Universiteit Leiden
The Netherlands

Opleiding Informatica

Neural Networks for Clobber

Teddy Etoeharnowo

Supervisors:
dr. W.A. Kosters & dr. H.J. Hoogeboom

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

26/01/2017

Abstract

Clobber is a strategic board game played on a chequered board with black stones on all black squares and white stones on all white squares. The game is played with two players and each of them is assigned to a color. Players take turns into capturing stones from the opponent, but only in adjacent squares, either horizontally or vertically. The winner is the player who makes the last move.

For this thesis we have developed different strategies to play Clobber with. These algorithms are Random, Monte Carlo, Monte Carlo Tree Search and a Neural Network. The focus lies on creating a Neural Network that can play better than any other algorithm and hopefully can play optimally.

After training our Neural Network by playing against all other algorithms and even itself, Monte Carlo Tree Search still manages to win most games against the Neural Network, especially when it has a large *playout* depth.

Acknowledgements

Many thanks to my supervisor dr. W.A. Kosters for guiding me throughout my thesis and for the weekly meetings that kept me motivated and inspired. I also thank dr. H.J. Hoogeboom for being my second supervisor.

And since this is my last project for my bachelor's degree, I would also like to take this opportunity to thank every teacher that has taught me throughout my years here at Snellius. They have equipped me with the necessary tools that a computer scientist needs to possess.

Finally, I thank my family and friends for being there for me.

Contents

Chapter 1

Introduction

Clobber is an abstract strategy game invented in 2001 by combinatorial game theorists Michael H. Albert, J.P. Grossman and Richard Nowakowski. It has also been investigated by the same people at Games-at-Dal 2001. It was then introduced at the 2002 Dagstuhl Seminar on Algorithmic and Combinatorial Game Theory [1]. Since 2005, it has also been one of the events in the Computer Olympiad.

Clobber is a board game in which two players have to take turns to capture pieces from their opponent. The player that can no longer make any moves is the one that loses the game.

The problem that we try to solve in this thesis is to play Clobber as good as possible. Different algorithms will be developed and compared to each other. These algorithms are Random, Monte Carlo, Monte Carlo Tree Search and a Neural Network. With these algorithms, the focus lies on creating a Neural Network that can hopefully play Clobber optimally. Inspired by AlphaGo, which is a computer program that which also uses Neural Networks, we hope to find similar results with our own algorithms for Clobber.

Figure 1.1: The ICGA Computer Olympiad 2011 took place in the city of Tilburg (NL). CLOBBER was part of the Computer Olympiads [2].

In Chapter 2 we will introduce the game rules. In Chapter 3 we will discuss papers that are related to this thesis. In Chapter 4 the strategies we have used will be discussed in detail. In Chapter 5 we will show the results of the experiments carried out. And lastly, in Chapter 6 conclusions will be drawn and future work will be discussed.

This paper is written as a bachelor thesis at the Leiden Institute of Advanced Computer Science (LIACS), the computer science institute of Leiden University. This thesis has been supervised by dr. W.A. Kosters and dr. H.J. Hoogeboom.

Chapter 2

Game rules

Clobber is usually played on a chequered board with black stones on all black squares and white stones on all white squares. Usually, in competitions, this board is of size 5×6 and when played with computers even of sizes 8×8 and 10×10.

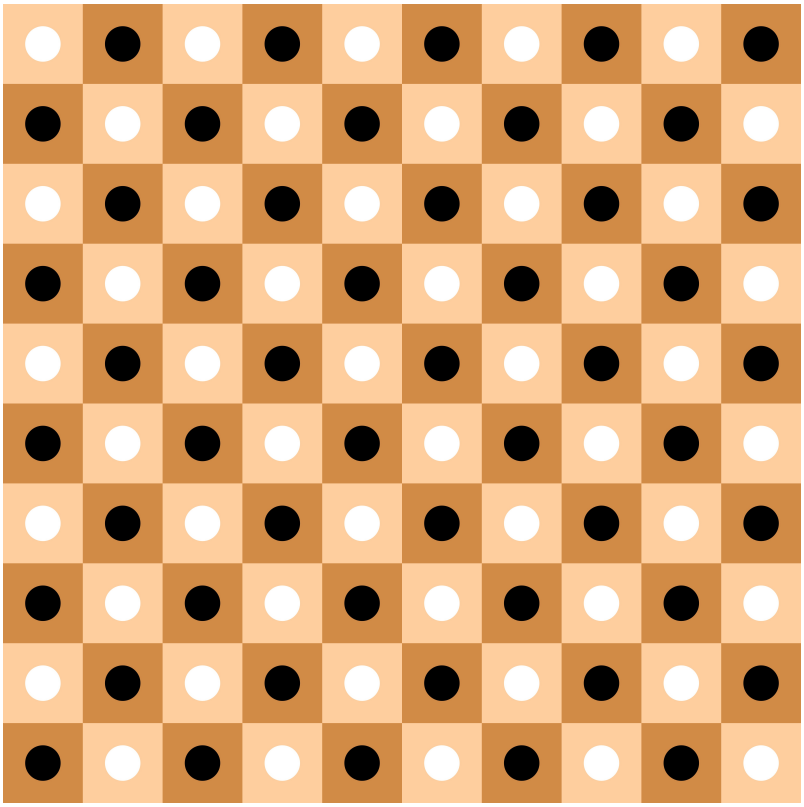

Figure 2.1: Computer Clobber start position.

There are two players. Each of them is assigned to a color. Players take turns and they have to move one of their own stones. A move is when a player takes his or her stone and captures a stone from the opponent directly adjacent, either vertically or horizontally, to the square the stone is on. The stone of the opponent has

to be removed from the board and the stone, that was used to capture, will move to this square. The player who cannot make a move loses the game. Because there is no position in which one player can move and the other cannot, since both players have a move if and only if there are adjacent stones of opposite colors, CLOBBER is an all-small game [3]. And this also means there is always exactly one winner.

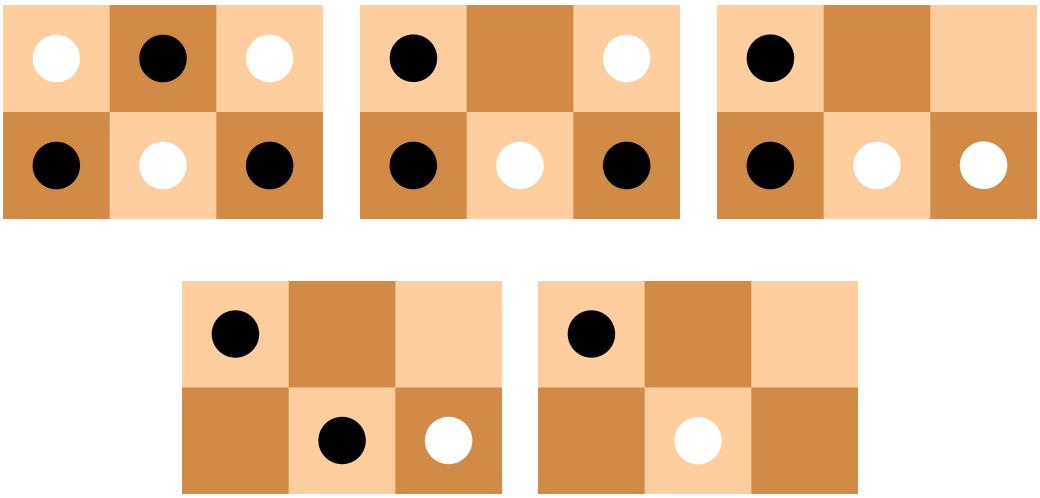

Figure 2.2: A possible game on a 2×3 board. White wins.

There exist a multitude of variations of the game CLOBBER. One variation is when players are allowed to hit their own stones.

Another variation would be to begin with a different initial board. Instead of a chequered board, a random initialized board may be used.

And lastly CLOBBER can also be played with more than only two colors. Although this comes with problems, like how the board must be initialized and how to determine who actually wins.

Chapter 3

Related Work

A paper by Albert et al. introduced the game Clobber. This paper shows that the rules easily generalize when playing on an arbitrary graph. The authors show that determining the value of the game is NP-hard [1].

Griebel and Uiterwijk have applied both a general NegaScout search with many possible enhancements and techniques from Combinatorial Game Theory (CGT) to show that these methods can be combined by incorporating endgame databases filled with CGT values into a NegaScout solver. Ultimately a database with exact CGT values was built for all (sub)games of up to 8 connected pieces. Reduction depends on board size, going down from 100% for the boards in the database to 75% for the 3×6 board [4].

A survey by Browne et al. of the literature to date about Monte Carlo Tree Search (MCTS), provides a snapshot of the state of the art of MCTS research. The authors outline the core algorithm's derivation, analyze the many variations and enhancements that have been proposed, and summarises the results from the key game and non-game domains to which MCTS methods have been applied [5].

Chapter 4

Strategies

In this chapter we will take a look at different strategies that were developed to play Clobber.

4.1 Random

The first algorithm used in this research is the Random algorithm. This algorithm is used for other algorithms, but is also used in the first stages of this research to make sure the Clobber program is working properly. Also, this sets a base line of how a game may look like when a person plays Clobber for the first time.

The algorithm works by firstly analyzing the current board to determine all legal moves it may make. The program does this by going through every square and checking whether this square has a stone of the player that currently has to make a move. It then checks if it can capture opponent stones. Then it uses a random number generator to choose between all legal moves.

4.2 Monte Carlo

For our next method, an algorithm was needed that can actually play this game at a respectable level. Considering the large state-space of a Clobber game, especially on larger boards, an algorithm like Minimax would take too many resources. The number of nodes increases exponentially and since one needs to get all the way to the end of a game to determine who will win, this method is considered infeasible for this problem.

Monte Carlo (MC) should be a significant better player than a Random player. It uses the Random algorithm explained previously to approximate the best move at a given time with a given board.

The algorithm works by trying each possible move from a given state. Each possible move gets to be played until the end for a set number of times (hereafter referred to as "*playouts*"). So this means that for each move made by MC, the algorithm actually plays the games *playouts* × *number of possible moves* times. This is done by using the Random algorithm. After a few *playouts*, the strength of each possible move can be roughly determined. At the end of each *playout*, the winner of each simulation will be determined. The move with the most wins in all of its *playouts*, for the player currently playing, is considered as the best move to do.

4.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) has been considerably popularized recently by Computer Go programs, most notably AlphaGo which was able to beat Lee Sedol, a 9-dan professional, in a five-game match [6].

MCTS should in theory be better than the conventional Monte Carlo method. MCTS uses a search tree to keep track of moves, giving MCTS the precision of a search tree and still the generality of random simulations. MCTS, like MC, also uses *playouts*, by selecting random moves. After a *playout* the tree will be updated, in order to make better nodes more likely to be chosen in the future. There is a difference between the number of actual game simulations in MC and MCTS; in MC it is: *playouts* × *number of possible moves*, whereas in MCTS the number of simulations is actually equal to the value of *playouts*. The ratio between exploration and exploitation is controlled by a special parameter.

The implementation used in this research is the most popular algorithm in the MCTS family, the Upper Confidence Bound for Trees (UCT) algorithm [5]. Figure 4.1 shows the UCT algorithm in pseudocode. Each node v has pieces of data associated with it: the game state $s(v)$, the associated action $a(v)$, the total simulated number of wins $Q(v)$ and the visit count $N(v)$. The term $\Delta(v, p)$ denotes the reward Δ associated with player p at node v.

The UCT algorithm can be divided into its functions used in our implementation:

- *UCTSearch*
- *TreePolicy*
- *Expand*
- *BestChild*

- *DefaultPolicy*
- *Backup*

UCTSearch starts of by creating a root node for the state the program is currently using the algorithm for. Each node contains a certain amount of data to keep track of game states, this will be discussed in the explanation of *Expand*. Within the computational budget that is given to the algorithm, *UCTSearch* calls the following functions: *TreePolicy*, *DefaultPolicy* and *Backup*. Subsequently, it uses the *BestChild* function one more time to determine which move the program should make next.

TreePolicy is used to return a node for *UCTSearch* to use. It checks whether a node, which starts at the root node, is a non terminal node. If it is, it returns the current node. If it is not, then it will either *Expand* or go to a child, until it finds a non terminal. If a node is not fully extended, it will simply add another child using *Expand* and returns that particular node. If the node is fully extended it will use *BestChild* to choose one of its children and then returns this child back to *UCTSearch*.

Expand adds a child to a node, which is not fully extended yet. Each node will contain:

- pointers to its children and parent
- the game state: strings which describes the board and which player is currently playing
- the number the node has been visited (*playouts*) and how many times these resulted in a win.

BestChild returns the best child. It does this by using the formula:

$$\frac{Wins(v')}{playouts(v')} + c\sqrt{\frac{2\ln playouts(v)}{playouts(v')}}$$

The child associated with the highest result from this formula will be returned. c is the tunable bias pa-

function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)
  return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

Figure 4.1: This figure shows the UCT algorithm in pseudocode from [5].
This code is a summary of UCT descriptions from several sources, notably [7].

rameter which regulates the ratio of exploration and
exploitation. A smaller c means less exploration and more exploitation; nodes with the highest simulated reward will be chosen, while a bigger c forces relatively less frequent visited nodes to be expanded also. In this case, c is usually $\frac{1}{\sqrt{2}}$ which was shown by Kocsis and Scepesvári [8] to satisfy the Hoeffding inequality with rewards in the range [0;1]. But when *UCTSearch* calls *BestChild* it uses a c of 0, because we do not want *BestChild* to use exploration at all at this stage.

DefaultPolicy makes moves at random until the game ends. And will return whether the current player will win this simulation.

Backup receives a node that was returned by *TreePolicy* and the score returned by *DefaultPolicy*. And it updates the search tree accordingly; every ancestor node will have their number of *playouts* incremented. And if *DefaultPolicy* simulated this node to a win it will also increment the number of wins for every ancestor.

4.4 Neural Network

After MC and MCTS, the Neural Network (NN) is ready to be developed. In the hope it will play better than both MC and MCTS, the Neural Network has to train against either of these methods beforehand. The steps that will be taken could look like this:

- Firstly, the Neural Network learns from games against a Player that selects its moves randomly.
- Then it should learn from MCTS.
- Then against itself.

The main Neural Network that is used in this research is a feedforward Neural Network. The Neural Network takes the state of the game as the input. The Neural Network has one output node. This node should give an estimation on how good a given state is, and depending on these numbers for all possible moves on a given state, the next move should be made.

The input nodes are connected to the next layer of nodes. This layer is called the hidden layer. In theory multiple layers are possible and can have an added value when looking at complex problems. These connections between nodes have weights which ultimately determine how each node interacts with each other to come up with the end value. Usually we start with randomly initialized weights, but in this project we also have a few instances where we use weights from other Neural Networks, to give the Neural Network a head start. When the Neural Network has been initialized with random weights, the Neural Network

is already able to return a result value after a state has been given to it. Though not reliable, the Neural Network gets its result by doing the following:

- First it receives the state of a game. The state of a game consists of the current board and whose turn it is. The Neural Network takes the board by mapping each square to one input node. If there is a black piece on a square, the node will get a value of 1. If the piece is white, the node will get a -1. And when it is empty it will get a 0. After that an extra node will be assigned to represent those turn it is.

- Next is to determine what the values will be of the hidden nodes. This is done by using a logistic sigmoid function as the activation function of, what we will refer to as, the *invalue* of a node. The *invalue* of a node is the sum of all input nodes multiplied by their corresponding weight to the given hidden node. The sigmoid function used is $g(x) = \frac{1}{1+e^{-\beta x}}$ with β dictating the slope of the sigmoid.

- In the same manner the output value will be determined using the hidden nodes and their weights in relation with the output node.

The strength of a Neural Network lies in its abilities to learn. This is where Backpropagation comes into play. Backpropagation is a method to update the weights of a network. The calculated output from the network will be compared to the desired output and this error value will be used to change the weights of the Neural Network. How the desired output is being determined will be discussed later. The weights will be updated accordingly:

- a value called Δ will be calculated for the output node. For this it also needs to know the *error*. The *error* is how much the output differs from the desired output.

$$\Delta = error \times g'(invalue\ of\ the\ outputnode)$$

with

$$g'(x) = \beta g(x)(1 - g(x))$$

- the Δs will also be calculated for the hidden nodes

$$\Delta_i = g'(invalue\ of\ a\ hidden\ node_i) \times W_i \times \Delta$$

with

$$W_i \leftrightarrow Weight\ of\ hidden\ node_i\ to\ the\ output\ node$$

- finally the weights will be updated

$$W_i \leftarrow W_i + \alpha \times value\ of\ hidden\ node_i \times \Delta$$

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times input_j \times \Delta_i$$

with α being the learning rate.

As for the training of the Neural Network, we use the following method: We let the Neural Network play against its opponent. During this game we will save the game states, which consist of both the board and whose turn it is. And when the game is over we will use these game states to learn from. We will send each game state, from beginning till the end, to the Neural Network for Backpropagation. When the game was won, each game state will receive a 1 as a target value when it is sent to the NN for Backpropagation, and -1 when it is a losing game. Every time a state is sent to the Neural Network to learn, the α value, within a game, will increase by 10%. This is done so that the last few rounds have a bigger impact on the Neural Network than the first few ones.

Chapter 5

Results

5.1 Random vs Random

In our first set of tests we let RANDOM play against RANDOM on various boards. This is to test whether starting a game may give a significant advantage to the player starting. Although the winner can be determined beforehand in games on smaller boards when the player plays perfectly, this cannot be done (yet) on larger boards.

Board sizes used are:

- 4×4; this is the largest square board used in the paper of Albert et al. [4].
- 5×4; this is the smallest board not included in the paper of Albert et al. [4].
- 8×8; standard board for computer players.
- 10×10; standard board for computer players.

We let the two RANDOM players play against each other for 120,000 milliseconds.

The results are shown in the Table B.1 in the Appendix.

The chance of winning is at each board still around 50%. Beginning a game does not seem to give a huge advantage when RANDOM players are playing.

5.2 MC vs Random

For our second test we measured the strength our Monte Carlo (MC) algorithm. MC should be a better player than a Random player. But how much better MC is in comparison to a Random player should also be measured. For this MC was let to play against our first algorithm, Random. MC was left to play against Random, with different depths and different boards. Also we let MC play both as the black player and the white player.

The results are shown in the Table B.2 in the Appendix.

In the case of MC with 10 *playouts*, it can already manage a better win rate than 88.4% on every board size that was given to it. With 100 and 1000 *playouts*, MC can reach 93.9% and 95.9% respectively. The win rate of MC generally increases with the number of *playouts*. But also when the number of *playouts* is high enough, the win rate increases for larger board sizes.

5.3 MCTS vs Random

For our next test we measured the strength our Monte Carlo Tree Search (MCTS) algorithm, by comparing it to our Random algorithm. We let MCTS play against Random, with different depths and different boards. Also we let MCTS play both as the black player and the white player.

The results are shown in the Table B.3 in the Appendix.

MCTS with *playouts* 10, 100 and 1,000 have winning rates above 92.4%, 97.7% and 96.8%. Again the win rate generally increases with *playouts*. The minimum winning rate may have decreased with 1,000 *playouts*, but this could be an imprecision due to the lack of a larger number of games.

5.4 MCTS vs MC

To make sure the results above are correct we let MCTS play against MC, with different depths and different boards. Both MCTS and MC were able to play as both black and white players.

The results are shown in the Table B.4 in the Appendix.

With the same number of playouts, MCTS wins from MC in most cases. The only cases in which MC won more times than MCTS were on 4×4 when MCTS was white and on a 10×10 with 1,000 *playouts*.

Lastly for our MCTS against MC comparison we let each algorithm calculate its next move for only 100 milliseconds. MCTS is a bit slower per *playout*, because it has to make new nodes and has to make a few more calculations. So, both algorithms were taken to play against each other with the same time resources to make it more fair.

The results are shown in the Table B.5 in the Appendix.

When giving both the same amount of time, 100ms, to calculate their best move, MCTS plays considerably better, with the exception of on a 4×4 board when MCTS was white.

5.5 Neural Network tuning

Before the Neural Network (NN) starts learning, it firstly has to be tuned to a moderately degree. The Neural Network's α, β and the number of nodes are tuned before it trains.

Firstly we take our NN with a 5×4 board, $\beta = 1.0$ and number of hidden nodes is equal to 20. And then the initial values of α is varied between the values 0.01, 0.02, 0.05, 0.10 and 0.20.

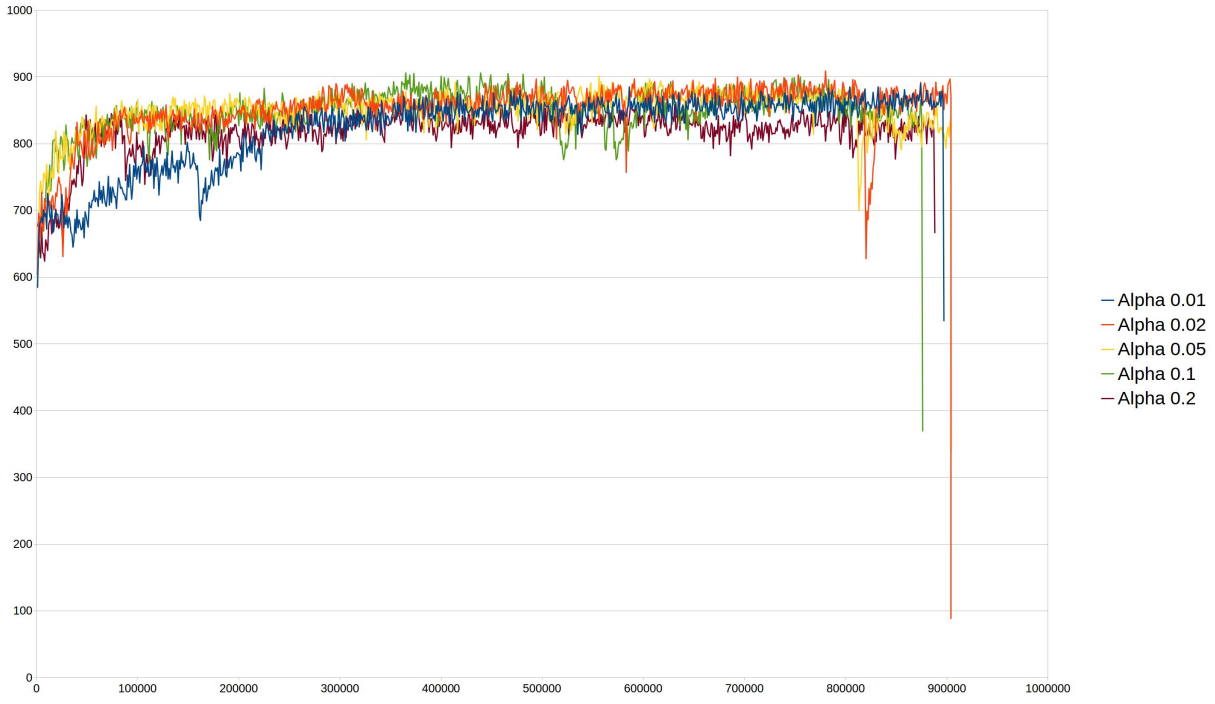

Figure 5.1: The comparison of NNs, with different initial α, learning against Random. On the y-axis is the win rate per 1000 games. On the x-axis is the number of games played in total.

According to the results, shown in Figure 5.1, there is not a huge difference between the chosen α values, although a value of 0.02 for α was by a little bit better than the rest at the end. Therefore, the value 0.02 was chosen as the initial value of α that will be used for further research. With the NN adjusted with the right α,

it now was taken with various βs to determine the best value for β.

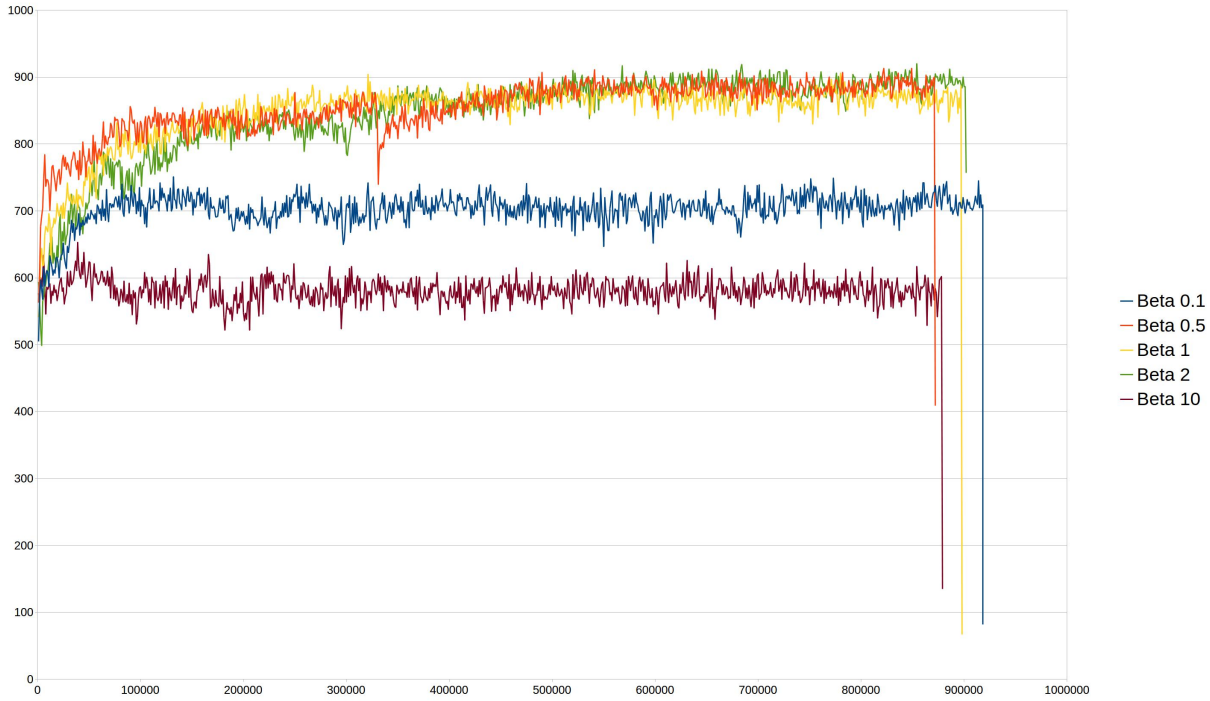

Figure 5.2: The comparison of NNs, with different β, learning against Random. On the y-axis is the winrate per 1000 games. On the x-axis is the number of games played in total.

In the case of β it clearly shows (Figure 5.2) that choosing the right beta has a significant impact. The values for β 0.10 and 10 clearly is worse. But there is not a huge different between the values 0.50, 1.0 and 2.0. But the value of 2.0 is just a bit better than the other values at the end. Therefore, the value 2.0 was chosen as the initial value of α that will be used for further research.

Both α en β wave now been set the only variable remaining is the number of nodes. For this we tried various nodes on different board sizes which were going to be used in the next few experiments. The board size 10×10 has been dismissed from the experiments, because this board seems to be too big for our Neural Network's capabilities. Instead games on a 6×6 board have been analyzed.

Board	# of hidden nodes	wins by NN
4×4	10	946
	20	987
	30	971
5×4	10	821
	20	923
	30	913
	40	956
6×6	20	537
	30	551
	40	477
	50	484
	60	522
8×8	20	489
	40	460
	60	453
	80	462
	100	451
	120	488

Table 5.1: Comparison of different number of hidden nodes on different board sizes.

These results, demonstrated in Table 5.1, shows
us that having 20 hidden nodes resulted in the
best outcome, with exception of on a 6×6 here the optimum lies on 30 nodes. So these are the numbers of hidden nodes used for each NN.

5.6 Neural Network vs Random Learning

For first part of the Neural Network's training we let it play against the Random player for 900,000 milliseconds. The purpose of this is to give the NN a rough idea of how the game works. After every 1,000 a Neural Network will be outputted for further research.

Board size	NN's color	ratio of wins (black) first set	# completed games	ratio of wins (black) last set
4×4	b	0.679	3,692,000	0.970
	w	0.524	5,242,000	0.180
5×4	b	0.567	2,683,000	0.900
	w	0.555	3,501,000	0.213
6×6	b	0.469	605,000	0.490
	w	0.499	645,000	0.527
8×8	b	0.510	151,000	0.465
	w	0.540	151,000	0.518

Table 5.2: Neural Network vs Random Learning. The first set is the the first 1,000 games. And the last set concerns the last 1,000 games.

As the results show (Table 5.2) learning from playing against a Random player make the NN, particularly on boards 4×4 and 5×4, play a lot better, e.g. on a 4×4 the black player increases its win rate from 67.9% to 97.9% after 3,692,000 games. With an 8×8 board the NN actually plays worse at the end than when it never trained against a Random player.

5.7 Neural Network vs MCTS learning

After learning against the Random player, the NNs were chosen to play against MCTS with *playout* depth of 10.

The results (Table 5.3), looking at "first set" comparing Pre- and Post-training in particular, show that learning from Random does not give an head start on learning the play patterns of MCTS, except for the black player on a 4×4, in this instance the NN increases its win rate from 3.4% to 30.9%. The board size 8×8 did not have enough plays ($<$1,000) in our experiments to be usable.

Also, when comparing the first set with the last set, the results demonstrates the NN actually does not learn well from playing against MCTS. In most cases this actually resulted in a lower win rate, e.g. the black Post-Training player on a 4×4 decreases its win rate from 30.9% to 5.1%.

	Board size	NN color	ratio of wins (black) first set	# completed games	ratio of wins (black) last set
Pre-Training	4×4	b	0.034	40000	0.037
		w	0.99	30,000	0.984
	5×4	b	0.052	26,000	0.072
		w	0.983	29,000	0.988
	6×6	b	0.021	4,000	0.012
		w	0.972	3,000	0.984
	8×8	b	0.016	562	N/A
		w	0.984	557	N/A
Post-Training	4×4	b	0.309	89,000	0.051
		w	0.960	66,000	0.998
	5×4	b	0.030	36,000	0.026
		w	0.977	26,000	0.989
	6×6	b	0.023	4,000	0.010
		w	0.964	3,000	0.975
	8×8	b	0.022	593	N/A
		w	0.983	538	N/A

Table 5.3: Neural Network vs MCTS learning. The first set is the the first 1,000 games. And the last set concerns the last 1,000 games. "N/A" in the last column shows there is not enough plays to reach 1000.

5.8 Neural Network vs Neural Network learning

After learning from both Random and MCTS. The NN has to play against another NN.

Board size	ratio of wins by black in the first set	number of completed games	ratio of wins by black in the last set
4×4	0.413	125,000	0
5×4	0.451	104,000	1
6×6	0.779	21,000	0.991

Table 5.4: Neural Network vs Neural Network learning. The first set is the the first 1,000 games. And the last set concerns the last 1,000 games.

These results (Table 5.4) tell us which NNs actually win, after a few games against another NNs. Only the winning NN was saved for the last comparison. When we look at the instance on the 4×4 board, both the NN playing black and the NN playing white are about as good as each other, since their win rates are 41.3% and 58.7% respectively. But at the end white wins every game of the last 1,000 games. This could mean the white NN gets better, but this also could mean the NN playing as black gets worse. To test this we have our last test set.

5.9 Neural Network vs MCTS

For our last set of experiments we compare NNs from different learning levels against MCTS with a *playout* of 100, to test the strength of the NNs. All of these NNs were collected from previous experiments described above.

These results (Table 5.5) show that learning from Random has the highest success rate, when looking at winning from MCTS with a *playout* depth of 100.

	NN's color	black wins in the first set	ratio of wins by black in the first set
NN0	b	1	0.01
	w	100	1
NN1	b	6	0.06
	w	99	0.99
NN2	b	3	0.03
	w	99	0.99
NN3	b	N/A	N/A
	w	99	0.99

Table 5.5: In this table the results are shown of the plays of different NNs from different levels. NN0 is a RANDOM initialized NN. NN1 has been learning for 900,000 milliseconds. NN2 is the same as NN1 plus it has been learning from playing against MCTS with *playout* depth of 10 for 300,000 milliseconds. NN3 is the same as NN2 plus it has been learning from playing against a NN2. There is only 1 NN3 in this table, because only the winner of our NEURAL NETWORK vs NEURAL NETWORK learning comparison was saved.

Chapter 6

Conclusions

We have developed and compared different strategies for playing the game Clobber. These strategies were Random, Monte Carlo, Monte Carlo Tree Search and a Neural Network. The algorithms have been compared to each other and nothing seems to defeat Monte Carlo Tree Search, particularly when it has a large *playout* depth.

Although we tried to make the Neural Network play as good as possible, the Neural Network could not learn good enough from other strategies. The Neural Network, using our method, does not learn particularly well after playing against either MCTS or another Neural Network, which is most likely why our Neural Network could not defeat our Monte Carlo Tree Search.

6.1 Future work

While training some kind of Simulated Annealing, e.g., lowering α the more the Neural Network has played, might have a positive impact.

Giving the Neural Network more resources than was given to it in this research could definitely improve performance

For future work a bigger Neural Network would be the next step. This can be done with more layers and thus will accommodate more complex systems, in particular for larger boards.

An option would be to use a different kind of Neural Network. A good example could be a Convoluted Neural Network, which is also used in Deep Learning programs which plays Go [9]. Altogether, much research can still be done in this field.

Bibliography

[1] M. Albert, J. Grossman, R. J. Nowakowski, and D. Wolfe, *An Introduction to Clobber*. Integers: Electronic Journal of Combinatorial Number Theory 5, 2005.

[2] I. Althöfer, *Computer Olympiad 2011 - Clobber*. `http://www.althofer.de/clobber/clobber-2011-mcjena+pan.jpg`, [accessed 10/01/2017].

[3] E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for Your Mathematical Plays*. A K Peters, Ltd., 2nd ed., 2001.

[4] J. Griebel and J. Uiterwijk, *Combining Combinatorial Game Theory with an α-β Solver for Clobber*. Proceedings BNAIC, 2016, pages 48–55.

[5] C. Browne, E. J. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. P. Liebana, S. Samothrakis, and S. Colton, *A Survey of Monte Carlo Tree Search Methods*. IEEE Trans. Comput. Intellig. and AI in Games 4, 2012, pages 1–43.

[6] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, *Mastering the Game of Go with Deep Neural Networks and Tree Search*. Nature 529, 2016, pages 484–503.

[7] S. Gelly and D. Silver, *Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go*. Artif. Intell. 175, 2011, pages 1856–1875.

[8] L. Kocsis, C. Szepesvári, and J. Willemson, *Improved Monte Carlo Search*. University Tartu, Estonia, Technical Report, 2006.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Appendix A

System

All tests in this thesis were run on the same system:

- Operating system: Ubuntu 14.04 LTS 64-bit
- Processor: Intel® Core™i7-3610QM CPU @ 2.30GHz × 8
- Graphics: NVIDIA® GeForce® 610M with 2GB DDR3 VRAM
- Memory: 7,7 GiB DDR3 @ 1600 MHz

Appendix B

Tables

This appendix will provide supplementary tables.

Board size	# of games	wins (black)	ratio wins (black)
4×4	6,518,800	3,342,954	0.512
5×4	4,927,263	2,768,973	0.562
8×8	769,984	382,272	0.496
10×10	348,574	170,402	0.489

Table B.1: Random vs Random

Playouts	board size	MC's color	# of games	# of wins (black)	ratio of wins (black)
10	4×4	b	69,252	66,476	0.960
		w	110,453	12,764	0.116
	5×4	b	36,066	34,167	0.947
		w	50,256	5,310	0.106
	8×8	b	560	521	0.930
		w	655	40	0.061
	10×10	b	111	104	0.937
		w	123	14	0.114
100	4×4	b	7,940	7,870	0.991
		w	12,143	737	0.061
	5×4	b	3776	3,721	0.985
		w	5,151	185	0.036
	8×8	b	63	62	0.984
		w	67	4	0.060
	10×10	b	11	11	1
		w	12	0	0
1000	4×4	b	789	784	0.994
		w	1,104	46	0.041
	5×4	b	330	330	1
		w	552	12	0.022
	8×8	b	6	6	1
		w	7	0	0
	10×10	b	2	2	1
		w	2	0	0

Table B.2: MC vs Random

Playouts	board size	MCTS's color	# of games	# of wins (black)	ratio of wins (black)
10	4×4	b	60,798	59,819	0.984
		w	93,677	7,080	0.076
	5×4	b	29,389	28,622	0.974
		w	41,528	2,563	0.062
	8×8	b	555	526	0.948
		w	625	24	0.038
	10×10	b	107	104	0.972
		w	110	4	0.036
100	4×4	b	6,824	6,819	0.999
		w	10,371	240	0.023
	5×4	b	3,106	3,099	0.998
		w	4,690	57	0.012
	8×8	b	58	57	0.983
		w	65	0	0
	10×10	b	10	10	1
		w	11	0	0
1000	4×4	b	742	742	1
		w	1,119	36	0.032
	5×4	b	309	309	1
		w	476	4	0.008
	8×8	b	6	6	1
		w	7	0	0
	10×10	b	2	2	1
		w	2	0	0

Table B.3: MCTS vs Random

Playouts	board size	MCTS's color	# of games	# of wins (black)	ratio of wins (black)
10	4×4	b	36,936	30,835	0.835
		w	37,937	22,710	0.599
	5×4	b	18573	13502	0.727
		w	19,027	8,466	0.445
	8×8	b	279	183	0.656
		w	321	121	0.377
	10×10	b	56	39	0.696
		w	58	27	0.466
100	4×4	b	4,548	4,408	0.969
		w	4,405	2,875	0.653
	5×4	b	1,968	1,673	0.850
		w	2,062	667	0.323
	8×8	b	33	25	0.758
		w	32	10	0.313
	10×10	b	6	4	0.667
		w	6	3	0.500
1000	4×4	b	472	472	1
		w	468	424	0.094
	5×4	b	184	179	0.973
		w	206	57	0.277
	8×8	b	4	3	0.750
		w	4	0	0
	10×10	b	1	0	0
		w	1	1	1

Table B.4: MCTS vs MC with the same *playout*

Time	board size	MCTS's color	# of games	# of wins (black)	ratio of wins(black)
100ms	4×4	b	399	399	1
		w	314	160	0.510
	5×4	b	259	247	0.954
		w	264	69	0.261
	8×8	b	76	67	0.882
		w	76	8	0.105
	10×10	b	45	34	0.756
		w	46	6	0.130

Table B.5: MCTS vs MC with the same amount of time resources