AI Learning Polar Tic-Tac-Toe: Simple Hurestic, Naive Bayes, Temporal Difference Neural Network, Minimax

Jacob Barthelmeh, Kaleb Himes, Angelica Davis

December 10, 2014

Abstract

This is an experiment to compare the ability of four different algorithms to win a polar tic tac toe game. The four algorithms used are; a simple heuristic, Naive Bayes Network, Temporal Difference Neural Network (TDNN) and the Minimax Algorithm. Algorithms are compared in their ability to learn and their ability to win when competing against each other.

1 Introduction

Use of AI in industry and commercial settings is on the rise. Such as with the use of TDNN's in robots [4]. With the broad applications for AI algorithms knowing the best algorithm to use for a problem is important. Since there is "no free lunch" (not an algorithm that is best at all) than the best fit needs to be determined either by mathmatical proof or imperical evedince , or by both. This paper goes through an experiment in finding the best AI algorithm for being used to solve a problem. By testing the performance of each algorithm it is shown that you can find evedience of which will be best suited.

In previous work Minimax algorithms have been used to solve chess moves and are well suited for board games in general [1]. We also hypothisise that the Minimax algorithm will perform best when implemented to play polar tic-tac-toe. This is because it will be looking ahead a couple moves and making

decisions based off of that and stored outcomes from previous games where as the TDNN and Naive Bayes algorithms are trying to learn how to make inferences given the current board state. Making inferences with this board may take more precalculations than what will be done in the experiments.

Previous work with the TDNN has been done on the backgammon board game [5]. It has shown to be able to play at a grand master level in backgammon. Here it will be implemented to make decisions on moves with the polar tic-tac-toe game. To simulate the process of learning the TDNN implementations gradient descent to update the weight values [8]. With having the weights updated it can adjust the predicted outcomes of board states to match what it is seeing in actual tests.

This paper is divided into an explenation of the problem followed by an explenation of each algorithm implemented. After defining the problem and showing the algorithms used there is a section on the experimental methods and on the that came from conducting the experiment. The end of the paper has sections to discuss the results of the experiment and at the very end a conclusion.

2 Problem Statement

Algorithms will be used to play Polar Tic-Tac-Toe. The game is played by alternating turns of who gets to pick a move on the board. A spot being controlled by a player is represented visually by a O or an X indicating which player has chosen the spot. If at any time a player has 4 spots in a row, either diagonally or in a straight line, they win the game. The game consists of four rings, with each ring containing 12 evenly spaced locations for a possible move. In this experiment it is only a legal move if the space being chosen has one edge touching a space that has previously been chosen by either player. This though is relaxed on the very first move since no previous moves have been made.

The problem to be solved by the algorithms being experimented with is to attempt learning information about the state space of the game. Each of the algorithms is to make perdictions on which move to make and ultimatly each will try to not lose.

2.1 Traditional Minimax Search

The Minimax algorithm used checks at each ply for the maximize value when looking for player ones potential moves. When playing as player two it looks to minimize the possible utility value. An adjustibale parameter set for the algorithm is the depth at which it searchs. Values are assigned by finding the max or min of the next ply. A terminating node with a win condition for player one is assigned the value of 5 and a loss condition is assigned -5. For ties the value of 0 is assigned to the node. All nodes in the tree are initialized with the value returned from the simple heuristic if player one, if player two than the node value is initialized as 0.

The process can be shown in the following pseudocode. [7]

function Minimax(state) return action
return arg max a $\in Actions(s)Min - Values(Result(state, a))$

function Max-Value(state) returns a utility value
if Terminal node than return utility of state
else set v to negative infinity
for each a in Actions(state) do
v set as the max between v and Min-Value(Result(s,a))
return v

function Min-Value(state) returns a utility value
if Terminal node than return utility of state
else set v to positive infinity
for each a in Actions(state) do
v set as the min between v and Max-Value(Result(s,a))
return v

The biggest difference between our implementation and the cononical version is that we have a cutoff point specified by depth instead of going all the way to terminal nodes. In the Minimax experimented with it also retains the state, action, and utility values in a tree for future use.

2.2 Minimax with Alpha-Beta Pruning

The ability to use Alpha-Beta Pruning is something that can be adjusted at runtime. It uses almost the same algorithm as the regualer Minimax but has the added check of alpha and beta values. Alpha is set to negative infinity and beta is set to positive infinity when the alpha-beta algorithm is started. This is to insure that an alpha and beta value will be found and that node values will not be over looked while the algorithm is operating. When looking for a minimum value for the next move if the value found is less than alpha the node is returned and the rest of the nodes are not explored. If the value is not less than or equal to alpha than beta is set to the minumum between the current beta value and the current node being explored. When looking for a maximum value for the next move if the value found for the current node being explored is less than or equal to the current beta value than the current node is returned and the rest of the nodes in that ply are not explored. If the value is less than the current beta value than the alpha value is updated to be the maximum value of either the current alpha value or the current node value.

In our implementation the Alpha-Beta Pruning can be called on an exisiting stored Minimax tree. This works by starting at the head of the tree and recursivily finding the max value or min value from possible moves while keeping track of an alpha value and beta value. It then marks nodes that the Alpha-Beta Pruning algorithm decided not to visit so that they will not be explored in the future either.

2.3 Heuristic Functions

In the simple heuristic implementation, the heuristic value is the number of similar player pieces in a line adjacent to each other that include the play to be evaluated. This value ranges from one to three since a value of four signifies a win. This can be exploited to make a move by determining from all possible moves which has the highest heuristic value. This heuristic is admissible but not the closest to the actual heuristic it could be since there is a condition where there could be two in a row with a space than a another spot where the player had moved. In this case the actual heuristic value should be three but will be evaluated to two.

$$V^{\pi}(s) = \max_{ain\mathcal{A}} \left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a)V(s') \right].$$

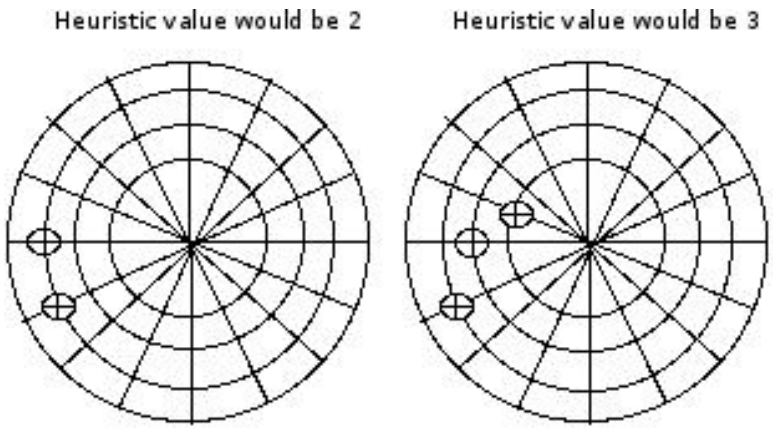

Figure 1: Simple Heuristic Examples

2.4 Win Checking with Resolution

The win checker used in the experiment determined a win, lose, or tie by using first order logic, resolution and unification.

player = p	north east neighbor = urn (for upper right neighbor)
neighbor = n	north west neighbor = uln (for upper left neighbor)
left neighbor = ln	south east neighbor = drn (for down right neighbor)
right neighbor = rn	south west neighbor = dln (for down left neighbor)
upper neighbor = un	
down neighbor = dn	

Table 1: Notation used in winchecker

Let p = X $\bigoplus O$

Let connected = 4 be a winning state If n is moved on by p, perform a local search with that node as the origin.

$\exists n \ni p \in O, X \bigwedge p \in n$

An example of one of the sections of logic would be the check horizontal portion.

check left:

n $\in \{a1, a2, a3, a4\} \Rightarrow ln \in \{l1, l2, l3, l4\}$

$\neg n \in \{a1, a2, a3, a4\} \bigwedge (n-1)0 \bigwedge p \in (n-1) \Rightarrow connected$

check right:

n $\in \{l1, l2, l3, l4\} \Rightarrow rn \in \{a1, a2, a3, a4\}$

$\neg n \in \{l1, l2, l3, l4\} \bigwedge (n+1)47 \bigwedge p \in (n+1) \Rightarrow connected$

While for each of the logic checks a connected value of 4 is not found it goes through the other 3 first order rules. If at the end no winner has been found, connected = 0, It then becomes the other players turn.

2.5 Game Evaluation with Classifiers

The canonical version of Naive Bayes was chosen to be the classifier in the project. Types for the board state when being run through the algorithm is broken up into average heuristic value of all current players pieces and number of plays in each of the 4 rings that a player has. Average heuristic is divided into being greater than or equal to two, while number of plays in a ring is divided into being higher than 5 or not. The heuristic value here is determined using the simple heuristic implementation described earlier.

There are three possible classes for the Naive Bayes network. It can classify the board state as a win for player one, a win for player two, or a tie. Along with classifying the board, the algorithm will also be able to give the probability with which the board state falls into that particular class.

In addition to being able to classify a board state the Naive Bayes algorithm will be able to make a move. This will be done by considering all possible moves and their probabilities for a given outcome. The board state with the probability that is highest for a desired outcome will be the move selected by the algorithm.

2.6 Temproal Diffrence Neural Network

When training the TDNN the first 20 games are trained against a random player. This is in hopes of reducing the risk of falling into an early local minimum. The remainder of n games chosen to be trained by a user is

than done by pitting the current TDNN against itself. After each game is played the final result is evaluated and each state leading up to the result is compared to the prediction given by the network. Gradient descent is than done using the error of the network and allowing for adjustments to be made to the networks weights. The following equation is used to adjust weights in the network.

$$\triangle w_t = \alpha(V_{t+1} - V_t)\sum_{k=1}^{t} {}^{t-k}\nabla_w V_k$$

Where alpha is the learning rate and V is the perdicted output.

For each game the network is trained for both players, having the weights adjusted from both players perspectives after each game has ended.

The topology of the network is 48 inputs, one hidden layer with 40 nodes, and 3 output nodes. 48 input nodes to represent each possible state for a move and 3 output nodes; one for player 1 win, one for player 2 win, and one for a tie perdicted.

3 Experimental Methods

In conducting experiments with the implemented algorithms we had them compete against each other 1000 games each. All algorithms competed against the other 3 and also against a random player. To get the performance result of using alpha beta pruning on minimax, it will also compete against the other 3 algorithms and against a random player while using alpha beta pruning.

4 Results

	vs	win	tie	loss
TDNN	Minimax	0	0	1000
TDNN	Simple Heuristic	0	0	1000
TDNN	Naive Bayes	507	0	493
TDNN	Random	948	0	52

Table 2: Tempral Diffrence Neural Network performance with 40 hidden nodes

This is a table 2 that shows win, loss, and ties against the Minimax, Simple Heuristic and Naive Bayes algorithms along with the results of it playing a random player.

	vs	win	tie	loss
Minimax	TDNN	1000	0	0
Minimax	Simple Heuristic	476	0	524
Minimax	Navie Bayes	1000	0	0
Minimax	Random	979	1	20
Minimax with Pruning	TDNN	1000	0	0
Minimax with Pruning	Simple Heuristic	474	0	526
Minimax with Pruning	Naive Bayes	1000	0	0
Minimax with Pruning	Random	982	1	17

Table 3: Minimax Performance with depth of 2

	vs	win	tie	loss
Naive Bayes	TDNN	487	0	513
Naive Bayes	Simple Heuristic	0	0	1000
Naive Bayes	Minimax	0	0	1000
Naive Bayes	Random	936	1	64

Table 4: Naive Bayes Performance

	vs	win	tie	loss
Simple Heuristic	TDNN	1000	0	03
Simple Heuristic	Minimax	494	0	506
Simple Heuristic	Naive Bayes	1000	0	0
Simple Heuristic	Random	991	0	9

Table 5: Simple Heuristic Performance

	win percentage
TDNN	36.4
Simple Heuristic	87.1
Naive Bayes	35.6
Minimax	86.4
Minimax with Pruning	86.4
Random	3.6

Table 6: Win percentage of algorithms in experiment

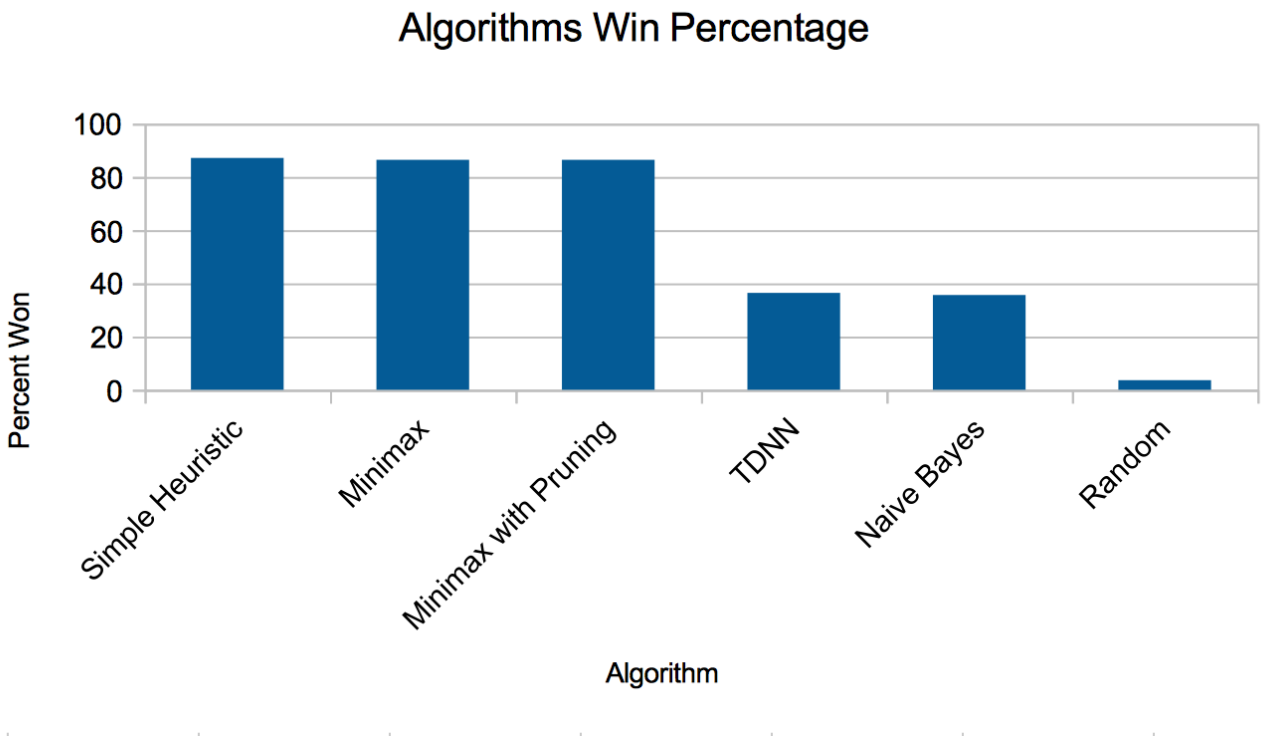

Figure 2: Win Percentage

5 Discussion

In all of the tests the Simple Heuristic and the Minimax algorithms outperformed the TDNN and the Naive Bayes network. This pattern was consistent through out all of the tests that were ran. All algorithms were able to do better than random but that does not provide us with much information about the performance of the algorithm.

6 Conclusions

While conducting the experiment it was shown that the simple heuristic and minimax algorthims performed best out of all of the implementations. It was also apparant while running the experiments that the Minimax tree gets very large relatively quickly when stored. The results lead to the suspicion that the topology of the TDNN is a large factor in it's ability to perform well.

In the experiment the TDNN did not perform well. We think this could be due to the amount of input nodes in comparision to the number of hidden nodes used. To test this we briefly ran a network that had 160 hidden nodes but still did not see a performance gain. Future exploration of the reason for it's lack of performance could be done in investigating the effect of having 48 input values. It may increase performance to do calculations to reduce the number of inputs since the state search space for backgammon in which it excelled was only 20.

References

[1] Daniel Higginbotham. An exhaustive explanation of minimax, a staple ai algorithm. *Flying Machine Studios*, 2012.

[2] T. Hill and P. Lewicki. Electronic statistics texbook. *StatSoft, Inc.*, 2013.

[3] Eamonn Keogh. Naive bayes classifier.

[4] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University, 1993.

[5] James McClelland. *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises*. Stanford, second edition, 2014.

[6] Charles Rich. Technical game development ii.

[7] Stuart Russell and Peter Norvig. *Artificial Intelligence : A modern Approach*. Pearson Education, Inc., third edition, 2010.

[8] David Silver. Gradient temporal difference networks. *JMLR: Workshop and Conference Proceedings*, 24:117–129, 2012.