The Implementation of Machine Learning in the Game of Checkers

William Melicher
Computer Systems Lab
Thomas Jefferson

June 9, 2009

Abstract

Most games have a set algorithm that does not change. This means that these programs cannot adapt to a situation or learn from mistakes that it makes. However if a machine could learn, then it could adapt to new situations and would have a nearly boundless skill level. Machine Learning programs can be beaten once, but against an opponent that does not change, it eventually will be able to beat it. The project that I am writing will learn how to play the game of checkers as it plays, by modifying itself after ever game played. It will review its play and if it played well it will play that way more often, if it did not it will avoid that way of playing.

1 Introduction

The game of checkers has been weakly solved by a computer program. The program generated every possible board combination and simply uses a brute force method of searching through these at every move. However most computers cannot do this and no personal computers can. A Machine learning program would be able to play at a high level of play without requiring huge data bases or large amounts of processing power for each move. The game of checkers has lower complexity level compared to other games like othello, chess, go, and others, but the use of machine learning in this program can

also be extended to those games and other situations that require a learning program.

The basic principle of machine Learning is that the program can use past examples of a situation to predict how an action will end up in the future. So a machine learning program would be able to continually adapt to the circumstances that it is in. This type of program would also be able to adjust its play when it is playing an opponent that does not change. The program needs to play games to learn how to play, and more games would give it more experience to learn how to play. The program needs to play against itself to get a large enough experience base to be able to play well.

This research could be applied to any situation that requires quickly solving many similar problems in succession. Any problem where previous data can be used to predict a future situation would benefit from a machine learning algorithm. The program would take the data from previous problems and the outcome from those problems and then use it to predict the outcome of a particular situation.

2 Background

Most programs that play games today use a search through a tree of moves and boards. Each different possible move and the board that would result from that move is evaluated and when all of the boards have been searched an evaluated; the program chooses the move that results in the best outcome quickest. This is called the minimax algorithm. Each player wants to act in their own interest, to either maximize or minimize an evaluation function called a heuristic. The heuristic determines which boards are good and which ones are bad, without being able to perfectly predict the results of the game. The minimax algorithm looks at all of the possible boards that can be reached from one state and then looks at all the boards that can be reached from another state, and so on and so on. At some depth the algorithm stops and then applies the evaluation function to determine the fitness of a board. At each level, the player whose turn it is will make the move that results in the best outcome for them. This algorithm is limited by two things, which are the depth of the decision tree that it searches, and the quality of the heuristic function. However this can become very costly in terms of time very quickly. There is not enough time to search through the entire decision tree for all but the simplest games. The heuristic for each game is different and a common

form of a heuristic is

$$H(s) = c_0 * F_0(s) + c_1 * F_1(s) + ... + c_n * F_n(s) \quad (1)$$

Where H(s) is the value of the board, Fi is a feature of the particular board for example a feature of a checkers game would be how many checkers you have compared to your opponent, and ci is the weight given to that particular feature (2 Olsen). The play of the AI increases in skill as you can more perfectly predict the future states of the board. This means having an AI that accurately describes the position of the board and the minimax algorithm can go to a large depth. Creating a heuristic for a game generally requires an expert and specially tailoring the heuristic to its application.

This is why a learning program would be useful. The program could learn its own evaluation function that would be more accurate and with less work than having a static function. One method of creating a learning program is to have the program learn by rote. One program designed by De Jong and Schultz (1) used an experience based that the program would reference when it made moves. The experience base stored all of the boards encountered and the moves that had been tried off of these boards. While this is similar to creating all of the possible board combinations and simply searching them exhaustively, this does not require as much memory and is used in conjunction with a static evaluation function. A different method of creating a learning program is described by Olsen (2), generalization learning. The program modifies the heuristic function each time it plays based on the outcome of the play. The program starts with a heuristic function that has all of the features weighted equally. After the program plays a game, it will change the values of the weights of the heuristic function based on the outcome of the game. This type of learning when implemented in a program, generally played very well during the middle game play, but less well in beginning and end game play, while the rote learning process played very well in end game play.

3 Developement

I am using a combination of the learning by rote algorithms and the generalization algorithms. I create a file and store all of board combinations that have been seen and the moves taken from that particular board. I then also use a changing heuristic to evaluate each board. The heuristic also learns

based on the outcome of the game. Temporal difference learning adjusts the evaluation function during play based on the difference between the evaluation function at one point and the evaluation function at a different time. This brings the evaluation function into a state of equilibrium toward the ideal evaluation function. Every time another move is made, the heuristic is changed based on the previous heuristic. The equation for this is

$$U(s) < - - U(s) + a(R(s) + U(s') - U(s)) \tag{2}$$

S is the current state of a system, U is the evaluation function of state s, is the learning coefficient and decreases as the number of times state s has been encountered increases and R is the reward that you get for state s.

Temporal difference learning improves the evaluation function by exploiting the differences in the evaluation function at each time. While there is no proof that says that a heuristic function that is evaluated closer to the end state of a board is better, it generally is. This makes sense because you can more easily tell whether a player will win or lose a game when the game is closer to the end. What the temporal difference algorithm does is change the heuristic function in the direction of the current board. So basically the program is making a prediction of how good the board will be for a player in the future knowing that this prediction will have some error in it. Then in the future it examines the previous prediction and identifies what was wrong with it and then changes the function. However what if a rare occurrence happened that was not predicted. This would drastically change the heuristic function, but this would change it in a bad way because the occurrence is not likely to happen and the heuristic will give a less than optimal prediction when this occurrence will not happen. So to protect the heuristic from rare occurrences you have to decrease the amount you change the function when you already have a lot of data. This makes sense because if you have a heuristic function that has been improving for one hundred games then the function is likely pretty good, so you don't want to change it a lot. But if you have a function that has only been improving for five games, then it is probably worse and you want to change it more.

The alpha value in the temporal difference equation is the thing that decreases the returns of the temporal difference learning equation. It does this by multiplying the amount that you change the weights of the heuristic by a progressively smaller amount as the program has seen more boards. The alpha value must be in the form of the equation a=1/n, where n is the

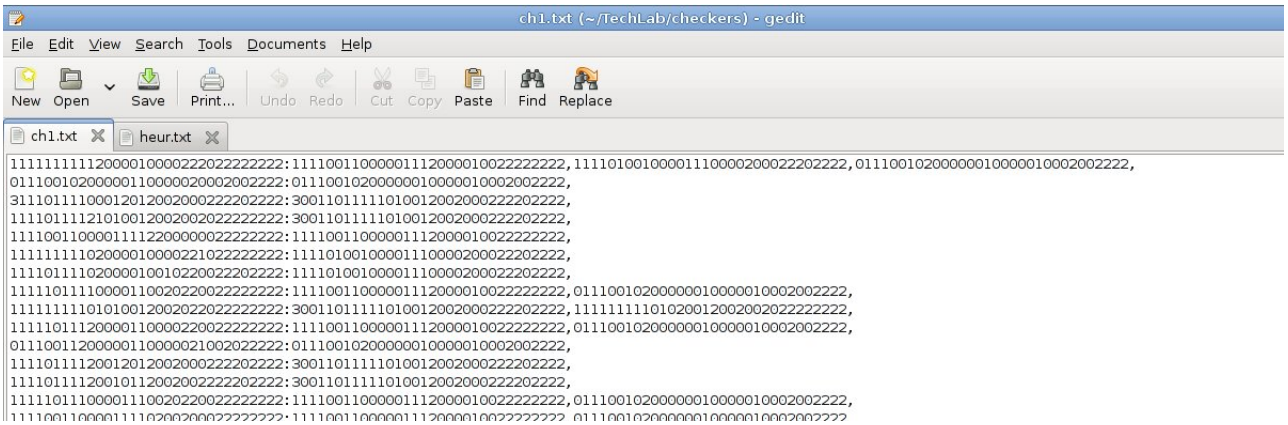
number of games played. This way the alpha function will be 0 when the program has played an infinite number of games. The alpha value function is important so that the program does not learn too fast and think that it knows things that are wrong but it also cannot learn too slowly so that the program is not taken off equilibrium too often by a rare occurrence. The alpha value function that I used in my checkers program was a=50/(49+n) where n is the number of boards played.

What the temporal difference learning algorithm changes is the weights of each of the terms in the function. The individual terms in the heuristic are specific aspects of the board that can be measured, for example one term might be the number of checkers that you have and the number of checkers that an opponent has, or how many kings you have compared to the opponent. Basically anything that affects the outcome of the game is quantified in a term. What the temporal difference learning algorithm does is improve the weights of these terms. It improves how much one specific term is worth. For example in checkers the thing that most decides the outcome of the game is the piece count of each side, so it would be worth more than how many kings each player has.

The learning by rote system is used together with the temporal difference learning algorithm. The program that I wrote keeps track of all the boards that it has seen in the time that it has been playing. This information is used in the alpha function of the temporal difference learner. The alpha function a=50/(49+n) requires that you have the information of how many boards are played before being able to change the weights of the heuristic effectively.

My program stores each board that was visited and the boards that can be reached from that board. It does this by converting each board state into a number. The number is a 32 digit number of base 5. There are 32 digits for each space on the board that a piece can be in, and base 5 for each piece that can be there. Then the program maintains a hashmap of the boards where a board is the key and each board that can be reached from that board is the value. At the end of play, the program prints the data out to a file for the next iteration of the program.

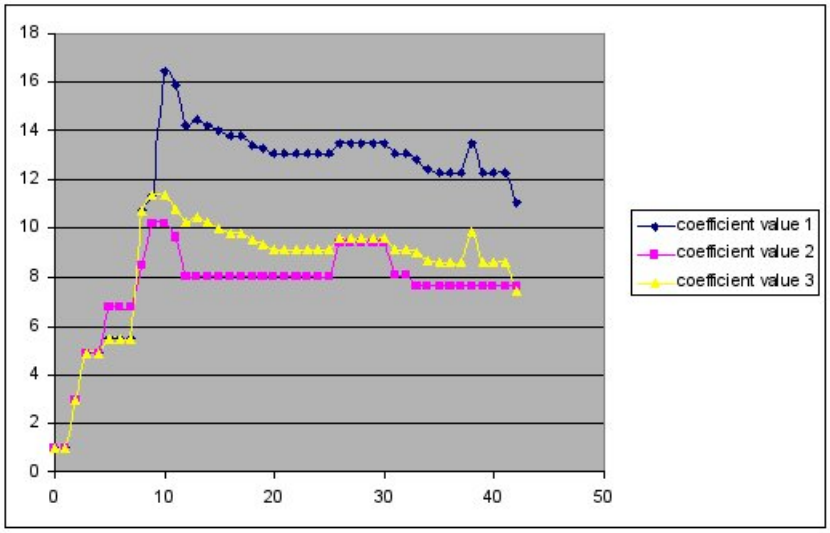
This is a copy of the output of the learning by rote file. On the left is a board represented by the string of numbers. On the right of the colon is the list of boards that can be reached from the original board.



4 Results

This graph shows the value of the terms of the heuristic over the course of a game. Weights number 1 is attached to the value of the piece differential of the board. Weight number 2 is attached to the value of the number of kings that are on the board. Weight number 3 is attached to the number of pieces that are on the side of the board. Each of the coefficients was initiated with the value of 1.0, and the coefficients were saved in a file containing the three doubles. In the beginning the AI realized that the weights were under valued and increased it at an increasing rate. At some point the program determined that the weight was overshot in the course of the game, so the program then begins to lower the heuristic by progressively less amounts. Each of the weights during the course of the game reached an optimal value.

The learning by rote method that I used stored the boards that were seen in the course of the learning. The database of boards that it maintained was used in the temporal difference learning so that it may decrease the changes of in the weights of the heuristics with the number of times that the program has seen a board. This method of storing the data did succeed and over time the data in the database increased. However the data in the board never became too much for the database to become unreasonable. This was a concern because the number of boards that were possible is much more than could be stored in a file, but the number of boards that were routinelly encountered by play was much less than the amount of boards that are possible.



5 Conclusion

Temporal difference learning is a good way to achieve equilibrium value for the weights of the heuristic. All of the values of the heuristics achieved an equilibirum value, in that there was little change after about twenty iterations of the learning program. Also, the temporal difference learner required relativelly few runs to have significant learning. Within twenty turns the amount oflearning was enough that it played to a reasonable level. The program pretty soon was noticably harder to beat from the perspective of a human player.

However sometimes the temporal difference learning requires the programmer's intervention on getting the program away from a false minimum in the function. Occassionally the program would achieve negative values for the weights. This means that the program would work away from the win and would try to lose. Because the program relies on the difference in the heuristic in two different points in time, the program will continue to get more and more negative heuristics. The reason for that is that as the program goes on it expects that the other player will evaluate a board in the same way, and that the opposing player will have the same goal as it. However when the program got negative values the programs goal becomes

to lose. This means that the heuristic values will continue to be more and more negative because the program is continually losing.

Temporal difference learning also does not require large amounts of memory. At the end of the learning period, my program only stored 2.5 kilobytes of data for each board and .55 kilobytes for storing the values of the heuristic. Learning did not require that the program have knowledge of game strategies, however it does require that the game move toward a win and be able to recognize a win. My program overall was a success and it became increasingly good at the game of checkers.

References

[1] De Jong, Kenneth A. and Alan Schultz, Using Experience-Based Learning in Game Playing., 1998.

[2] Olsen, Daniel, Learning to Play Games From Experience: An Application of Articial Neural Networks and Temporal DifferenceLearning. 1993.

[3] Norvig, Peter and Stuart Russel. Artificial Intelligence, a Modern Approach. New Jersey Pearson Education, 2003.

[4] Rich, Elaine and Kevin Knight. Artificial Intelligence. New York McGraw1991.