Using Neural Networks in the Static Evaluation Function of a Computer Chess Program

A master's thesis by
Erik Robertsson
January 2002

ABSTRACT

Neural networks as evaluation functions have been used with success in game playing search algorithms in games such as backgammon. In the game of chess, this approach has not been as successful. This may be due to the fact that classifying a chess position is such a complex task and would require a too complex classification function. Maybe it would be more successful to use neural networks to classify a small part of a chess position, a strategic concept such as king safety or pawn structure, and incorporate this in an ordinary chess evaluation function? This thesis is trying to investigate how useful this approach is.

The method used is to try to apply the approach on a simple endgame position, where the results are easily determined, and evaluate the effectiveness of the approach.

PREFACE

I would like to thank my supervisor

Jan Eric Larsson
Associate Professor,
Department of Information Technology,
Lund University, Sweden

I would also like to take the opportunity to thank my friend Benny Antonsson for inspiration and advice concerning the search engine.

1 INTRODUCTION 5

1.1 MACHINES PLAYING CHESS 5
1.2 STRATEGY VERSUS TACTICS 5
1.3 GOALS 6

2 GAME TREE SEARCHING 7

2.1 METHODS AND ALGORITHMS 7
2.1.1 Alphabeta Cut Offs 8
2.1.2 The Transposition Table 9
2.1.3 Further Improvements 10
2.2 THE EVALUATION FUNCTION 10

3 NEURAL NETWORKS 12

3.1 THE PERCEPTRON 12
3.2 BACK PROPAGATION 13
3.3 GENERALIZATION 14

4 METHODS 16

4.1 A SIMPLE ENDGAME 16
4.2 NEURAL NETWORKS IN THE EVALUATION FUNCTION 17
4.3 THE LAYOUT AND THE TRAINING OF THE NETWORKS 17

5 TESTS AND RESULTS 19

5.1 A FIRST ATTEMPT 19
5.2 SINGLE NODE OUTPUT 19
5.3 ROTATION 20
5.4 ENHANCED TRAINING 21
5.5 MATE SEQUENCES 21
5.6 RESULTS 21

6 CONCLUSIONS 22

7 FURTHER IMPROVEMENTS 23

A CHESS ENGINE 24

B REFERENCES 25

1 INTRODUCTION

The game of chess has its origin in India nearly one and a half millennium ago. It has since then spread all over the world, and has with a few exceptions looked the same ever since. It is safe to say that chess is an extremely complicated game. Yet the rules of the game [1] are fairly simple and can be taught to a person in the time span of, say, an hour. Maybe it is the combination of these two facts that has made the game so popular. Nowadays, hundreds of thousands of people play chess, and several shelf-metres have been written about chess, treating the different aspect of the game: the opening, the middle game and the endgame.

1.1 Machines Playing Chess

Mastering the game of chess has always been held in high regard and is said to demand intelligence, creativity, imagination, logic and fighting sprit. Of these, logic is the most apparent aspect. The "world" of the game is well defined, as are the rules of how to make a move, and when the game is over. Since we have machines capable of logic operations, one obvious question is: If we make a machine that knows how to play chess, can we say that it has the qualities mentioned above? That is, can we say that such a machine is, in some sense, intelligent?

The question, though out of the scope of this thesis, is relevant, and perhaps the reason to why so many have been interested in creating chess playing machines.

One of the first attempts to make a machine play chess, was made in 1890 by the Spanish scientist Torres y Quevedo. It was a mechanical machine that had the ability to conduct a mate with rook and king versus a lone king. British crypto analysts D. Mitchie and Alan Turing took other early steps in the 1950s. They had worked together in Bletchley Park, were they had come in contact with one of the first computers ever created, the Colossus. The Colossus was created for one purpose: to break the German enigma codes, but Mitchie and Turing saw other possibilities for this new invention. In 1951, Turing created a "paper machine," i.e., a set of rules on paper, called Turochamp, which played, and lost, a game of chess against a beginner [2]. Today, the table has turned. We have seen the world chess champion Gary Kasparov being defeated by a mainframe computer called Deep Blue, and for a small sum of money, one can purchase a program for a home computer, that can give even professional chess players a rough match.

1.2 Strategy versus Tactics

When creating a computer program that plays chess, a conflict you inevitably have to face is that of strategic understanding versus tactical understanding. The faster a program you write, the more you can investigate about the possible outcome of the move you choose to make (tactics). In contrast, the more strategic you want your choice of move to be, the slower your program becomes. This is a serious problem, because a chess game is most often played in a limited amount of time.

Most of the top chess programs of today, are tactically better than most of the human chess players. As the evolution of faster computers continues at a fast rate, this fact will only grow stronger. Perhaps it is time for chess programmers to move the focus to the strategic aspect of the game [7][8].

1.3 Goals

The sad truth is that defining abstract strategic concepts in concrete source code is a hard thing to do. You have to weigh different strategic concepts with each other and with material values[1] in order to get a good estimate of the position. The reality of this is that you often have to spend long and hard hours of trimming and tuning your program.

The above has led me to wondering if perhaps using neural networks is a good and effective way of defining strategic concepts. A neural network is a well-known method in artificial intelligence that has the ability of learning a classification by example. Neural networks can also generalize classification rules from a classification set, which gives the ability to classify new instances that have some resemblance to the set of training examples.

My idea is to use different, relatively small neural networks, to evaluate different strategic aspects of a position.

The goal of this thesis is to answer the following questions:

- Are neural networks capable of learning strategic chess concepts?
- How much work does it take to make neural networks learn strategic chess concepts?
- Are neural networks effective in a chess-playing program, given the critical aspect of speed?

A description of algorithms and methods used in game play is given in chapter 2. Chapter 3 describes the function of perceptrons and neural networks. Chapter 4 will discuss the methods that I have used to be able to reach the goals stated above. In chapter 5 test and test results will be presented. In chapter 6 the test results are discussed and evaluated and in chapter 7 I will speculate about further improvements.

[1] The material value is defined as the sum of the values of the pieces on the board. See 2.2 Static evaluation.

2 GAME TREE SEARCHING

In this chapter, different methods and algorithms for planning ahead in game play will be discussed. The natural method for this kind of planning is to investigate every possible outcome of the available moves, and thereafter choose the most profitable of these. To illustrate this, you can create a tree, called a game tree, where each node corresponds to a position, and the arcs from a node corresponds to moves leading to new positions. A leaf in a tree is defined as a node with no outgoing arcs.

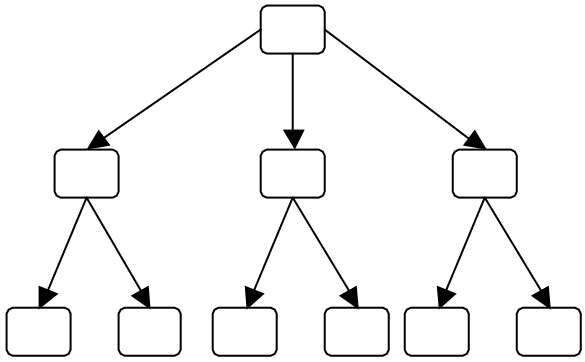

Fig. 2.1 An example of a game tree. The boxes are nodes representing positions and the arrows are arcs representing moves.

We define W as the width of the tree, that is, the average number of moves from any given position, and D as the depth of the tree. The average number of positions in any given tree is then W^D. The approach of investigating every outcome is only possible if the size of the game tree is reasonably small. In the case of chess, and many other games, this is not so. Instead you conduct a search to a limited depth, and apply an evaluation function to the leaves of the tree, giving an *estimate* of how good each position is.

2.1 Methods and Algorithms

When finding the best move to make in a game such as chess, you have to take into account that you are playing against an opponent. You do this by assuming that your opponent always makes the best move he can do.

We design the evaluation function so that it gives a high value when the position is profitable for the side you are investigating and a low value otherwise. Since you want to maximize your profit and your opponent wants minimize your profit, and since your opponent is in the move on every other level, this means that you alternate between selecting the move with the maximum profit and the move with the minimum profit on every other level. This is called the minimax algorithm.

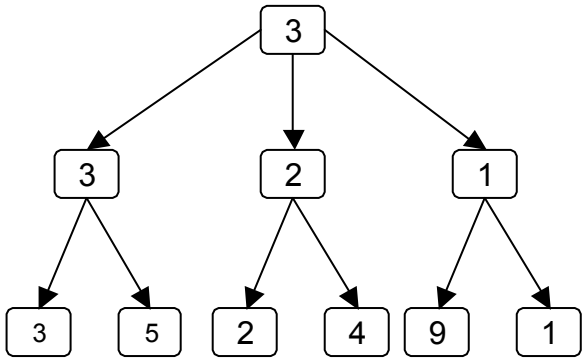

Fig. 2.2 The minimax algorithm. At depth 2 we are looking for a maximum value and at depth 3 for a minimum value.

The deeper the search and the better evaluation function we have, the better output will we get from the minimax algorithm.

Mate and stalemate is taken care of by the tree search when we find a node where there are no legal moves. If the side that cannot move is in check we have found a mate, otherwise the position is stalemate.

The average number of moves in a chess position is about 35. As an example of how big a game tree in chess can get, if we wanted to do a full search of a position to the depth of 10 plies[2], we would have to run the evaluation function on $35^{10} = 2{,}75 * 10^{15}$ positions. Generally, the evaluation function is relatively time consuming, so in order to be able to conduct deeper searches, we need a method of reducing the number of positions in a game tree.

2.1.1 Alphabeta Cut Offs

Alphabeta is an algorithm, which will give exactly the same results as a minimax search, without having to evaluate as many leaves. This is done by keeping track of the upper and lower bounds of the search, the alphabeta window, and stop the search when values falls outside this window. For instance, when we have searched the first (leftmost) sub tree of figure 2.2, we know that we will not do worse than 3. The lower bound is then 3, since we are subsequently only interested in moves that result in positions with better values than 3. When we go on to the second sub tree, we find that the first leaf gives the opponent the opportunity to make a move resulting in a position of value 2. We can now draw the conclusion that it is no use to continue the search in this sub tree, since it is already worse than the first sub tree. Instead we go on with the search in the third sub tree.

[2] A ply is a game play term defined as a half move, were a move is defined as first one side makes a move, and then the other side makes a reply.

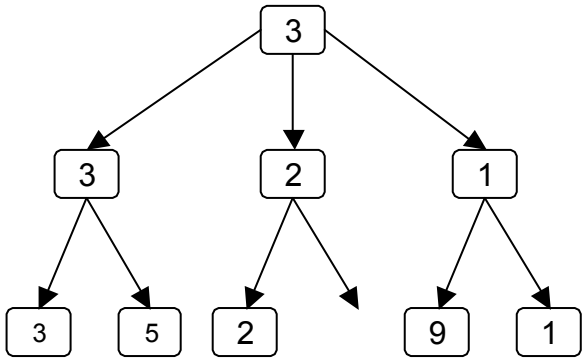

Fig. 2.3 The alphabeta algorithm. When searching the middle sub tree we find that the first leaf results in a position with the value 2. Since this is already worse than the leftmost sub tree, it is no idea to continue the search in this sub tree.

This is what is called an alphabeta cut off. It is clear that the tree in figure 2.3 will give the same result as the tree in figure 2.2, though it has executed one less call to the evaluation function. It can be shown that the alphabeta algorithm can search a game tree twice as deep as the minimax algorithm in the same amount of time [3].

2.1.2 The Transposition Table

In the game of chess, especially in the endgame, so-called transpositions occur quite frequently. A transposition is when two different sequences of moves from a given position result in the same position.

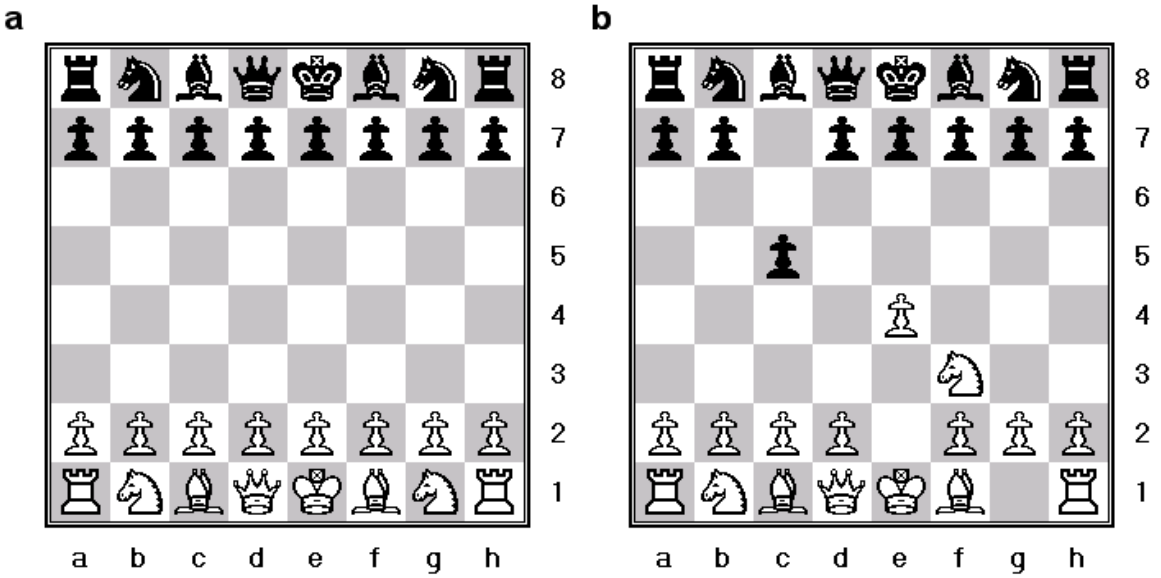

Fig. 2.4 Transposition. The two move sequences 1. e4 c5 2. Nf3 and 1. Nf3 c5 2. e4 will both transpose position a into position b.

For example consider in figure 2.4. Starting with position 2.4a, the two move sequences 1. e4 c5 2. Nf3 and 1. Nf3 c5 2. e4 will both result in position 2.4b. This means that our game tree will have a lot of nodes that contain the same position. This fact can be used in to reduce the number of positions in the game tree that we have to search and evaluate.

A position that has been searched or evaluated can be stored in a hash table, along with it is estimated value and what search depth this value is based on. Next time before we evaluate or

search a position, we first check if we can find the position in the hash table. If so, and if the search depth of the stored position is greater than or equal to the desired search depth we can simply use the previously calculated estimate thus saving the work of the search or the evaluation.

If implemented efficiently, a search in a hash table is done in $O(1)$ time [4]. The loss of time for using a hash table in the search algorithm is therefore not significant.

2.1.3 Further Improvements

There are a lot of methods of improving the alphabeta search algorithm. One of the best and easiest methods is to search the moves in a specific order. If you search good moves first, you will get more cut offs in the search tree.

The problem is how to know which moves are good. This is an area of game tree searching where you can benefit a lot from experimenting. You can for example start to examine moves, which take one of the opponent's pieces, or moves centralizing pieces. Another method is to store a list of moves which most frequently have caused a cut off in the search. Later searches will then examine these moves first. The idea is that a move that is good in a certain position in the game tree is probably a good move in other positions in the game tree.

2.2 The Evaluation Function

The natural starting point when creating an evaluation function is to give each piece a material value, and sum up the material values for all the pieces for each side. This is called material balance. It is common to estimate a position in centipawns (hundredth of a pawn). The material value of a pawn is then 100 centipawns and the material value of a queen could be, say, 900 centipawns. When you have the material balance you can start considering strategic aspects of the position. For example, if one side has a double pawn, you can subtract 10 centipawns from his material value. This type of modification of the material balance is done for both sides and for a lot of different aspects of the position.

Here are a few things that are useful to consider when creating an evaluation function for chess positions:

King safety
Since losing the king means losing the game, you have to make sure that it is hard to attack and easy to defend your king.

Piece placement
Pieces are usually better placed in the centre of the board rather than in the edges.

Development
It is important to see to that you get all your pieces in play, and to not let pawns or pieces hinder the development of your pieces.

Pawn structure
Generally, you would not want to defend a pawn with a piece because a piece often has a better task (like defending or attacking other pieces). Instead you want to defend a pawn with a pawn. Therefore it is important to make sure that your pawns are connected. A pawn island is a group of pawns that are not connected to other pawns. A pawn island can form a chain of pawns that protect each other, thus reducing the weakness of the island to the base of the chain. What you want to do is to minimize the number of pawn islands. Also, you want to avoid isolated pawns and double pawns. A double pawn is two pawns on the same file.

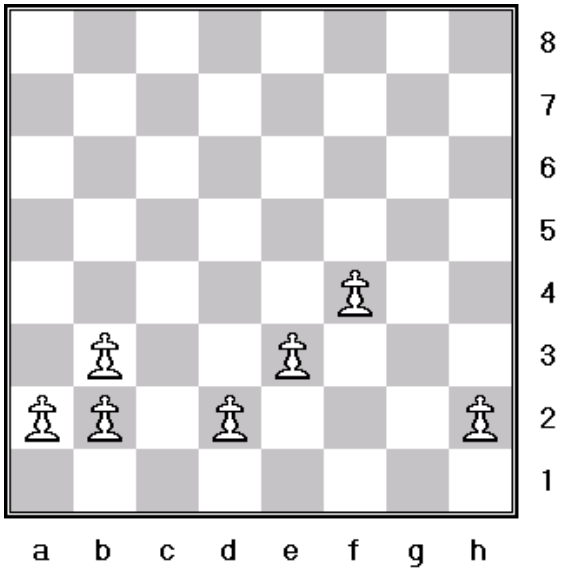

Figure 2.5 Pawn structure. The group a2, b2, b3 is a pawn island with a double pawn at b2,b3. The group d2, e3, f4 is a pawn island in a chain with the base d2. The pawn at h2 is isolated.

Influence over the centre
Since pieces are better placed in the centre, this is a natural battleground. You want to control the centre squares with pieces and pawns in order to keep your opponents pieces out of the centre and in order to be able to place pieces of your own there.

Endgame knowledge
The endgame is the phase of a chess game when most of the pieces and pawns have been traded off, and the objective mainly is to promote pawns and to mate your opponent. In this phase some of the strategic aspects change. Pawns become more valuable, and the king is to be reckoned with as a striking force. During the endgame the king is better placed in the centre. This is especially important when trying to mate a king with for example a king and a rook. The strategy is to drive the opponent king first out of the centre to the edge of the board and then from the edge to the corner, where the mate occurs.

3 NEURAL NETWORKS

Since the human brain is the very base of our definition of intelligence, it would seem sensible to mimic the brain in our attempts to create an intelligent machine. A neural network is such an attempt and I will give a brief description of the fundamentals of neural networks in this chapter.

The brain consists of a vast number of neurons that are connected to each other. A neuron is a relatively simple processing unit that basically gets input, called stimulation, from other neurons and sends output, called activation, to other neurons. Both the input and the output can be either inhibitory or excitatory. Depending on the input, the neuron determines if, and how much, the neuron should stimulate the other neurons. The state and flow of the neurons and their activations determine the information that is represented. The power of calculation in the human brain comes not from the neuron per se, but from the fact that all the neurons process information at the same time [3].

3.1 The Perceptron

The perceptron has the same characteristics as a neuron. It has a set of inputs, and sends the output 0 or 1 depending on the input. Each input has a weight that defines if the input should be inhibitory or exhibitory. An inhibitory weight has a negative value and an exhibitory weight has positive value. We define the inputs as $(x_1, x_2, \ldots, x_n) \in R$ and the weights as $(w_1, w_2, \ldots, w_n) \in R$ with values $\{0 \ldots 1\}$.

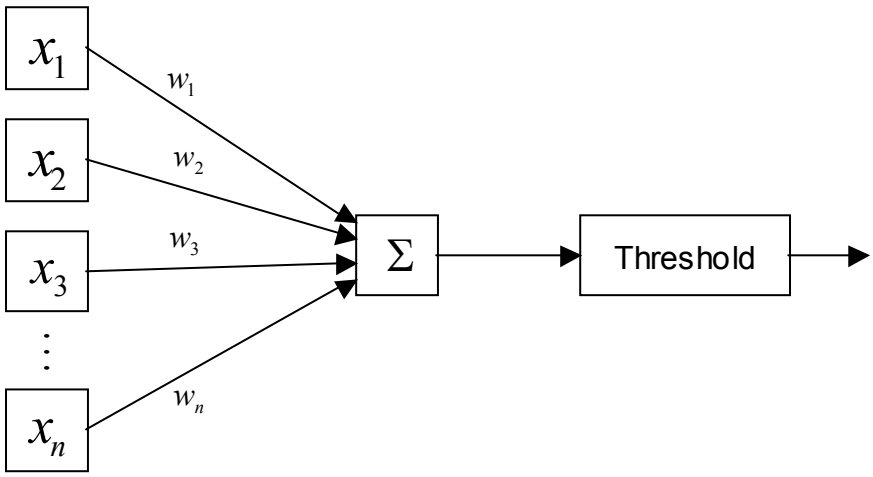

Figure 3.1 The perceptron. The output of the perceptron is 1 if the weighted sum of its inputs is greater than the threshold. Otherwise the output is 0.

The weighted sum of the perceptrons inputs, $g(x)$ is defined as

$$g(x) = \sum_{i=1}^{n} x_i w_i$$

If $g(x)$ is greater than the adjustable threshold, t, then the output $o(x)$ of the perceptron is 1, otherwise, the output is 0.

$$o(x) = \begin{cases} 1 & if\ g(x) > t \\ 0 & if\ g(x) < t \end{cases}$$

The beautiful thing about a perceptron is that everything it can compute, it can learn to compute [5]. Learning is done by letting the perceptron produce output for a set of inputs called the training set. The training set also contains the desired output for each input. The weights are then adjusted according to gradient descent to minimize the error between the actual output of the perceptron and the expected output [5]. This way, the perceptron can learn to match pairs of input and output. The time span in this learning is measured in epochs, where an epoch is the adjustment of the weights for every instance in the training set. The training is continued until the mean error is as low as desired.

The problem with the perceptron is that it can only learn to classify problems that are linearly separable [5].

3.2 Back Propagation

In order to be able to classify problems that are not linearly separable, we need to arrange a set of perceptrons as a connected network consisting of multiple layers. We do this by having an input layer, much the same as the inputs in a perceptron, a number of hidden layers and an output layer. The hidden layers are called hidden, because we do not know, and are not interested, in what they contain. We are only interested in the input and its corresponding output.

Neural networks can be arranged in different ways, and use different learning algorithms. Examples of different kinds of neural networks are Boltzman machines, Temporal Difference Delay Networks and Competitive Learning Networks. This thesis uses a Feed-Forward Network with Back Propagation as learning algorithm. The reason for this is that I started experimenting with this type of network because it is simple to implement. I did not see any use in trying other types of networks or learning algorithms since the results I got was good enough.

Now, in the brain, layers of neurons can loop back and affect previous layers of neurons. In order to avoid the complexity that this produces, we use networks that do not have any loop back circuits. This kind of network is called a Multilayer Feed-Forward Network. A network where every node of one layer is connected to every node of the next layer is called a fully connected network. In this model, computation is done separately for each layer rather than in parallel. The first hidden layer calculates its output, based on the input from the input layer. The output of the first hidden layer becomes the input of the next layer and so on.

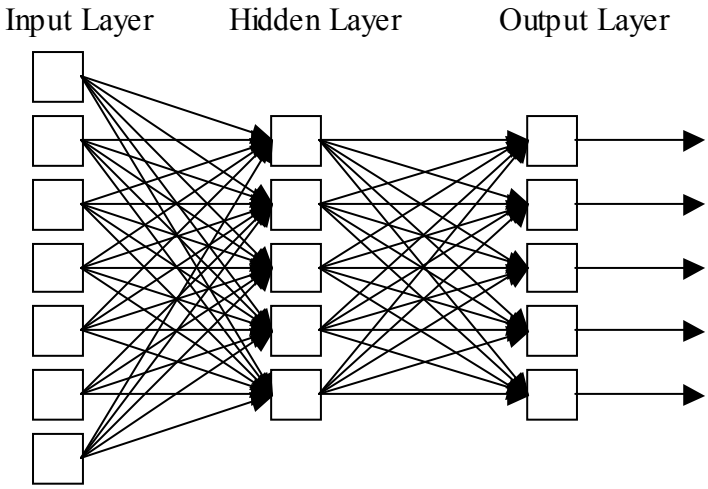

Figure 3.2 A fully connected Multilayer Feed-Forward Network. The hidden layer calculates its output, which becomes the input for the next layer.

The input layer is designed so as to be able to represent the instance of what we want to classify and the output layer is designed so that it is able to represent the kind of output we want. As for the hidden layer, in general, the larger the hidden layer or layers, the more information the network can learn to classify. The drawback is that a larger network takes more time to train.

The adjustment of the weights in Back Propagation is done by gradient descent, but since we now have multiple layers, we need to distribute the error over the different layers of weights. This is done by derivation of the activation function [6].

In the perceptron we used a threshold as an activation function. As this function is not derivable, we need another activation function. We use the sigmoid function:

$$o(x) = \frac{1}{1 + e^{-g(x)}}$$

The output of the sigmoid function is a real value, and thus the output of the nodes in the output layer of the network is a real value. The derivation of the sigmoid function is out of the scope of this thesis. For a full description of the Back Propagation algorithm see [4, 5, 6].

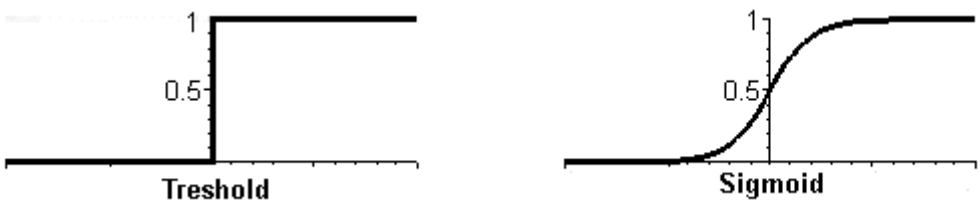

Figure 3.3 The threshold and the sigmoid. Back Propagation needs a derivable activation function.

3.3 Generalization

Simply using a neural network for the purpose of memorizing a set of examples is a total waste of energy. If the learning process has been successful, the network will have made

generalizations from the training set [5]. This means that the network can produce reasonably correct output from input that is not part of the training set, but similar to it.

As we are adjusting the weights in our network according to some training set, it is useful to set aside a subset of the training set that is not used for adjusting the weights. This set, which we call the test set, is instead used to test how good the network has learned to generalize from the training set.

4 METHODS

In this chapter I will be discussing the means by which I have tried to incorporate neural networks in the evaluation function of a chess playing software. The layout and training methods of the networks used in chapter 5 is also described.

4.1 A Simple Endgame

In experimenting with using neural networks as a part of an evaluation function I have concentrated on a simple endgame where one player has a rook and a king, and the other player has only his king left on the board. This endgame is a simple win for the player with the rook. The strategy consists of forcing the opponent's king to one side of the board, and then forcing the king to a corner where the mate is conducted. Here is an example.

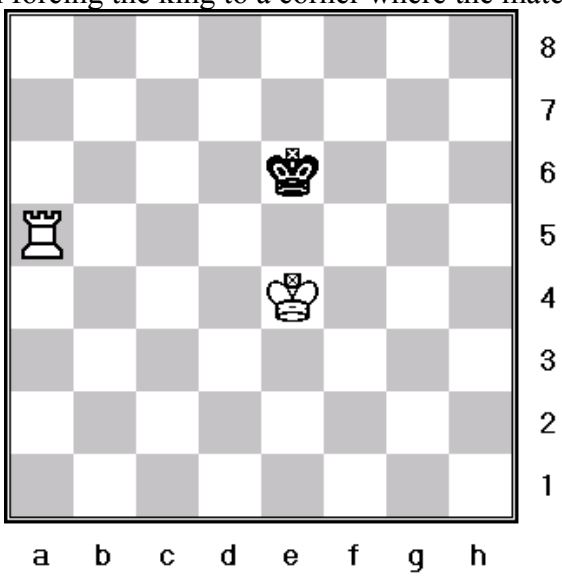

Figure 4.1 White to move mates by forcing the black king into the corner.

White mates easily with for instance this move sequence 1.Ra6+ Ke7 2.Kd5 Kf7 3.Ke5 Ke7 4.Ra7+ Kd8 5.Kd6 Ke8 6.Rb7 Kf8 7.Ke6 Kg8 8.Kf6 Kh8 9.Kg6 Kg8 10.Rb8 mate.

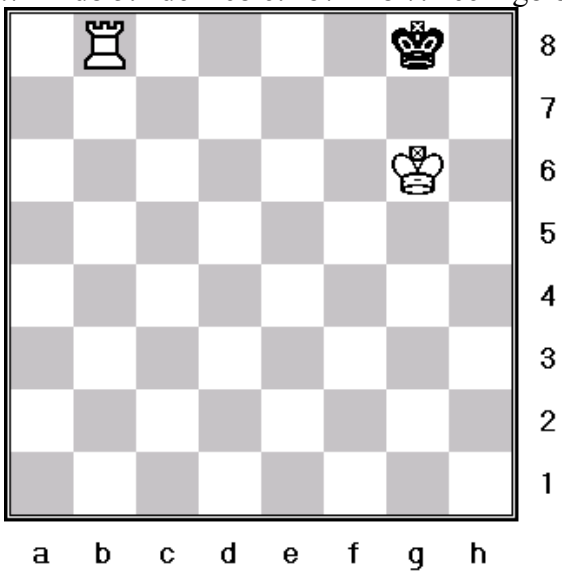

Figure 4.2 Black is checkmate

4.2 Neural Networks in the Evaluation Function

Usually an evaluation function has special cases for handling different type of endgames. For instance when one side only has the king left the position is judged by how close the lone king is to a corner and how close the attacking pieces are to the lone king.

For this project, when discovering that we have an endgame with king and rook versus king, the outcome of the evaluation function is determined solely by a purposely trained neural network. The side with the extra rook is for simplicity here always called white. The training sets are also designed to only consist of position where white have the extra rook. When using the network in the evaluation function, colours are easily flipped to simulate that we have a position where white has the extra rook.

4.3 The Layout and the Training of the Networks

Due to speed considerations I have tried to limit the size of the network. In the input layer each piece is represented by 8 * 8 nodes where the value 1 means that the piece is on this given square, and the value 0 means that the piece is not. Furthermore an extra node is included in the input layer to represent the side to move. The size of the hidden layer in all tests is 20 nodes. Finally the output layer is a couple of nodes depending on the type of output desired for the specific test.

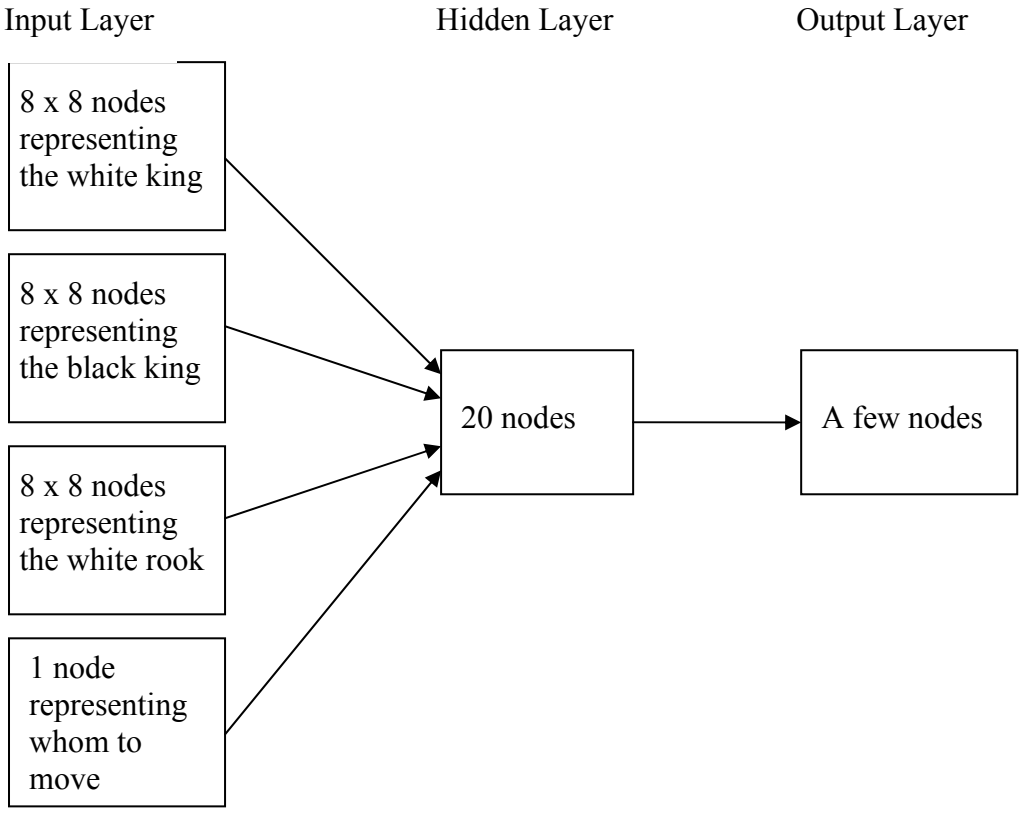

Figure 4.3 The Layout of the neural networks

The desired output matching the input in the training sets is given by the number of moves that you can force mate in the position. This is taken from an endgame database, in which all positions with king and rook versus king is calculated. Endgame databases provide means of perfect information for the positions it contains. The networks are trained with the input and

matching output until the mean error is below a certain value. When this occurs we say that the training has converged and that the network has learned to classify the training set with acceptable accuracy.

The layout of the input layer is obviously not optimal regarding the size of the neural network. One can easily find a smaller input layer to represent the same position with for instance bit representation of the ordinal number of the square for each piece. The reason I chose the representation in figure 4.3 was to get the topological aspect of the placement of the pieces since the how good a position in this endgame is depends heavily on how close the enemy king is to the corner, and how close the kings are to each other.

In order to evaluate the effectiveness of the network it is tested in the evaluation function of a search engine as described in 4.2. Testing is done by letting the engine play a 5 minute game, a 2 minute game and a 1 minute game in 20 positions with rook and king versus king against perfect opposition (i.e. an endgame database). The search engine used is one that I have written for the purpose and it is described in Appendix A.

5 TESTS AND RESULTS

In this section I will present the tests I have done with neural networks in a simple endgame. I will also give my reflections on the test results.

5.1 A First Attempt

I wanted to start as simple as possible to see if neural networks could classify this problem at all, so I started with a test set consisting of 100 random positions. As the desired output I wanted the number of moves to mate, a value between 0-31. Thus the layout of the neural network is as follows: 193 nodes in the input layer, 20 nodes in the hidden layer and 5 nodes in the output layer for a binary output value.

The training converged with a mean error sum less than 0.0001 after 200 epochs. This suggested that classification of this problem should work, but when testing the network on some positions from the test set I found that when the pieces were close to each other the network often reported mate in 0. Binary output seemed to be a bad idea perhaps due to the fact that if you for instance have a position with mate in 4, the output only has to be erring on one node to report that the position is mate in 0.

5.2 Single Node Output

Instead of having binary output, my next idea was to have the output represented by one node, where the better the position is the closer the value comes to 1 and vice versa. To achieve this I let the desired output for the positions in the training set be

$$\frac{31-n}{31}$$

where n is the number of plies to mate.

This seemed to work quite well. The training converged after about 500 epochs and the network also seemed to give reasonable response to the positions in the test set. Testing the network in the search engine showed that it had actually learned something useful. The enemy king is driven towards the edge, but the engine can't find the mate except in two trivial cases. Here is a sample of the performance of the network in the search engine:

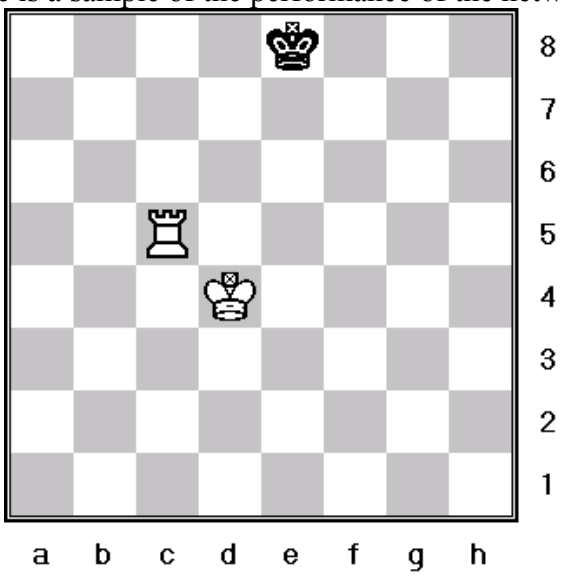

Figure 5.1 White to move.

1. Ke5 Kf7 2. Ra5 Ke7 3. Ra7+ Kd8 4. Ke6 Kc8 5. Rd7 Kb8 6. Kf6 Ka8 7. Re7 Kb8 8. Rd7 Kc8 9. Ke6 Kb8 10. Re7 Kc8 11. Kf6 Kd8 12. Ke6 Kc8 13. Kf6 Kd8

14. Rf7 Kc8 15. Kg7 Kb8 16. Rd7 Kc8 17. Re7 Kd8 18. Ra7 Kc8 19. Re7 Kb8 20. Rd7 Kc8 21. Re7
{Draw by repetition} 1/2-1/2

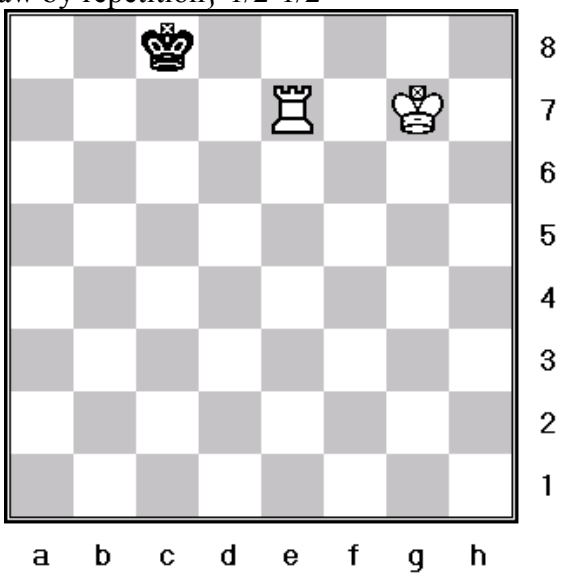

Figure 5.2 The neural network failed to mate the black king.

5.3 Rotation

The single node output was clearly better than binary output. Maybe the poor results were caused by the training set not being comprehensive enough. My next try was therefore to include the symmetrical rotations of the positions in the training set, resulting in a training set of 400 positions.

Surprisingly, with this network, the engine did find the mate in all twenty positions in the five minute test games. It did not always find the quickest mate and looking at the games you can see that it sometimes does awkward moves.

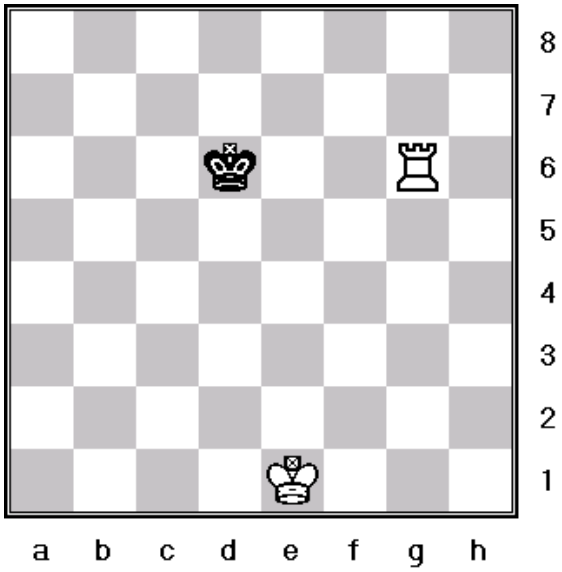

Figure 5.3 The search engine with the neural network finds the mate.

1... Kd5 2. Ke2 Ke4 3. Ra6 Ke5 4. Kf3 Kd5 5. Kf4 Kc5 6. Ke5 Kb5 7. Rd6 Kc4 8. Ke4 Kc5 9. Rd1 Kc4 10. Rd5 Kc3 11. Rc5+ Kb4 12. Kd4 Kb3 13. Rb5+ Kc2 14. Rb8 Kd2 15. Rb2+ Kd1 16. Kd3 Ke1 17. Rd2 Kf1 18. Re2 Kg1 19. Ke4 Kf1 20.

Ke3 Kg1 21. Rf2 Kh1 22. Kf4 Kg1 23. Kg3 Kh1 24. Rf1#
{White mates} 1-0

In the games with faster time controls results were not as good, suggesting when used in more shallow searches the information from the network is not quite enough.

5.4 Enhanced Training

In order to test if more extensive training would result in any improvements, I tried training with the exactly same conditions as in test 5.3, but this time I let the training go on until the mean error sum was less than 0.00001.

Training converged after about 1600 epochs, but the results were not that impressive. Using this network, the engine found mate in two more positions in the 1 and 2 minute games.

5.5 Mate Sequences

In my final attempt, I tried adding three mating sequences to the training set to see if less randomness would improve information in the network.

Training converged with a mean error sum of less than 0.0001 after 453 epochs. With this method further improvements were made in the 1 minute games, and mate was found in 12 out of 20 positions.

5.6 Results

In figure 5.3 the results of the test games for the neural networks are presented.

	5 min. game	**2 min. game**	**1 min. game**
Test 5.1	0	0	0
Test 5.2	2	2	2
Test 5.3	20	15	7
Test 5.4	20	17	9
Test 5.5	20	17	12

Figure 5.3 The number of mates found by the neural networks in a search engine in 20 random positions with king and rook versus king.

Measuring the speed performance in these positions I've found that the implementation of the evaluation function using neural networks was about 3-4% slower than an implementation that does not use neural networks.

6 CONCLUSIONS

Reflecting on the results presented in chapter 5 I find it quite surprising that the networks do so well, especially in the tests which only use random positions. Part of the results can be attributed to the brute force of the search engine as seen in the loss of results in faster games.

It is fairly obvious though, that neural networks are capable of learning strategic chess concepts. Using only random training positions with perfect information of how good the position is, the network learned, as is shown in the game in Figure 5.3, to drive the enemy king towards the edge and then towards the corner and also mate the opponent.

It can be quite hard to figure out what has gone wrong when the network will not learn what you want it to learn. You just have to try to figure it out with trial and error, which can result in quite a bit of work. Also, a bit of experimenting with the parameters and the training set in order to perfect the knowledge learned by the network is required and the best solution is possibly to have a fully handcrafted training set, which cover all the aspects of the strategic concepts that is to be learned.

In the examples used in this thesis, the gain of using neural networks is very questionable, since writing a set of rules that drive the king towards the edge and the corner is a trivial task. The speed of the implementation is also a factor in this case, since using a neural network in the evaluation of the simple endgame resulted in a considerable loss of speed.

7 FURTHER IMPROVEMENTS

There may be areas where the strategies are not that easily defined, for example king safety or pawn structure, were the approach of neural networks may be more effective in both speed and ease of implementation than in the simple endgame used in this thesis. Both king safety and pawn structure are a costly tribute to the evaluation function, and perhaps using neural networks as a substitute will not result in a loss of speed.

The layout of the neural network as shown in Figure 4.3 can easily be modified to handle the pawn structure of a position. You only need 8*8 nodes to represent white's pawn structure and another 8*8 nodes for black's pawn structure. This network is even smaller than the one used in the tests in chapter 5, so the speed issues will be less of a problem.

The problem in this case is to find a reasonable oracle that tells us how good or bad the pawn structure is. One way to test if neural networks can learn to classify pawn structures is to use the evaluation of a strong chess program as an oracle. The benefit of this is that the implementation is easy, but the downside is that the network can only learn what the chess program evaluates.

Another way of implementing an oracle can be to use grandmaster games and let the grandmaster who won be the oracle [9].

A CHESS ENGINE

The chess engine used for testing the neural networks in an evaluation function is written in C++ and uses most standard techniques for tree searching such as:

- Alphabeta cut offs
- Transposion tables
- Move ordering (killers, history moves, mvv/lva)
- Null move pruning
- Iterative deepening
- Quiescence search
- Extensions

The most basic concepts considered in the evaluation function are:

- Material values
- King safety
- Pawn structure (doubled pawns, isolated pawns, free pawns, pawn islands)
- Control of centre
- Development of pieces.

In an average middle game position the engine searches 6-7 ply's in a couple of seconds on a 300 MHz Pentium.

The engine is called StAndersen and plays regularly on the Free Internet Chess Server (FICS):

```
Statistics for StAndersen(C)    (Last disconnected Thu Dec  6, 13:03 CET 2001):

           rating     RD      win    loss    draw   total   best
Blitz       1779     43.9      60      19      12      91   1779 (06-Dec-2001)
Standard    1895    106.9      18       4       0      22
Lightning   1810     64.0      32       7       4      43   1810 (06-Dec-2001)
Wild        ----    350.0       0       0       0       0
Bughouse    ----    350.0       0       0       0       0
Crazyhouse  ----    350.0       0       0       0       0
Suicide     ----    350.0       0       0       0       0
```

StAndersen is available for download at http://surf.to/StAndersen

B REFERENCES

Published references

[1] J. Eade, "Chess for Dummies," Hungry Minds, Inc, 1996

[2] T.D. Harding, "The New Chess Computer Book," Pergamont Press, Great Britain 1985

[3] Russel, Norvig, "Artificial Intelligence – a modern approach," Prentice Hall, New Jersey 1995

[4] Cormen, Leiserson, Rivest, "Introduction to Algorithms," The MIT Press, England, 1996

[5] E. Rich, K. Knight "Artificial Intelligence, 2nd ed., Chapter 18," McGraw-Hill, 1991

[6] D.E. Rummelhart, J.L McClelland, "Parallel Distributed Processing Vol. 1-2," MIT Press 1986

Electronic references

[7] E. Schröder, "Chess in 2010," http://www.rebel.nl/ches2010.htm, 2000

[8] V. Diepenveen, "Diep explained," http://www.students.cs.ruu.nl/~vdiepeve/homepage/about.html, 1997

[9] A Nowatzyk, "Deep Thought Evaluation Tuning," http://www.tim-mann.org/DT_eval_tune.txt, 2000