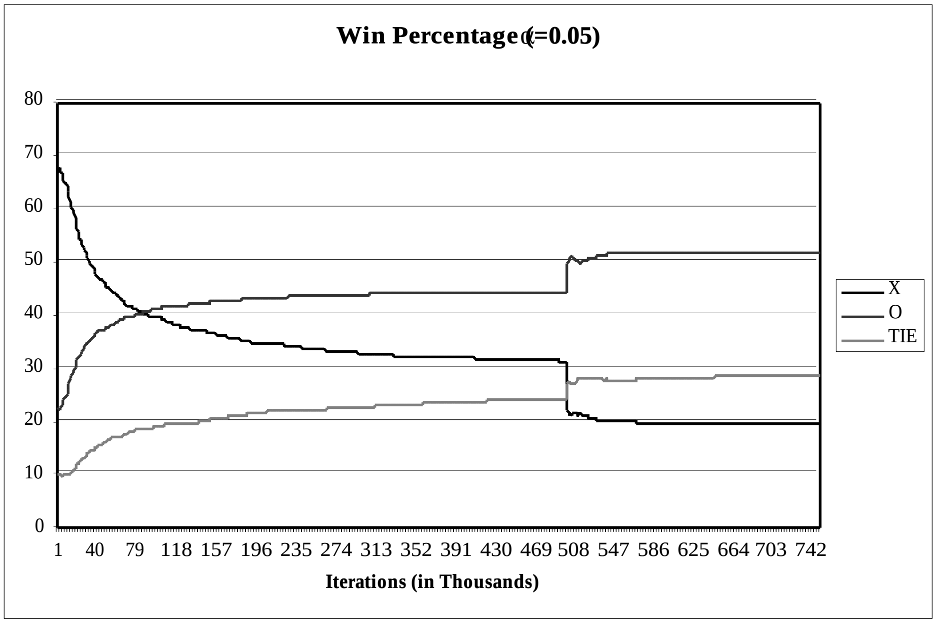

Figure 6.2, Percentage of Victories (λ=0.5, α=0.05)

Figure 6.3 shows the case for α=0.10. In this case, TD(λ) learned quicker and was able to surpass the random player in about 72,000 iterations. Again, as it was reset at the 500,000 mark, it began to perform better; however, as shown in figure 6.3, it was chattering some after it had leveled off at approximately 550,000 iterations. After one and one-half million iterations, TD(λ) had stored 2,226 states and was winning the game 50.5%, drawing a tie 30.3%, and losing 19.2% of the games played. Although α=0.10 was able to correctly estimate the majority of states required to beat the random player quickly, it was apt to worsening its state estimation position as time continued.

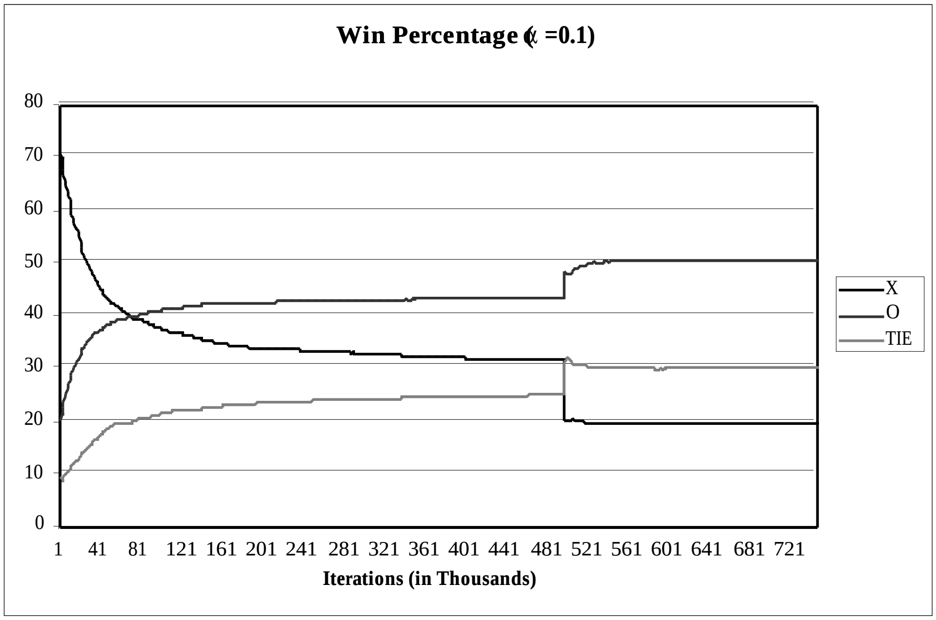

Figure 6.3, Percentage of Victories (λ=0.5, α=0.1)

Finally, figure 6.4 shows the case for α=0.15. This case showed that the larger the learning rate parameter, the estimations became more unstable. The TD(λ) player took 92,000 iterations to finally best the random player. As the graph shows, it was very prone to overshooting its estimates during updates. After one and one-half million iterations, TD(λ) had stored 2,291 states (the most of any of the other tests); however, it was winning the game only 49.8%, drawing a tie 29.5%, and losing 20.7% of the time. This learning rate parameter gave the worst results of the three, with α=0.05 giving the best results, just as it had with the Markov Decision Process in section 4.4.

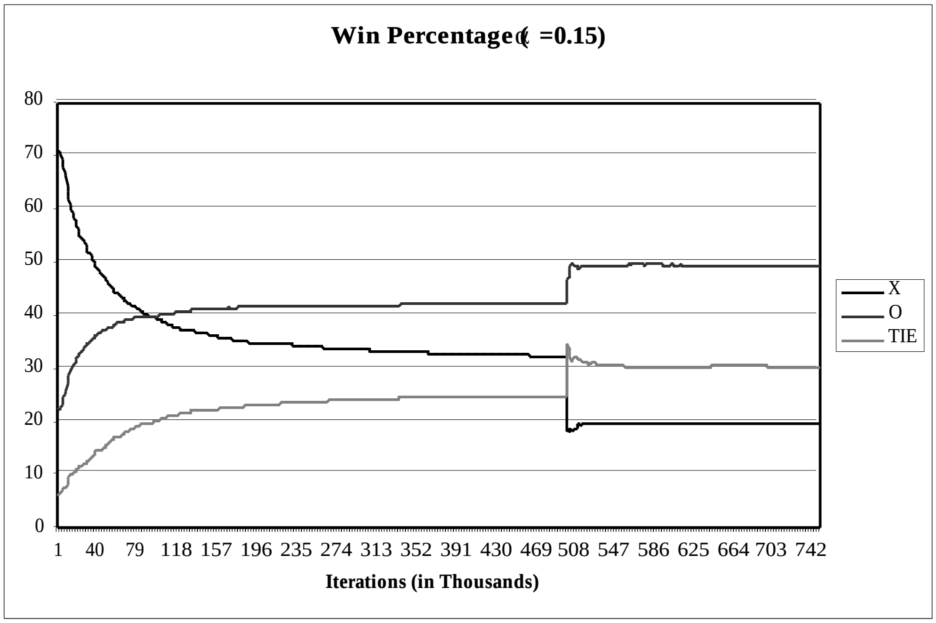

Figure 6.4, Percentage of Victories (λ=0.5, α=0.15**)**

With the learning rate parameter chosen as 0.05, experiments were performed to find the optimal reward for a draw game. Initially, a draw game received a reward of 0. The reward for winning is, of course, 1.0 and conversely for losing, -1.0. This gives a function to which all states lie between –1.0 and 1.0. However, should a tie be worth a reward of some value? Trying four separate values for ties gave some interesting results.

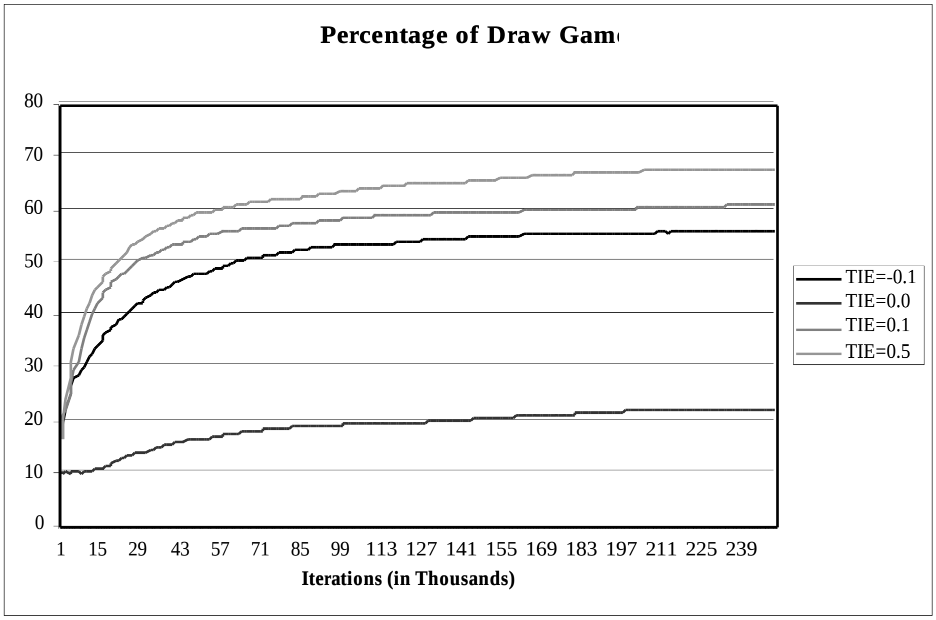

Figure 6.5, Percentage of Games Ending in a Draw For Each Reward

As shown in figure 6.5, any non-zero draw value heavily weighted the game to play for a tie as opposed to playing for a win. For this reason, the draw reward was chosen to be 0. With a learning rate and a draw reward set, the *trace-decay parameter (λ)* must now be decided. Varying trace decays from 0.0 to 0.6 were tested before deciding that the results were not improving with each increase in the *trace-decay parameter.* As shown in figure 6.6, the number of games won by the player increased slightly faster for those cases with a higher λ; however, after several thousand iterations, all λ values settled upon the same curve.

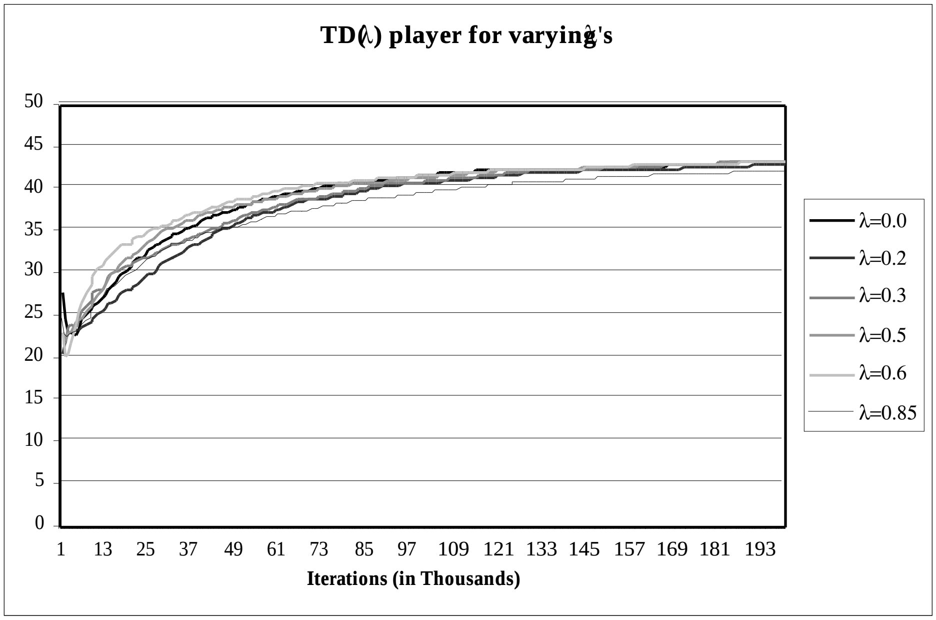

Figure 6.6, Percentage of Victories for Differing λ

Because the O position is playing a game of defense-first, O typically cannot mount an offensive strategy to win the game. With TD(λ) playing O, the random player was always at an advantage until TD(λ) adapted its states enough to more accurately predict the outcome. As a final experiment, TD(λ) was taught to estimate the X player to compare with the previous O player results.

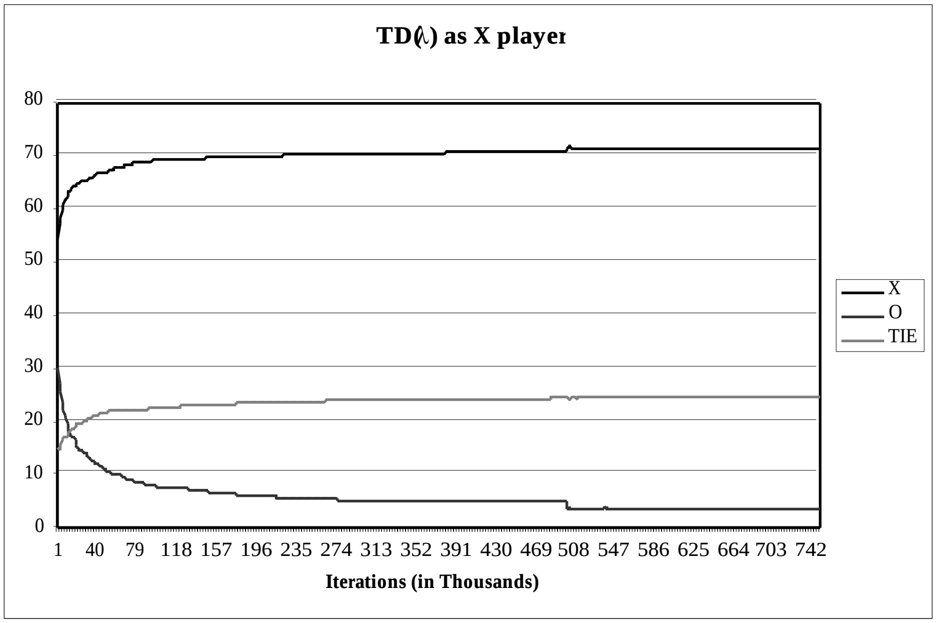

Figure 6.7, Percentage of Wins with TD(λ) as X

As shown in figure 6.7, with λ=0.5 and α=0.05, the results were significantly improved over the case of TD(λ) playing as O with the same parameters. After being trained for one and one half million iterations, it had generated 1,489 states. This was significantly less than the states generated as the O player, in part, because the X player has such a piece advantage that it can quickly win a game with a wrong move by the O player. In fact, after the training set was complete, X was winning 71.2% of the matches while O won 3.8% and games ended in a draw 25% of the time.

After all of the training iterations for varying values of α, λ, and draw rewards, it was time to perform a subjective test to see how well TD(λ) could actually play the game of Tic-Tac-Toe against a worthy human opponent, as opposed to the random player. Because the best results against the random player were achieved with α=0.05 and a draw rewarded with 0, those training sets were utilized against a human opponent: the author. For varying λ, the game was played against the human opponent for twenty iterations to test its prowess.

	Number of Training Iterations	Number of States	Games won by Human	Games won by TD(λ)	Draw Games
λ=0.0	10,000,000	2,305	6	4	10
λ=0.2	10,000,000	2,318	9	1	10
λ=0.3	10,000,000	2,316	10	2	8
λ=0.5	10,000,000	2,324	5	5	10
λ=0.6	10,000,000	2,327	6	2	12
λ=0.5 (computer X)	1,500,000	1,489	0	0	20
λ=0.5 (tie=-0.1)	10,000,000	2,418	6	2	12

Table 6.1, TD(λ) vs. Human Opponent

As shown in table 6.1, the results—though not remarkable—were respectable for a table based look-ahead method. The best results for TD(λ) as player O were achieved with λ=0.5 where the computer and the human opponent each won 5 games while ending in a draw 10 games; therefore, the match ended in a draw. The best overall results were achieved with TD(λ) playing as the X player. With only 15% of the training iterations and 64% of the number of state estimations as compared to the O player, λ=0.5 case, it was able to prevent the human opponent from winning and forcing a tie every time.

After playing hundreds of games against the varying trained sets, some basic patterns were seen that the TD(λ) player had developed. Firstly, TD(λ) did learn the basic strategy of Tic-Tac-Toe. It was able to consistently avoid giving up the power position to the opponent which made the game more difficult for the opponent to win. Secondly, it ultimately learned only a defensive game rather than a winning game. TD(λ) was able to play a fairly decent game; however, it was not a truly formidable opponent.

TD(λ) did learn the very basic strategy of Tic-Tac-Toe (arguably, the only strategic move in Tic-Tac-Toe): when playing the O position, if X takes any of the corners, O must take the center position or O will lose. In every single game in which the human opponent was X, if X chose a corner position, TD(λ) would take the center position to prevent X from winning the game outright. In the cases in which TD(λ) played X, it always chose a corner position to begin with, forcing O into the defensive strategy of taking the center position. Interestingly, TD(λ) would never begin the game with a weak position (one of the non-corner border positions). Because these positions offer no piece advantage (unless used for the win or a block), TD(λ) developed a strategy to select the more important positions first.

Unfortunately, TD(λ) became primarily focused upon preventing the opponent from winning (even with the cases of a draw being rewarded with 0 or –0.1). Once this realization was made, it became much easier to beat the computer because winning positions for the computer could be left open.

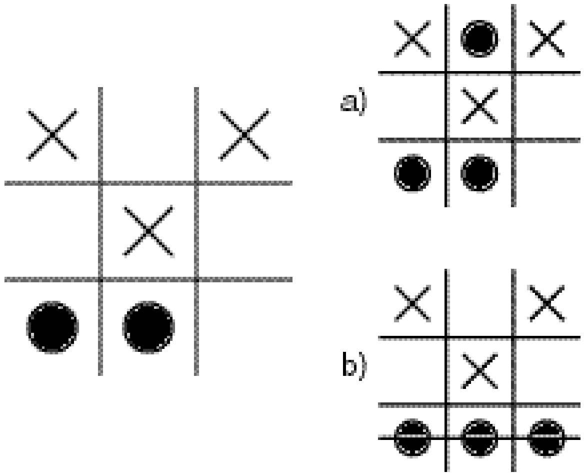

Figure 6.8, Board Position with O’s turn to move. (O would choose a. over the winning position b.)

As shown in figure 6.8, if TD(λ) (playing as O) had the opportunity to choose a winning move or a blocking move, it always elected the blocking move. This allows a human opponent to quickly develop a strategy of always placing pieces offensively to win and not concentrate upon preventing a computer win. In the cases in which the computer was able to both block and setup a winning position, the computer would always choose the winning position. So, if the computer placed a piece to block and in doing so had two potential winning moves remaining (always a winning position against any opponent), the computer would win upon its next move.

In reviewing this problem, I found the major shortcoming of the basic TD(λ) table-based implementation: the program must have an estimate for every single state in order to be effective. In every single case investigated in which the computer chose a defensive move over the winning move, the winning move’s board estimation was unknown. Unfortunately, even after 10 million iterations, there were still states missing from its estimation. Therefore, for a more comprehensive state space estimation, more exploration is needed. By increasing the random probability of an exploratory move did not assist with this problem.

Additionally, increasing the number of training iterations from one and one-half million to ten million made little difference. In fact, the problem that it may be more inherent with the random player. Essentially, the TD(λ) table-based algorithm can only estimate those states which it sees. In the case of a random player, with so many combinations, it is unlikely that the random player would show the TD(λ) player the “wiser” choices to be made.

To attempt exploration of all states, the results of the TD(λ) as the X player were used to train a new TD(λ) O player. The results for the O player were not promising. After one and one-half million iterations, X won 50%, O won 7%, and the game ended in a draw 43% of the time. In playing a game against this trained player, the human opponent won all 20 of the games. As a final experiment, TD(λ) as the X player was used to further tune the states learned from the O player, α=0.05, λ=0.5, tie=0.0 case. This case was trained for one and one-half million iterations. The results were worse than the previous experiment. The already trained O player could best the trained X player only 7.1% of the time, while losing 49.9% of the time. In fact, in pitting this previously decent opponent against the human opponent, the computer lost all 20 games.

Upon studying the resulting state estimations of those O players dominated by the trained TD(λ) X player it was found the O player was—quite literally—beaten into submission. It was so difficult for it to win a game initially that the only states it was able to update and accurately estimate were losing states. Eventually every move it was making was leading toward a “reward” of –1.0. This constant negative reinforcement pushed all states toward a negative goal, thus, figuratively crushed its spirit.

6.3. Back-Propagation and Neural Networks

Neural Networks are a method of fitting a non-linear function to closely approximate the problem domain. Neural Networks are composed of neurons that are connected in a layered network. For a network, a minimum of two layers is required: an input layer which is connected to the output layer. Every layer can have as many neurons as are required. These neurons are then connected to neurons in the previous (if any) and next (if any) layers. Each connection to other neurons is weighted. So, one neuron may have a stronger signal to a target neuron than a second neuron may have to the same target neuron.

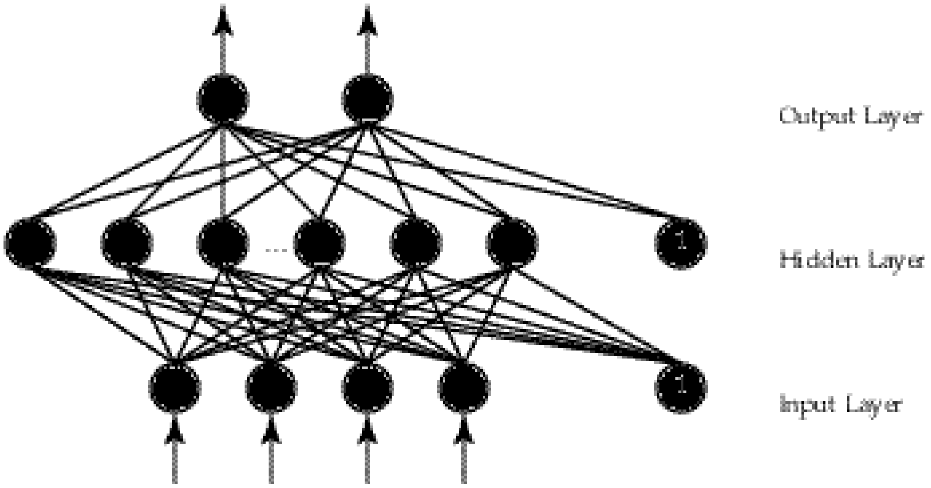

Figure 6.9, Typical Neural Network

As shown in figure 6.9, there is an input layer which “feeds” its activations to the hidden layer which, in turn, “feeds” activations to the output layer. On both the input and hidden layers, a bias is added which is a fixed neuron acting as a constant for the equation being approximated. Through back-propagation, a network can be adjusted so that it better approximates the result that is to be achieved. Back-propagation is a gradient-descent method that gradually lessens the output error. For instance, the error could be immediately calculated (if the expected results are known) and the weights adjusted accordingly;

however, to keep the network from becoming unstable, the error is gradually lessened. Once the error is known, the weights can be adjusted in the direction of the gradient from the output layer back down to the input layer, hence the name, back-propagation.

Typically, back-propagation is employed in supervised learning mode. During supervised learning, fixed input is fed to the network where the expected output is known. The expected output is then compared to the actual output of the network. The error is computed between the two values, and this error is propagated back through the network. This process continues until the network has correctly fit its function to generate the correct results. The hope, oftentimes, is that by training the network on many different inputs, it will eventually develop a generic function for most input cases—an example of this is Neurogammon.

To utilize a neural network, one typically performs three tasks:

- Feed the network the input data and retrieve the resulting output value(s). The output is the result of the non-linear function to which our neural network is computing.
- If training the neural network, compute the error between the expected and actual outputs
- Propagate the error back through the network, allowing the neurons to adjust their weights.

This process is repeated until the network no longer needs to be trained. Typically, a training session lasts for tens or hundreds of thousands of iterations until the network is sufficiently trained to generate the correct output for the given set of training input.

The first step is feed-forward. The input layer neurons are excited to some activation by the initial data. From these new values contained in the input layer, we feed the data forward to the hidden layer. The hidden layer's excited neurons can then feed their data forward to the output layer. The activations of neurons on one layer are dependent upon the weights and the activations of the previous layer. As we can see in equation 6.1, an activation is a function of the summation of all weighted neuron activations connected to it from the previous layer.

$$a_i^n(t) = f\left(\sum_{j \in n-1} \omega_{ij}^{n-1,n}(t) a_j^{n-1}(t) \right) \tag{6.1}$$

where,

$$f(x) = \frac{1}{1+e^{-x}} \tag{6.2}$$

therefore,

$$f'(x) = \frac{e^x}{\left(1+e^x\right)^2} = f(x)\left[1 - f(x)\right] \tag{6.3}$$

The term, $a_i^n(t)$, is defined to be the activation of neuron *i* on layer *n* at time, *t*. Conversely, ω_{ij}^{mn} is defined to be the weight between neuron *i* on layer *m* to neuron *j* on layer *n*. Equation 6.2 is the standard sigmoidal function for generating a neural activation between 0 and 1. Equation 6.3 is the derivative of equation 6.2 that will be used during back-propagation.

The feed forward step is very fast to compute, and, in fact, once a network has been trained, it can calculate advanced non-linear functions very rapidly. During training, after the activations of the output layer are computed, they are compared to the expected output to compute the error as shown in equation 6.4.

$$\delta_i^{\check{O}}(t) = \left(desired_i^{\check{O}}(t) - a_i^{\check{O}}(t)\right) \tag{6.4}$$

Where *desired* is the vector of expected outputs at time *t*, and $\check{O}$ is the vector of neurons on the output layer. To update the weights, the output layer errors are propagated to the previous layer. Once these weights are updated, the error is propagated back to the weights from the previous layer, and, so forth, until the input layer is reached. Equation 6.5 is the weight update method.

$$\omega_{ij}^{mn}(t+1) = \omega_{ij}^{mn}(t) + \alpha \nabla_{w^{mn}} \delta(t) \tag{6.5}$$

The gradient of the weights with respect to the error must be found to solve equation 6.5. The learning rate is defined as α. Substituting the chain rule into equation 6.5 results in:

$$\omega_{ij}^{mn}(t+1) = \omega_{ij}^{mn}(t) - \alpha \frac{\partial \delta_k(t)}{\partial \omega_{ij}^{mn}(t) a_i^m(t)} \frac{\partial \omega_{ij}^{mn}(t) a_i^m(t)}{\partial \omega_{ij}^{mn}(t)} \tag{6.6}$$

Which reduces to:

$$\omega_{ij}^{mn}(t+1) = \omega_{ij}^{mn}(t) + \alpha \sigma_j^n(t) a_i^m(t) \tag{6.7}$$

where,

$$\sigma_j^n(t) \equiv \frac{\partial \delta_k(t)}{\partial \omega_{ij}(t) a_i(t)} \tag{6.8}$$

The back-propagation process computes $\sigma_j^n(t)$ as shown in 6.9.

$$\sigma_i^n(t) = \begin{cases} \delta_i^n(t) f'\left(a_i^n(t)\right) & \textit{where } n = \check{O} \\ \left(\sum\limits_{j \in n+1} \sigma_j^{n+1} \omega_{jk}^{n,n+1}\right) f'\left(a_i^n(t)\right) & \textit{otherwise} \end{cases} \tag{6.9}$$

The general back-propagation of neural networks has become quite popular for many tasks that do not fit a standard, linear function or tasks that require constant modification of the parameters such as visual recognition, speech recognition, and others (including the aforementioned Neurogammon).

6.4. Temporal Difference and Neural Networks

The temporal difference method for updating states can also be used to help solve a gradient descent method as shown in section 4.6. The TD(λ) method for neural networks; however, is slightly different from the standard back-propagation method. In the standard back-propagation, an error is propagated back to every neuron, which, in turn, adjusts its weights for every other neuron signal that contributed to that error. In the case of utilizing TD(λ), the back-propagation is used to generate the eligibility of each neuron. After each time step, the temporal difference is computed and adjusts each neuron based upon their eligibility to the error. These eligibilities determine which neuron weights contribute to the current temporal difference, allowing the properly assigned update.

The TD(λ) gradient descent method follows the same form as with the standard neural network; however, to correspond with the method, $V_i^m(t)$ is used to represent the activation for neuron *i* on layer *m* at time *t*, and $\theta_{ij}^{mn}(t)$ represents the weight between neuron *i* on layer *m* to neuron *j* on layer *n* at time *t*. The feed-forward step as shown in equation 6.10 is identical to equation 6.1 with the new variable definitions.

$$V_i^n(t) = f\left(\sum_{j \in n-1} \theta_{ij}^{n-1,n}(t) V_j^{n-1}(t)\right) \tag{6.10}$$

where, *f* and *f'* are defined by equations 6.2 and 6.3.

Substituting into equation 4.5, the temporal difference error is computed by equation 6.11.

$$\delta_i^{\check{O}}(t) = r_i(t) + \gamma V_i^{\check{O}}(t+1) - V_i^{\check{O}}(t) \tag{6.11}$$

Where $r_i(t)$ is the reward, and γ is the discount parameter. The weights can be updated based upon equation 6.12, which is adapted from the temporal difference gradient descent method, equation 4.15.

$$\theta_{ij}^{mn}(t+1) = \theta_{ij}^{mn}(t) + \alpha\left(\delta_k^{\check{O}}\right) e_{ijk}^{mn}(t) \quad \textit{where} \;\; k \in \check{O} \tag{6.12}$$

Where $e_{ijk}^{mn}(t)$ is the eligibility of the connection between neuron *i* on layer *m* to neuron *j* on layer *n* to the output layer, unit *k*. Therefore, a change to an output unit affects all other neurons through their “eligibility” to that output node. The eligibility is determined by equation 4.16. Substituting for the gradient results in equation 6.13.

$$e_{ijk}^{mn}(t+1) = \gamma\lambda e_{ijk}^{mn}(t) + \sigma_{kj}^{n}(t+1) V_i^{m}(t+1) \tag{6.13}$$

Where $\sigma_{kj}^{n}(t)$ is computed with the recursive back-propagation method as defined by equation 6.14:

$$\sigma_{ij}^{mn}(t) = \begin{cases} f'\left(V_j^n(t)\right) & \mathit{if} \quad m = \underset{\mathrm{V}}{n} \\ 0 & \mathit{if} \quad m \in O \ \& \ m \neq n \\ \left(\sum_k \sigma_{ik}^{n,n+1} \theta_{jk}^{n,n+1} \right) f'\left(V_j^n(t)\right) & \mathit{otherwise} \end{cases} \tag{6.14}$$

Utilizing the gradient descent method as defined in section 4.6, combined with the standard back-propagation neural-network, a neural-network can employ real-time, reinforcement temporal difference learning. This alleviates the need for supervised learning (one of the major drawbacks to useful neural networks) allowing the network to “learn” on its own during the tasks it is set up to solve.

6.5. TD(Neural-Net) Implementation

To implement solving Tic-Tac-Toe with the temporal difference based backpropagation approach, a neural network player was added to the same code base as described in section 6.1. This gives the software four potential players to be tested and compared: human, random, TD(λ), and TD(Neural-Net). The TD(Neural-Net) player utilizes the neural network process described in section 6.4. The code is based upon Sutton’s pseudo-code for a temporal based neural network. The network is a three-layer network with a single hidden layer. The hidden layer size was set to eight neurons for each of the results. Both the input and output layers varied based upon each experiment. Additionally, the rewards were varied for each experiment.

The network was implemented using the standard function provided by Sutton [Sutton, 1992]. The first step, feed forward, is used to generate an estimate of the current state using the neural network. Using equation 6.10, Sutton created a subroutine (as shown in code 6.2) which will feed the current input layer through the network to calculate the current output layer.

```
sub feed_forward
    value := 0
    for j in hidden_layer
        for i in input_layer
            value := value + θ[i][j]*V[i]
        end for
        V[j] := f(value)
    end for
    value := 0
    for k in ouput_layer
        for j in hidden_layer
            value := value + θ[j][k]*V[j]
        end for
        V[k] := f(value)
    end for
end
```

Code 6.2, Subroutine for Computing the Output of the Network

As shown in code 6.2, each neuron’s weighted connection to the layer below is summed and computed with the sigmoidal function as defined by equation 6.2. After feeding the input layer through, the output layer can be used as the estimation for determining the next move. After a move is made the current state must be compared with the previous state estimated by the previous move. In addition to applying any reward received for the last move, the temporal error from the previous estimation to the current estimation must be computed using equation 6.11. This error is utilized to adjust the weights such that the estimation of the last move and the estimation of the current move both follow a continuous function.

```
sub update_weights
    for k in output_layer
        error[k] := r[k] + γ*V[k](t+1) – V[k](t)
        for j in hidden_layer
            θ[j][k] := θ[j][k] + β*error[k]*e[j][k]
            for i in input_layer
                θ[i][j] := θ[i][j] + α*error[k]*e[i][j][k]
            end for
        end for
    end for
end
```

Code 6.3, Subroutine for Computing the Change in Weights of the Network

Code 6.3 implements equation 6.12 to update the weights that connect every neuron in the network. In code 6.3, α and β are *learning rate parameters* which scale the error to follow a slower slope of the gradient-descent. The weight is changed slightly from its previous value by how much it contributed to the current error. How the weight contributed to the current error is determined by the eligibility. The eligibilities define how each neuron is connected to the output layer and how heavily that neuron's activation contributed to the error. The eligibilities are updated after each move as shown in code 6.4.

```
sub update_eligibility
    for j in hidden_layer
        for k in output_layer
            e[j][k] := γ*λ*e[j][k]+f′(V[k])*V[j]
            for i in input_layer
                e[i][j][k] := γ*λ*e[i][j][k] +
                    f′(V[k])*θ[j][k]*f′(V[j])*V[i]
            end for
        end for
    end for
end
```

Code 6.4, Subroutine for Backpropagation of Eligibility Values

As shown in code 6.4, the eligibility decays with the *trace decay parameter*, λ. This allows move contributions to decay over time (e.g., a move made at the initial point of the game may be less crucial to a win or loss than a move made at the end). The eligibility is changed slightly based upon is previous eligibility (scaled by the *trace decay*) and the change computed by the back-propagation procedure for solving the gradient descent. Equation 6.3 is employed to compute the derivative of our sigmoidal function. Because the activation (the result of equation 6.2) is known, the second form of the derivative using the values known for $f(x)$ is used.

There were three different networks developed to test how well TD(Neural-Net) could learn the game of Tic-Tac-Toe. The first was based directly upon the initial work of Tesauro with TD-Gammon. The second, still based upon the concepts of Tesauro's work was modified to better estimate the outcome of a Tic-Tac-Toe game. The third and final, was based upon TD(Neural-Net), but deviated with a different sigmoidal function and neural network computation result. Each network was tested with a constant *learning-rate parameter* of λ=0.5. Additionally, each network used a hidden layer size of eight neurons. Each network was trained against itself using varying rewards and lastly trained against a "learned" player from the TD(λ)-based table implementation in section 6.2.

The first experiment built is very similar to Tesauro's TD-Gammon. Two neurons represent each position on the Tic-Tac-Toe board. The first neuron is set to one if an X occupies that position or the second neuron is set to one if it is occupied by an O piece; otherwise, both are set to 0. This gives eighteen neurons with odd numbered neurons representing X and even representing O. The final two neurons tell the network which player has the current turn. The first neuron is set to one for X or the second neuron is set to one for O. The output layer consists of a single neuron which represented the probability of victory for the given state and player. It is hoped that the neural network would learn to be an accurate state estimator for any given Tic-Tac-Toe state. By continually feeding the game state, the network should return an estimation of the probability of victory. This allows the player to use the estimator to determine the better move. The rewards for this experiment were first tried as 0 for a loss, 0.5 for a draw, and 1 for a win. In the second trial, the rewards were changed to –1 for a loss, 0 for a draw, and 1 for a win.

The second neural experiment is based upon the first experiment with one change to the output layer. The output layer was increased to two neurons to estimate the likelihood of a win and the likelihood of a draw game. The input vector was created using the same process as described in the first neural network. The estimator is then used differently by the player. The player would feed forward a possible state and compare the likelihood of a victory vs. a draw game. The player would choose the outcome that had the greatest possibility of occurring. The rewards were computed differently in that there were two output neurons. If the outcome were a draw game, the neuron estimating a draw would be rewarded with a 1, and the neuron estimating a win would be rewarded with a 0. For a winning outcome, the rewards were reversed. In the case of a loss, both neurons were given a reward of a 0 or a –1 depending upon the test trial for obtaining results.

The final neural experiment was still based upon the TD(Neural Net); however, its function was slightly changed. Instead of being an estimator for the probability of a win, it was altered to give an estimation of the expected reward (similar to the method as studied in Section 6.2). There was a single output neuron that returned a result from –1 to 1 as the estimation of the final reward to be given. In order to achieve a result from –1 to 1, our sigmoidal function (equation 6.2) must be changed from its standard 0 to 1 form. Using equation 6.15, our neuron’s activation will range from –1 to 1.

$$f(x) = \frac{1 - e^{-x}}{1 + e^{-x}} \tag{6.15}$$

With the derivative being:

$$f'(x) = \frac{2e^{x}}{\left(1 + e^{x}\right)^{2}} = 2f(x)\left[1 - f(x)\right] \qquad (6.16)$$

Using equations 6.15 and 6.16 in the feed-forward and back propagation computations allows the network to vary from –1 to 1. The rewards were used exactly as in Section 6.2. In the case of a win, the player was given a reward of 1 and in the case of a loss, a –1 was rewarded. For a draw game, no reward was given.

6.6. Results of TD(Neural-Network)

The procedure to train the neural network is somewhat different than that of the TD(λ) procedure in section 6.1. The neural network requires a repetition of the states in order to begin to develop an estimation for that state. For this reason, the neural network cannot be trained against a random player. The neural network needs a constant stream of games that are played similarly for it to recognize and adapt its weights. The best source of endless games in such a manner is against itself. For all three of the neural network experiments described in section 6.5, the neural network was primarily trained against itself to achieve the results. In all cases, a test was also run to see how well the network would train against a known, average player. The source for average player is the TD(λ) player using results from section 6.2.

The first neural network is a direct implementation of the system Tesauro created for TD-Gammon adapted to the game of Tic-Tac-Toe. With separate input neurons representing each player and an output neuron intended to return the probability of victory for the given player, it was trained against itself for 500,000 iterations to test how well it had progressed.

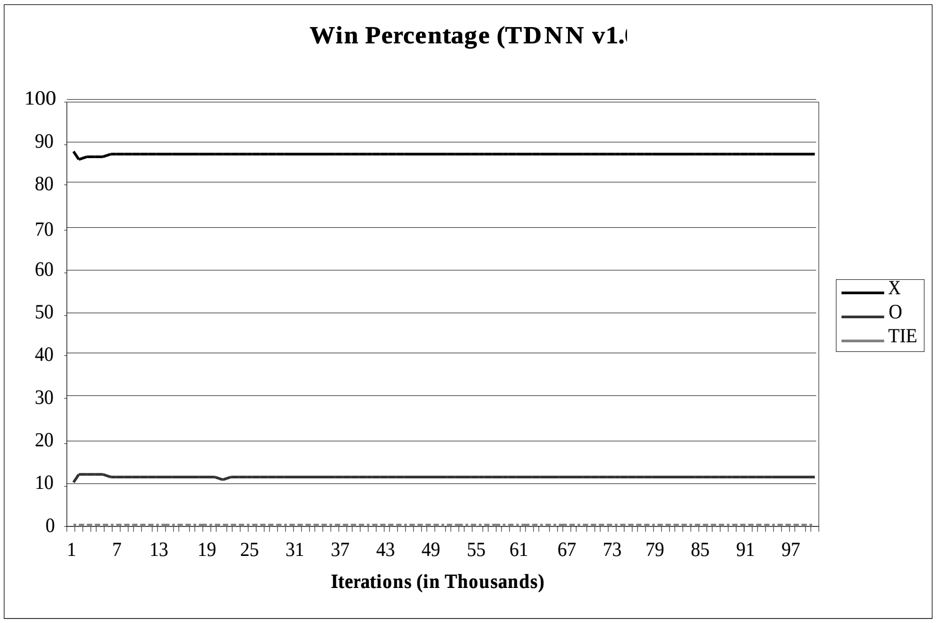

Figure 6.10, Percentage of Victories for TDNN v1.0

As shown in figure 6.10, the neural network did not seem to improve either of its players for the first 100,000 iterations (which are consistent for all iterations). This result was somewhat surprising as it would be assumed that as the neural network becomes a more accurate estimator and each player is utilizing that estimator, the number of tie games would rise dramatically approaching the vast majority of game outcomes. This did not occur in this case. In fact, closer inspection of the neural network at this time shows that the output of the network has asymptotically approached one. Every input fed through the network results in an output activation very close to one.

Inspection of the process used for TDNN v1.0 reveals that the sigmoidal function returns a result of 0.5 when the input is 0. Therefore, the initial activation is 0.5. Upon each iteration, more neurons are excited, which increases the activation, pushing it further toward one on the sigmoidal. The temporal error in that case is positive which forces the network to try and increase the positive value of each activation. More excited neurons are fed forward, exacerbating the issue. Because there are no negative rewards given to the network, there is never a negative temporal error and it therefore is always approaching one.

To try to alleviate this situation, TDNN v1.1 was altered to give different rewards. A reward of 1 was given to the network for a winning game, while a "correction" reward of –1 was given to the network for a losing game. In all other cases, a reward of 0 was given. TDNN v1.1 was then trained against itself for 500,000 games to see if it could return better results.

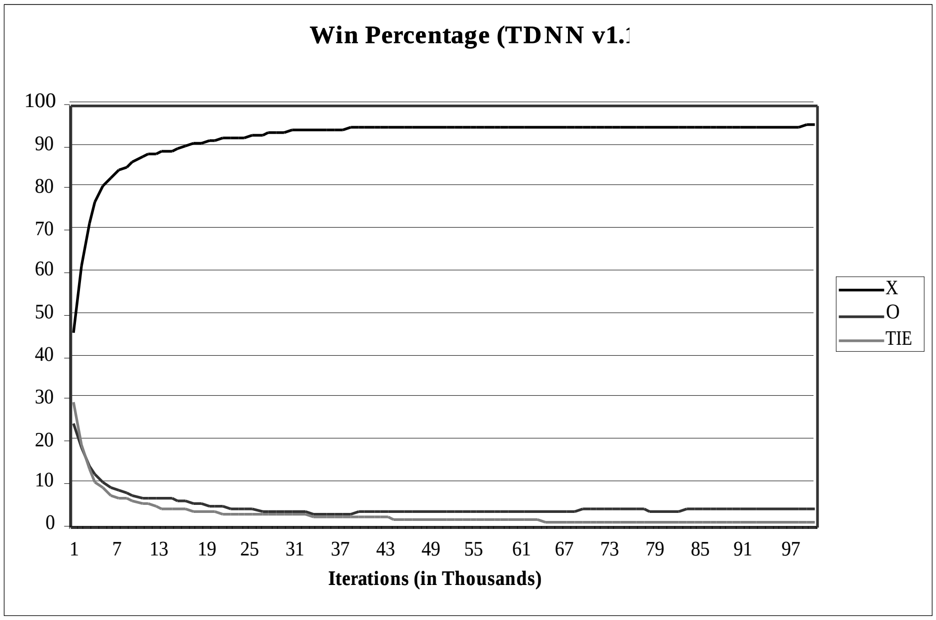

Figure 6.11, Percentage of Victories for TDNN v1.1

As shown in figure 6.11, TDNN v1.1 did not fare any better than TDNN v1.0. The results of TDNN v1.1 continued the trend in which the estimator cannot properly compute the state for a given board of Tic-Tac-Toe. The probability of winning function was not anywhere near approximating a valid probability for the given state. There appeared to be no learning of the Tic-Tac-Toe function (if one exists) at all.

The possibility existed that the network had fallen into an extreme local minimum and could not work its way out; thus, it was simply playing the same game (or few) over and over. I decided to attempt training the network against a player who had already learned the game. I used the α=0.05, λ=0.5 case from section 6.2. This player had faired well in the O position, so I trained the neural network playing the X position against the TD(λ) player.

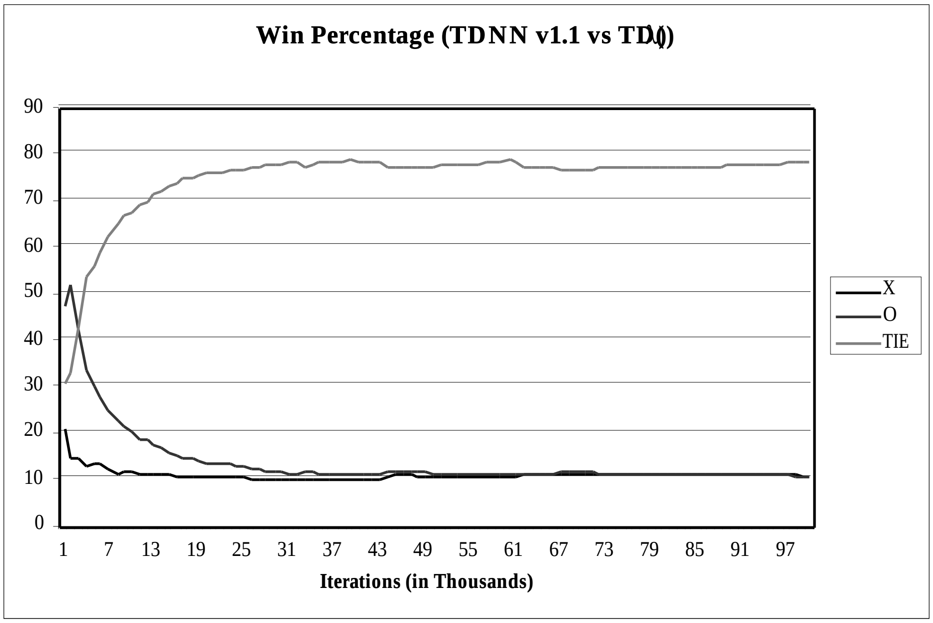

Figure 6.12, Percentage of Victories for TDNN v1.1 vs. TD(λ)

As shown in figure 6.12, the neural network did not perform well against a proper trainer. In fact, the network only won about 10% of the games, with another 10% ending as a draw, and the remaining 80% won by the inferior positioned O player. This proved to me that the network was not learning the game as presented to it.

Because of the failed attempts with TDNN v1.0 and v1.1, a change was made to the algorithm to attempt to better represent the game of Tic-Tac-Toe with the neural network. In the case of TD-Gammon, Tesauro's initial version contained a single ouput neuron to estimate the probability of a win. On later iterations, he added more neurons to represent the various outcomes of backgammon (victory, gammon, or backgammon). For TDNN v2.0, I added a second output neuron to represent the probability of a draw game. So, for TDNN v2.0, there were two output neurons, each representing the probability of occurrence. The first represented the probability of victory, the second, probability of a draw game.

After testing TDNN v2.0, its results were identical to TDNN v1.0. In fact, upon closer inspection of the output layer after complete training, the neuron representing the probability of a victory always returned a value near zero while the neuron representing a draw game returned a value near one. No matter the input, both neurons would return activations very close to these values. For similar reasons as the change from TDNN v1.0 to TDNN v1.1, I altered the rewards for the output neurons. If a victory was achieved, the neuron for victory was rewarded with a 1 while the draw game was given a –1. If a draw game occurred, the values were reversed. In the case of a loss, both neurons were given a reward of –1.

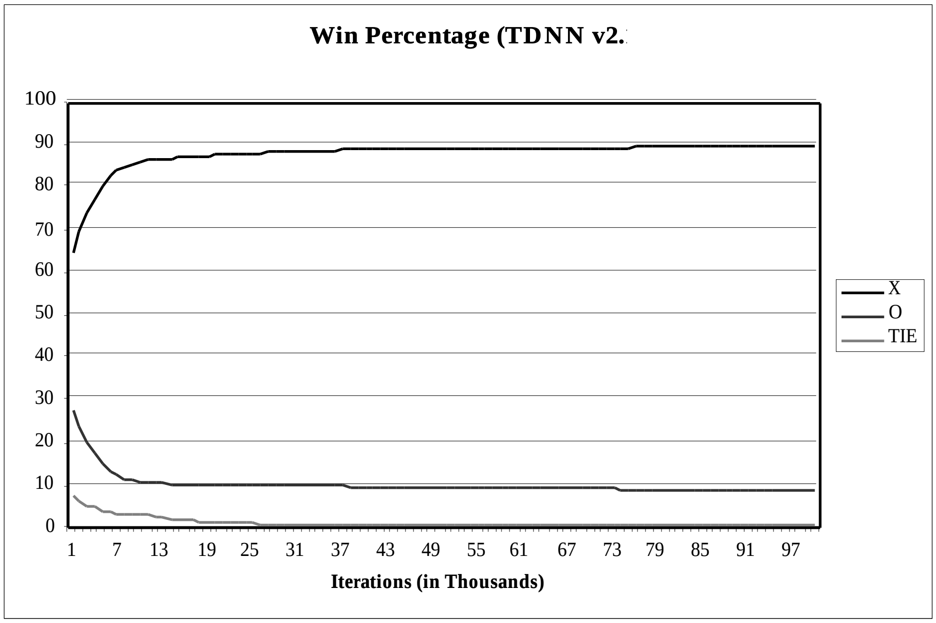

Figure 6.13, Percentage of Victories for TDNN v2.0

As shown in figure 6.13, TDNN v2.1 seems no different than TDNN v1.1. In fact, the results from TDNN v2.1 were no better than TDNN v1.1. The output layer seemed to have little effect on how the network responded to learning games presented to it. Again, a test was run to see if the network would perform better if trained against an average quality opponent. In tests versus the TD(λ) opponent, it was able to better play the opponent as it could play to a draw game as often as allowing the opponent to win the game as shown in figure 6.14.

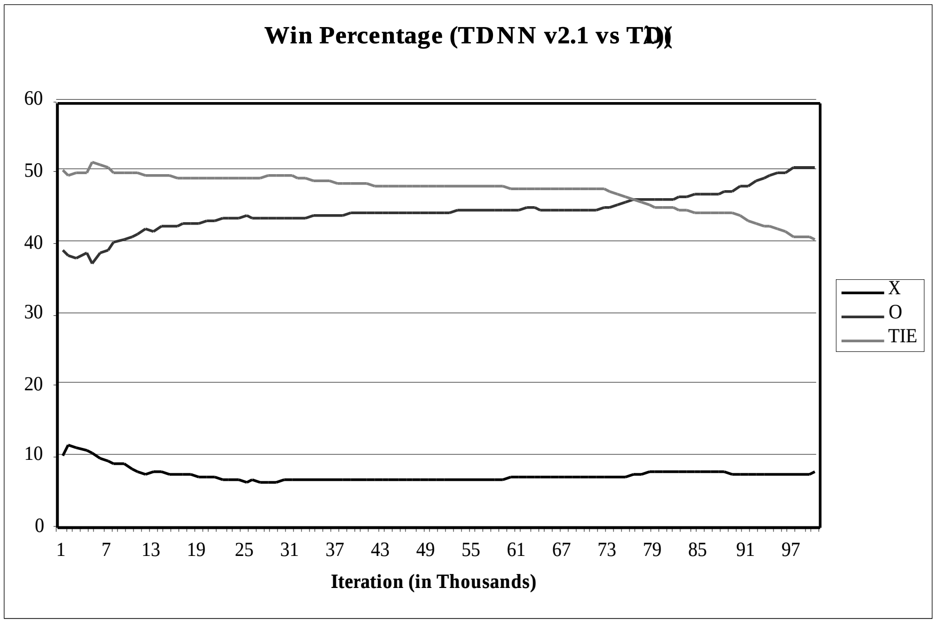

Figure 6.14, Percentage of Victories for TDNN v2.1 vs. TD(λ)

Because the results for TDNN v1 and v2 were not promising, a test was conducted to see how well each would perform against a human opponent. I played against each trained network for twenty games. The results of the tests against humans, based upon earlier results, were not surprising. As shown in table 6.2, only the player who fared well against the TD(λ) player was able to play a game that was not dominated.

TDNN	Number of Training Iterations	Games won by Human	Games won by TD(λ)	Draw Games
v1.0	500,000	20	0	0
v1.1	500,000	20	0	0
v1.1 trained against TD(λ)	500,000	20	0	0
v2.0	500,000	20	0	0
v2.1 *	500,000	20	0	0
V2.1 trained against TD(λ)	500,000	13	2	5

Table 6.2, TDNN vs. Human Opponent

Players marked with an asterisk (*) did not even follow the basic rule of Tic-Tac-Toe: as the O player, if X does not choose the centerpiece, O must take the center. Every single player seemed to follow the same pattern. It would always choose the same positions (if available) in the same order no matter what the opponent had chosen. These seemed to suggest either the computer was not learning anything at all or that the training games rarely were consistent (or possibly too consistent) to train the network to recognize a pattern.

I decided to break somewhat from the basic method outlined as the TD(λ) gradient descent method with neural networks. I changed the sigmoidal function to return a value from –1 to 1 so that the reward estimation could be accurately predicted as was performed with the TD(λ) player in section 6.2. At first, the input layer was altered for TDNN v3.0. The input vector was comprised of 10 neurons. A single neuron represented each board position and the final neuron represented which player had the current turn. A –1 represented the O player while the X player was represented with a 1. If no player occupied the space, it was left as 0. This input vector, however, proved to be flawed because when training against itself, the same board position would be presented (with either a positive or negative activation) for each player with a different reward. The network would try to correct its error from a 1 reward, then, attempt to correct the error from a –1 reward resulting in very large offsetting weights which quickly became too large for a 32-bit processor to handle. Therefore, TDNN v3.0 was returned to using the standard input vector as used for every other version.

As with previous versions, TDNN v3.0 was trained against itself for 500,000 iterations. The results were more consistent with expectation in that the number of ties was more recurring than in previous attempts. The X player still held the major advantage and won most of the games; however, the number of draw games steadily increased as shown in figure 6.15.

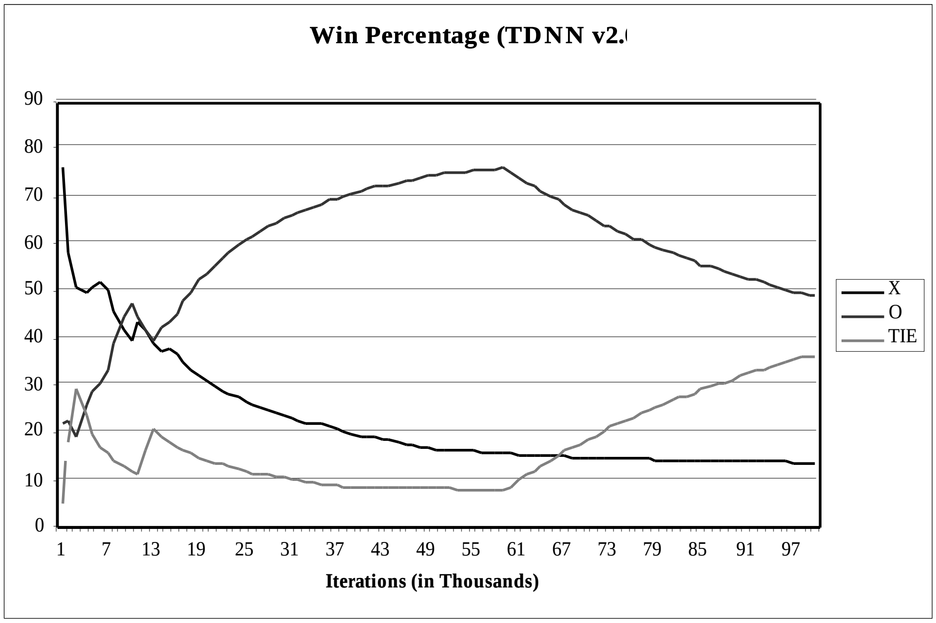

Figure 6.15, Percentage of Victories for TDNN v3.0

Based on how well it trained against itself, the hope was that it would fair well against a trained opponent. Playing TDNN v3.0 against the TD(λ) player would test how well the player had learned the game. As figure 6.16 shows, however, the results were much poorer than expected. In fact, the TDNN v3.0 player did not play as well against a trained opponent as the earlier TDNN v2.1 had.

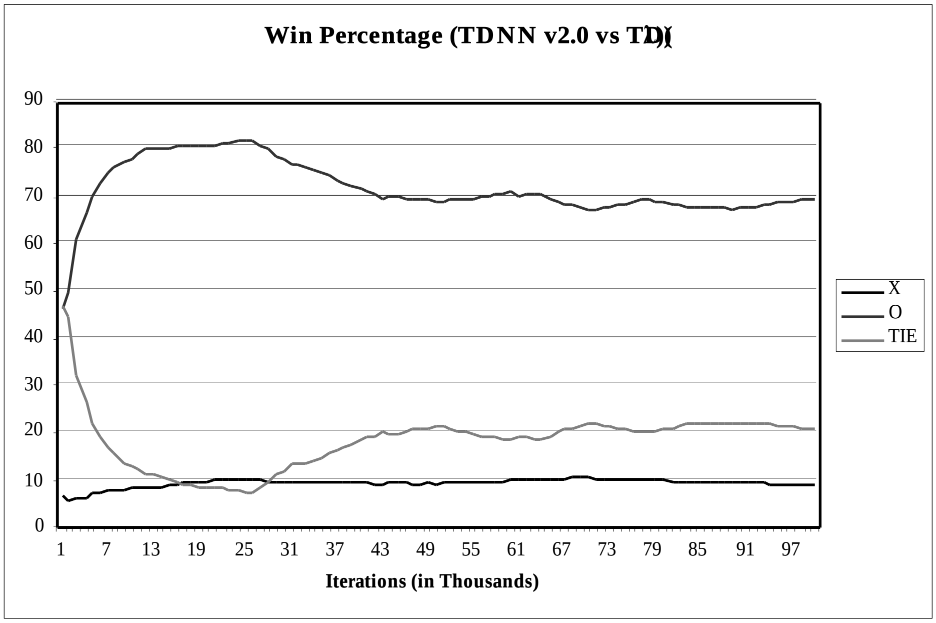

Figure 6.16, Percentage of Victories for TDNN v3.0 vs TD(λ)

Because TDNN v3.0 was based on a concept more similar to that of TD(λ) with a reward estimator for a given state, it was hoped that it would learn to compute (or estimate) a given state as opposed to storing a value such as the TD(l) player. Unfortunately, it was not able to learn the function. Even after 1.5 million iterations, it played a rather poor game. As table 6.3 shows, the various TDNN v3.0 networks were not able to perform at a level at which it could compete with a human.

Trainer	Number of Training Iterations	Games won by Human	Games won by TD(λ)	Draw Games
Itself	500,000	20	0	0
TD(λ)	500,000	20	0	0
Random	1,500,000	20	0	0

Table 6.3, TDNN v3.0 vs. Human Opponent

Inspection of the reward estimations for TDNN v3.0 revealed a similar problem to what had occurred with TDNN v1.0. All states fed forward through the input resulted in an estimation that approached 1. One primary difficulty with a neural implementation of Tic-Tac-Toe is that for a given input and a given output, one minor change in the input (changing the user who is currently at play) must force the network to "invert" the estimation. In other words, for a given board state for Player X, the estimation for a reward may be 0.78; however, feed the same state through the estimation for Player O, and the reward estimation would be roughly –0.78 because the likelihood for Player O is a loss.

6.7. Tic-Tac-Toe Conclusions

In the Tic-Tac-Toe problem domain, there were two distinctly different results in the use of the concepts of temporal difference. In the use of TD(λ) as a linear, table-based lookup method, the results were extremely positive, particularly, as player X in which the human opponent could not beat it. The second use of TD(λ) in the non-linear, neural network case did not achieve results which would be construed as "learning." Although, this was the case, it does not cast doubt upon the algorithm, more it requires thought as to the application.

Temporal difference methods as described by Sutton [Sutton, 1998] were primarily focused upon linearly independent input patterns. In the case of utilizing TD(λ) as a linear predictor, it worked quite well. In the case of utilizing TD(λ) with a non-linear neural network, the results were very poor. Possibly this speaks more toward the simplistic game of Tic-Tac-Toe. Tic-Tac-Toe is a very basic Markovian process (a state depends only upon the current state to reach the following) without non-linear aspects such as strategy, middle-games, etc. The application of neural networks to real-world problems requires selection of a proper problem, and the game of Tic-Tac-Toe may very well be a case in which the network did not fit the problem. In many cases, a problem that combines prediction and control would not converge with the TD(λ) algorithm. It can "get stuck in a self-consistent but non-optimal predictor/controller" [Tesauro, 1992].

One of the most important considerations in the implementation of a multi-layered neural network is that of data representation. A method with which to express the current state and extract meaningful results from the network as related to that state is different for every problem. If expressed incorrectly, the network may not have enough (or too many minor) data points to accurately understand the state or its outcome. Tesauro outlines two classifications [Tesauro, 1992]: a) *lookup table* in which there enough parameters such that the network may store the correct output for every state, and b) *compact* representation where the parameters are far fewer than the states and the network must derive the underlying task. In the second case, which is what was attempted with TD(Neural-Network) for Tic-Tac-Toe, the representation is all important. In fact, TD(λ) was described for the first case in which all states could be stored separately, and may not be suitable in standard form for the case in which it must compactly represent the model. Even with the compact model, the gradient-descent method is only capable of finding local optima for each state, not a global-state independent optimum.

Although the TD(Neural-Network) was not successful at implementing the game of Tic-Tac-Toe, the TD(λ) player did adapt quite well and was able to play a very reasonable game of Tic-Tac-Toe without an *a priori* knowledge of the underlying game. In the linear case, it was shown to be a very successful algorithm. In fact, a comparison between the TD(λ) implementation of Tic-Tac-Toe and of TD-Gammon reveals that an extremely complicated, non-linear function such as BackGammon performed extremely well utilizing TD(Neural-Network). It could, potentially, be implemented utilizing the TD(λ) linear based algorithm; however, with the number of possible states exceeding even that of Chess, it would be impractical and require years to train.

The reverse is true for TD(Neural-Network) applied to Tic-Tac-Toe. Tic-Tac-Toe has approximately 3,000 viable game states without a complicated game structure. It shows itself a very solvable problem with the linear TD(λ) player thus practically proving that it is too simple a problem to be solved with the multi-layered neural network. The problem domain being linear makes it difficult for the non-linear network to fit its hyper-plane to. The addition complexity of properly assigning all parameters within the neural network (*trace decay, discount*, *learning rate*, and others) make it very difficult to properly fit a function to the game of Tic-Tac-Toe.

The temporal difference methods have had many successful applications in recent years including robotic control, automated task scheduling, playing games such as BackGammon and Go at world-class levels, and channel allocation for cellular radio-based traffic. The concepts introduced with reinforcement learning and applied via the temporal difference methods have applications in most problem spaces; however, as was shown with the two examples exhibited, one must properly choose the temporal difference method for their particular application.

7. Conclusions

Mankind has tried to understand the learning process since the earliest days of history. To understand how we learn is to understand, possibly, what most gives us our most human of attributes: the ability of abstraction. People have dreamt of building machines that could think independently since the days of Pascal and later Babbage. To achieve this goal, researchers have closely followed the advancements of biologic learning research. De Morgan eventually applied this research in developing the "*laws of thought* and thus took the first step towards AI software" [Hofstadter, 1979].

As machine learning techniques progress, those which require the least supervision and learn on their own are the methods that are typically seen as holding the most promise. Samuel's method of searching the tree space for all possible moves in conjunction with an adaptive state estimator is viewed as an historical revolution in artificial intelligence. Now, with faster, cheaper, and larger capacity hardware, programs such as Deep Blue can beat the top chess player in the world utilizing look-ahead algorithms. When humans play games, they employ basic look-ahead algorithms to formulate strategy; however, this is where we differ from the machine. A human may abstract, formulate, and strategize whereas the machine is merely performing a look ahead to find the optimal board position for itself. How similar is a look-ahead of board estimations to a human playing the game? Quite similar as a matter of fact. The primary difference, however, is in the state estimation being performed. Although we do evaluate state estimations, it tends to be highly abstracted (oftentimes, humans will sacrifice valuable pieces as a way to "trick" our opponent). This is where TD-Gammon excelled.

TD-Gammon introduced the first major successful application of the concept of reinforcement learning. By watching the game and comparing where you are with where you thought you would be in the past allows us to develop a method to learn independently of a teacher or supervisor. TD-Gammon was able to learn the game of backgammon in such a way that it could accurately predict superior board positions (even if it may have appeared inferior). In fact, although the estimated probability value by TD-Gammon may have been incorrect, every estimation was consistent with the other. This parallels human state estimation. Every person estimates their current states differently than another; however, people are consistent with themselves. Despite this error in probabilistic calculation, TD-Gammon, according to top players in the world, was unusually cunning in its move selections, often seemingly attempting to lull its opponent into a trap.

Although the success of TD-Gammon cannot directly translate to other problem domains, such as Tic-Tac-Toe, the basic principles of reinforcement learning, developed by Sutton and employed by Tesauro, have proven to be an important new direction for the research of machine learning. As with all other aspects of artificial intelligence, ideas are "borrowed" from many different disciplines and adapted to fit machine learning model. Temporal difference with its roots in dynamic control systems has given us a way to learn on-line; to learn by doing—by exploring. It is this exploration—inquisitiveness—which allows humans to broaden their knowledge of the world around them. It is what makes the temporal difference method rather unique in the field. Samuel first realized the importance of exploration in his work before the concept began to again attract attention in the late 1980's. Independent exploration is an extremely important aspect in biological learning. Curiosity may have killed the cat, but it certainly learned about its environment along the way.

Alan Turing created the “imitation game” [Hofstadter, 1979] which we now simply call the *Turing Test.* The premise was that if a computer could “converse” with a human and the human was incapable of determining whether they had held a conversation with a computer or another human, then true intelligence has been achieved. Has TD-Gammon (or Deep Blue, or other championship game playing algorithms) passed the *Turing Test*? An argument could be made that, indeed, they have certainly passed the test. If an expert was to play a remote game of backgammon against another expert or against TD-Gammon and could not distinguish between the two, then at least for the domain of BackGammon, the test has been passed.

It is interesting that all definitions of intelligence come back to tests of how well they relate to human intelligence. Maybe this is the key to why mankind pushes for machine learning. For, the final outcome would finally allow us to “play god” by creating a machine in the image of ourselves.

Appendix A. MDP Code

The following is the source code in C++ used to implement the random walk example in Chapter 4.

```
/*************************************************************************
 *  This program will perform the MDP random walk with either TD(0),
 *  Monte-Carlo (MC), or TD(l).  It performs a certain number of
 *  iterations (each iteration executes until an endpoint is reached).
 *
 *  Written by Brian Powell
 *  December, 1999
 *  for Master's Thesis
 *  University of Colorado at Denver
 *
 *************************************************************************/
#include <iostream>
#include <math.h>
#include <unistd.h>
#include <vector>

using namespace std;  //introduces namespace std

// Constants defining our solution method
enum {
  gTD_0 = 1,
  gMC,
  gTD_l
};

// The main subroutine for the application
int main()
{
  cout << "Compute our random walking example with TD(0), TD(l), or MC" << endl;

  // Seed the random generator
  srand(getpid());

  // Declare our variables
  int method = gTD_0;           // Which method do we wish to use?
  int num_states = 7;           // How many states do we have(+2 for the goal
states)
  int iterations = 100;         // How many iterations should we perform?
  double alpha = .01;           // The alpha value to use during computation
  double gamma = 1.0;           // The gamma value to use during computation
  double lambda = 0.0;          // The trace decay parameter for TD(lambda)

  // Set everything up: Initialize everything to the .5 estimation
  vector<double> states(num_states, 0.5);
  // Ask the user which method to perform
  cout << "Would you like to perform: " << endl << "1) TD(0)" << endl;
  cout << "2) MC" << endl << "3) TD(l)" << endl << "[1] ?";
  cin >> method;
  cout << "Performing " << ( (method==1) ? "TD(0)" : ((method==2) ? "MC" :
"TD(l)") ) << endl;
```

```
  // If we are using TD(lambda), we need to find out the lambda value to use
  if ( method == gTD_l ) {
    cout << "Lambda Value: ";
    cin >> lambda;
    cout << "using l=" << lambda << endl;
  }

  // Go through all of the iterations
  for ( int iter=0; iter<iterations; iter++ ) {
    // The goal states have no estimation
    states[0] = states[num_states-1] = 0.0;

    // Where we currently our
    int current = (num_states+1)/2;

    // Store the moves we made
    vector<int> moves;

    // Create the eligibility trace
    vector<double> e(num_states, 0.0);

    // Let's print out everything thus far and compute the RMS-error
    double rms_error = 0.0;
    cout << iter << ",";
    for (int i=1; i<num_states-1; i++) {
      double tmp = states[i] - ( (double)i/double(num_states-1) );
      rms_error += tmp*tmp;
      cout << states[i] << ",";
    }
    cout << sqrt(rms_error/double(num_states-2)) << endl;

    // Set the default reward
    double r = 0.0;

    // Randomly walk until we reach an end point
    while ( current && current < num_states-1 ) {
      // randomly decide where to go next
      int next = int(rand()*100/RAND_MAX);
      next = ( next%2 ) ? current + 1 : current - 1;

      // Are we at the reward point?
      if ( next == num_states-1 ) r = 1.0;

      // Perform the action based upon our method
      switch ( method ) {
        case gTD_0:
          // Update our value based upon the immediate reward
          states[current] = states[current] +
                            alpha*(r + gamma*states[next] - states[current] );
          break;
        case gMC:
          // Just push this move onto our stack of moves
          moves.push_back(next);
          break;
        case gTD_l:
          // Update our value based upon the immediate reward and update all
previous
          // eligibilities
          double d = r + gamma*states[next] - states[current];
          e[current] += 1.0;
          moves.push_back(current);
          for (int j=0; j<moves.size(); j++) {
            int x = moves[j];
            states[x] += alpha*d*e[x];
            e[x] = gamma*lambda*e[x];
          }
          break;
        default:
          break;
      }

      // Move to the next state and continue
      current = next;
    }

    // If we are performing MC, we need to go back through our moves, and update
based
    // upon the final reward
    if ( method == gMC ) {
      current = (num_states+1)/2;
      for (int i=0; i<moves.size(); i++) {
        states[current] = states[current] + alpha*(r - states[current]);
        current = moves[i];
      }
    }
  }

  return 0;
}
```

Appendix B. Tic-Tac-Toe Code

The following code is the C++ implementation of the Tic-Tac-Toe problem as described in Chapter 6. It consists of several C++ classes fully implemented within their respective header file and a single main.cc file as the primary program handler.

The files, listed in alphabetic order with source files first are:

- main.cc
- HumanPolicy.h
- matrix.h
- Player.h
- Policy.h
- RandomPolicy.h
- TDlambdaPolicy.h
- TDNeuralPolicy.h
- TDSuttonNet.h

Appendix B.1. main.cc

The file, main.cc, is the primary entry point and flow control for the Tic-Tac-Toe program.

```
/*************************************************************************
 *  Tic-Tac-Toe Research Code:  This code implements the game of
 *  Tic-Tac-Toe in any game board size.  It is based upon a Policy object
 *  which dictates how a particular player object plays.  The defined
 *  policies include: random, human, TD-lambda, and TD-Neural Net.
 *
 *  Written by Brian Powell
 *  January, 2000
 *  for Master's Thesis
 *  University of Colorado at Denver
 *
 *************************************************************************/

#include <iostream>
#include <fstream>
#include <math.h>
#include <string>
#include <unistd.h>
#include <time.h>
#include <vector>
#include <map>
#include "matrix.h"
#ifdef macintosh
#include <console.h>
#endif

using namespace std;  //introduces namespace std

// Declarations
enum {
  kEmpty = -1,
  kPlayerX = 0,
  kPlayerO,
  kNoWin,
  kDraw
};

// Type Definitions
typedef matrix< int > tBoard;
typedef map< string, double , less< string > > tBoardMap;

// Global Variables
double  gAlpha = 0.05;
double  gBeta = 0.1;
double  gGamma = 1.0;
double  gLambda = 0.0;
bool    gDebug = false;
int     gGameSize = 3;
int     gInputSize = gGameSize*gGameSize*2+2;
int     gHiddenSize = 8;
int     gOutputSize = 1;
```

```
// Headers Required
#include "Player.h"
#include "Policy.h"

// Prototypes
Policy *create_policy(string name, string file);
int game_over(tBoard &board);
void display_board(tBoard &board);

// Policies we use
#include "HumanPolicy.h"
#include "RandomPolicy.h"
#include "TDlambdaPolicy.h"
#include "TDNeuralPolicy.h"

// "Local" Globals
TDSuttonNet   *gNeuralNet = NULL;

/////////////////////////////////////////////////////////////////////
// Main
//
// The entry point for the program
/////////////////////////////////////////////////////////////////////
int main( int argc, char **argv )
{
#ifdef macintosh
  argc = ccommand(&argv);
#endif

  // Set up
  srand(time(NULL));
  int     iterations = 10;
  string  player1 = "h";
  string  file1;
  string  player2 = "r";
  string  file2;

  // Get the input from the user
  int cur_argc = 0;
  while ( ++cur_argc < argc ) {
    if ( *argv[cur_argc] != '-' ) continue;
    switch ( argv[cur_argc][1] ) {
      case 'a':
        if ( ++cur_argc < argc ) {
          gAlpha = atof(argv[cur_argc]);
        }
        break;
      case 'b':
        if ( ++cur_argc < argc ) {
          gBeta = atof(argv[cur_argc]);
        }
        break;
      case 'd':
        gDebug = true;
        break;
      case 'g':
        if ( ++cur_argc < argc ) {
          gGamma = atof(argv[cur_argc]);
        }
        break;
      case 'i':
        if ( ++cur_argc < argc ) {
          iterations = atoi(argv[cur_argc]);
        }
        break;
      case 'l':
        if ( ++cur_argc < argc ) {
          gLambda = atof(argv[cur_argc]);
        }
        break;
      case '1':
        if ( ++cur_argc < argc ) {
          player1 = argv[cur_argc];
          if ( cur_argc+1 < argc && *argv[cur_argc+1] != '-' ) {
            file1 = argv[++cur_argc];
          }
        }
        break;
      case '2':
        if ( ++cur_argc < argc ) {
          player2 = argv[cur_argc];
          if ( cur_argc+1 < argc && *argv[cur_argc+1] != '-' ) {
            file2 = argv[++cur_argc];
          }
        }
        break;
      default:
        cerr << "Usage: " << argv[0] <<
         " [-a alpha]\n[-d]\n[-g gamma][-i iter]\n[-l lambda]\n" <<
          "[-1 type [ filename ] ] [-2 type [ filename ] ]\n"
          "where type is one of (h,r,t)" << endl;
        exit(-1);
        break;
    }
  }

  // Create the Policies
  vector< Policy* > policies;
  policies.push_back(create_policy(player1, file1));
  policies.push_back(create_policy(player2, file2));

  // Create the Players
  vector< Player > players;
  players.push_back(Player(kPlayerX));
  players.push_back(Player(kPlayerO));

  // Iterate over the number of times to play
  for (int played=1; played<=iterations; played++ ) {
    // Create the Board
    tBoard board(gGameSize,gGameSize,kEmpty);

    // Reset the policies
    for ( size_t i=0; i<policies.size(); i++ ) {
      policies[i]->reset();
    }

    // Reset the players
    for ( size_t i=0; i<players.size(); i++ ) {
      players[i].reset();
    }

    // Set the first player
    int last = kPlayerX;
    int current = kPlayerX;

    // Set up the winner
```

```
    int winner = kNoWin;

    // Play the game
    while ( winner == kNoWin ) {
      // Tell the player to move
      policies[current]->move(board, players[current]);

      // How are we doing?
      winner = game_over(board);
      if ( winner == players[current].piece()) players[current].winner(true);

      // Learn from this move
      policies[current]->learn(board,players[current],( winner != kNoWin ) );

      // Swap Players
      last = current;
      current = ( current == kPlayerX ) ? kPlayerO : kPlayerX;
    }

    // Update the winner and loser
    if ( winner != kDraw ) {
      players[last].won();
      players[current].lost();
    }
    else {
      players[last].tied();
      players[current].tied();
    }

    // The game is over, tell the loser about the outcome
    policies[current]->learn(board, players[current], true);

    // Display the results
    if ( played%1000 == 0 ) {
      cout << played << " " << players[0].wins() << " " << players[1].wins() <<
" ";
      cout << players[0].ties() << endl;
    }
  }

  // Print Stats
  cout << iterations << " Games Played" << endl;
  cout << "X Won " << players[0].wins() << "( " <<
    double(players[0].wins())/double(iterations)*100.0 << "% )" << endl;
  cout << "O Won " << players[1].wins() << "( " <<
    double(players[1].wins())/double(iterations)*100.0 << "% )" << endl;
  cout << "Tied " << players[0].ties() << "( " <<
    double(players[1].ties())/double(iterations)*100.0 << "% )" << endl;

  // Save the policies (if needed)
  for ( size_t i=0; i<policies.size(); i++ ) {
    policies[i]->save();
  }

  // Clean up
  delete gNeuralNet;

  return 0;
}

//////////////////////////////////////////////////////////////////////
// create_policy
//
```

```
// Create a policy based upon the user request
//////////////////////////////////////////////////////////////////////
Policy *create_policy(string name, string file)
{
  Policy *pol = NULL;
  switch (name[0]) {
    case 'h':
      pol = new HumanPolicy(file);
      break;
    case 'n':
      // The reason for this convoluted gNeuralNet is that we want to use
      // the same network for any of the Neural Policies we use
      if ( gNeuralNet == NULL )
        gNeuralNet = new TDSuttonNet(gInputSize, gHiddenSize, gOutputSize,
file);
      pol = new TDNeuralPolicy(gNeuralNet);
      break;
    case 'r':
      pol = new RandomPolicy(file);
      break;
    case 't':
      pol = new TDlambdaPolicy(file);
      break;
    default:
      cerr << "An Incorrect Policy was specified" << endl;
      exit(-1);
  };
  return pol;
}

//////////////////////////////////////////////////////////////////////
// display_board
//
// Print the board for the user
//////////////////////////////////////////////////////////////////////
void display_board(tBoard &board)
{
  string div = " ";
  int x = board.cols();
  int y = board.rows();
  cout << " ";
  for ( int i=0; i<x; i++ ) {
    cout << " " << i;
    div += "--";
  }
  cout << endl;
  for ( int j=0; j<y; j++ ) {
    cout << j << "|";
    for (int i=0; i<x; i++ ) {
      cout << ((board[i][j] != kEmpty) ?
              (board[i][j] == kPlayerX)?"X":"O" : " ") << "|";
    }
    cout << endl << div << endl;
  }
}


//////////////////////////////////////////////////////////////////////
// game_over
//
// Check to see if the game has ended due to a player winning
// or simply a tie game.
//////////////////////////////////////////////////////////////////////
```

```
int game_over(tBoard &board)
{
  int x = board.cols();
  int y = board.rows();
  bool match = false;

  // Check across each row
  for ( int j=0; j<y; j++ ) {
    match = true;
    int player = board[0][j];
    for ( int i=1; i<x; i++ ) {
      if ( board[i][j] != player || board[i][j] == kEmpty ) match = false;
    }
    if ( match ) return player;
  }

  // Check down each column
  for ( int i=0; i<x; i++ ) {
    match = true;
    int player = board[i][0];
    for ( int j=1; j<y; j++ ) {
      if ( board[i][j] != player || board[i][j] == kEmpty ) match = false;
    }
    if ( match ) return player;
  }

  // Check the diagonals if the board is larger than 2
  int player = board[0][0];
  match = true;
  for ( int i=0, j=0; i<x && j<y; i++, j++ ) {
    if ( board[i][j] != player || board[i][j] == kEmpty ) match = false;
  }
  if ( match ) return player;

  player = board[x-1][0];
  match = true;
  for ( int i=x-1, j=0; i>=0 && j<y; i--, j++ ) {
    if ( board[i][j] != player || board[i][j] == kEmpty ) match = false;
  }
  if ( match ) return player;

  // Check for any remaining spots
  for (int i=0; i<x; i++) {
    for (int j=0; j<y; j++) {
      if ( board[i][j] == kEmpty ) return kNoWin;
    }
  }

  // It is a draw
  return kDraw;
}
```

Appendix B.2. HumanPolicy.h

The file, HumanPolicy.h, defines a subclass of Policy which is responsible for interacting with a human player to determine the game playing policy.

```
/*************************************************************************
 * humanPolicy.h
 * Brian Powell, 1999
 * Master's Thesis, University of Colorado at Denver
 * Dr. William Wolfe, Advisor
 *
 * This class implements a policy of asking for input to allow for
 * human play.
 *************************************************************************/

#ifndef _HUMAN_POLICY_H
#define _HUMAN_POLICY_H

#include "Policy.h"

class HumanPolicy : public Policy {
public:
  // Default Constructor
  HumanPolicy(string filename) : Policy(filename) {};

  // Destructor
  ~HumanPolicy() {};

  // Allow the person to make a move
  void move( tBoard &board, const Player &plyr ) {
    // Show the person the board move
    display_board(board);

    // Input the location
    int x = -1;
    int y = -1;

    while ( x < 0 ) {
      cout << ((plyr.piece() == kPlayerX) ? "X" : "O") <<
        " Enter a Move (x y)> ";
      cin >> x >> y;
      if ( x > 2 || y > 2) x=-1;
      if ( x >= 0 && board[x][y] != kEmpty ) x=-1;
    }
    board[x][y] = plyr.piece();
  };

  // Humans can learn for themselves
  void learn( tBoard &board, const Player &plyr, bool game_over ) {
    if ( game_over && board.rows() ) {
      if ( plyr.winner() ) {
        cout << "Congratulations, You Won!" << endl;
      }
      else if ( plyr.loser() ) {
        cout << "Sorry, you lost." << endl;
      }
      else {
        cout << "Game ended in a Draw" << endl;
      }
    }
  };
};

#endif
```

Appendix B.3. matrix.h

The file, matrix.h, defines a basic C++ templated class for dealing with a two-dimensional array of values.

```
/*************************************************************************
 * matrix.h
 * Brian Powell, 1999
 * Master's Thesis, University of Colorado at Denver
 * Dr. William Wolfe, Advisor
 *
 * This class implements a basic matrix utilizing a vector of STL vectors.
 * It is used as a way to store and manage the data generated in
 * backprop.
 *************************************************************************/

#ifndef _MATRIX_H_
#define _MATRIX_H_

#include <vector.h>
#include <stdlib.h>

template < class T >
class matrix {

public:
  // Default Constructor
  matrix( int cols=1, int rows=1, T initial=0 ) :
      myData()
  {
    for ( int i=0; i<cols; i++ ) {
      myData.push_back( vector< T >((size_t)rows, initial) );
    }
  };

  // Copy Constructor
  matrix( const matrix &orig ) :
    myData( orig.myData )
  {
  }

  // Destructor
  ~matrix( void ) {
  }

  // Assignment Operator
  matrix& operator=( const matrix &orig ) {
    if ( this == &orig ) return *this;
    myData = orig.myData;
    return *this;
  }

  // Column & Row Accessor
  inline int cols( void ) const { return (int)myData.size(); };
  inline int rows( void ) const { return (int)myData[0].size(); };
```

```
// Resize the matrix
inline void resize( int x, int y ) {
  myData.erase( myData.begin(), myData.end() );
  for (int i=0; i<x; i++) {
    myData.push_back( vector< T >((size_t)y) );
  }
}

// Data Accessor
vector< T > &operator[]( int i ) {
  return myData[i];
}

// Output Operator
friend ostream &operator<<( ostream &out, const matrix &orig ) {
  int x = orig.cols();
  int y = orig.rows();
  out << x << " " << y << endl;
  for ( int j=0; j<y; j++ ) {
    for ( int i=0; i<x; i++ ) {
      out << (const T)orig.myData[i][j] << " ";
    }
    out << endl;
  }
  return out;
}

// Input Operator
friend istream &operator>>( istream &in, matrix &orig ) {
  int x, y;
  in >> x >> y;
  if ( x > 0 && y > 0 ) {
    orig.resize(x, y);
    for ( int j=0; j<y; j++ ) {
      for ( int i=0; i<x; i++ ) {
        in >> orig.myData[i][j];
      }
    }
  }
  return in;
}

// Equal Operator
friend bool operator==( const matrix &mat1, const matrix &mat2 ) {
  int x = mat1.cols();
  int y = mat1.rows();
  if ( x != mat2.cols() || y != mat2.rows() ) return false;
  for ( int i=0; i<x; i++ ) {
    for ( int j=0; j<y; j++ ) {
      if ( mat1.myData[i][j] != mat2.myData[i][j] ) return false;
    }
  }
  return true;
}

// Comparison Operators
inline friend bool operator<( const matrix &mat1, const matrix &mat2 ) {
  int x = mat1.cols();
  int y = mat1.rows();
  if ( x < mat2.cols() || y < mat2.rows() ) return true;
  for ( int i=0; i<x; i++ ) {
    for ( int j=0; j<y; j++ ) {
        if ( mat1.myData[i][j] < mat2.myData[i][j] ) return true;
      }
    }
    return false;
  }
  inline friend bool operator>( const matrix &mat1, const matrix &mat2 ) {
    return ! ( mat1 < mat2 );
  }

  // Utility methods

  // Compute the sum of the matrix
  inline T sum( void ) const {
    T total = 0;
    for ( int i=0; i<cols(); i++ ) {
      for ( int j=0; j<rows(); j++ ) {
        total += (const T)myData[i][j];
      }
    }
    return total;
  }

  // Compute the average value of the matrix
  inline T average( void ) const {
    return sum()/T( cols() * rows() );
  }

  // Randomize every value in the matrix with the given Max
  inline void randomize( T min, T max ) {
    T delta = max - min;
    for ( int i=0; i<cols(); i++ ) {
      for ( int j=0; j<rows(); j++ ) {
        myData[i][j] =  min + rand() * delta/RAND_MAX;
      }
    }
  }

private:
  vector< vector< T > >         myData;
};

template< class T >
class mat_compare {
public:
  bool operator()( const matrix< T > &mat1, const matrix< T > &mat2 ) const {
    return mat1 < mat2;
  }
};


#endif
```

Appendix B.4. Player.h

The file, Player.h, defines an abstract C++ class that defines the interface for dealing with a player of the game of Tic-Tac-Toe.

```
/*************************************************************************
 * Player.h
 * Brian Powell, 1999
 * Master's Thesis, University of Colorado at Denver
 * Dr. William Wolfe, Advisor
 *
 * This class implements a player contesting a game
 *************************************************************************/

#ifndef _PLAYER_H
#define _PLAYER_H

class Player {
public:
  // Default Constructor
  Player(int piece=kPlayerX) :
    myPiece(piece),
    myWins(0),
    myLosses(0),
    myTies(0),
    myWinner(false),
    myLoser(false) {};

  // Copy Constructor
  Player(const Player &orig) :
    myPiece(orig.myPiece),
    myWins(orig.myWins),
    myLosses(orig.myLosses),
    myTies(orig.myTies),
    myWinner(orig.myWinner),
    myLoser(orig.myLoser) {};

  // Destructor
  ~Player(void) {};

  // Utility methods

  // Set members
  inline void winner(bool val) { myWinner = val; myLoser = !val; };
  inline void loser(bool val) { myLoser = val; myWinner = !val; };
  inline void piece(int val) { myPiece = val; };
  inline void reset() { myWinner = myLoser = false; };

  // Get Members
  inline bool winner(void) const { return myWinner; };
  inline bool loser(void) const { return myLoser; };
  inline int piece(void) const { return myPiece; };

  // Stats
  inline int wins(void) const { return myWins; };
  inline void won(void) { myWins++; winner(true); };
  inline int losses(void) const { return myLosses; };
  inline void lost(void) { myLosses++; loser(true); };
  inline int ties(void) const { return myTies; };
  inline void tied(void) { myTies++; myLoser = myWinner = false; };

private:
  int     myPiece;                // The piece I control
  int     myWins;                 // How many times have I won?
  int     myLosses;               // How many times have I lost?
  int     myTies;                 // How many times have I tied?
  bool    myWinner;               // Did I win the current game?
  bool    myLoser;                // Did I lose the current game?
};

#endif
```

Appendix B.5. Policy.h

The file, Policy.h, defines an abstract C++ class that defines the interface for dealing with the policy for determining how a player attempts the game.

```
/*************************************************************************
 * Policy.h
 * Brian Powell, 1999
 * Master's Thesis, University of Colorado at Denver
 * Dr. William Wolfe, Advisor
 *
 * This class is an abstract interface to a policy.  The policy must
 * decide upon the move to make and (possibly) learn from that move.
 *************************************************************************/

#ifndef _POLICY_H
#define _POLICY_H

#include "Player.h"

class Policy {

public:
  // Default Constructor
  Policy(string filename="") : myFileName(filename) {};

  // Destructor
  virtual ~Policy() {};

  // Abstract Public Methods

  // Make a board move based upon this policy
  virtual void move( tBoard &board, const Player &plyr ) = 0;

  // Learn from the last move based upon this policy
  virtual void learn( tBoard &board, const Player &plyr, bool game_over ) = 0;

  // Reset anything that may need resetting
  virtual void reset(void) {};

  // Save anything that may need saving
  virtual void save(void) {};

protected:
  string    myFileName;       // The filename to load and save our data with
};

#endif
```

Appendix B.6. RandomPolicy.h

The file, RandomPolicy.h, defines a subclass of Policy that implements a playing policy of randomly choosing an open board position.

```
/*************************************************************************
 * RandomPolicy.h
 * Brian Powell, 1999
 * Master's Thesis, University of Colorado at Denver
 * Dr. William Wolfe, Advisor
 *
 * This class implements a Policy that simply selects a random free
 * board position.
 *************************************************************************/

#ifndef _RANDOM_POLICY_H
#define _RANDOM_POLICY_H

#include "Policy.h"

class RandomPolicy : public Policy {
public:
  // Default Constructor
  RandomPolicy(string filename) : Policy(filename) {};

  // Destructor
  ~RandomPolicy() {};

  // Allow the person to make a move
  void move( tBoard &board, const Player &plyr ) {
    // Input the location
    int x = -1;
    int y = -1;
    while ( x < 0 ) {
      x = int( ( rand() * 3.0 ) / RAND_MAX );
      y = int( ( rand() * 3.0 ) / RAND_MAX );
      if ( x > 2 || y > 2) x=-1;
      if ( x >= 0 && board[x][y] != kEmpty ) x=-1;
    }
    board[x][y] = plyr.piece();
  };

  // Random players have nothing to learn
  void learn( tBoard &board, const Player &plyr, bool game_over ) {
    if ( game_over && board.rows() ) {
      if ( plyr.winner() ) {
      }
      else if ( plyr.loser() ) {
      }
      else {
      }
    }
  };
};
#endif
```

Appendix B.7. TDlambdaPolicy.h

The file, TDlambdaPolicy.h, defines a subclass of Policy that implements a playing policy for the TD(λ) algorithm. This policy adapts its weights upon later estimations based upon the reward.

```
/*************************************************************************
 * TDlambdaPolicy.h
 * Brian Powell, 1999
 * Master's Thesis, University of Colorado at Denver
 * Dr. William Wolfe, Advisor
 *
 * This class implements the TD(lambda) algorithm as an estimator in my
 * policy for the game.  If no lambda is specified, it runs as TD(0).
 * If lambda = -1.0 then we are not learning, but simply using our policy.
 *************************************************************************/

#ifndef _TDLAMBDA_POLICY_H
#define _TDLAMBDA_POLICY_H

#include "Policy.h"

const double kInitialize = 0.0;  // Initialize each state

class TDlambdaPolicy : public Policy {
public:
  // Default Constructor
  TDlambdaPolicy(string filename) :
    Policy(filename),
    myV(),
    myE(),
    myMoves(),
    myLearning(true),
    myCurrent(),
    myNext()
  {
    if ( myFileName.length() ) {
      ifstream in((char *)myFileName.data());
      if ( in ) {
        // Read in each record
        while ( ! in.eof() ) {
          string board;
          double value;
          in >> board >> value;
          myV[board] = value;
        }
      }
      cout << "SIZE: " << myV.size() << endl;
    }
    // Set if we are learning
    myLearning = ( gLambda >= 0.0 );
  };
  // Destructor
  ~TDlambdaPolicy() {};

  // We need to figure out the best move
  // We will typically grab the move with the highest state
  // of probability for winning; however, we need to explore
  // our space as well; therefore, we will (at random intervals)
  // choose a move which may not be optimal.  We will only do this
  // during learning.
  void move( tBoard &board, const Player &plyr ) {
    int rows = board.rows();
    int cols = board.cols();
    // This will randomly pick a number between 0 and 20.  If it is
    // 7 (and we are in learning mode) then we will simply pick
    // a random move.
    bool random = ( int( ( rand() * 20.0 ) / RAND_MAX ) == 7 ) && myLearning;

    // Debugging
    if ( gDebug ) cout << "Random : " << ((random)?"t":"f") << endl;

    // Perform TD
    if ( ! random ) {
      double best = -1.0;
      int x = -1;
      int y = -1;
      for ( int i=0; i<cols; i++ ) {
        for ( int j=0; j<rows; j++ ) {
          if ( board[i][j] == kEmpty ) {
            board[i][j] = plyr.piece();
            string str = board_string(board);
            double val = kInitialize;

            // Initialize unknown board positions
            tBoardMap::iterator iter = myV.find(str);
            if ( iter != myV.end() ) {
              val = myV[str];
            }

            // If this board position is the best so far
            if ( val != kInitialize && val > best ) {
              if ( gDebug )
                cout << "New Best: " << i << "," << j << " VALUE: " << val <<
endl;
              best = val;
              x = i;
              y = j;
            }

            // Set our board back to where we were and keep searching
            board[i][j] = kEmpty;
          }
        }
      }

      // See if we found a move, if not, it is random
      if ( x != -1 ) board[x][y] = plyr.piece();
      else random = true;
    }

    if ( random ) {
      if ( gDebug ) cout << "choosing randomly" << endl;
      vector < tBoard > moves;
      for ( int i=0; i<cols; i++ ) {
        for ( int j=0; j<rows; j++ ) {
          if ( board[i][j] == kEmpty ) {
            board[i][j] = plyr.piece();
            moves.push_back(board);
            board[i][j] = kEmpty;
          }
        }
      }

      // Randomly select from our boards
      int move = int( ( rand() * (double)moves.size() ) /  RAND_MAX );
      if ( moves.size() == 1 ) move = 0;
      board = moves[move];
    }

    // Save this status
    myNext = board_string(board);

    // Debugo
    if ( gDebug ) {
      cout << "Moved: " << myNext << " Estimate: " << myV[myNext] <<endl;
    }

    // Push this onto our moves
    myMoves.push_back(myNext);
  };

  // TDlambda learns from the previous moves
  void learn( tBoard &board, const Player &plyr, bool game_over ) {
    double reward = 0.0;

    if ( game_over ) {
      myNext = board_string(board);

      // Set the reward on whether or not the player won, lost, or tied
      if ( plyr.winner() ) reward = 1.0;
      else if ( plyr.loser() ) reward = -1.0;
      else reward = 0.0;
      if ( gDebug ) cout << "REWARD: " << reward << endl;
    }

    // If we are learning, and everything is up-to-date, adjust our states
    if ( myLearning && myCurrent != "" && myNext != "" ) {
      double delta = reward + gGamma*myV[myNext] - myV[myCurrent];
      myE[myCurrent] += 1.0;
      if ( gDebug ) {
       cout << "Current: " << myCurrent << endl;
       cout << "Next: " << myNext << endl;
       cout << "r=" << reward << " delta=" << delta << " e="
            << myE[myCurrent] << endl;
      }
      for (vector< string >::iterator i=myMoves.begin(); i!=myMoves.end(); i++)
{
        myV[*i] += gAlpha * delta * myE[*i];
        myE[*i] = gGamma * gLambda * myE[*i];
        if ( gDebug ) {
          cout << "STATE: " << myV[*i] << endl;
          cout << "ELIG: " << myE[*i] << endl;
        }
      }
    }
    myCurrent = myNext;
    myNext = "";
  };

  // Reset the current and previous states
  void reset( void ) {
    // Erase the eligibility traces
    myE.erase(myE.begin(), myE.end());

    // Erase the moves we made
    myMoves.erase(myMoves.begin(), myMoves.end());

    // Get rid of the history
    myCurrent = "";
    myNext = "";
  };

  // Save the states out to disk
  void save( void ) {
    ofstream out((char *)myFileName.data());
    if ( out && myLearning ) {
      // Go over the map, writing it out
      tBoardMap::iterator iter = myV.begin();
      while ( iter != myV.end() ) {
        if ( iter->second != kInitialize && iter->first != "") {
          out << iter->first << " " << iter->second << endl;
        }
        iter++;
      }
    }
  };

  string board_string(tBoard &board) {
    string str = "";
    for ( int j=0; j<board.rows(); j++ ) {
      for (int i=0; i<board.cols(); i++ ) {
        str += ((board[i][j] != kEmpty) ? (board[i][j] == kPlayerX)?"X":"O" :
"B");
      }
    }
    return str;
  }

private:
  tBoardMap         myV;        // The policy
  tBoardMap         myE;        // The eligibility trace
  vector< string >  myMoves;    // Track the moves we've made
  bool              myLearning; // Whether we are learning or not
  string            myCurrent;  // The current board I am working with
  string            myNext;     // The next board state I move to

};

#endif
```

Appendix B.8. TDNeuralPolicy.h

The file, TDNeuralPolicy.h, defines a subclass of Policy that implements a playing policy for the TD(Neural-Network) algorithm. This class uses an instantiation of the TDSuttonNet class as its estimator. The policy determines errors from previous state estimations based upon its current state and reward. This error is then fed to the TDSuttonNet to propagate backwards through the network.

```
/*************************************************************************
 * TDNeuralPolicy.h
 * Brian Powell, 2000
 * Master's Thesis, University of Colorado at Denver
 * Dr. William Wolfe, Advisor
 *
 * This class implements a Policy that simply selects a TDNeural free
 * board position.
 **************************************************************************/

#ifndef _TD_NEURAL_POLICY_H
#define _TD_NEURAL_POLICY_H

#include "Policy.h"
#include "TDSuttonNet.h"

class TDNeuralPolicy : public Policy {
public:
  // Default Constructor
  TDNeuralPolicy(TDSuttonNet *hopfield) :
    Policy(),
    myNet(hopfield),
    myLearning(true),
    myCurrent(),
    myNext()
  {
    // The current is the baseline of the net output
    myCurrent = hopfield->output();
  };

  // Destructor
  ~TDNeuralPolicy() {};

  // Allow the policy to make a move
  void move( tBoard &board, const Player &plyr ) {
    int rows = board.rows();
    int cols = board.cols();
    // This will randomly pick a number between 0 and 20.  If it is
    // 7 (and we are in learning mode) then we will simply pick
    // a random move.
    bool random = ( int( ( rand() * 20.0 ) / RAND_MAX ) == 7 ) && myLearning;
    // Debugging
    if ( gDebug ) cout << "Random : " << ((random)?"t":"f") << endl;

    // Perform TD
    if ( ! random ) {
      double best = -1.0;
      int x = -1;
      int y = -1;
      for ( int i=0; i<cols; i++ ) {
        for ( int j=0; j<rows; j++ ) {
          if ( board[i][j] == kEmpty ) {
            board[i][j] = plyr.piece();
            vector< double > vec = board_vector( board, plyr.piece() );
            myNet->input( vec );
            myNet->feed_forward();
//          double val = (myNet->output())[0];
            double val = max((myNet->output())[0], (myNet->output())[1]);

            // If this board position is the best so far
            if ( val > best ) {
              if ( gDebug )
                cout << "New Best: " << i << "," << j << " VALUE: " << val <<
endl;
              best = val;
              x = i;
              y = j;
            }

            // Set our board back to where we were and keep searching
            board[i][j] = kEmpty;
          }
        }
      }

      // See if we found a move, if not, it is random
      if ( x != -1 ) board[x][y] = plyr.piece();
      else random = true;
    }

    if ( random ) {
      if ( gDebug ) cout << "choosing randomly" << endl;
      vector < tBoard > moves;
      for ( int i=0; i<cols; i++ ) {
        for ( int j=0; j<rows; j++ ) {
          if ( board[i][j] == kEmpty ) {
            board[i][j] = plyr.piece();
            moves.push_back(board);
            board[i][j] = kEmpty;
          }
        }
      }

      // Randomly select from our boards
      int move = int( ( rand() * (double)moves.size() ) /  RAND_MAX );
      if ( moves.size() == 1 ) move = 0;
      board = moves[move];
    }

    // Save this status
    myNext = board_vector( board, plyr.piece() );
  };

  // Policy can learn from its moves
```

```
  void learn( tBoard &board, const Player &plyr, bool game_over ) {
    vector< double > reward((myNet->output()).size(), 0.0);

    if ( gDebug ) {
      cout << "in Learn" << endl;
      print_vector("Last Result: ", myCurrent);
    }

    if ( game_over ) {
      myNext = board_vector( board, plyr.piece() );

      // Compute the Reward
      if ( plyr.winner() ) {
        reward[0] = 1.0;       // Win
//        reward[1] = -1.0;      // Tie
      }
      else if ( plyr.loser() ) {
        reward[0] = -1.0;   // No Tie
//        reward[1] = -1.0;   // No Tie
      }
      else {
        reward[0] = 0.0;     // Tie
//        reward[1] = 1.0;      // Tie
      }

      print_vector("REWARD: ", reward);
    }

    // Get the output from our move
    myNet->input(myNext);
    myNet->feed_forward();
    vector< double > output = myNet->output();

    // If we are debugging, let's see what the output is
    print_vector("New Output", output);

    // Push the values through to see what happens

    // Create our TD error
    vector< double > error = output;
    for ( int k=0; k<(int)myCurrent.size(); k++ ) {
      error[k] = reward[k] + gGamma*output[k] - myCurrent[k];
    }

    if ( gDebug ) cout << "ERROR: " << error[0] << endl;

    // Update the weights based on our TD error
    myNet->update_weights(error);

    // Feed forward again
    myNet->feed_forward();

    // The current is now the last move's output we made
    myCurrent = myNet->output();

    // Debugging
    print_vector("Next: ", myCurrent);

    // Update the eligibilities
    myNet->update_eligibility();
  };

  // Reset the current and previous states
  void reset( void ) {
    // Erase the move history
    myCurrent.erase(myCurrent.begin(), myCurrent.end());
    myNext.erase(myNext.begin(), myNext.end());

    // Allow the network to reset
    myNet->reset();

    // Set the current to the opening output
    myCurrent = myNet->output();
  };

  // Create an input vector for the given board and player
  vector< double > board_vector( tBoard &board, int player ) {
    double kXvalue = 1.0;
    double kOvalue = 1.0;
    double kBvalue = 0.0;
    vector< double > board_vec((size_t)gInputSize, kBvalue);

    // go over the board and put a one in the positions occupied by each player
    for ( int j=0; j<board.rows(); j++ ) {
      for ( int i=0; i<board.cols(); i++ ) {
        int xpos = ( i+j*board.cols() ) * 2;
        int opos = xpos+1;
//        int xpos = ( i+j*board.cols() );
//        int opos = xpos;
        if ( board[i][j] == kPlayerX ) board_vec[xpos] = kXvalue;
        else if ( board[i][j] == kPlayerO ) board_vec[opos] = kOvalue;
      }
    }

    // Specify who's turn it is
    if ( player == kPlayerX ) board_vec[gInputSize-2] = kXvalue;
    else if ( player == kPlayerO ) board_vec[gInputSize-1] = kOvalue;

    return( board_vec );
  }

private:
  TDSuttonNet      *myNet;      // The neural net we are using
  bool              myLearning; // Whether we are learning or not
  vector< double >  myCurrent;  // The last output vector we had
  vector< double >  myNext;     // The next board evaluation I move to
};

#endif
```

Appendix B.9. TDSuttonNet.h

The file, TDSuttonNet.h, defines a class for handling and manipulating a gradient-descent based TD(λ) algorithm via a Neural Network. This is the code used for TDNN v3.0.

```
/*************************************************************************
 * TDSuttonNet.h
 * Brian Powell, 2000
 * Master's Thesis, University of Colorado at Denver
 * Dr. William Wolfe, Advisor
 *
 * This class implements a Hopfield-based neural network using temporal
 * difference errors and the backpropagation to compute the gradient of
 * the eligibility traces with respect to the error.  The network that
 * is implemented is a 3-layer (single hidden layer) network.
 ************************************************************************/

#ifndef _TD_SUTTONNET_H
#define _TD_SUTTONNET_H

#include <string>
#include <vector>
#include <fstream.h>
#include "matrix.h"
#include "math.h"

// Global methods

// Compute the sigmoidal output of a neuron
template < class T >
inline T output_function( T &x ) {
  double e = exp((double)-x);
  return T(1.0 - e)/T(1.0 + e);
}

// Compute the derivative value of the output
template < class T >
inline T output_function_prime ( T &x ) { return T(2.0)*x*(T(1.0) - x); }

// Debug function for displaying the contents of our vector
template < class T >
inline void print_vector( string msg, vector< T > &x ) {
  if ( gDebug ) {
    cout << msg;
    for(int i=0; i<(int)x.size(); i++) cout << x[i] << " ";
    cout << endl;
  }
}


class TDSuttonNet {
public:
  // Default Constructor
  TDSuttonNet( int input, int hidden, int out, string file = "",
               double bias = 1.0 ) :
```

```
  myInputSize(input+1),                      // Add one for the bias neuron
  myHiddenSize(hidden+1),                    // Add one for the bias neuron
  myOutputSize(out),
  myInputLayer((size_t)myInputSize, 0.0),
  myHiddenLayer((size_t)myHiddenSize, 0.0),
  myOutputLayer((size_t)myOutputSize, 0.0),
  myWeightInHid(myInputSize, myHiddenSize, 0.0),
  myWeightHidOut(myHiddenSize, myOutputSize, 0.0),
  myEligInOut(),
  myEligHidOut(myHiddenSize, myOutputSize, 0.0),
  myFileName(file),
  myBias(bias)
{
  // Construct the eligibility from the input to the output
  for ( int k=0; k<myOutputSize; k++ ) {
    myEligInOut.push_back( matrix< double >(myInputSize, myHiddenSize, 0.0) );
  }

  // Randomly build the weights
  myWeightInHid.randomize( -0.1, 0.1 );
  myWeightHidOut.randomize( -0.1, 0.1 );

  // If we have a file, we need to load our weights
  if ( myFileName.length() ) {
    ifstream in((char *)myFileName.data());
    if ( in ) {
      // Read in each of the weights
      in >> myWeightInHid >> myWeightHidOut;
    }
  }

  // Initialize everything
  initialize();

  if ( gDebug ) {
    cout << "INITIAL IN-HID WEIGHTS: " << myWeightInHid;
    cout << "INITIAL HID-OUT WEIGHTS: " << myWeightHidOut;
  }
};

// Destructor
~TDSuttonNet() {
  // If we have a file, we need to save our weights
  if ( myFileName.length() ) {
    ofstream out((char *)myFileName.data());
    if ( out ) {
      out << myWeightInHid << myWeightHidOut << endl;
    }
  }
};

// Accessors
inline void input( vector< double > &val ) {
  if ( val.size() == (size_t)myInputSize ) myInputLayer = val;
  else if ( val.size() == (size_t)myInputSize - 1 ) {
    myInputLayer = val;
    myInputLayer.push_back(myBias);
  }
}
inline vector< double > &input( void ) { return myInputLayer; };
inline vector< double > &output( void ) { return myOutputLayer; };

// Initialize the neural network
```

```
void initialize( void ) {
  feed_forward();
  update_eligibility();
}

// Feed forward the input through the network
void feed_forward( void ) {
  // Set the initial bias neurons
  myInputLayer[myInputSize-1] = myBias;
  myHiddenLayer[myHiddenSize-1] = myBias;

  // For Debug
  print_vector("Forward Input: ", myInputLayer);

  // Feed Forward from input to hidden layer
  for ( int j=0; j<myHiddenSize-1; j++ ) {
    double val = 0.0;
    for ( int i=0; i<myInputSize; i++ ) {
      val += myInputLayer[i] * myWeightInHid[i][j];
    }
    myHiddenLayer[j] = output_function(val);
  }

  // For Debug
  print_vector("Forward Hidden:", myHiddenLayer);

  // Feed forward from the hidden layer to output layer
  for ( int k=0; k<myOutputSize; k++ ) {
    double val = 0.0;
    for ( int j=0; j<myHiddenSize; j++ ) {
      val += myHiddenLayer[j] * myWeightHidOut[j][k];
    }
    myOutputLayer[k] = output_function(val);
  }

  // For Debug
  print_vector("Forward Out:", myOutputLayer);
}

// Update the weights using the back-propagation values of the eligibility
void update_weights( vector< double > &error ) {
  for ( int k=0; k<myOutputSize; k++ ) {
    for ( int j=0; j<myHiddenSize; j++ ) {
      if ( gDebug )
        cout << "DELTA: " << gAlpha * error[k] * myEligHidOut[j][k] << " ";
      myWeightHidOut[j][k] += gBeta * error[k] * myEligHidOut[j][k];
      for ( int i=0; i<myInputSize; i++ ) {
        myWeightInHid[i][j] += gAlpha * error[k] * myEligInOut[k][i][j];
      }
    }
  }
}

// Use backpropagation to find the eligibility gradient
void update_eligibility( void ) {
  vector< double >  output((size_t)myOutputSize, 0.0);

  // To repeat calculating this value again and again, we'll
  // store the output derivative for calculations
  for ( int k=0; k<myOutputSize; k++ ) {
    output[k] = output_function_prime(myOutputLayer[k]);
  }
  print_vector("ELIG OUT: ", output);
```

```
    // Let's back-propagate everything through
    for ( int j=0; j<myHiddenSize; j++ ) {
      for ( int k=0; k<myOutputSize; k++ ) {
        myEligHidOut[j][k] = gGamma*gLambda * myEligHidOut[j][k] +
          output[k]*myHiddenLayer[j];
        if ( gDebug ) cout << "ELIGIBILITY: " << myEligHidOut[j][k] << endl;
        for ( int i=0; i<myInputSize; i++ ) {
          myEligInOut[k][i][j] = gGamma*gLambda * myEligInOut[k][i][j] +
                        ( output[k] * myWeightHidOut[j][k] *
                        output_function_prime(myHiddenLayer[j]) *
myInputLayer[i] );
        }
      }
    }
  }

  // Reset the states
  void reset( void ) {
    // Erase all of the layer information
    myInputLayer.erase(myInputLayer.begin(), myInputLayer.end());
    myHiddenLayer.erase(myHiddenLayer.begin(), myHiddenLayer.end());
    myOutputLayer.erase(myOutputLayer.begin(), myOutputLayer.end());

    // Rebuild the layer information
    myInputLayer = vector< double >((size_t)myInputSize, 0.0);
    myHiddenLayer = vector< double >((size_t)myHiddenSize, 0.0);
    myOutputLayer = vector< double >((size_t)myOutputSize, 0.0);

    // Erase the eligibilities
    myEligInOut.erase(myEligInOut.begin(), myEligInOut.end());

    // Rebuild the eligibilites
    for ( int k=0; k<myOutputSize; k++ ) {
      myEligInOut.push_back( matrix< double >(myInputSize, myHiddenSize, 0.0) );
    }
    myEligHidOut = matrix< double >(myHiddenSize, myOutputSize, 0.0);

    // Initialize
    initialize();
  };

private:
  int                         myInputSize;      // The size of the input layer
  int                         myHiddenSize;     // The size of the hidden layer
  int                         myOutputSize;     // The size of the output layer
  vector< double >            myInputLayer;     // The input layer data
  vector< double >            myHiddenLayer;    // The hidden layer data
  vector< double >            myOutputLayer;    // The output layer data
  matrix< double >            myWeightInHid;    // The weights from the input to
hidden layer
  matrix< double >            myWeightHidOut;   // The weights from the hidden
to output
  vector< matrix< double > >  myEligInOut;      // The eligibilities from input
to output
  matrix< double >            myEligHidOut;     // The eligibilities from hidden
to output
  string                      myFileName;       // The file to restore/save our
states
  double                      myBias;           // The value of the bias neuron
};

#endif
```

Appendix C. MDP Data Result Tables

The following table is the output of a single run for the TD(0.5) case in the Markov Decision Process as shown in Section 4.X. The output shows the state estimation values for each state A through E and the RMS Error for all five states.

Iteration	A	B	C	D	E	RMS Error
1	0.5000	0.5000	0.5000	0.5000	0.5000	0.2357
2	0.5000	0.5000	0.5000	0.5234	0.5938	0.2082
3	0.5000	0.5000	0.5075	0.5529	0.6674	0.1894
4	0.4500	0.4750	0.4942	0.5417	0.6674	0.1694
5	0.4050	0.4787	0.4973	0.5159	0.6674	0.1601
6	0.4050	0.4787	0.4973	0.5477	0.7007	0.1481
7	0.3645	0.4569	0.5019	0.5514	0.6803	0.1350
8	0.3645	0.4569	0.5019	0.5802	0.7123	0.1237
9	0.3645	0.4569	0.5162	0.6230	0.7615	0.1111
10	0.3645	0.4569	0.5162	0.6488	0.7853	0.1070
11	0.3281	0.4248	0.4947	0.6276	0.7640	0.0903
12	0.3281	0.4380	0.5354	0.6227	0.7682	0.0943
13	0.3281	0.4546	0.5516	0.6605	0.8060	0.0940
14	0.3281	0.4546	0.6280	0.6840	0.7918	0.1088
15	0.3281	0.4546	0.6280	0.7052	0.8126	0.1087
16	0.2952	0.4221	0.5832	0.7053	0.7754	0.0851
17	0.2952	0.4221	0.6036	0.7235	0.7979	0.0890
18	0.2952	0.4221	0.6036	0.7411	0.8181	0.0905
19	0.3963	0.4576	0.5609	0.7458	0.8103	0.1255
20	0.3567	0.4317	0.5410	0.6947	0.7905	0.1001
21	0.3210	0.4428	0.4933	0.6755	0.7905	0.0869
22	0.3210	0.4549	0.5382	0.6558	0.8054	0.0905
23	0.2660	0.4107	0.5215	0.6245	0.7860	0.0638
24	0.2660	0.4107	0.5215	0.6786	0.8059	0.0587
25	0.2856	0.4440	0.5150	0.7043	0.8228	0.0750
26	0.2856	0.4440	0.5753	0.6955	0.8000	0.0824
27	0.2615	0.4602	0.5282	0.6599	0.7810	0.0757
28	0.2354	0.4365	0.5366	0.6143	0.7810	0.0666
29	0.3244	0.4230	0.5461	0.6275	0.7573	0.0920
30	0.3244	0.4386	0.5643	0.6332	0.7791	0.0940

Iteration	A	B	C	D	E	RMS Error
31	0.3244	0.4386	0.5643	0.6588	0.8012	0.0907
32	0.3244	0.4386	0.6043	0.6608	0.8071	0.0975
33	0.3244	0.4386	0.6043	0.6851	0.8264	0.0972
34	0.3244	0.4386	0.6232	0.7161	0.8438	0.1036
35	0.2919	0.4264	0.6056	0.7019	0.8194	0.0859
36	0.3265	0.4666	0.6153	0.7016	0.8042	0.1083
37	0.3011	0.4558	0.5817	0.6580	0.8042	0.0902
38	0.2506	0.4080	0.5613	0.6428	0.7817	0.0626
39	0.2255	0.3797	0.5318	0.6199	0.7817	0.0479
40	0.2255	0.3797	0.5318	0.6470	0.8035	0.0398
41	0.2255	0.3797	0.5499	0.6934	0.8347	0.0420
42	0.2255	0.3797	0.5499	0.7319	0.8603	0.0512
43	0.2030	0.3615	0.5296	0.6801	0.8603	0.0279
44	0.1827	0.3355	0.4998	0.6501	0.8603	0.0159
45	0.1827	0.4003	0.5119	0.6340	0.8743	0.0391
46	0.1827	0.4003	0.5543	0.6758	0.8504	0.0402
47	0.1827	0.4003	0.5781	0.7101	0.8653	0.0524
48	0.1827	0.4003	0.5781	0.7324	0.8788	0.0587
49	0.1827	0.4003	0.5781	0.7531	0.8909	0.0658
50	0.1827	0.4003	0.5781	0.7723	0.9018	0.0731
51	0.1708	0.3617	0.5046	0.7346	0.8763	0.0382
52	0.1708	0.3617	0.5046	0.7550	0.8887	0.0484
53	0.1538	0.3341	0.4765	0.7046	0.8567	0.0233
54	0.1384	0.3084	0.4483	0.6457	0.8567	0.0319
55	0.1245	0.2844	0.4223	0.6130	0.8567	0.0522
56	0.1245	0.3140	0.4617	0.6402	0.8710	0.0339
57	0.1245	0.3140	0.5606	0.6357	0.8536	0.0379
58	0.1245	0.3410	0.5744	0.6627	0.8683	0.0415
59	0.1245	0.3410	0.5744	0.6899	0.8815	0.0452
60	0.1245	0.3410	0.6072	0.7119	0.8933	0.0616
61	0.1245	0.3410	0.6072	0.7354	0.9040	0.0679
62	0.1245	0.3410	0.6072	0.7570	0.9136	0.0747
63	0.1245	0.3410	0.6072	0.7770	0.9222	0.0817
64	0.1121	0.3132	0.5666	0.7397	0.9222	0.0649
65	0.1131	0.2960	0.4987	0.7074	0.9222	0.0526
66	0.1018	0.2721	0.4664	0.6704	0.9222	0.0583
67	0.1018	0.2721	0.4999	0.7122	0.9300	0.0623
68	0.0916	0.2499	0.4561	0.6530	0.9300	0.0693
69	0.0916	0.2499	0.5149	0.7308	0.9008	0.0655
70	0.1114	0.2743	0.4375	0.6596	0.9008	0.0549

Iteration	A	B	C	D	E	RMS Error
71	0.1114	0.2743	0.4681	0.7246	0.9106	0.0581
72	0.1533	0.2875	0.5314	0.7530	0.8968	0.0543
73	0.1380	0.2664	0.4965	0.7133	0.8968	0.0480
74	0.1380	0.2664	0.4965	0.7368	0.9071	0.0560
75	0.1242	0.2466	0.4636	0.6964	0.9071	0.0583
76	0.1409	0.2982	0.3821	0.5945	0.9071	0.0727
77	0.1409	0.2982	0.4355	0.6670	0.9121	0.0495
78	0.1409	0.2982	0.4355	0.6959	0.9209	0.0540
79	0.1409	0.2982	0.4355	0.7224	0.9288	0.0605
80	0.1409	0.2982	0.4355	0.7466	0.9359	0.0678
81	0.1409	0.2982	0.4355	0.7687	0.9423	0.0753
82	0.1409	0.2982	0.4355	0.7890	0.9481	0.0827
83	0.1409	0.2982	0.4355	0.8075	0.9533	0.0897
84	0.1409	0.2982	0.5217	0.7889	0.9428	0.0765
85	0.1205	0.2574	0.5269	0.7571	0.8955	0.0643
86	0.1205	0.2971	0.5581	0.7415	0.8891	0.0557
87	0.1383	0.3232	0.4804	0.7096	0.8035	0.0284
88	0.1383	0.3232	0.4804	0.7288	0.8231	0.0324
89	0.1383	0.3232	0.5126	0.7437	0.8408	0.0376
90	0.1383	0.3232	0.5126	0.7614	0.8567	0.0460
91	0.1383	0.3232	0.5440	0.7748	0.8710	0.0565
92	0.1356	0.2638	0.5151	0.7373	0.8710	0.0499
93	0.1356	0.2638	0.5151	0.7571	0.8839	0.0579
94	0.1356	0.2638	0.5151	0.7902	0.9008	0.0719
95	0.1356	0.2638	0.5151	0.8062	0.9108	0.0794
96	0.1356	0.2638	0.5151	0.8211	0.9197	0.0864
97	0.1356	0.3052	0.5663	0.7454	0.9277	0.0652
98	0.1165	0.2606	0.5028	0.7165	0.9277	0.0620
99	0.1165	0.2606	0.5028	0.7413	0.9349	0.0689
100	0.1726	0.3435	0.5312	0.7265	0.8477	0.0313

Appendix D. Tic-Tac-Toe – TD(λ) Data Result Tables

The following data shows the state estimations computed and stored for the TD(λ) X player as described in section 6.2. Each state is represented by a string of nine characters beginning at the upper, left and working across and down to the bottom, right. A ‘B’ represents an empty space while ‘X’ & ‘O’ represent each player.

State	Estimation	State	Estimation	State	Estimation
BBBBBBBBX	0.5308	BXOBBXBBB	0.3445	XBBBBOBBX	0.0509
BBBBBBBXB	0.5482	BXOBBXBOX	0.0119	XBBBBOBXB	0.2116
BBBBBBOXX	0.0418	BXOBBXOBX	0.0500	XBBBBOXBB	0.1156
BBBBBBXBB	0.4887	BXOBBXXBO	-0.0250	XBBBBOXOX	-0.0346
BBBBBBXOX	0.2306	BXOBOBBXX	-0.1860	XBBBBXBBO	0.1695
BBBBBBXXO	-0.0456	BXOBOBXXB	-0.0538	XBBBBXBOB	0.2452
BBBBBOBXX	0.0772	BXOBOXBBX	-0.0526	XBBBBXOBB	0.1119
BBBBBOXBX	0.2773	BXOBOXBXB	-0.1288	XBBBBXOOX	-0.0802
BBBBBOXXB	0.0700	BXOBOXXBB	0.4291	XBBBOBBBX	-0.0364
BBBBBXBBB	0.4927	BXOBOXXXO	-0.3460	XBBBOBBXB	0.4310
BBBBBXBOX	0.1985	BXOBXBBBB	0.5518	XBBBOBOXX	-0.0765
BBBBBXBXO	0.2971	BXOBXBBOX	0.6067	XBBBOBXBB	0.0442
BBBBBXOBX	0.0760	BXOBXBOBX	0.9200	XBBBOBXOX	-0.3269
BBBBBXOXB	0.1327	BXOBXBXBO	0.1670	XBBBOBXXO	-0.0066
BBBBBXXBO	0.3349	BXOBXBXOB	0.4703	XBBBOOBXX	-0.0975
BBBBBXXOB	0.3865	BXOBXOBBX	0.7657	XBBBOOXBX	-0.1426
BBBBOBBXX	-0.0061	BXOBXOXBB	0.0643	XBBBOOXXB	-0.0188
BBBBOBXBX	-0.0529	BXOBXOXOX	0.4808	XBBBOXBBB	0.1641
BBBBOBXXB	0.0044	BXOBXXBBO	0.4842	XBBBOXBXO	0.0621
BBBBOXBBX	-0.0501	BXOBXXBOB	0.5338	XBBBOXOBX	-0.2062
BBBBOXBXB	0.0863	BXOBXXOBB	0.3877	XBBBOXOXB	0.1750
BBBBOXXBB	0.1571	BXOBXXOOX	1.0000	XBBBOXXOB	-0.1204
BBBBOXXXO	-0.1451	BXOBXXXOO	0.4894	XBBBXBBBO	0.5661
BBBBXBBBB	0.6898	BXOOBBBXX	-0.0475	XBBBXBBOB	0.4094
BBBBXBBOX	0.6861	BXOOBBXXB	0.0250	XBBBXBOBB	0.5071
BBBBXBBXO	0.5633	BXOOOXBXX	-0.0998	XBBBXBOXO	0.5360
BBBBXBOBX	0.4807	BXOOOXXBX	0.0500	XBBBXBXOO	0.4768
BBBBXBOXB	0.5048	BXOOOXXXB	0.4634	XBBBXOBBB	0.4799

State	Estimation	State	Estimation	State	Estimation
BBBBXBXBO	0.3490	BXOOXBBBX	0.4634	XBBBXOBXO	0.1913
BBBBXBXOB	0.7612	BXOOXBXBB	0.6994	XBBBXOOXB	0.4448
BBBBXOBBX	0.6537	BXOOXBXOX	0.4068	XBBBXOXBO	-0.3417
BBBBXOBXB	0.6471	BXOOXOXBX	1.0000	XBBBXOXOB	0.6054
BBBBXOOXX	0.4792	BXOOXXBBB	0.6380	XBBBXXBOO	0.2836
BBBBXOXBB	0.3872	BXOOXXBOX	0.3127	XBBBXXOBO	0.2467
BBBBXOXOX	0.8119	BXOOXXOBX	0.4566	XBBBXXOOB	-0.3312
BBBBXOXXO	-0.3787	BXOOXXXBO	0.5545	XBBOBBBBX	0.1290
BBBBXXBBO	0.6449	BXOXBBBBB	0.1316	XBBOBBBXB	0.3534
BBBBXXBOB	0.6539	BXOXBBOXB	-0.2094	XBBOBBXBB	0.1837
BBBBXXOBB	0.4983	BXOXBBXOB	-0.0215	XBBOBOXBX	-0.0590
BBBBXXOOX	0.6493	BXOXBOBXB	-0.0629	XBBOBOXXB	-0.0491
BBBBXXOXO	0.5709	BXOXBXBBO	-0.0530	XBBOBXBBB	0.2233
BBBBXXXOO	0.7798	BXOXBXOOX	-0.0025	XBBOBXBOX	0.0903
BBBOBBBXX	-0.0215	BXOXOBBXB	-0.1850	XBBOOBBXX	0.0573
BBBOBBXBX	0.3076	BXOXOBXBB	0.1638	XBBOOBXXB	0.0805
BBBOBBXXB	0.2014	BXOXOBXOX	0.0862	XBBOOXBBX	0.1751
BBBOBXBBX	0.0950	BXOXOOBXX	-0.0356	XBBOOXBXB	0.7324
BBBOBXBXB	0.0547	BXOXOOXBX	0.8953	XBBOOXOXX	-0.0952
BBBOBXXBB	0.2084	BXOXOXBBB	-0.2386	XBBOOXXOX	-0.1931
BBBOOXXBX	0.2378	BXOXOXBXO	-0.3698	XBBOXBBBB	0.6951
BBBOXBBBX	0.5206	BXOXOXXBO	-0.1599	XBBOXBBXO	0.6472
BBBOXBBXB	0.6201	BXOXXBBBO	-0.4208	XBBOXBOXB	0.7341
BBBOXBOXX	-0.2784	BXOXXBBOB	0.3703	XBBOXBXBO	0.1744
BBBOXBXBB	0.7604	BXOXXBOBB	0.4423	XBBOXBXOB	0.7795
BBBOXBXOX	0.7197	BXOXXBOOX	1.0000	XBBOXOBXB	0.8083
BBBOXBXXO	0.5093	BXOXXBXOO	-0.2131	XBBOXOXBB	0.7261
BBBOXOBXX	0.5479	BXOXXOBBB	0.0648	XBBOXOXXO	0.0992
BBBOXOXBX	1.0000	BXOXXOBOX	0.4238	XBBOXXBBO	0.4235
BBBOXOXXB	0.5674	BXOXXOOBX	1.0000	XBBOXXBOB	0.4282
BBBOXXBBB	0.6068	BXOXXOXOB	-0.4696	XBBOXXOBB	0.5423
BBBOXXBOX	0.4810	BXXBBBBBO	0.0090	XBBOXXOXO	0.5118
BBBOXXBXO	0.5213	BXXBBBBOB	0.0833	XBBOXXXOO	0.3885
BBBOXXOBX	-0.3502	BXXBBBOBB	0.0178	XBBXBBBBO	0.0235
BBBOXXOXB	0.1980	BXXBBBOOX	0.0567	XBBXBBBOB	-0.0292
BBBOXXXBO	0.6073	BXXBBBOXO	0.0233	XBBXBBOBB	0.0598
BBBOXXXOB	0.5665	BXXBBOBBB	0.2195	XBBXBBOXO	0.0295
BBBXBBBBB	0.5164	BXXBBOBOX	0.0492	XBBXBOBBB	0.0707
BBBXBBBOX	0.2710	BXXBBXOBO	-0.1082	XBBXBXOOB	-0.0650
BBBXBBBXO	0.2196	BXXBBXOOB	-0.1513	XBBXOBBBB	0.0655

State	Estimation	State	Estimation	State	Estimation
BBBXBBOBX	0.3149	BXXBOBBBB	-0.0181	XBBXOBBOX	-0.0482
BBBXBBOXB	0.2931	BXXBOBBOX	0.5245	XBBXOBBXO	0.2468
BBBXBBXBO	0.0065	BXXBOBBXO	-0.1811	XBBXOBOXB	-0.3218
BBBXBBXOB	0.0760	BXXBOBOBX	0.2902	XBBXOOBXB	0.4025
BBBXBOBBX	0.2848	BXXBOBXBO	-0.1378	XBBXOOOXX	-0.3410
BBBXBOBXB	0.2972	BXXBOBXOB	0.0955	XBBXOXBOB	-0.0390
BBBXBOOXX	0.0339	BXXBOOBBX	-0.0534	XBBXOXOBB	-0.2093
BBBXBOXBB	0.0813	BXXBOOBXB	-0.1170	XBBXOXOOX	-0.1855
BBBXBOXOX	0.0712	BXXBOOOXX	-0.0407	XBBXOXOXO	-0.3806
BBBXBOXXO	-0.1083	BXXBOOXBB	-0.0814	XBBXXBBOO	-0.1692
BBBXBXBBO	-0.0191	BXXBOOXOX	-0.0500	XBBXXBOBO	-0.2668
BBBXBXBOB	-0.0279	BXXBOXBBO	-0.0978	XBBXXBOOB	-0.3634
BBBXBXOBB	-0.0170	BXXBOXBOB	-0.0254	XBBXXOBBO	-0.2585
BBBXOBBBX	0.0958	BXXBOXOBB	-0.1966	XBBXXOBOB	0.1867
BBBXOBBXB	0.1402	BXXBOXOXO	-0.2772	XBBXXOOBB	0.0093
BBBXOBOXX	0.0203	BXXBOXXOO	-0.0792	XBBXXOOXO	-0.4591
BBBXOBXBB	-0.0387	BXXBXBBOO	-0.3851	XBOBBBBBX	0.1163
BBBXOOBXX	0.1687	BXXBXBOBO	-0.5915	XBOBBBBXB	0.0996
BBBXOOXBX	0.3332	BXXBXBOOB	-0.2925	XBOBBBXBB	0.1410
BBBXOXBBB	0.0179	BXXBXOBBO	0.4229	XBOBBBXOX	0.1382
BBBXOXOBX	-0.2009	BXXBXOBOB	0.3594	XBOBBOXBX	0.1609
BBBXOXXBO	-0.1320	BXXBXOOBB	0.0900	XBOBBOXXB	-0.0612
BBBXXBBBO	0.4244	BXXBXOOOX	0.3639	XBOBBXBBB	0.5587
BBBXXBBOB	0.5960	BXXOBBBBB	0.1206	XBOBBXOXB	0.0975
BBBXXBOBB	0.6581	BXXOBBOBX	-0.1560	XBOBOBBXX	-0.1448
BBBXXBOOX	0.7404	BXXOBBXOB	0.2282	XBOBOBXXB	0.0349
BBBXXBOXO	0.4477	BXXOBOBBX	0.0956	XBOBOXBBX	-0.0117
BBBXXBXOO	0.4227	BXXOBOXBB	-0.0230	XBOBOXBXB	0.2469
BBBXXOBBB	0.5393	BXXOBOXOX	0.0590	XBOBOXXXO	0.1897
BBBXXOBOX	0.6234	BXXOBOXXO	-0.0650	XBOBXBBBB	0.4557
BBBXXOBXO	0.3315	BXXOBXOXO	0.0793	XBOBXBBXO	0.1497
BBBXXOOBX	0.5558	BXXOOBBBX	-0.4260	XBOBXBOXB	0.7753
BBBXXOOXB	0.6464	BXXOOBOXX	-0.7226	XBOBXBXBO	-0.2798
BBBXXOXBO	-0.2956	BXXOOBXBB	-0.0789	XBOBXBXOB	0.4054
BBBXXOXOB	0.4489	BXXOOBXOX	-0.1176	XBOBXOBXB	-0.3177
BBOBBBBXX	0.0790	BXXOOBXXO	-0.3017	XBOBXOXBB	-0.3842
BBOBBBXBX	0.0669	BXXOOXBBB	0.0590	XBOBXXBBO	0.6530
BBOBBBXXB	0.0021	BXXOOXOXB	0.1307	XBOBXXBOB	0.3984
BBOBBXBBX	0.0272	BXXOOXXBO	0.0046	XBOBXXOBB	0.7861
BBOBBXBXB	0.1974	BXXOOXXOB	1.0000	XBOBXXOXO	1.0000

State	Estimation	State	Estimation	State	Estimation
BBOBBXXBB	0.0863	BXXOXBBBO	-0.0107	XBOBXXXOO	0.5378
BBOBOXBXX	-0.1681	BXXOXBBOB	-0.1319	XBOOBBBXX	-0.0273
BBOBOXXBX	-0.0498	BXXOXBOBB	-0.4271	XBOOBBXBX	-0.0238
BBOBXBBBX	0.5140	BXXOXBOOX	-0.1616	XBOOBXXBB	0.0139
BBOBXBBXB	0.4604	BXXOXOBBB	0.4672	XBOOOXBXX	-0.0975
BBOBXBOXX	0.3311	BXXOXOBOX	1.0000	XBOOOXXBX	0.3925
BBOBXBXBB	0.5068	BXXOXOOBX	-0.1796	XBOOOXXXB	0.4874
BBOBXBXOX	0.0886	BXXOXXBOO	-0.3794	XBOOXBBXB	0.5559
BBOBXBXXO	-0.5931	BXXOXXOBO	-0.9934	XBOOXBXBB	0.2280
BBOBXOBXX	0.4049	BXXOXXOOB	-0.4312	XBOOXBXXO	-0.5413
BBOBXOXBX	0.4065	BXXXBBBOO	-0.0477	XBOOXOXXB	0.0040
BBOBXOXXB	-0.5055	BXXXBBOBO	-0.1782	XBOOXXBBB	0.5639
BBOBXXBBB	0.6284	BXXXBOBBO	0.0949	XBOOXXBXO	0.6369
BBOBXXBOX	0.4023	BXXXBOOXO	0.0975	XBOOXXOXB	1.0000
BBOBXXBXO	0.5411	BXXXBOXOO	0.3698	XBOOXXXOB	0.5337
BBOBXXOBX	0.4267	BXXXOBBBO	-0.3009	XBOXBBBBB	0.0328
BBOBXXOXB	0.5158	BXXXOBBOB	0.4769	XBOXBBBXO	-0.0687
BBOBXXXBO	0.5720	BXXXOBOBB	0.4251	XBOXBBOBX	0.0485
BBOBXXXOB	0.5217	BXXXOBOOX	1.0000	XBOXBOOXX	0.0500
BBOOXBBXX	-0.0333	BXXXOBOXO	0.2133	XBOXBXBBO	0.1864
BBOOXBXBX	0.4425	BXXXOBXOO	0.0692	XBOXBXBOB	-0.1620
BBOOXBXXB	0.4027	BXXXOOBBB	-0.0547	XBOXBXOOX	-0.2202
BBOOXXBBX	-0.1316	BXXXOOBOX	0.5612	XBOXBXOXO	0.0952
BBOOXXBXB	0.4277	BXXXOOBXO	-0.0844	XBOXOBBBX	-0.2902
BBOOXXOXX	-0.1551	BXXXOOOBX	0.4302	XBOXOBBXB	-0.3126
BBOOXXXBB	0.4050	BXXXOOOXB	0.6015	XBOXOOBXX	0.1262
BBOOXXXOX	0.4107	BXXXOOXBO	-0.1394	XBOXOXBOX	-0.1855
BBOOXXXXO	0.4335	BXXXOOXOB	0.2830	XBOXOXBXO	-0.2668
BBOXBBBBX	0.1518	BXXXOXBOO	-0.7080	XBOXXBBBO	-0.4331
BBOXBBBXB	-0.0415	BXXXOXOBO	-0.2649	XBOXXBBOB	0.0885
BBOXBBXBB	0.0277	BXXXOXOOB	-0.0148	XBOXXBOBB	0.3124
BBOXBBXXO	-0.1751	BXXXXOBOO	-0.1691	XBOXXBOXO	-0.0421
BBOXBOBXX	0.2003	BXXXXOOBO	-0.0393	XBOXXOBBB	-0.4949
BBOXBXBBB	-0.0222	BXXXXOOOB	-0.1903	XBOXXOOXB	0.1707
BBOXBXBXO	0.0500	OBBBBBBXX	0.0621	XBXBBBBBO	0.1122
BBOXBXOBX	-0.0622	OBBBBBXBX	0.1343	XBXBBBBOB	0.0146
BBOXOBBXX	-0.0214	OBBBBBXXB	0.0761	XBXBBBOBB	0.1206
BBOXOXBXB	0.0000	OBBBBXBBX	-0.0339	XBXBBOBBB	0.2862
BBOXOXXBB	-0.0238	OBBBBXBXB	0.0049	XBXBBOOBX	0.1108
BBOXOXXOX	-0.0500	OBBBBXXBB	0.1442	XBXBBOXBO	0.1852

State	Estimation	State	Estimation	State	Estimation
BBOXOXXXO	0.0476	OBBBOXXBX	-0.0595	XBXBBXBOO	-0.1116
BBOXXBBBB	0.4394	OBBBXBBBX	0.5243	XBXBOBBBB	0.0647
BBOXXBBOX	0.3377	OBBBXBBXB	0.3632	XBXBOBBXO	0.5609
BBOXXBBXO	-0.1452	OBBBXBOXX	-0.0302	XBXBOBOBX	0.0550
BBOXXBOBX	0.8016	OBBBXBXBB	0.4649	XBXBOBOXB	0.6418
BBOXXBOXB	0.4696	OBBBXBXOX	0.2280	XBXBOBXBO	0.1545
BBOXXBXBO	-0.5146	OBBBXBXXO	0.1118	XBXBOBXOB	-0.2382
BBOXXBXOB	0.1992	OBBBXOBXX	0.3303	XBXBOOBBX	-0.0847
BBOXXOBBX	0.4863	OBBBXOXBX	0.4019	XBXBOOBXB	0.0037
BBOXXOBXB	-0.1920	OBBBXOXXB	-0.1310	XBXBOOXBB	-0.0994
BBOXXOOXX	1.0000	OBBBXXBBB	0.2522	XBXBOOXOX	-0.5367
BBOXXOXBB	-0.1775	OBBBXXBOX	0.2602	XBXBOOXXO	-0.2047
BBOXXOXOX	0.4452	OBBBXXBXO	0.4783	XBXBOXBBO	0.0181
BBXBBBBBB	0.5626	OBBBXXOBX	-0.5969	XBXBOXBOB	-0.1233
BBXBBBBOX	0.3809	OBBBXXOXB	-0.6355	XBXBOXOXO	0.2201
BBXBBBBXO	0.2496	OBBBXXXBO	0.6789	XBXBOXXOO	0.0455
BBXBBBOBX	0.1008	OBBBXXXOB	0.4588	XBXBXBBOO	-0.3509
BBXBBBOXB	0.2252	OBBOXBBXX	-0.3666	XBXBXBOBO	-0.0963
BBXBBBXBO	0.0664	OBBOXBXBX	0.3839	XBXBXBOOB	-0.0468
BBXBBBXOB	0.0853	OBBOXBXXB	0.4077	XBXBXOBBO	0.5441
BBXBBOBBX	0.0716	OBBOXXBBX	-0.2751	XBXBXOBOB	0.3549
BBXBBOBXB	0.3533	OBBOXXBXB	0.0131	XBXBXOOBB	0.3465
BBXBBOXBB	0.1306	OBBOXXXBB	0.4643	XBXBXOOXO	0.4092
BBXBBOXOX	-0.0251	OBBOXXXOX	0.5150	XBXOBBBBB	0.3328
BBXBBXBBO	0.0436	OBBOXXXXO	1.0000	XBXOBBBXO	0.1874
BBXBBXBOB	0.0483	OBBXBBBBX	0.1545	XBXOBBOXB	0.2162
BBXBBXOBB	-0.0733	OBBXBBBXB	0.1115	XBXOBBXBO	0.0918
BBXBOBBBX	0.4353	OBBXBBXBB	0.0724	XBXOBBXOB	0.0500
BBXBOBBXB	0.2281	OBBXBBXOX	-0.0381	XBXOBOOXX	-0.0500
BBXBOBXBB	0.0025	OBBXBBXXO	-0.0500	XBXOBOXXO	-0.1209
BBXBOBXXO	-0.1373	OBBXBOXBX	0.0372	XBXOBXBOB	0.0500
BBXBOOBXX	-0.0459	OBBXBXBBB	0.1626	XBXOOBBBX	-0.2175
BBXBOOXBX	-0.1460	OBBXBXBOX	-0.0500	XBXOOBBXB	-0.0407
BBXBOOXXB	-0.0957	OBBXBXOBX	0.1030	XBXOOBOXX	0.1935
BBXBOXBBB	0.0320	OBBXOBXBX	-0.0017	XBXOOBXOX	-0.5367
BBXBOXBXO	-0.0795	OBBXOXBBX	-0.0892	XBXOOXBXO	0.3955
BBXBOXOXB	-0.0442	OBBXOXBXB	-0.2865	XBXOOXOXB	0.9991
BBXBXBBBO	0.4196	OBBXOXOXX	-0.0545	XBXOOXXBO	0.1579
BBXBXBBOB	0.3258	OBBXXBBBB	0.6183	XBXOOXXOB	-0.0850
BBXBXBOBB	0.4987	OBBXXBBOX	0.6091	XBXOXBBBO	0.4336

State	Estimation	State	Estimation	State	Estimation
BBXBXBOOX	0.4094	OBBXXBBXO	0.7000	XBXOXBBOB	0.4987
BBXBXBOXO	0.5300	OBBXXBOBX	0.5744	XBXOXBOBB	0.5626
BBXBXOBBB	0.7442	OBBXXBOXB	0.4730	XBXOXBOXO	0.3707
BBXBXOBOX	0.6619	OBBXXBXBO	0.2946	XBXOXOBBB	1.0000
BBXBXOBXO	0.7461	OBBXXBXOB	0.4081	XBXOXOBXO	1.0000
BBXBXOOBX	0.2401	OBBXXOBBX	0.4131	XBXOXOOXB	1.0000
BBXBXOOXB	0.5247	OBBXXOBXB	0.3896	XBXOXXBOO	0.0880
BBXBXXBOO	-0.4706	OBBXXOOXX	0.4106	XBXOXXOBO	-0.6279
BBXBXXOBO	-0.3677	OBBXXOXBB	-0.0382	XBXOXXOOB	-0.0871
BBXBXXOOB	-0.3152	OBBXXOXOX	0.4180	XBXXBBOOB	-0.2270
BBXOBBBBX	0.0351	OBBXXOXXO	-0.0704	XBXXBOBOB	0.0500
BBXOBBBXB	0.2272	OBOBBXXXB	-0.0500	XBXXBOOOX	0.0500
BBXOBBXBB	0.1389	OBOBXBBXX	-0.6906	XBXXOBBBO	0.2117
BBXOBOBXX	-0.0975	OBOBXBXBX	-0.1757	XBXXOBBOB	-0.0271
BBXOBOXBX	0.1191	OBOBXBXXB	-0.6410	XBXXOBOBB	0.1285
BBXOBOXXB	-0.1776	OBOBXXBBX	-0.1241	XBXXOBOOX	0.0984
BBXOBXBBB	-0.0461	OBOBXXBXB	-0.5119	XBXXOBOXO	0.5904
BBXOOBBXX	-0.3801	OBOBXXOXX	-1.0000	XBXXOOBBB	0.0653
BBXOOBXBX	-0.3261	OBOBXXXBB	0.2322	XBXXOOBOX	-0.0590
BBXOOXBXB	0.1924	OBOBXXXOX	-0.5357	XBXXOOBXO	1.0000
BBXOXBBBB	0.4650	OBOBXXXXO	-0.1839	XBXXOOOBX	0.5620
BBXOXBBOX	0.3704	OBOOXXBXX	-0.9775	XBXXOOOXB	0.4241
BBXOXBBXO	0.4383	OBOOXXXBX	-0.6308	XBXXOXBOO	-0.7622
BBXOXBOBX	-0.3578	OBOOXXXXB	0.0393	XBXXOXOBO	-0.2126
BBXOXBOXB	0.2463	OBOXBBBXX	-0.0934	XBXXOXOOB	-0.2649
BBXOXOBBX	0.4352	OBOXBXBXB	-0.2076	XBXXXOBOO	-0.0091
BBXOXOBXB	0.5759	OBOXBXOXX	-0.1855	XBXXXOOBO	-0.5690
BBXOXOOXX	-0.0198	OBOXOXBXX	-0.5123	XBXXXOOOB	-0.0160
BBXOXXBBO	-0.0351	OBOXOXXBX	-0.2392	XOBBBBBBX	-0.0383
BBXOXXBOB	0.2146	OBOXOXXXB	-0.2262	XOBBBBBXB	0.4262
BBXOXXOBB	-0.3752	OBOXXBBBX	0.2859	XOBBBBXBB	0.2269
BBXOXXOXO	-0.5077	OBOXXBBXB	-0.1326	XOBBBBXXO	0.1631
BBXXBBBBO	0.1819	OBOXXBOXX	-0.1010	XOBBBOBXX	0.0093
BBXXBBBOB	0.2745	OBOXXBXBB	-0.3479	XOBBBXBBB	0.3434
BBXXBBOBB	0.1729	OBOXXBXOX	-0.4456	XOBBOBBXX	0.1335
BBXXBBOXO	0.0300	OBOXXBXXO	-1.0000	XOBBOBXXB	0.4195
BBXXBBXOO	0.2499	OBOXXOBXX	-0.0810	XOBBOXBXB	0.1891
BBXXBOBBB	0.2464	OBOXXOXBX	-0.5921	XOBBOXOXX	0.2214
BBXXBOBXO	0.0116	OBOXXOXXB	-0.9490	XOBBOXXXO	0.4017
BBXXBOXOB	0.0801	OBXBBBBBX	0.1133	XOBBXBBBB	0.7360

State	Estimation	State	Estimation	State	Estimation
BBXXBXBOO	-0.1609	OBXBBBBXB	0.1001	XOBBXBBXO	0.4826
BBXXOBBBB	0.5298	OBXBBBXBB	0.0571	XOBBXBOXB	0.6224
BBXXOBBOX	-0.0442	OBXBBBXXO	-0.0228	XOBBXBXBO	0.3377
BBXXOBBXO	-0.0517	OBXBBOXXB	-0.0251	XOBBXBXOB	1.0000
BBXXOBOBX	0.6338	OBXBBXBBB	0.1303	XOBBXOBXB	0.4906
BBXXOBXOB	-0.1466	OBXBBXBXO	-0.0326	XOBBXOXBB	0.3743
BBXXOOBBX	0.5603	OBXBBXOXB	-0.0472	XOBBXOXXO	0.0322
BBXXOOBXB	0.6972	OBXBOBBXX	0.1054	XOBBXXBBO	0.6291
BBXXOOXBB	0.0557	OBXBOBXBX	0.1440	XOBBXXBOB	0.6601
BBXXOOXOX	-0.0926	OBXBOBXXB	-0.3405	XOBBXXOBB	0.4069
BBXXOOXXO	-0.1300	OBXBOXXBB	-0.1159	XOBBXXOXO	0.4867
BBXXOXBBO	-0.1439	OBXBXBBBB	0.4084	XOBBXXXOO	1.0000
BBXXOXBOB	-0.1421	OBXBXBBOX	0.3770	XOBOBBXBX	0.1549
BBXXOXOBB	-0.0271	OBXBXBBXO	0.6666	XOBOBBXXB	0.2303
BBXXOXOXO	-0.0451	OBXBXBOBX	-0.0492	XOBOBXXOX	-0.0500
BBXXOXXOO	-0.4013	OBXBXBOXB	0.2689	XOBOOXBXX	0.9306
BBXXXBBOO	-0.3696	OBXBXOBBX	0.0292	XOBOOXXBX	0.1134
BBXXXBOBO	0.3211	OBXBXOBXB	0.5646	XOBOOXXXB	0.4172
BBXXXBOOB	0.3482	OBXBXOOXX	-0.5334	XOBOXBBXB	0.6365
BBXXXOBBO	0.6061	OBXBXXBBO	0.3073	XOBOXBXBB	0.7264
BBXXXOBOB	0.4927	OBXBXXBOB	0.1076	XOBOXBXXO	0.5579
BBXXXOOBB	0.4515	OBXBXXOBB	-0.5482	XOBOXOXXB	1.0000
BBXXXOOOX	0.3596	OBXBXXOXO	0.1752	XOBOXXBBB	0.6359
BBXXXOOXO	0.4188	OBXOBBXXB	-0.0750	XOBOXXOXB	0.3606
BOBBBBBXX	0.0216	OBXOBXXXO	-0.0476	XOBOXXXBO	0.2794
BOBBBBXBX	0.0199	OBXOOXXXB	-0.2528	XOBOXXXOB	1.0000
BOBBBBXXB	0.0604	OBXOXBBBX	-0.2766	XOBXBBBBB	0.2376
BOBBBXBBX	0.0282	OBXOXBBXB	-0.4511	XOBXBBBOX	-0.1234
BOBBBXBXB	0.0750	OBXOXOBXX	-0.1259	XOBXBBBXO	0.0263
BOBBBXOXX	0.0718	OBXOXXBBB	-0.4450	XOBXBXBOB	-0.0926
BOBBBXXBB	0.1354	OBXOXXBXO	0.0162	XOBXBXOOX	-0.0797
BOBBOXXBX	-0.2672	OBXXBBBBB	0.3057	XOBXOBBBX	-0.0262
BOBBOXXXB	0.0431	OBXXBBOBX	0.0591	XOBXOBBXB	0.0480
BOBBXBBBX	0.3993	OBXXBBXOB	-0.0032	XOBXOBOXX	-0.0951
BOBBXBBXB	0.6339	OBXXBOXBB	0.1562	XOBXOOBXX	0.4638
BOBBXBOXX	0.0913	OBXXBOXOX	0.0816	XOBXOXBBB	-0.2306
BOBBXBXBB	0.4997	OBXXBXBBO	0.0380	XOBXOXOBX	-0.4596
BOBBXBXOX	0.0829	OBXXBXXOO	0.0508	XOBXXBBBO	0.2516
BOBBXBXXO	0.0671	OBXXOBBBX	0.7224	XOBXXBBOB	0.5301
BOBBXOBXX	0.4090	OBXXOBBXB	0.0161	XOBXXBOBB	0.4619

State	Estimation	State	Estimation	State	Estimation
BOBBXOXBX	0.4822	OBXXOBOXX	0.5441	XOBXXBOXO	0.2869
BOBBXOXXB	0.2006	OBXXOBXBB	-0.2979	XOBXXOBBB	0.3443
BOBBXXBBB	0.5033	OBXXOBXOX	-0.0407	XOBXXOBXO	-0.1404
BOBBXXBOX	0.5349	OBXXOOBXX	0.6232	XOBXXOOXB	0.3582
BOBBXXBXO	0.5055	OBXXOOXXB	0.1054	XOOBBBBXX	0.2112
BOBBXXOBX	-0.0186	OBXXOXBBB	0.1384	XOOBBBXBX	-0.0875
BOBBXXOXB	0.3553	OBXXOXOXB	-0.1851	XOOBBXOXX	0.2262
BOBBXXXBO	0.5100	OBXXXBBBO	0.4764	XOOBOXBXX	0.0920
BOBBXXXOB	0.8203	OBXXXBBOB	0.3806	XOOBOXXBX	-0.3017
BOBOXBBXX	0.2302	OBXXXBOBB	0.5158	XOOBOXXXB	0.9997
BOBOXBXBX	0.3654	OBXXXBOOX	0.4437	XOOBXBBXB	0.5483
BOBOXBXXB	0.4921	OBXXXBOXO	1.0000	XOOBXBXBB	0.6015
BOBOXXBBX	0.2533	OBXXXOBBB	0.5921	XOOBXBXXO	-0.5335
BOBOXXBXB	0.1830	OBXXXOBOX	0.4593	XOOBXOXXB	0.2195
BOBOXXOXX	-0.2078	OBXXXOBXO	1.0000	XOOBXXBBB	0.7698
BOBOXXXBB	0.5039	OBXXXOOXB	0.4623	XOOBXXBXO	0.5717
BOBOXXXOX	1.0000	OOBBBXBXX	-0.0725	XOOBXXOXB	1.0000
BOBOXXXXO	0.5395	OOBBXBBXX	-0.0328	XOOBXXXBO	0.4785
BOBXBBBBX	0.0521	OOBBXBXBX	-0.2458	XOOBXXXOB	1.0000
BOBXBBBXB	0.2406	OOBBXBXXB	-0.4916	XOOOXBXXB	0.4479
BOBXBBXBB	-0.0901	OOBBXXBBX	-0.4273	XOOOXXBXB	0.2442
BOBXBBXOX	0.0263	OOBBXXBXB	0.0353	XOOOXXXBB	0.3832
BOBXBOXBX	0.0233	OOBBXXOXX	-0.9373	XOOXBOBXX	0.0500
BOBXBXBBB	0.1195	OOBBXXXBB	-0.3302	XOOXBXBBB	-0.0209
BOBXOBBXX	0.0311	OOBBXXXOX	-0.1038	XOOXBXBOX	-0.1262
BOBXOXBBX	-0.3076	OOBBXXXXO	-0.0576	XOOXBXBXO	0.1855
BOBXOXBXB	0.0802	OOBOXXBXX	-0.4013	XOOXBXOBX	-0.0952
BOBXXBBBB	0.5198	OOBOXXXBX	-0.1036	XOOXOXBBX	-0.4013
BOBXXBBOX	0.5623	OOBOXXXXB	-0.0305	XOOXOXBXB	0.0497
BOBXXBBXO	0.2133	OOBXBBXBX	-0.0444	XOOXXBBBB	0.5575
BOBXXBOBX	0.4369	OOBXBXXOX	-0.0975	XOOXXBBXO	-0.2103
BOBXXBOXB	0.4315	OOBXOXXBX	-0.5599	XOOXXBOXB	1.0000
BOBXXBXBO	-0.0709	OOBXXBBBX	0.1559	XOOXXOBXB	-0.0743
BOBXXBXOB	0.3554	OOBXXBBXB	0.2545	XOXBBBBBB	0.2021
BOBXXOBBX	0.4468	OOBXXBOXX	-0.4744	XOXBBBXBO	0.0263
BOBXXOBXB	0.2135	OOBXXBXBB	-0.3430	XOXBBOXBB	0.0286
BOBXXOOXX	0.5055	OOBXXBXOX	-0.1182	XOXBBOXXO	0.4013
BOBXXOXBB	0.0378	OOBXXBXXO	-0.2169	XOXBBXOBB	-0.0167
BOBXXOXOX	1.0000	OOBXXOBXX	-0.4153	XOXBBXXOO	0.0975
BOBXXOXXO	-0.0732	OOBXXOXBX	-0.0881	XOXBOBBBX	-0.2406

State	Estimation	State	Estimation	State	Estimation
BOOBXBBXX	-0.3086	OOBXXOXXB	-0.3491	XOXBOBBXB	0.1215
BOOBXBXBX	-0.4119	OOXBBBXBX	0.1555	XOXBOBXXO	0.0500
BOOBXBXXB	-0.2538	OOXBBXBXB	0.1822	XOXBOOXXB	-0.0778
BOOBXXBBX	-0.3952	OOXBBXXXO	-0.1493	XOXBOXBBB	-0.0488
BOOBXXBXB	0.2461	OOXBOXXXB	-0.0712	XOXBOXOXB	0.6194
BOOBXXOXX	0.2840	OOXBXBBBX	0.4948	XOXBXBBBO	0.1623
BOOBXXXBB	0.1090	OOXBXBBXB	0.6408	XOXBXBBOB	0.6132
BOOBXXXOX	-0.2820	OOXBXBOXX	-0.6376	XOXBXBOBB	0.0436
BOOBXXXXO	-0.5040	OOXBXOBXX	0.5011	XOXBXOBBB	0.8281
BOOOXXBXX	-0.2665	OOXBXXBBB	0.5054	XOXBXOBXO	0.6267
BOOOXXXBX	-0.3856	OOXBXXBXO	1.0000	XOXBXOOXB	0.4551
BOOOXXXXB	-0.3224	OOXBXXOXB	-0.2272	XOXBXXBOO	0.0434
BOOXBXBXB	-0.0294	OOXOBXXXB	0.3366	XOXBXXOBO	-0.6273
BOOXBXOXX	-0.0500	OOXOXBBXX	0.2227	XOXBXXOOB	0.0466
BOOXOXBXX	-0.2649	OOXOXXBXB	-0.1087	XOXOBBBBX	-0.0238
BOOXOXXBX	-0.2262	OOXXBBBXB	0.0500	XOXOBBBXB	-0.0748
BOOXOXXXB	-0.0025	OOXXBBXOX	0.0448	XOXOBBOXX	0.1426
BOOXXBBBX	-0.3806	OOXXBOBXX	0.0500	XOXOBBXXO	0.0862
BOOXXBBXB	-0.0606	OOXXBOXXB	0.2649	XOXOBOBXX	0.0025
BOOXXBOXX	-0.0736	OOXXBXBBB	0.1499	XOXOBOXBX	-0.0025
BOOXXBXBB	-0.3304	OOXXBXOXB	0.2649	XOXOBOXXB	0.0727
BOOXXBXOX	-0.2482	OOXXBXXBO	-0.0075	XOXOOBBXX	0.1925
BOOXXBXXO	-0.9605	OOXXOBBXX	1.0000	XOXOOBXBX	-0.5123
BOOXXOBXX	-0.1339	OOXXOBXBX	-0.0522	XOXOOBXXB	-0.0475
BOOXXOXBX	-0.0492	OOXXOBXXB	-0.1897	XOXOOXBXB	0.1267
BOOXXOXXB	-0.3017	OOXXOXBXB	0.2448	XOXOXBBBB	0.7060
BOXBBBBBX	0.5935	OOXXOXXBB	-0.1426	XOXOXBBXO	0.4892
BOXBBBBXB	0.3651	OOXXXBBBB	0.7600	XOXOXBOXB	0.5359
BOXBBBOXX	0.0274	OOXXXBBOX	1.0000	XOXOXOBXB	1.0000
BOXBBBXBB	-0.0009	OOXXXBBXO	1.0000	XOXOXXBBO	0.5269
BOXBBBXXO	-0.0384	OOXXXBOBX	0.4633	XOXOXXBOB	1.0000
BOXBBOXXB	0.2311	OOXXXBOXB	0.5164	XOXOXXOBB	0.4212
BOXBBXBBB	0.0891	OOXXXOBBX	0.3562	XOXXBOBBB	0.0023
BOXBBXXBO	-0.0263	OOXXXOBXB	0.4416	XOXXBOBOX	-0.0952
BOXBOBXBX	-0.2978	OXBBBBBBX	0.2253	XOXXBXBOO	-0.0975
BOXBOBXXB	0.0092	OXBBBBBXB	-0.0781	XOXXBXOOB	-0.0975
BOXBOXBXB	0.1151	OXBBBBOXX	-0.1071	XOXXOBBBB	-0.0294
BOXBOXXBB	-0.0801	OXBBBBXBB	0.2691	XOXXOBBXO	0.5614
BOXBOXXXO	-0.4258	OXBBBBXXO	-0.0936	XOXXOBOBX	-0.2640
BOXBXBBBB	0.6960	OXBBBOXBX	0.1735	XOXXOOBBX	-0.0904

State	Estimation	State	Estimation	State	Estimation
BOXBXBBOX	1.0000	OXBBBXBBB	0.0556	XOXXOOBXB	0.1769
BOXBXBBXO	0.6267	OXBBOBBXX	-0.0698	XOXXOXBBO	-0.1190
BOXBXBOBX	0.3879	OXBBOBXXB	-0.1965	XOXXXBBOO	-0.2282
BOXBXBOXB	0.4082	OXBBOXBBX	0.4639	XOXXXBOBO	-0.4697
BOXBXOBBX	0.7118	OXBBOXBXB	-0.1510	XOXXXBOOB	-0.2270
BOXBXOBXB	0.6380	OXBBOXOXX	-0.3017	XOXXXOBBO	0.3147
BOXBXOOXX	0.3481	OXBBOXXBB	0.2117	XOXXXOBOB	1.0000
BOXBXXBBO	0.4383	OXBBOXXOX	0.5481	XOXXXOOBB	0.5575
BOXBXXBOB	0.5556	OXBBXBBBB	0.5212	XXBBBBBBO	-0.0149
BOXBXXOBB	0.1443	OXBBXBBOX	0.4145	XXBBBBBOB	0.0197
BOXBXXOXO	0.3139	OXBBXBOBX	-0.1204	XXBBBBOBB	0.0254
BOXOBBXBX	0.1146	OXBBXBXBO	0.5652	XXBBBBOXO	0.0500
BOXOBXBXB	0.0325	OXBBXBXOB	0.5938	XXBBBOBBB	0.0597
BOXOOXXXB	0.3027	OXBBXOBBX	0.5263	XXBBBOBXO	-0.1871
BOXOXBBBX	0.3537	OXBBXOXBB	0.5925	XXBBOBBBB	-0.0725
BOXOXBBXB	0.4798	OXBBXOXOX	0.3845	XXBBOBBOX	0.1466
BOXOXBOXX	-0.1381	OXBBXXBBO	0.4844	XXBBOBBXO	0.0468
BOXOXOBXX	1.0000	OXBBXXBOB	0.3767	XXBBOBOBX	0.0274
BOXOXXBBB	0.5024	OXBBXXOBB	-0.5517	XXBBOBOXB	-0.1847
BOXOXXBXO	0.3079	OXBBXXOOX	-0.0530	XXBBOBXBO	0.2055
BOXOXXOXB	-0.4783	OXBBXXXOO	1.0000	XXBBOBXOB	0.1218
BOXXBBBBB	0.3209	OXBOBBBXX	-0.1113	XXBBOOBBX	-0.0273
BOXXBBXOB	-0.0810	OXBOBXXXO	0.0500	XXBBOOBXB	-0.2282
BOXXBOBBX	0.0391	OXBOOXBXX	-0.1405	XXBBOOOXX	-0.4013
BOXXBOBXB	0.1316	OXBOOXXBX	1.0000	XXBBOOXBB	-0.1713
BOXXBOXOX	-0.0835	OXBOOXXXB	0.0002	XXBBOOXOX	0.1300
BOXXBOXXO	0.1855	OXBOXBBBX	0.1554	XXBBOOXXO	-0.9373
BOXXBXOBB	0.1124	OXBOXBXBB	0.7120	XXBBOXBBO	0.3006
BOXXBXXOO	0.0483	OXBOXBXOX	0.2802	XXBBOXBOB	0.1823
BOXXOBBBX	0.2933	OXBOXOXBX	1.0000	XXBBOXOBB	-0.2124
BOXXOBBXB	0.5765	OXBOXXBBB	0.1790	XXBBOXOXO	-0.2882
BOXXOBOXX	0.4381	OXBOXXBOX	-0.2934	XXBBOXXOO	0.9999
BOXXOBXBB	-0.0844	OXBOXXXBO	1.0000	XXBBXBBOO	-0.1057
BOXXOBXXO	-0.0312	OXBOXXXOB	0.5061	XXBBXBOBO	-0.6589
BOXXOOBXX	0.5278	OXBXBBBBB	0.1978	XXBBXBOOB	-0.4962
BOXXOOXBX	0.1311	OXBXBOOXX	0.0975	XXBBXOBBO	-0.2316
BOXXOOXXB	0.1855	OXBXBXOOX	0.0500	XXBBXOBOB	0.1708
BOXXOXBBB	0.0102	OXBXOBBBX	0.1544	XXBBXOOBB	0.0451
BOXXOXBXO	-0.1334	OXBXOBXBB	-0.0512	XXBBXOXOO	-0.0150
BOXXOXOXB	0.2190	OXBXOOXBX	0.4640	XXBOBBBBB	0.2671

State	Estimation	State	Estimation	State	Estimation
BOXXOXXBO	-0.1426	OXBXOOXXB	-0.0975	XXBOBBBXO	-0.0274
BOXXXBBBO	0.4141	OXBXOXBOX	0.4155	XXBOBBOBX	0.2688
BOXXXBBOB	0.5580	OXBXOXOBX	-0.1343	XXBOBOBXB	-0.0568
BOXXXBOBB	0.5514	OXBXOXOXB	-0.1426	XXBOBOOXX	-0.2748
BOXXXBOOX	1.0000	OXBXOXXOB	-0.1426	XXBOBOXBB	0.0500
BOXXXBOXO	0.4839	OXBXXBBBO	0.4451	XXBOBOXOX	-0.2262
BOXXXOBBB	0.6267	OXBXXBBOB	0.5428	XXBOBOXXO	-0.0975
BOXXXOBOX	1.0000	OXBXXBOBB	0.4105	XXBOBXOBB	0.1308
BOXXXOBXO	0.3662	OXBXXBOOX	0.4787	XXBOBXOOX	0.4013
BOXXXOOBX	0.4062	OXBXXBXOO	1.0000	XXBOOBBBX	-0.2306
BXBBBBBBB	0.4470	OXBXXOBBB	0.4899	XXBOOBBXB	-0.2056
BXBBBBBOX	0.3223	OXBXXOOBX	0.4507	XXBOOBOXX	-0.4867
BXBBBBBXO	0.0331	OXBXXOXBO	-0.4503	XXBOOBXBB	-0.0500
BXBBBBOBX	0.2382	OXBXXOXOB	0.3920	XXBOOXBBB	0.0097
BXBBBBOXB	-0.0631	OXOBBBXXB	0.0558	XXBOOXBOX	0.3446
BXBBBBXBO	0.1571	OXOBBXBBX	-0.0330	XXBOOXBXO	0.4416
BXBBBBXOB	0.3334	OXOBOXBXX	-0.1078	XXBOOXOBX	-0.0872
BXBBBOBBX	0.3082	OXOBOXXBX	0.2437	XXBOOXOXB	0.0022
BXBBBOBXB	-0.0544	OXOBOXXXB	-0.1251	XXBOOXXBO	0.5297
BXBBBOXBB	0.1118	OXOBXBBBX	0.5912	XXBOOXXOB	0.0926
BXBBBOXXO	-0.1264	OXOBXBXBB	0.5125	XXBOXBBBO	0.3343
BXBBBXBBO	0.2190	OXOBXOXBX	0.4182	XXBOXBBOB	0.3533
BXBBBXBOB	0.1435	OXOBXXBBB	0.3366	XXBOXBOBB	0.3468
BXBBBXOBB	-0.0644	OXOBXXBOX	0.4578	XXBOXBXOO	0.3139
BXBBOBBBX	0.0028	OXOBXXOBX	-0.2220	XXBOXOBBB	0.4165
BXBBOBBXB	-0.1441	OXOBXXXBO	1.0000	XXBOXOXBO	-0.4753
BXBBOBOXX	-0.3082	OXOBXXXOB	0.3349	XXBOXOXOB	1.0000
BXBBOBXBB	0.3033	OXOOBXXXB	0.0975	XXBOXXBOO	-0.1109
BXBBOBXOX	0.2050	OXOOXBXBX	0.3646	XXBOXXOBO	0.0162
BXBBOBXXO	-0.0800	OXOOXXBBX	-0.4176	XXBOXXOOB	-0.3445
BXBBOOXBX	0.0633	OXOOXXXBB	0.4922	XXBXBOBOB	-0.0018
BXBBOOXXB	-0.1869	OXOXBBOXX	-0.1405	XXBXBOOXO	-0.0905
BXBBOXBBB	0.3182	OXOXBBXBB	0.0965	XXBXOBBOB	0.1415
BXBBOXBOX	0.0441	OXOXBBXXO	-0.3366	XXBXOBOOX	0.1407
BXBBOXBXO	0.0094	OXOXBOBXX	0.4867	XXBXOOBOX	0.4867
BXBBOXOBX	-0.2371	OXOXBOXXB	0.1855	XXBXOOBXO	-0.2017
BXBBOXOXB	-0.1776	OXOXBXBXO	-0.0500	XXBXOOOXB	-0.0855
BXBBOXXBO	0.0565	OXOXBXXBO	-0.0476	XXBXOXBOO	-0.0073
BXBBOXXOB	0.6875	OXOXOBBXX	0.0905	XXBXOXOBO	-0.2649
BXBBXBBBO	0.2814	OXOXOBXXB	0.0336	XXBXOXOOB	-0.3366

State	Estimation	State	Estimation	State	Estimation
BXBBXBBOB	0.5913	OXOXOXBBX	-0.2262	XXBXXOBOO	-0.3698
BXBBXBOBB	0.4424	OXOXOXXBB	-0.3095	XXBXXOOBO	-0.9764
BXBBXBOOX	0.6721	OXOXXBBBB	0.5072	XXBXXOOOB	-0.0455
BXBBXBXOO	0.4817	OXOXXBBOX	0.5304	XXOBBBBBB	0.0734
BXBBXOBBB	0.6145	OXOXXBOBX	1.0000	XXOBBOBBX	0.0495
BXBBXOBOX	0.5725	OXOXXBXBO	0.0447	XXOBBOXOX	0.2262
BXBBXOOBX	0.3014	OXOXXBXOB	0.3660	XXOBBXBBO	0.0614
BXBBXOXBO	-0.4093	OXOXXOBBX	0.4894	XXOBBXOBB	-0.1573
BXBBXOXOB	0.4414	OXOXXOXBB	-0.4603	XXOBOBBBX	0.1154
BXBBXXBOO	0.2642	OXXBBBBBB	-0.0259	XXOBOBBXB	-0.2109
BXBBXXOBO	-0.5863	OXXBBBOBX	-0.1318	XXOBOBXBB	-0.0377
BXBBXXOOB	-0.1585	OXXBBOBBX	-0.0106	XXOBOBXOX	0.3076
BXBOBBBBX	0.2052	OXXBBOBXB	-0.0166	XXOBOBXXO	-0.3260
BXBOBBBXB	0.0390	OXXBBOXXO	-0.0810	XXOBOOBXX	-0.1855
BXBOBBOXX	-0.2693	OXXBBXOXO	-0.3017	XXOBOOXBX	-0.0634
BXBOBBXBB	0.2994	OXXBOBBXB	-0.0867	XXOBOOXXB	-0.8063
BXBOBXBBB	0.2067	OXXBOBXBB	-0.1807	XXOBOXBBB	-0.0076
BXBOBXBXO	0.0693	OXXBOBXOX	0.1624	XXOBOXBOX	-0.1260
BXBOBXXOB	0.0302	OXXBOOBXX	-0.1787	XXOBOXXBO	0.4475
BXBOOBXXB	-0.0809	OXXBOOXXB	-0.4596	XXOBOXXOB	0.5775
BXBOOXBBX	-0.0309	OXXBOXBBB	-0.1789	XXOBXBBBO	-0.3506
BXBOOXBXB	0.0598	OXXBOXOXB	-0.2262	XXOBXBBOB	-0.0209
BXBOOXXBB	0.7278	OXXBOXXOB	-0.0049	XXOBXBOBB	0.1491
BXBOOXXOX	0.2004	OXXBXBBBO	0.2793	XXOBXBXOO	-0.5015
BXBOOXXXO	-0.3017	OXXBXBBOB	-0.0422	XXOBXOBBB	-0.3720
BXBOXBBBB	0.5287	OXXBXBOBB	-0.3492	XXOBXOXOB	0.0268
BXBOXBBOX	0.4630	OXXBXBOOX	-0.5418	XXOBXXBOO	-0.6130
BXBOXBOBX	-0.4425	OXXBXOBBB	0.5255	XXOBXXOBO	-0.1498
BXBOXBXBO	0.3718	OXXBXOBOX	0.3627	XXOBXXOOB	0.2480
BXBOXBXOB	0.5886	OXXBXOOBX	-0.5591	XXOOBBBXB	-0.0544
BXBOXOBBX	0.7440	OXXBXXBOO	-0.4143	XXOOBBOXX	-0.0975
BXBOXOXBB	0.3908	OXXBXXOBO	-1.0000	XXOOBBXOX	0.0500
BXBOXOXOX	1.0000	OXXBXXOOB	-0.9562	XXOOBBXXO	-0.1404
BXBOXXBBO	0.4202	OXXOBBBXB	-0.0752	XXOOBOBXX	-0.0975
BXBOXXBOB	0.2371	OXXOBBXBB	0.1402	XXOOBOXBX	0.1293
BXBOXXOBB	-0.0745	OXXOBBXOX	0.2649	XXOOBXOBX	0.1448
BXBOXXOOX	-0.0756	OXXOBBXXO	0.0395	XXOOBXOXB	0.0500
BXBOXXXOO	0.3361	OXXOBOXXB	-0.1324	XXOOOBBXX	-0.3017
BXBXBBBBO	-0.0581	OXXOOBBXX	-0.4596	XXOOOBXXB	-0.3073
BXBXBBBOB	0.1059	OXXOOBXBX	0.3115	XXOOOXBBX	-0.1311

State	Estimation	State	Estimation	State	Estimation
BXBXBBOBB	0.0975	OXXOOBXXB	-0.4013	XXOOXBBBB	0.4583
BXBXBBOOX	0.0500	OXXOOXBXB	-0.2262	XXOOXBXBO	-0.4657
BXBXBBXOO	0.0705	OXXOOXXBB	-0.0043	XXOOXBXOB	0.5114
BXBXBOBBB	0.2648	OXXOXBBBB	-0.4925	XXOOXOXBB	0.0012
BXBXBOOBX	0.0158	OXXOXBBOX	-0.2395	XXOOXXBBO	0.3834
BXBXBOOXB	0.0548	OXXOXOBBX	0.1460	XXOOXXBOB	0.3761
BXBXBXOBO	-0.1443	OXXOXXBBO	-0.3039	XXOOXXOBB	1.0000
BXBXOBBBB	0.0928	OXXOXXBOB	-0.2822	XXOXBBOXO	-0.3017
BXBXOBBXO	-0.0721	OXXXBOOXB	0.0500	XXOXBOOXB	-0.1855
BXBXOBXBO	-0.0875	OXXXBOXBO	0.1426	XXOXBXBOO	0.0500
BXBXOBXOB	0.0397	OXXXBXOBO	-0.2262	XXOXBXOBO	-0.1855
BXBXOOBBX	0.3176	OXXXOBBBB	-0.0499	XXOXOBBXO	-0.2649
BXBXOOOXX	-0.1713	OXXXOBBOX	0.5310	XXOXOOBBX	0.1660
BXBXOOXOX	0.5155	OXXXOBOBX	0.4075	XXOXOOBXB	-0.9694
BXBXOXBBO	-0.1262	OXXXOBXOB	-0.1847	XXOXOXBBO	-0.1000
BXBXOXBOB	0.0404	OXXXOOBXB	-0.1179	XXOXOXBOB	-0.2177
BXBXOXOBB	-0.2056	OXXXOOXBB	-0.1334	XXOXXBBOO	-0.9764
BXBXOXOOX	0.0279	OXXXOXBOB	0.2105	XXOXXBOBO	-1.0000
BXBXOXXOO	0.0975	OXXXOXOBB	-0.2556	XXOXXBOOB	-0.0311
BXBXXBBOO	0.0009	OXXXXBBOO	-0.0027	XXOXXOBOB	-0.2153
BXBXXBOBO	-0.3952	OXXXXBOBO	-0.0595	XXOXXOOBB	-0.3182
BXBXXBOOB	0.2792	OXXXXBOOB	-0.4590		
BXBXXOBBO	-0.0594	OXXXXOBBO	1.0000		
BXBXXOBOB	0.2314	OXXXXOBOB	0.2808		
BXBXXOOBB	0.4898	OXXXXOOBB	0.5201		
BXBXXOOOX	0.4779	XBBBBBBBB	0.5518		
BXBXXOXOO	0.0865	XBBBBBBOX	0.0520		
BXOBBBBBX	0.1944	XBBBBBBXO	0.1727		
BXOBBBBXB	-0.1032	XBBBBBOBX	0.0826		
BXOBBBXBB	0.2570	XBBBBBOXB	0.3194		
BXOBBBXXO	-0.1620	XBBBBBXBO	0.1491		

References

Balkenius, C. (1994). “Biological Learning and Artificial Intelligence”, Lund University Cognitive Studies, LUCS 30.

Brooks, R. A. (1991). “New Approaches to Robotics”, Science, 253, pp. 569-595.

Dayan, P. (1992). “The Convergence of TD(λ) for general λ”, *Machine Learning,* 8 v.3, pp. 341-362.

Griffith, A. K. (1974). “A Comparison and Evaluation of Three Machine Learning procedures as Applied to the Game of Checkers”, *Artificial Intelligence*, v.5, pp. 137-148.

Hertz, J., Anders, K., Palmer, R.G. (1991). *Introduction to the Theory of Neural Computation.* Addison-Wesley, Redwood City, CA.

Hofstadter, D. R. (1979). *Gödel, Escher, Bach: An Eternal Golden Braid*, Basic Books, Inc., New York, New York.

Kaelbling, L., Littman, M., Moore, A. (1996). “Reinforcement Learning: A Survey”, *Journal of Artificial Record,* v.4, pp. 237-285.

Lorenz, K. (1977). *Behind the Mirror,* Methuen & Co., Ltd., London.

Macfarlane, D. A. (1930). “The Role of Kinesthesis in Maze Learning”, *University of California Publications in Psychology*, v.4, pp. 277-305.

Nilsson, Nil J. (1996). “Introduction to Machine Learning”, Nil J. Nilsson.

Premack, D. (1971). “Catching Up on Common Sense, or Two Sides of a Generalization: Reinforcement and Punishment”, *On the Nature of Reinforcement*, edited by Glaser, R., New York: Academic Press, New York.

Russel, S., Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*, Prentice Hall, New Jersey.

Samuel, A.L. (1959). “Some Studies in Machine Learning Using the Game of Checkers”, *Computers and Thought,* edited by Feigenbaum, E. and Feldman, J., MIT Press, 1995.

Shannon, C.E. (1950). “Programming a computer for playing chess”, *Philosophical Magazine*, v. 41, pp. 256-275.

Shapiro, Stuart C. (1987). *Encyclopedia of Artificial Intelligence*, John Wiley and Sons, Inc., New York, New York, 1987, pp 88-93.

Sutton, R.S. (1988). “Learning to Predict by the Method of Temporal Differences”, *Machine Learning*, v. 3, pp. 9-44.

Sutton, R. S., Barto, A. G. (1990). “Time-Derivative Models of Pavlovian Reinforcement”, *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, edited by Gabriel, M, Moore, J., MIT Press, Cambridge, MA, pp. 497-538.

Sutton, R.S., Bonde, A. Jr. (1992). “Nonlinear TD / Backprop pseudo C-code”, File from Richard Sutton’s Website, © 1993 GTE Laboratories Incorporated.

Sutton, R. S., Barto, A. G. (1998). *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA.

Tesauro, G, Sejnowski, T.J. (1989). “A Parallel Network that Learns to Play Backgammon”, *Artificial Intelligence*, 1989, No. 39, pp. 357-390.

Tesauro, G, (1992). “Practical Issues in Temporal Difference Learning”, *Machine Learning*, 1992, v. 8, pp. 257-277.

Tesauro, G. (1994). “TD-Gammon, a Self-Teaching Bachgammon Program, Achieves Master-Level Play”,*Neural computation*, v. 6, No. 2, pp. 215-219.

Tesauro, G. (1995). “Temporal Difference Learning and TD-Gammon”, *Communications of the ACM*, 1995, v.38, No. 3, pp. 58-68.

Thorndike, E. L. (1911). *Animal Intelligence.* Hafner, Darien, CT.

Turing, A. M. (1950). “Computing Machinery and Intelligence”, *Mind*, v. 59, 1950, pp. 433-460. Reprinted *Computers and Thought*, edited by Feigenbaum, E. and Feldman, J., MIT Press, 1995.

Webster’s New World Dictionary, (1982) edited by Guralnik, D. B., Simon & Schuster.