# Temporal Difference Learning in the Game of Othello

## CS701 Final Project Report


Will Ernst
Middlebury College
Middlebury, VT
wernst@middlebury.edu

Ghassan Gedeon Achi
Middlebury College
Middlebury, VT
ggedeonachi@middlebury.edu



## ABSTRACT

*This paper explores the implementation of the temporal-difference learning algorithm TD($\lambda$) to train an Artificial Intelligence agent to play the board game 'Othello'. The training strategies compared are: learning from self-play, learning against a fixed opponent, and learning against a random opponent. These training strategies are used on two player types, a constant semi-random player and a decreasingly semi-random player. The players are trained using a multi-layer perceptron neural network, which is updated using the TD($\lambda$) algorithm. Our results show that the algorithm trains the network best with self play and an added stochastic element, and that for our basic network setup, $\lambda = 1$ yields the best results. A network trained in this way was able to decisively beat an Alpha-Beta Search Algorithm. We conclude that our implementation works decently to train an Othello player, but increased training iterations and improved training methods would likely yield better success.*



## Keywords

CS701; TD($\lambda$); Othello; Reinforcement Learning; Temporal Difference Learning; Neural Networks


## 1. INTRODUCTION

This paper adapts Gerald Tesauro's implementation of the TD($\lambda$) algorithm on the game of backgammon to the game of 'Othello' (also known as 'Reversi' ). The game is played on an 8x8 board and works by having 'Black' and 'White' players take turns placing pieces on the board. The initial board configuration has 4 pieces, 2 for each player, placed in the center of the board in a diagonal configuration [Figure (1)]. In Othello, Black goes first; all subsequent moves by the players must flank one or more of the opponent's pieces−meaning that you must place your piece such that it surrounds one or more of the opponent pieces either vertically, horizontally or diagonally. Flanked pieces are captured and are replaced with pieces of your own color. The game goes on until neither player has any valid moves. Once the game has ended the player with the most pieces wins. The project was motivated by our interest in getting some experience with the emerging field of machine learning, as well as our interest in developing an agent capable of strong play while being trained completely independently.

In order to build an Othello player, we need a way to evaluate the states of the game (configurations of the board). Because the game tree size for 8x8 Othello is estimated at $10^5$4, a look up table or similar approach is not practical.

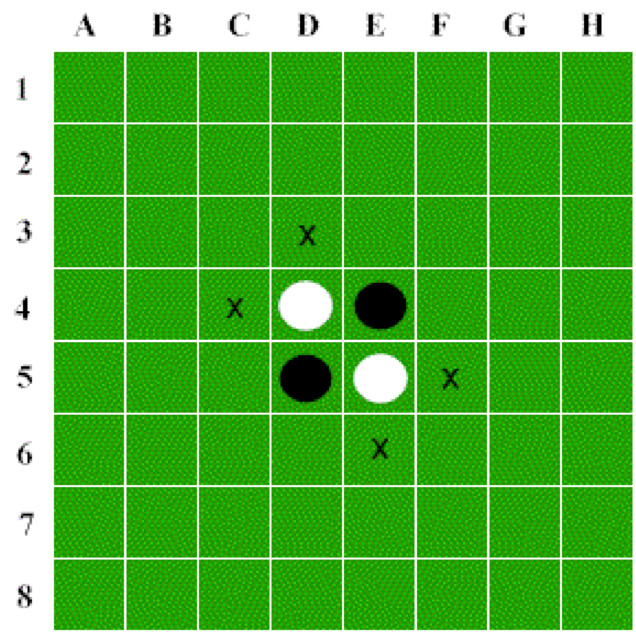



Figure 1: Initial board state for Othello, where x's are valid moves for Black

Instead, we use a neural network to evaluate states. We use a Neural Network with 1 input layer, 1 hidden layer and 1 output layer. The input of our network is a vectorized representation of the board state and the output of the network is its associated evaluation. To calculate the output of a given board state, the input is fed through the network. The easiest way to visualize this process is to think of the neurons as a node. Each of the input neurons is connected to every one of the neurons in the hidden layer, and each of the hidden neurons are connected to the output layer [Figure (2)]. Each connection has a corresponding weight, and the output of each neuron is equal to the sum of all the inputs multiplied by their weight. To normalize and smooth out these connections, we use a sigmoid function. This returns an output between 0 and 1 for all neurons in our Network. Once a board state has been fed through the network, we can evaluate its strength based on the final output of the network. The final output of the network can be interpreted as the probability of Black winning from that state. Given this interpretation, Black will choose a move that maximizes this output, and White will choose a move that minimizes this output.

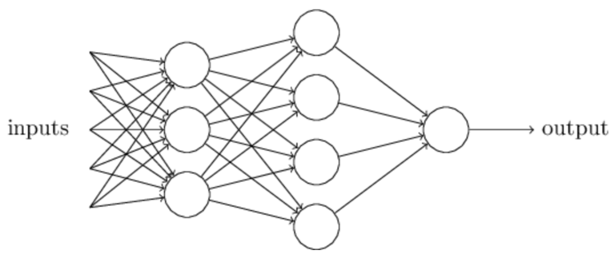



Figure 2: A simplistic representation of a Neural Network with 1 hidden layer and 1 output layer

At each time step−after every move−the network calculates the error associated with the given move and back-propagates this error through the network. Back-propagation is the process of updating the weights of the connections to correct for the error. The goal of the back-propogation procedure is to update the weights in a way that this error is minimized. Finally, at the end of each game a pre-defined reward is given for a win, loss, or tie. The values of these final rewards are what defines the interpretation of the network output. Further details on how this back-propagation works will be discussed in our Methods section.

Othello is a deterministic game, in that it contains no stochastic elements, such as a dice roll. In training though, the addition of a stochastic element can be helpful to explore unseen board states. As such, we use a stochastic element in our training iterations. We call this element the exploration rate ($\epsilon$), and determines how often the player will choose a random move during training, as opposed to choosing the optimal move determined by the Network.

In this project, we have devised two types of networks trained with distinct implementations of the exploration rate. The first of these networks, which we named NN-Fixed, has a fixed exploration rate. The second of these Networks, this one called NN-Decreasing, has an exponentially decreasing exploration rate. The analysis of these two-distinct implementation is our project's main contribution, and sheds some light on the optimal implementation of temporal difference learning in a deterministic environment.

Our second contribution is seeing how each of these networks perform with different Lambda ($\lambda$) values. The $\lambda$ value determines the rate of decay in error correction for previous time steps. After some testing, we found that high lambda values yielded the best results, so we chose to test the extreme case $\lambda = 1.0$ vs a slightly less drastic case $\lambda = 0.9$.

Our third contribution is seeing how the network learns versus various opponents. For this purpose, we created 3 additional opponents against which our networks train against. These opponents are: (1) a fully *Random* agent whose moves are selected randomly, (2) a *Positional Value* opponent, where tiles have a hardcoded value based on how good or bad they are in traditional play [7], (3) an AI agent using the *Alpha-Beta* search. In our training experiments, both NN-Fixed and NN-Decreasing are trained against opponents (1) and (2), as well as against themselves, using $\lambda = 1.0$ and $\lambda = 0.9$ . An analysis of these contributions will be covered in our Results section where we evaluate the performance of the various combination of Network types, lambda values and opponent type. All networks are trained for $125,000$ games, and their performance against a random agent for 500 games is used as a measure of strength. Additionally, we use the score of the networks against (2) *Positional-Value* agent and (3) *Alpha-Beta* agent as a gauge of performance over time.

Our final contribution is an implementation of a user-friendly GUI interface for the Game, where the user can choose to play against any of our trained Neural Network, or any of the other agents that we created. As of now the selection of opponent is code based, but in the future we are hoping to create a full fledged web application where the user can select the opponent of choice from a drop down menu.

# 2. RELATED WORK

## 2.1 TD-Gammon

The work of this paper was motivated by Gerald Tesauro's TD-Gammon [6]. TD-Gammon, developed in 1992, is a backgammon program that uses an artificial neural network trained by the TD($\lambda$) algorithm. The program was a major success at the time, and led to new developments in backgammon strategy and theory.

The program used a standard multilayer perceptron network as a function approximator for the probability of the black player winning at a certain state. The network uses the the TD($\lambda$) back-propagation algorithm to update its weights. The goal of the weight updates, as it is with most temporal difference learning algorithms, is to minimize the temporal-difference error. This error is the difference of the neural network output in time $t + 1$ and the neural network output in time $t$.

The TD gammon program runs as follows: at each timestep the program calculates each possible move using a one ply look ahead (this means that it only looks one move ahead of its current state). Those moves are vectorized into neural network inputs and are fed into the network. The feed-forward output of the network is a vector of four outputs corresponding to the four possible outcomes of either White or Black winning either a normal win or a gammon. Due to extreme rarity, triple gammon end games were not included. The best value for the player is chosen and the weights are updated using TD($\lambda$). When a game is over, instead of calculating a new move, the board state is assigned a vector (for the four endgame possibilities) with a 1 assigned to the observed endgame state and 0s assigned to the rest.

During training, the neural network itself is used to select moves for both sides. With this training paradigm, the program only learns from what its own moves are. This paradigm is used throughout the entire training, even at the beginning when the weights of the network are random. Although at the beginning the network has no sensible strategy, it is improved through self play as it observes successful sequences of moves. The results of TD-Gammon were impressive. Although the initial state of the program had no knowledge of the game, it was able to develop basic strategies for the game early on, such as hitting the opponent, playing safe, and building new points. This was observed after just a few thousand games. After several tens of thousands of training games, more sophisticated strategies were observed. In examination of the weights from the input to hidden layer, the network revealed interesting spatially organized patterns of positive and negative weights, corresponding to useful features of the game. This implies that TD-Gammon was capable of automatic "feature discovery", which is one of the original goals of game learning research.

To evaluate the success of the program, it was tested against world-class human opponents. The first iteration of the program, version 1.0, achieved respectable results against Bill Robertie, Paul Magriel, and Malcolm Davis, (ranked 11th in the world at the time), with a net loss of 13 points and an average loss rate of about 1 quarter point per game. Version 2.0 of the program added more training iterations and a 2-ply look ahead. Playing Kent Goulding, Kit Woolsey, Wilcox Snellings, former World Cup Champion Joe Sylvester, and former World Champion Joe Russell, the program had a net loss of only 7 points. Version 2.1 of the program achieved near parity to Robertie, who only managed to beat the program in the last game resulting in a narrow 1-point victory.

## 2.2 Othello research

Previous research has been done on using reinforcement learning (RL) algorithms to solve the game Othello. Work by Ree et al. explores Othello with three learning strategies: Learning by self-play, learning from playing against a fixed opponent, and learning from playing against a fixed opponent while learning from the opponent's moves as well. Their work examines three commonly used RL algorithms: Q-learning, Sarsa, and TD-learning. Results from their testing show that each algorithm has its own optimal training strategy. Q-learning and Sarsa performed best when trained along with a fixed opponent, while TD-learning performed best when trained through self-play. Additionally they found that learning from the opponent's as well as the player's own moves moves generally performed worse than just learning from the player's own moves [7].

# 3. METHODS

## 3.1 Network Layers

To implement the TD($\lambda$) algorithm we use a neural network with an input layer with 64 neurons, one hidden layer with 50 neurons, and an output layer with single neuron. The output neuron is used to evaluate the board state and can be interpreted as the probability that Black wins from that state.

Since neural networks require vector inputs, we convert the 8x8 board state into a vector of size 64 with value of 0, 1 and $-1$, which respectively correspond to an empty tile, tile containing a Black piece and a tile containing a White piece.

The hidden layer is comprised of 50 neurons. This choice comes from testing various implementations of the network, and picking the network that showed strong learning performance while remaining not too computationally demanding. Since each additional hidden neuron adds 65 connections$-$64 to the input layer and 1 to the output layer$-$the computational cost of adding one or more is rather steep.

## 3.2 Sigmoid Neurons

Perceptrons were developed in the 1950's and 1960's by a scientist named Frank Rosenblatt, who was inspired by the earlier work of Warren McCulloch and Walter Pitts. These are the most basic type of neurons that can be used in a

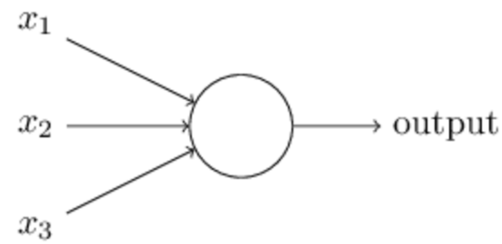


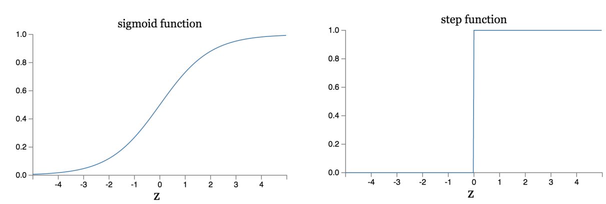

Figure 3: Visual representation of a neuron in a Neural Network

neural network. Perceptrons take inputs with corresponding weights, and output a value of 0 or 1. The output is normalized based on a threshold. If if the weighted sum of the inputs is less than the threshold, then the output is 0, and the output is 1 otherwise [Figure (3), Equation (1)][1].

$$output = \begin{cases} 0, & \text{if } \sum_j w_j x_j \leq Threshold \\ 1, & \text{if } \sum_j w_j x_j > Threshold \end{cases} \quad (1)$$

$$output = \frac{1}{1+e^{-z}} \quad (2)$$

The issue with perceptrons is that small changes in the weights can cause major changes in the output of the neuron, as such perceptron networks for complicated task can be very hard to train. With that in mind, we chose to use sigmoid neurons for our implementation of the networks. Visually these look the same as perceptrons [Figure(3)], but the output of the neuron is passed into a sigmoid function [Equation (2)], where $z$ is the sum of the weights multiplied by the input of the neuron. This returns a value between 0 and 1. A graphical representation of the outputs of perceptron vs sigmoid neurons can be seen in [Figure (4)].




(a) Percetron output value (b) Sigmoid output value

Figure 4: Different between Perceptron and Sigmoid output

## 3.3 TD($\lambda$) Algorithm

The TD($\lambda$) algorithm was designed by Richard Sutton in 1988. Most prediction learning methods at the time assigned credit by means of the difference between a predicted and actual value. Sutton's method, on the other hand, assigned credit by means of the difference between temporally successive predictions [2]. The network weights are updated according to the following rule [3]:

$$w_{ij}^{t+1} = w_{ij}^t + \alpha(P^{t+1} - P^t)e_{ij}^t \quad (3)$$

where:

$w_{ij}^t$ is the weight at time $t$ from node $i$ to $j$

$\alpha$ is the learning rate parameter

$O$ is the 'Set' of outputs given by the output layer

$P_k^t$ is output value of the output node $k$ at time $t$

$e_{ij}^t$ is the eligibilty trace matrix at time t

The weights are updated through this method in every turn of the game. The TD error is calculated as [3]:

$$(P^{t+1} - P^t) \tag{4}$$

The goal of the program is to minimize this error. If it does so, then the algorithm will have found a sequence of moves that perfectly predicts the outcome of the game. The learning rate, $\alpha$, is a weight parameter for the error to determine how much the total error will affect the weight change. The eligibility matrix is used in the algorithm to determine which weights are 'eligible' to be updated. When TD error occurs and learning needs to be done, only eligible states are assigned credit for the error. This matrix keeps a running memory of every credit assignment that has occurred in the training. The matrix is calculated as follows [3]:

$$e_{ij}^t = \sum_{n=1}^{t} \lambda^{t-n} \frac{\partial P^n}{\partial w_{ij}^n} \tag{5}$$

where:

$\lambda$ is the decay parameter for the eligibility matrix

The eligibility matrix is running sum of the previous credit assignments in the network [4]. Previous assignments are decremented at each timestep by the value lambda, so that credit assignments early in training have less of an effect on current credit assignments. The credit assignment is based on a matrix of partial derivatives of the network output with respect to each weight. This determines how much each weight affects the output of the network, and thus how much credit it should get for the error. These partial derivatives are calculated as follows [3]:

$$\frac{\partial P^t}{\partial w_{ij}^t} = \delta_j^{t+1} y_i^{t+1} \tag{6}$$

$$\delta_i^t = \frac{\partial P^t}{\partial s_i^t} = \begin{cases} y_i^t(1-y_i^t), & \text{if } i \in O \\ \sum_j \in FO_i \frac{\partial P^t}{\partial s_j^t} \frac{\partial s_j^t}{\partial y_i^t} \frac{\partial y_i^t}{\partial s_i^t} & \\ = \sum_j \in FO_i \delta_j^t w_{ij}^t y_i^t (1-y_i^t) & \text{otherwise} \end{cases} \tag{7}$$

where:

$\delta_i^t$ is the error at node $i$

$y_i^t$ is the output of node $i$ at time $t$ (where $i$ is in the layer before $j$)

From these equations, we can see that the final change for each weight is the a weighted (by $\alpha$ ) temporal difference error again weighted by how much credit that weight has in the final outcome. The inclusion of a lambda value is considered a bridge between one step TD and Monte Carlo methods [4]. For $\lambda = 0$, only the current credit assignment is factored into the weight update, while for $\lambda = 1$, all previous credit assignments are equally factored into the weight update. Setting $\lambda$ to a value between 0 and 1 gives the result algorithm a mix of both approaches.

Learning and Training: To teach the network, multiple games are played. For the results presented in this paper, the network was trained with $125,000$ games. Each epoch of training runs as follows:

1. While $(iterations < totaliterations)$
2.     If black move:
3.         Move and learn
4.     If white move:
5.         Move and learn

Note that with each game, the eligibility matrix is reset, but the weights of the network are not. This is because the eligibility matrix is supposed to keep track of previous moves in the game. For all games, the move in one game is independent of the moves of another, so we do not want to eligibility matrix remembering from past games. In this implementation of TD($\lambda$), while playing a game each move represents a time $t$. At each move the program executes the following:

1. Observes the current state $s^t$
2. If $s^t$ is endgame:
3.     Set $s^{t+1} = 0$ if white wins, $s^{t+1} = 0.5$ if tie, $s^{t+1} = 1$ if black wins
4. Else:
5.     Given $s^t$, the program calculates $A$ the possible afterstates of $s^t$
6.     For each $a$ in $A$:
7.         Vectorize $a$ into an input for NN
8.         The NN calculates the output of the of $a$
9.         Set $a$ to $s^{t+1}$ if it has the greatest output value
10.     Vectorize $s^t$
11.     Place a piece on the board in the position corresponding with state $s^{t+1}$
12. Given $s^t$ and $s^{t+1}$, run TD( $\lambda$) to adjust the weights

In order to understand the output of the neural network, specific rewards should be passed at the end of the game. These rewards are 1 if Black wins, 0 if White wins, and 0.5 if there is a tie. These values are chosen because at a state when Black wins, Black has a 100% chance of winning. The same can be seen for the other values. The error based on these rewards is passed through to the previous moves made by the player in through the eligibility matrix. Thus, these moves get credit for reaching the reward given at the end of the game.

## 3.4 Player Types

*NN* – This player is the standard neural network player. It decides its moves based on the neural network output. We found that this player did not train the network well during self play. In training, this player does a poor job at exploring a wide variety of different states. This is because Othello lacks a stochastic element. So, when a network trained by this player was given a state that it had not seen before, it performed poorly. This is the player used for testing the neural network.

*NN-Fixed* – This player decides its moves randomly a preset percentage of the time and moves based on the neural network otherwise. The benefit of this player in training is that it has a stochastic element that the Neural Network player does not. To implement this, at each move a random number between 0 and 1 is selected. If the random number is above some threshold, then the player moves based on the neural network and moves randomly otherwise. We call this threshold the exploration rate ($\epsilon$). This player is mainly used in training, and shows better training results than the Neural Network player.

*NN-Decreasing* – This player decides its moves randomly a decreasing percentage of the time and moves based on the neural network otherwise. This player has similar benefits to the Semi Random Neural Network player, but reinforces more of its moves in later iterations of training. $\epsilon$ is calculated by an exponential function:

$$\epsilon = -(1-a) * e^{\frac{10(i-n)}{i}} + a \tag{8}$$

Where:

$a$ is the initial exploration rate

$i$ is the current iteration

$n$ is the total number of iterations

At the point when $i = n$, the threshold is 1 and chooses only from the neural network.

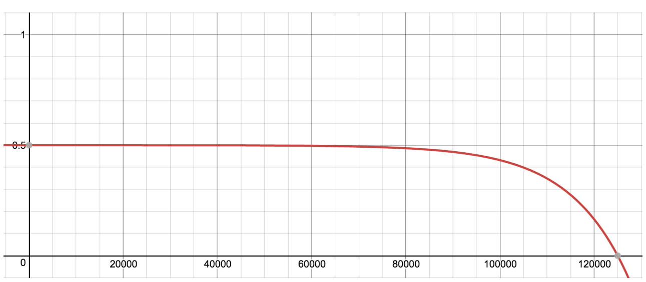



Figure 5: Graphical Representation of the exponentially decreasing learning rate with a = 0.5 , n = 125,000

*Random* – This player decides all its moves randomly. This player is mainly used in testing. Because Othello has no stochastic element, it is difficult to test a trained neural network against non-stochastic players. In these cases of testing, the same game is played every time. With this player, a different game is played in each round of testing so a more accurate win percentage for the network can be calculated.

*Position Values* – This is a fixed player with a heuristic value assigned to each of the 64 board tiles. It makes its moves based on the maximum tile value of the possible moves. This player is mainly used in testing.

*Greedy* – This player makes its moves in order to maximize its score. It uses a one ply look ahead to determine which move is best. This player is mainly used in testing.

*Alpha-Beta* – This player implements the alpha-beta pruning algorithm to its best possible move. The alpha-beta pruning algorithm is a tree search algorithm that minimizes the number of nodes visited in a minimax tree. It is able to forecast possible board states for $k$ timesteps in the future. It selects the move that maximizes its score at state$^{t+k}$, assuming that the other player also moves to maximize its score. This player is mainly used in testing.

- The Source Code for our project can be found at the following link *https://github.com/wernst/td-othello*

- The Repository also contains a Subset of the Networks we saved during training, a text file containing the <*Iterations*><Win$*Percentages*> during training for the respective networks, and finally vectorized version of the graphs found in the Results Sections.

# 4. RESULTS AND DISCUSSION

In the following section we analyze (1) the learning performance of the implementation of a exponentially decreasing exploration rate ($\epsilon$ starting at 0.5) as opposed to a fixed exploration rate ($\epsilon = 0.5$), (2) the effect of setting $\lambda = 0.9$ vs. setting $\lambda = 1.0$ and (3) how our Self-trained Neural Networks performed against our strongest competing Agents *Alpha-Beta* and *Positional-Value*. Finally, (4) we will briefly cover GUI applet that we have developed for users to have a convenient interface which they can use to play against the agent of their choice. Our best implementations reached performances against a random agent of around 90%.

## 4.1 NN Dec vs NN Fixed

Looking at Figure (6) and Figure (7), we see very similar patterns in learning for the two different types of network. Given the implementation of the exponential exploration decay, we were expecting the networks to behave very similarly until we reach game 80, 000, at which time the exploration rate would start decreasing. We had initially hypothesized that when this happened *NN-Decreasing* would start refining its weights to discover subtler patterns in play style that *NN-Fixed* would be unable to find. This, however did not happen. In both models, and against all opponents, we saw very similar win percentages. There is one small difference in the models that is quite interesting, but somewhat hard to spot. If we carefully look at the difference in win percentages between Figure (6b) and Figure (7b), specifically for the *red* line representing self-play, we notice that the line in Figure (7b) is much smoother than Figure (6b) and this starts right around the 80, 000 games. This is likely due to the exploration going down and the Network converging towards its final (and ideally optimal) weight configuration. Overall the decreasing exploration rate does not seem to provide too much of a benefit, however if a modified form of this decaying exploration rate were implemented right after the network reaches its maximum win percentage, it could prevent the network from experiencing the sharp dips that can be seen throughout the training. This theorized *adaptive* exploration rate $\epsilon$ is one possible avenue for future research.

## 4.2 Lambda Value Choice

During the course of training, both Networks – *NN-Decreasing* and *NN-Fixed* – with $\lambda = 0.9$ performances suffered right around reaching the 50, 000 iteration mark [Figure (6), Figure (7)]. This decrease in performance seems to happen right after the network reaches optimal performance, but our Networks with $\lambda = 1.0$ didn't seem to suffer from the same issue. During the period of decline, the values outputted by the networks with $\lambda = 0.9$ approached 1. By the end of training, the evaluation of almost every board state would output a value $> 0.99$. Although we are unsure of exactly why this occurs, we did notice that this pattern occurred earlier with larger $\alpha$ values. Although the goal of the TD($\lambda$) algorithm is to converge to the global minimum of the function approximator, this convergence has only been proven for linear networks and linearly independent sets. In the general case, there is no guarantee that the algorithm will converge even to a local minimum [5]. This finding seems to suggest that for our implementation of the Neural Network a $\lambda = 1.0$ is optimal, but more research would have to be done to fully conclude this.

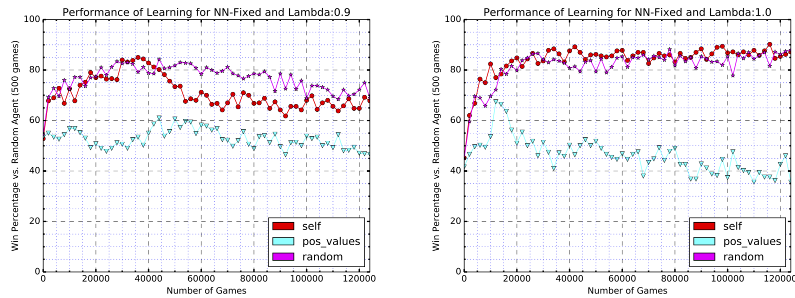
## 4.3 Opponent Types

Having established that the $\lambda = 0.9$ causes a detrimental form of convergences in our network, let us now turn our attention back to Figures (6b) and Figure (7b) to examine the way that the various opponents it trained against affected the Learning Rate.

*Positional-Value* $-$ Immediately it becomes clear that the network struggles to learn when playing against Positional Value. We had expected that the network would have a hard time beating the agent in the early stages of the game. But we also expected the network (especially *NN-Decreasing*) to eventually find a weight combination that would allow it to defeat the *Positional-Value* agent and start learning from playing against it. However, we notice that the performance against *Random*, stagnates around or even below the 50%. When we examined the output, we noticed that *Positional-Value* was winning a significant percentage of the games, which means that neither of our NN were receiving any reward and as such the network was unable to optimize its weight to find a combination that worked best.

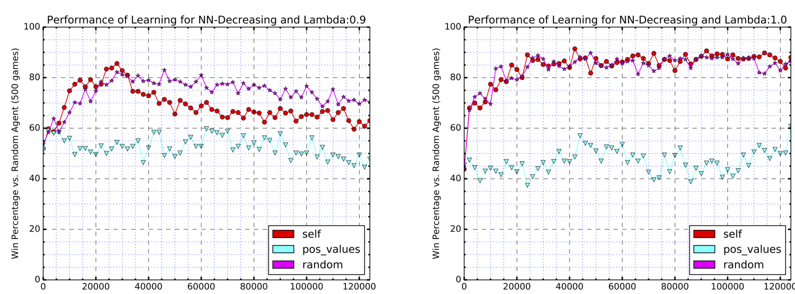
*Random* $-$ We were rather surprised at how well our network learned from a fully random opponent. We had not expected that our Neural Networks would learn almost as well against a fully random opponent, as it would against itself. This seems to confirm that high exploration rate allow for quicker learning. It should be noted that in the latter stages of training, the Networks training against a random opponent experienced more spikes and dips in learning that those learning from self-play. Additionally, these results may be biased as the testing benchmark is against *Random*. In these networks' performance against stronger players the results were not impressive.

*Self-Play* $-$ as we had expected this showed the best results, reaching a win percentage against Random of around 90%. Since this model seemed worked best, we decided to conduct further experiments and displayed our Networks score vs. both the *Positional-Value* agent, and *Alpha-Beta* with a max depth of 4 at various intervals in our training, see [Figure (8)] and [Figure (9)]. In Othello the score can quickly switch since a single move can cause a significant of the board to switch colors. As such, scores are not always the best indicator of learning, but we were curious to see if we noticed any patterns in the data. In general, being above the 32-point threshold-indicated by the blue line should indicate a win (however there are rare exceptions). Unfortunately, no subtle pattern emerged, but on a general case, we noticed that our data-points were generally above




(a) NN-Fixed with Lambda 0.9 (b) NN-Fixed with Lambda 0.9

Figure 6: Graphs for Neural Network with Fixed Exploration Rate




(a) NN-Decreasing with Lambda 0.9 (b) NN-Decreasing with Lambda 0.9

Figure 7: Graphs for Neural Network with Decreasing Exploration Rate

the 32-point threshold against *Alpha Beta*, implying it was able to defeat the *Alpha Beta* player consistently. We theorize that our models moves are much superior to *Alpha-Beta* in the early games, since *Alpha-Beta* has the advantage of looking 4 moves ahead during the end game.

Against *Positional-Value* [Figure (9)], the results were quite poor, and this seems to further reinforce what seemed to be true in training, which is that *Positional-Value* is a very strong opponent.

## 4.4 GUI implementation

The final contribution of this paper is a user-friendly environment in which the player can play against an agent of his choice. The Layout of the applet can be seen in Figure (10). We hope to be able to get the applet online soon. Once this happens, the applet's url will be linked from the following addresses:

- http://www.cs.middlebury.edu/ wernst
- http://www.cs.middlebury.edu/ ggedeonachi

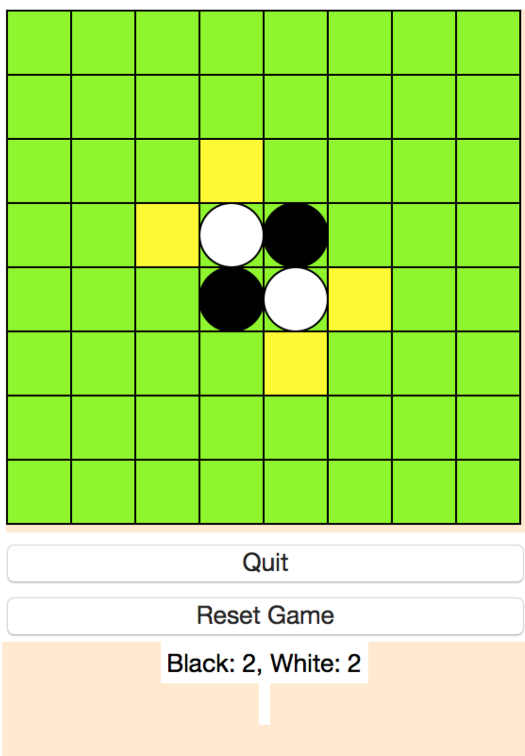
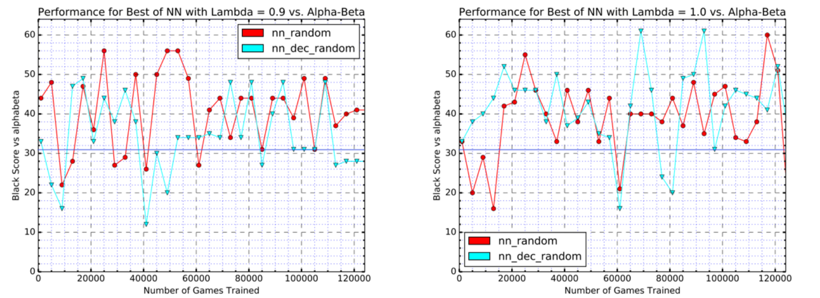



Figure 10: Graphical User Interface for Othello

# 5. CONCLUSION

In this project we compare two $\lambda$ values (0.9 and 1.0) in their ability to train an Othello player using the TD($\lambda$) algorithm. We found that both values worked best training with self-play. Using self play as the training paradigm, $\lambda = 1$ worked best for our TD($\lambda$) implementation on training an Othello player. This version of the algorithm is most similar to Monte Carlo methods [4] .

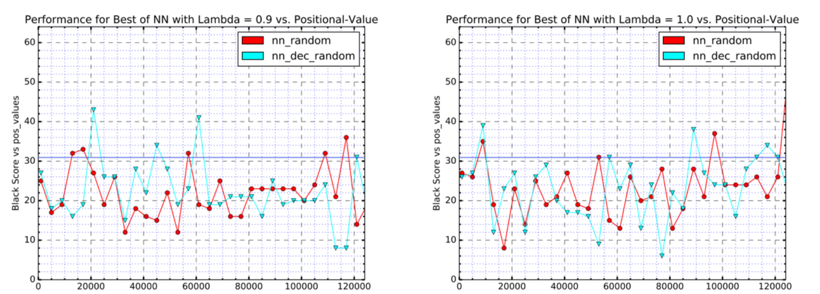
For future work with respect to our implementation, we would like to continue to explore combinations of $\alpha$ values, $\lambda$ values, network setups, and randomness to determine which results in the best training inputs. Using more advanced




(a) NN with Lambda 0.9 vs. Alpha-Beta

(b) NN with Lambda 1.0 vs. Alpha-Beta

Figure 8: Score of Self-Trained Neural Networks vs. Alpha-Beta




(a) NN with Lambda 0.9 vs. Positional-Value

(b) NN with Lambda 1.0 vs. Positional-Value

Figure 9: Score of Self-Trained Neural Networks vs. Positional-Value

network techniques, such as changing of $\alpha$ values and $\lambda$ values through the training process is another possibility. We also would like to determine what caused our network during training to inflate all its outputs toward 1, when $\lambda < 1$. In general, we feel that this method of learning could be expanded to number of board games.

## 6. ACKNOWLEDGMENTS

The authors would like to thank Professor Daniel Scharstein and Professor Ananya Christman for providing assistance on this project. Additionally, the authors would like to thank Professor Jeff Dunham for allowing access to his computational resources.

## 7. REFERENCES

[1] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[2] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[3] R. S. Sutton. Implementation details of the td () procedure for the case of vector predictions and backpropagation. *GTE Lab., Waltham, MA, Tech. Rep. TN87-509.1*, 1989.

[4] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[5] G. Tesauro. Practical issues in temporal difference learning. In *Reinforcement Learning*, pages 33–53. Springer, 1992.

[6] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[7] M. Van Der Ree and M. Wiering. Reinforcement learning in the game of othello: learning against a fixed opponent and learning from self-play. In *2013 IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning (ADPRL)*, pages 108–115. IEEE, 2013.