Reinforcement 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(λ) 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(λ) algorithm. Our results show that?

Keywords

CS701; LaTeX;

1. INTRODUCTION

This paper adapts Gerald Tesauro's TD(λ) algorithm the game of backgammon and implements it for the game of 'Othello', or as it is more commonly known 'Reversi' [Cite]. The game is played on a $8 * 8$ board and works by having a '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 see [Figure (1)] for a visual representation. 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−the highlighted tiles in [Figure (1)] illustrate these types of move. Flanked pieces are captured and are replaced with pieces of your own color. The game goes on until one player either has no more valid moves or the board is full. 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.

We used a Neural Network with 1 input layer, 1 hidden layer and 1 output layer. The input of our network was a vectorized representation of the board state and the output of the network was 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 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, see [Figure (2)] for simplistic visual representation.

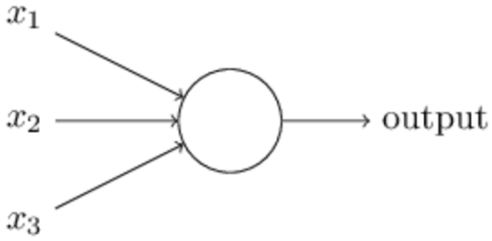


Figure 1: Initial Board State for Othello

Each connections has a corresponding weight, and the output of each neuron is equal to the sum of all the weights multiplied by their input. To normalize and smooth out these connections, we use the sigmoid of these sums, 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. It is assumed that the greatest value for all possible moves represent the optimal move for the network.

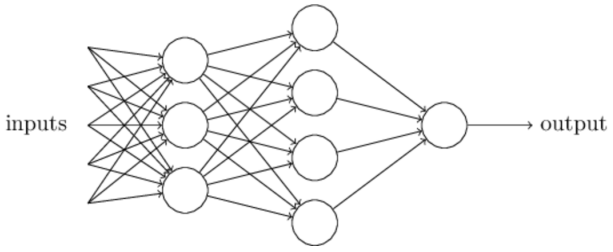


Figure 2: A simplistic representation of a Neural Network with 1 hidden layer and 1 output layer

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. If the output value for the current state is greater than the value for the previous state the network deems the previous move 'good'; on the other hand, if the value for the previous state is greater than the value for the current state, the network deems the previous move 'bad'. Finally, at the end of each game a pre defined reward is respectively given for a win, loss and tie. This reward is used

to calculate final error and is then back-propagated through the network. Further details on how this back-propagation works will be discussed in our Methods section.

Othello is a deterministic game, unlike Backgammon where dice rolls make it is stochastic. As such, to get our Network to explore new game strategies we had to add a Stochastic element to the TD(λ) algorithm. This element is called the exploration rate (ϵ), and determines how often our Network will explore moves that are not the expected to be optimal.

In this project, we have devised two types of Network with distinct implementations of the exploration rate. The first of these Network, 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 reinforcement learning in a deterministic environment.

Our second contribution is seeing how each of these networks perform with different Lambda (λ) values. The lambda value determines the rate of decay in error correction for previous time steps. A $\lambda = 0.0$ means that no feedback occurs beyond the current time step, and a $\lambda = 1.0$ means that feedback occurs without decay arbitrarily far back in time. 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 opponent. For this purpose, we created 3 additional opponent against which our network would 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 [Cite Othello], (3) an AI agent using the Alpha-Beta search. Unfortunately, Alpha-Beta proved to be extremely slow even at a depth of 2, so we chose to only use it as a means of testing our final networks. Thus, both NN-Fixed and NN-Decreasing were trained against opponents (1) and (2), as well as being trained against themselves, and each of these combinations was test with $\lambda = 1.0$ as well as $\lambda = 0.9$. 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 were trained for $125,000$ games, and we used their performance against a random agent for 500 games was used as a measure of strength −this choice will be explained in the Methods section.

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. TD-Gammon, developed in 1992, is a backgammon program that uses an artificial neural network trained by the TD(λ) 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(λ) backpropagation 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(λ). When a game is over, instead of calculating a new move, the board state is assigned a value of 0 or 1 for white winning the game or black winning the game, respectively.

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 terms of the setup of the network, it was noted that substantial improvements in performance were observed when the size of the network and amount of training experience increased. 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), 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

Here is a sample citation [**?**].

3. METHODS

3.1 Network Layers

To implement the TD(λ) algorithm we used 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 was used to evaluate the board state.

Since Neural Networks require vector inputs we converted our $8 * 8$ 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, see [Figure (3)] for a visual representation of a board state and its corresponding input vector.

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

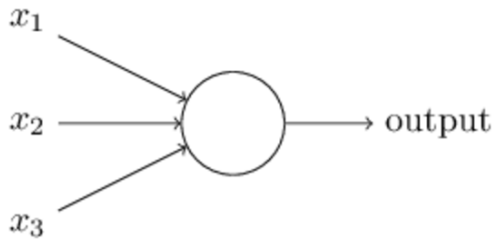
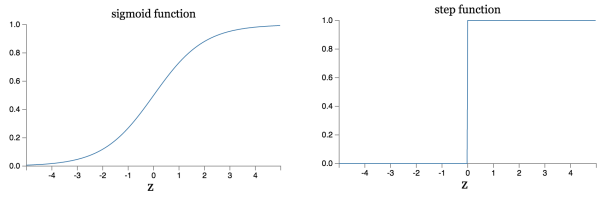


Figure 3: Visual representation of a neuron in a Neural Network

Perceptron's 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 you can use in a Neural Network. Perceptron's take inputs with corresponding weights, and output a value of 0 or 1. This work by setting a threshold, and output corresponding value for outputs below or above the determined threshold. [Figure (4)] and [Equation (1)] show the basic functioning of a Perceptron [CITE HANDWRITTEN].

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

The issue with perceptron's 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 Perceptron [Figure(4)], but the output of the neuron is passed into a sigmoid function [Equation ()], 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 (5)].



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

Figure 4: Different between Perceptron and Sigmoid output

3.3 TD(λ)

The TD(λ) 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]:

$$w_{ij}^{t+1} = w_{ij}^{t} + \alpha \sum_{k \in O} (P_k^{t+1} - P_k^t) e_{ij}^t \quad (2)$$

Where

w_{ij}^{t+1} is the weight at time t from node i to j

α 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:

$$\sum_{k \in O} (P_k^{t+1} - P_k^t) \quad (3)$$

In the case of Othello, there is only one output node, so the summation is not necessary. In this paper?s othello implementation, endgame states pass in a hardcoded of 0 for white winning, 0.5 for a tie, and 1 for black winning. 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, α, 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]:

$$e_{ijk}^{t} = \sum_{n=1}^{t} \lambda^{t-n} \frac{\partial P_k^n}{\partial w_{ij}^n} \tag{4}$$

Where

λ is the decay parameter for the eligibility matrix

The eligibility matrix is running sum of the previous credit assignments in the network [1]. 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 the 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]:

$$\frac{\partial P_k^n}{\partial w_{ij}^n} = \delta_{kj}^{t+1} y_i^{t+1} \tag{5}$$

$$\delta_{ki}^{i} = \frac{\partial P_k^t}{\partial s_i^t} = \begin{cases} y_i^t(1-y_i^t), & \text{if } k = i \\ 0 & \text{if } k \in O \text{ and } k \neq i \\ \sum_j \in FO_i \frac{\partial P_k^t}{\partial s_j^t} \frac{\partial s_j^t}{\partial y_i^t} \frac{\partial y_i^t}{\partial s_i^t} & \\ = \sum_j \in FO_i \delta_{kj}^t w_{ij}^t y_i^t (1-y_i^t) & \text{otherwise} \end{cases} \tag{6}$$

Where:

δ_{ki}^{i} is the error at node j

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 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. 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 λ 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(?), 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(λ) to adjust the weights

3.4 Player Types

Neural Network – 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.

Semi Random Neural Network – This player decides its moves based on the neural network only a preset percentage of the time and moves randomly 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 is selected. If the random number is below some threshold, then the player moves based on the neural network and moves randomly otherwise. This player is mainly used in training, and shows better training results than the Neural Network player.

Decreasingly Random Neural Network – This player decides its moves based on the neural network only an increasing percentage of the time and moves randomly otherwise. This player has similar benefits to the Semi Random Neural Network player, but reinforces more of its moves in later iterations of training. The threshold for the randomness is calculated by an exponential function:

$$T = (1 - a) * e^{10 * \frac{(i-n)}{i}} + a \quad (7)$$

Where:

t is the threshold

a is the initial threshold

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.

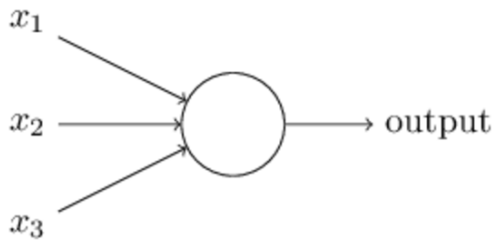


Figure 5: Graphical Representation of the exponentially decreasing learning rate

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.

- A description of the methods you are using (which itself could have sub-sections). If you have created a new application, you should describe which tools you have used to do so and the steps of your implementation. You should also briefly describe each of the tools you used (e.g. Ruby on Rails, MongoDB, D3, etc.). If you have proposed a new algorithm to solve a problem, you should provide the details of your algorithm along with pseudocode. If your algorithm is quite complex, you should describe its running time.
- Description of any data you used and explanation of why you chose this particular data (possibly organized into sub-sections).
- Link to a GitHub, BitBucket or other repository if relevant.

4. RESULTS

This section should include the following:

- Details and explanations of results obtained (which itself could have sub-sections). This is where you should provide tables, graphs, and/or figures that illustrate your results. If you have created a new application, include screenshots of it in action. You should also provide a link to your application if it is web-based.

You should entitle these sections and sub-sections with names that describe the key points (for example, instead of "methods we use", the heading "Statistical-Based Learning" would be more informative). The Methods and Results sections should together be approximately 3–4 pages in length.

5. DISCUSSION

This section should begin with a brief summary of your results. Next, provide a more detailed synopsis of your results: What new knowledge do they offer? What lessons did you learn? What is the main take-away message of your work? Finally, you should provide a brief critique of your own work: Point out the specific attributes that you feel are extremely positive and note any weaknesses or limitations. Discuss how your project could be improved. You can also discuss possible extensions of your work. This section should be approximately 1 page in length.

6. ACKNOWLEDGMENTS

This section is optional; it is a location for you to acknowledge grants, assistance, etc. For example, this report template was adapted from Prof. Christman.