Universiteit Leiden
The Netherlands

Opleiding Informatica

Using the Rectified Linear Unit activation function

in Neural Networks for Clobber

Laurens Damhuis

Supervisors:
dr. W.A. Kosters & dr. J.M. de Graaf

BACHELOR THESIS

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

30/01/2018

Abstract

A Neural Network based approach for playing the game Clobber has been implemented. This approach has been shown to be extremely good at playing other abstract strategy games like Go, Chess and Shogi. Clobber is a two-player board game, the first person unable to move loses. We use ReLU and Leaky ReLU to train against a random opponent and use the resulting network to play against a Monte Carlo opponent and achieve a win rate of over 50%.

We look at various different techniques to create a Neural Network, including two variants of the activation function, ReLU and Leaky ReLU. We also vary the structure of the Neural Network by using different numbers of hidden nodes as well as different numbers of hidden layers. We introduce a Temporal Learning Rate which weights moves made later in the game more.

Contents

Chapter 1

Introduction

The game of Clobber [AGNW05] is an abstract strategy board game in which two players play against each other. The game was introduced in 2001 by combinatorial game theorists Michael H. Albert, J. P. Grossman, Richard Nowakowski and David Wolfe. The goal of the game is to eliminate all possible moves the opponent can make and in doing so one wins the game. Clobber has been featured in tournaments at the ICGA Computer Olympiad since 2005, see Figure 1.1.

In this thesis we discuss several AI agents for playing Clobber, with a focus on a Neural Network based approach. The agents that will be created are Random, Monte Carlo and Neural Network. We will test these agents to determine under what conditions the Neural Network is able to learn near optimal play. The Neural Network is a feedforward network using backpropogation and the Rectified Linear Unit (ReLU) as the activation function for the nodes in the network.

Figure 1.1: An AI agent called Pan playing Clobber at the ICGA Computer Olympiad 2011 [Alt17].

1.1 Thesis Overview

The rules of Clobber will be explained in Chapter 2, including some of its variants. Related work done on the game and the techniques used will be discussed in Chapter 3. Chapter 4 describes the workings of the different agents that have been implemented and what decisions were made during implementation. Chapter 5 discusses the results of the experiments and in Chapter 6 we draw conclusions and discuss what future work could be done.

This bachelor thesis was supervised by Walter Kosters and Jeannette de Graaf at the Leiden Institute of Advanced Computer Science (LIACS) from Leiden University.

Chapter 2

Clobber

Clobber is a two-player strategy game usually played on a chequered $m \times n$ board on which white stones are placed on every white square and black stones on every black square; see Figure 2.1 for a starting position on a 6×5 board. The two players take alternating turns "clobbering" an opponent's stone. This is done by taking one of your stones and moving it onto a square that is currently occupied by an opponent's stone and that is directly next to it either horizontally or vertically. The opponent's stone is then removed from the game and the square your stone was on is now empty. The win condition is to be the last player to be able to make a move, which is called *normal play*. Because it is impossible for one player to still have available moves while the opponent does not, the game of Clobber is an *all-small game*. his also means there is always one winner with no possible draws. The game of Clobber is a partizan game, which means it is not impartial, as the moves that can be made by one player are different from the other player [Sie13], but it does meet the other requirements of being impartial: there are two players who alternate turns, a winner is picked when neither player can make a move, there is a finite number of moves and positions for both players and there is no element of chance.

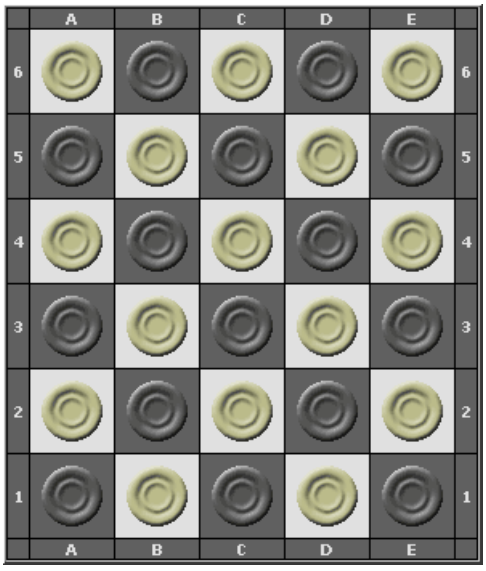

Figure 2.1: Starting position for Clobber on a 6×5 board.

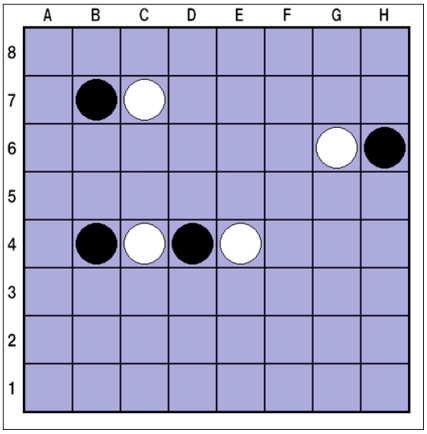

Figure 2.2: A game state where multiple smaller games are played, starting player wins [Gri17].

In competitions Clobber is usually played on different sizes of boards; usually a board of size 6×5 is used between human players and board sizes of 8×8 or even 10×10 are used for games between computer players.

Clobber positions such as the one in Figure 2.2 can be approached from a Combinatorial Game Theory perspective [Sie13] since the three unconnected groups of stones each creating their own smaller game of Clobber with just one winner each. By combining these values one can determine the winner of the entire game.

There are several different variants of Clobber. One of them is a version of Clobber called Cannibal Clobber where you are allowed to capture your own pieces as well as your opponent's pieces. Another variant is Solitaire Clobber [DDF02] in which there is only one player and the goal is to remove as many stones from the board as possible. And finally the game of Clobber does not need to be played on a chequered $m \times n$ board but instead can be played on any arbitrary undirected graph with one stone on each vertex [AGNW05]. This includes variants where the stones are not in a chequered pattern at the start but can be in any pattern, e.g., random.

Chapter 3

Related Work

The game of Clobber was introduced in a paper by Albert et al. in 2005 [AGNW05]. In this paper the authors show that you can play this game on any arbitrary undirected graph and the paper also shows that determining the value of the game is NP-hard.

A basic Neural Network approach has been implemented by Teddy Etoeharnowo in his bachelor thesis [Eto17], which has shown that winning on smaller boards against a random player is fairly easy to achieve, but larger boards are much more difficult to achieve high win rates on. He also implemented a Monte Carlo Tree Search based agent which played better than his Neural Network agent on all different board sizes. A NegaScout search was applied to Clobber by Griebel and Uiterwijk [GU16]; they used Combinatorial Game Theory (CGT) to calculate very precise CGT values of (sub)games and used these to reduce the number of nodes investigated by the NegaScout algorithm by $\sim 75\%$. Many other aspects of Combinatorial Game Theory are described in [Sie13].

Neural Network based approaches have been shown to be extremely good at learning board games that were classically very hard to create AI agents for. These new agents could compete with high level human players and have recently been able to defeat the world champions of Go, Chess and Shogi [SHS+17, SSS+17], the programs use Deep learning techniques, Monte Carlo Tree Search, Reinforcement Learning, and often specialized hardware. In this thesis we only used Reinforcement Learning.

The Rectifier activation function has been used successfully for different tasks [JKRL09]. Several variants of this activation function also exist which have been used to solve specific problems [NE10].

Chapter 4

Agents

Now the different agents for playing Clobber will be explained, namely Random, Monte Carlo and Neural Network. In particular, we describe how these different agents determine what move will be played. The different choices that were made for each agent will also be explained here. Every single agent will abide by the rules of Clobber by only picking moves from a list which contains all valid moves for the current player.

4.1 Random

Random is a very simple player which picks its move by randomly choosing a move from all available moves, where every move has the same odds of being picked. Because there is only one type of move that can be made this agent is not biased towards any play style.

4.2 Pick First

Another agent that was useful to create, similar to Random in its simplicity, was an agent which always picks the first possible move in the list of available moves. Because the move list is always generated ordered in the same way this agent always picks the same move. The move list is generated by going through every direction of every square, starting in the first row and column and incrementing the column until the last column is reached after which the row is incremented and the column is reset to the first one until we reach the final column of the final row. So this player has a tendency to play as near to the upper leftmost corner as possible.

4.3 Monte Carlo

The next agent uses the Monte Carlo algorithm, which employs random playouts to find the move that has the highest chance of winning. The algorithm does this by calculating a score for every possible move it can

make, and plays the move with the highest score. This score is determined by doing a set amount of playouts for every possible move using the Random algorithm until the end of the game. One playout is one full play through of the game from a given position using just the random algorithm. If the game is won in a playout the score of the initial move is incremented by 1. The algorithm does a set number of playouts per possible move, and we let *playouts* denote this number. In a given position the total number of games played is $playouts \times k$, with k being the number of possible moves available to the Monte Carlo player on a given board. By doing enough random playouts the strength of a certain move can be approximated fairly well.

An improvement on the basic Monte Carlo algorithm is called Monte Carlo Tree Search (MCTS) [BPW+12]. This method consists of the following: a game tree is built and a policy is used to determine what node in this game tree to expand; a simulation of the game is then run after which the game tree expands and the policy can select a new node to expand, see Figure 4.1. This algorithm was implemented by Teddy Etoeharnowo [Eto17] and was shown be an improvement over regular Monte Carlo. For this research only the basic Monte Carlo algorithm will be considered.

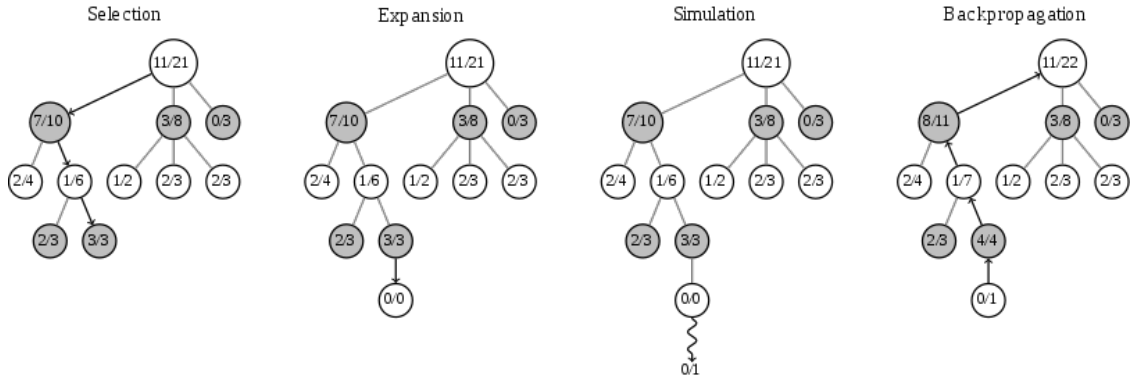

Figure 4.1: The structure of the Neural Network [Dic18].

4.4 Neural Network

The Neural Network agent that has been created in this research utilizes a feedforward Neural Network with the game state as the input layer, and one output node that gives the score of the board in the current state. These input and output nodes are connected through a number of fully connected hidden layers, see Figure 4.2. The activation function used is the Rectified Linear Unit (ReLU).

The way in which the Neural Network is used to determine what moves to play is similar to Monte Carlo in that it determines a score for every possible move that can be made, and, once every available move has a score assigned, it picks the move with the highest score. These scores are calculated by temporarily making every move and using the resulting board as input for the Neural Network and comparing the outputs of all possible moves.

The score of a board is calculated using the following steps:

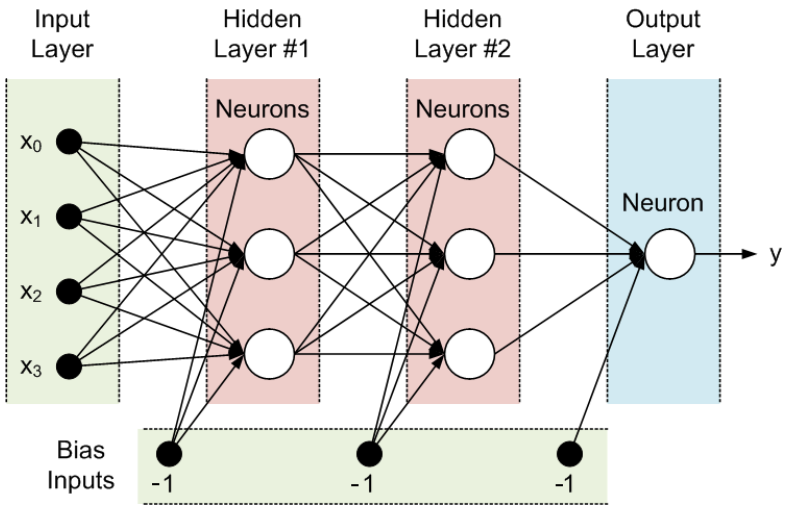

Figure 4.2: The structure of the Neural Network [Mor17], this network has two hidden layers.

- First the entire game state is loaded into the input nodes. Each square on the board is mapped to one input node exactly. If the square contains a stone of the player whose turn it currently is, the value is set to 1, if it is an opponent's stone it is set to -1, and if the square does not contain a stone it is set to 0. The number of input nodes is equal to the board size $+1$; this extra node is the bias node.
- Secondly the values of the nodes in the hidden layers have to be calculated. This value is calculated in two steps. Firstly the invalue of a node needs to be calculated, which is done by taking the sum over all input nodes times the weight that connects the node with the input node. After the invalue of a node has been calculated one calculates the value of the node by using the activation function. This step is then repeated for every hidden layer but instead of the sum over the input nodes the values in the previous layer in the network are used.
- This continues until the value of the output node can be calculated by repeating the same process as used for nodes in previous layers using the sum of all values of the previous layer times their respective weights and putting this invalue through the activation function to get the value of the output node.

The activation function that was used for our network is the Rectified Linear Unit (ReLU) which is defined as follows:

$$g(x) = \max(0, x)$$

A variant of this activation function is called Leaky ReLU [MHN13], which uses a small positive gradient when the input of the unit is negative. This variant was shown to perform as well as regular ReLU but could help combat the dying ReLU problem where all possible input of the network results in an output of 0. The ReLU function and Leaky ReLU function are shown in Figure 4.3, we use a value of 0.01 for a for Leaky ReLU. Leaky ReLU is defined as follows:

$$g(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01 \times x & \text{otherwise} \end{cases} \tag{4.1}$$

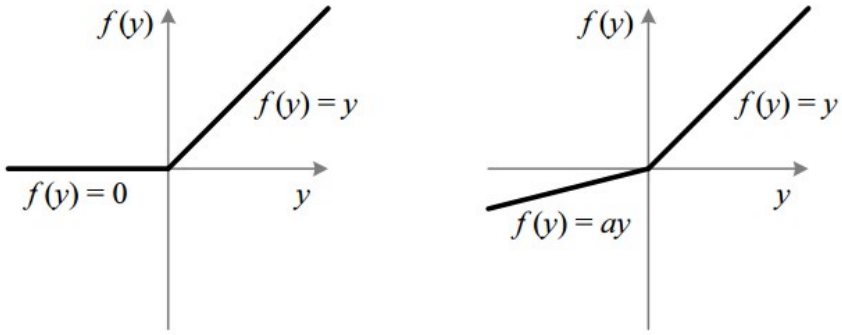

Figure 4.3: ReLU and Leaky ReLU [Sha18].

After a game has been played the result of the game will be used to train the network to play better in the future, rewarding play if the game was won and punishing if the game was lost. This is called Reinforcement Learning [SB18], and is done using backpropogation, which is the process by which the weights in the network are updated. The algorithm compares the value the Neural Network returns for a given position with the result of the game. The positions the Neural Network decides on during play are stored and after a game has finished are all used to train the Neural Network. The boards are passed into the update function in a first in first out system, which means that the first move of the game will also be the first to update the weights of the network. We also introduce a temporal learning factor τ, which changes the learning rate α depending on how far into the game a position is. This τ increases the learning rate after every board that is passed to the backpropogation function which means that when τ is positive α will increase after every board and decrease if τ is negative. After one whole game has been passed through the backpropogation function we reset α to the initial value. The following formula is used for this process:

$$\alpha \leftarrow \alpha(1 + \tau)$$

The weights of the Neural Network will be updated using the following algorithm:

- Compute the output of the Neural Network for a given position using the same algorithm as before. This also gives us the *invalue* of every hidden node and the output node as well as the *value* of all the hidden nodes.

- Compute the error value and the Δ value of the output node with *target* being the result of the game that was played. We use *target* in case of a win and $-$*target* in case of a loss. We let *output* be the value the Neural Network outputs for the given position, we let $g'(x)$ denote the derivative of the activation function used, (4.2) is used in the case of ReLU and (4.3) is used in the case of Leaky ReLU. We then let

$$error = target - output$$

$$\Delta = error \times g'(invalue\ of\ output\ node)$$

$$g'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ undefined & \text{if } x = 0 \end{cases} \tag{4.2}$$

$$g'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ undefined & \text{if } x = 0 \end{cases} \tag{4.3}$$

- Next the Δ_j of every hidden node j will be calculated using the Δ values of the nodes in the previous layer and the weight $W_{j,\,i}$ connecting node i from the previous layer with node j:

$$\Delta_j = g'(invalue_j) \times \sum_i (W_{j,\,i} \times \Delta_i)$$

- After this the weights $W_{j,i}$ can be updated according to the following formula with α being the learning rate:

$$W_{j,\,i} \longleftarrow W_{j,\,i} + \alpha \times value_j \times \Delta_i$$

After being updated the Neural Network should be slightly better at playing Clobber than before. How good the Neural Network is able to learn Clobber depends on quite a few different variables that all need to be tuned. Some of these include how many hidden layers there are, the number of hidden nodes every hidden layer has, and what the value of α is. Some of the values need to be tuned for different sized boards since a 2×2 board can be learned very easily but a 10×10 board cannot.

Chapter 5

Evaluation

In this chapter we will let the agents play against each other to see which one has the best odds of winning. We will have a large focus on the tuning of the Neural Network, since there are many variables that determine how well it is able to learn the concepts of the game. We examine board sizes from 4×4 up to 10×10, and we will only look at chequered board initialization.

The experiments were run using Bash on Ubuntu on Windows using gcc version 5.4.0. Two different computers were used to run experiments, one running an Intel i7-3820 and the other running an Intel i5-7300HQ, which have different single- and multithreaded performance. Multiple experiments were run at the same time on both computers, which increased the amount of time it took to complete each single experiment. This means that comparing execution times of the different experiments would result in unfair comparisons and also means that playing for a set amount of time would not always result in the same number of games being played.

5.1 Random and Pick First

The first two agents that will play against each other will be Random and Pick First. Both will play against themselves and against each other, both as the starting player and as the other player. To approximate the win rate for every combination 100,000 games will be played for each of them. In the case of the Pick First agent playing against itself all randomness is removed from play and this should result in either the first player winning all games or losing all games. The results are shown in Table 5.1. It took only a few minutes to run all games.

Players	Board size	Black wins
Random (black) vs Random (white)	4×4	51,367
	4×5	56,330
	5×5	54,522
	6×6	50,578
	7×7	52,149
	8×8	50,435
	9×9	51,021
	10×10	50,424
Pick First (black) vs Random (white)	4×4	34,261
	4×5	50,797
	5×5	47,136
	6×6	33,656
	7×7	32,842
	8×8	27,911
	9×9	26,960
	10×10	24,276
Random (black) vs Pick First (white)	4×4	47,431
	4×5	61,634
	5×5	68,520
	6×6	66,460
	7×7	71,389
	8×8	72,019
	9×9	75,398
	10×10	75,901
Pick First (black) vs Pick First (white)	4×4	0
	4×5	100,000
	5×5	0
	6×6	0
	7×7	100,000
	8×8	0
	9×9	100,000
	10×10	100,000

Table 5.1: Random and Pick First playing on different board sizes.

As can be seen the Random player has a slight edge if it is the starting player against another random player and is most pronounced on 4×5 and 5×5 board sizes. Pick First in most cases is a fair bit weaker than a Random player. When Pick First has the first move it only is able to compete on 5×4 and 5×5 board sizes,

when it is not first it is only able to win more than half the games on 4×4 boards. The results for Pick First playing against itself are in line with the prediction we made.

5.2 Monte Carlo

The second agent we will look at will be the Monte Carlo agent. We will let it play against Random and Pick First with different values for the number of playouts and on different board sizes, the number of games played has been reduced to only 1000 due to Monte Carlo being extremely slow as the board size and number of playouts is increased, only playing a few games per second in the worst case. The results are shown in Table 5.2 and Table 5.3.

Board size	playouts	Monte Carlo wins
4×4	5	915
	10	957
	20	969
	50	987
4×5	5	916
	10	948
	20	958
	50	979
6×6	5	904
	10	935
	20	947
	50	967
8×8	5	878
	10	914
	20	944
	50	969

Table 5.2: Monte Carlo and Random on different board sizes and different number of playouts.

Board size	playouts	Monte Carlo wins
4×4	5	972
	10	973
	20	992
	50	999
4×5	5	979
	10	984
	20	987
	50	988
6×6	5	965
	10	987
	20	987
	50	994
8×8	5	971
	10	981
	20	993
	50	997

Table 5.3: Monte Carlo and Pick First on different board sizes and different number of playouts.

These results show that even when the number of playouts is fairly low the Monte Carlo player has a good win rate against Random and Pick First. Using a higher number of playouts raised the win rate under all circumstances.

5.3 Neural Network

The last agent we will be looking at is the Neural Network agent. This agent must have its parameters tuned for every different board size that it will play on. The list of parameters is as follows:

- α, the rate of learning.
- *target*; the value of a win, $-target$ in the case of a loss, by having different values of *target* we can reward or punish the Neural Network more.
- number of hidden nodes.
- number of hidden layers.
- τ; the temporal learning rate or by how much later moves are weighed more heavily.

We will start by having the Neural Network play against a Random player on a 4×4 board. For all experiments from now on we will let the Neural Network learn for 1,000,000 games, after which we stop training the

network and let it play against the same player for another 100,000 games to determine its win rate, unless we note otherwise. We will start off with only one hidden layer with twenty hidden nodes plus one bias node and we will keep the temporal factor at 0.0. The weights of the network will be initialized randomly between 0 and 1 except for the weights of the bias nodes which are all set to 0.1.

Learning rate \ Target	1.0	10.0	50.0	100.0	150.0	200.0
0.0000002	35,179	70,634	76,454	79,639	84,096	76,817
0.000002	36,200	81,847	85,506	91,645	87,609	84,445
0.00002	34,294	95,343	94,842	97,521	97,955	90,521
0.0002	93,288	97,101	34,170	33,961	34,111	33,868

Table 5.4: Neural Network vs. Random on a 4×4 board, win rate for $100,000$ games.

The results in Table 5.4 show a wide range of play, with a low of 33,868 and a high of 97,584. The observation that the win rate goes to about 34% is the result of the dying ReLU problem [Kar18], where a unit in the network only outputs 0 for all possible inputs. In the scenario that a large part of the network dies, the output of the network will be 0 for almost all inputs, which means all moves have the same value and the first move in the move list will be picked, since this is the default move. This means that the Neural Network will play the same moves as the Pick First algorithm would. This issue is often the result of the learning rate being set too high, but this was not the case when the target was 1.0, where the highest learning rate is the only one that did not die or got close to dying. The best learning rate for a 4×4 board is 0.00002 with a target of 150.0, with a few other combinations of parameters close behind it. The Neural Network got close to winning all games ($\sim$ 98%) but still got beaten by Random. To improve the result we let the Neural Network train with the parameters we found for $10,000,000$ games to see if it is able to learn even better play after more games. This resulted in a win rate of around $\sim$ 99% after $2,000,000$ games, which it stayed around for the remainder of training. This means that the network was not able to create a model that was good enough to achieve flawless play, since it has been shown that on a 4×4 board the first player to move is winning [GU16].

Next we increase the board size to 4×5 and to 6×6 and ran the experiments with similar parameters as before, The results are shown in Table 5.5 and Table 5.6.

Learning rate \ Target	1.0	10.0	50.0	60.0	100.0	150.0	200.0	350.0
0.0000002	52,156	66,007	78,270	77,049	73,242	75,033	75,663	73,110
0.000002	52,197	69,994	85,790	78,864	82,016	78,520	84,801	81,105
0.00002	60,073	81,864	90,381	85,992	84,845	82,945	90,344	51,190
0.0002	83,201	87,297	50,975	51,311	51,022	51,092	51,146	51,312

Table 5.5: Neural Network vs. Random on a 4×5 board, win rate for $100,000$ games.

Learning rate \ Target	1.0	10.0	50.0	100.0	150.0	200.0	500.0	1000.0
0.00000002	41,318	51,312	36,200	33,710	33,996	33,894	62,419	64,526
0.0000002	34,477	34,813	59,872	64,528	60,147	64,543	33,849	33,659
0.000002	33,546	33,317	61,020	33,848	33,834	33,512	33,396	33,390
0.00002	33,614	33,800	33,642	33,436	33,557	33,539	33,770	33,769

Table 5.6: Neural Network vs. Random on a 6×6 board, win rate for $100,000$ games.

These results show that it is more difficult for our Neural Network to learn how to play against a Random player. We also see that on larger boards our network dies very often, especially on 6×6 boards where a majority of the chosen parameters resulted in the network dying. On 4×5 boards it is a bit more difficult to determine if a network is dead due to the win rate of a dead network against random being $\sim 51\%$. We can also already see that different parameters perform differently on different board sizes.

To improve the performance of our network, we will increase the number of hidden nodes and hidden layers in our network so that a more complex model can be created.

First we will increase the number of hidden layers to 2 and continue play on 4×5 and 6×6 boards. The results are shown in Table 5.7 and Table 5.8.

Learning rate \ Target	1.0	10.0	50.0	100.0	150.0	200.0	500.0	1000.0
2×10^{-9}	50,861	51,790	53,247	51,316	73,788	68,660	77,000	83,106
2×10^{-8}	51,246	49,737	62,065	51,190	78,512	78,984	83,786	85,676
2×10^{-7}	51,240	60,879	80,194	79,331	85,844	85,831	51,257	51,103
2×10^{-6}	51,019	51,033	82,209	81,741	55,850	51,019	51,024	51,037
2×10^{-5}	51,189	51,190	51,189	80,475	51,198	51,155	51,900	51,190
2×10^{-4}	51,316	51,316	51,189	70,130	51,494	50,618	51,314	51,316

Table 5.7: Neural Network vs. Random on a 4×5 board with 2 hidden layers, win rate for $100,000$ games.

Learning rate \ Target	1.0	10.0	50.0	100.0	150.0	200.0	500.0	1000.0
2×10^{-12}	49,448	49,344	48,065	47,750	45,542	45,644	44,907	47,790
2×10^{-11}	46,764	46,094	49,646	49,754	48,108	44,626	48,514	50,693
2×10^{-10}	48,874	42,381	33,849	52,521	50,549	50,592	51,518	33,610
2×10^{-9}	34,366	33,459	33,692	33,815	33,544	34,070	60,623	64,218
2×10^{-8}	34,268	34,204	33,980	51,264	33,293	33,754	33,773	33,794
2×10^{-7}	33,703	33,449	33,621	33,608	33,471	33,375	33,815	33,813

Table 5.8: Neural Network vs. Random on a 6×6 board with 2 hidden layers, win rate for $100,000$ games.

These results show no improvement for 6×6 and 4×5 boards over the first experiment with one hidden layer. It should be noted that the learning rate had to be much lower to prevent the network from dying. This could mean that the network had to train longer than $1,000,000$ games to achieve a better result, so we picked the best parameters for a 6×6 board, learning rate 2×10^{-9} and target 1000.0, and let it train for $20,000,000$ games. This resulted in a win rate of $\sim 73\%$ against a Random opponent.

Instead of increasing the number of hidden layers we now change the number of hidden nodes; for this we will stick with a 4×5 board and try different numbers of hidden nodes. The results are shown in Table 5.9, Table 5.10, Table 5.11, Table 5.12 and Table 5.13.

Learning rate \ Target	1.0	10.0	50.0	100.0	150.0	200.0	500.0	1000.0
2×10^{-9}	52,539	56,471	65,414	65,474	63,228	65,944	69,396	68,386
2×10^{-8}	50,286	61,779	69,548	67,965	73,575	79,349	69,417	74,814
2×10^{-7}	50,859	75,779	72,566	82,750	81,276	80,686	73,284	81,005
2×10^{-6}	53,095	77,880	82,921	86,702	88,850	87,191	88,070	87,868
2×10^{-5}	55,980	85,748	89,488	91,498	81,191	51,187	51,092	51,039
2×10^{-4}	76,697	91,806	50,877	51,336	51,045	51,314	50,988	46,323

Table 5.9: Neural Network vs. Random on a 4×5 board with 30 hidden nodes, win rate for $100,000$ games.

Learning rate \ Target	1.0	10.0	50.0	100.0	150.0	200.0	500.0	1000.0
2×10^{-9}	52,000	53,762	51,003	55,589	62,127	65,495	68,881	76,250
2×10^{-8}	52,525	53,550	70,752	70,659	72,386	72,024	70,420	72,458
2×10^{-7}	51,496	74,034	72,299	75,640	69,567	72,766	77,587	86,360
2×10^{-6}	51,018	77,183	84,518	85,934	89,056	89,600	89,418	93,512
2×10^{-5}	64,083	87,134	92,096	92,929	51,187	51,189	51,188	51,187
2×10^{-4}	79,706	93,674	51,315	51,315	51,315	51,435	51,336	51,335

Table 5.10: Neural Network vs. Random on a 4×5 board with 40 hidden nodes, win rate for $100,000$ games.

Learning rate \ Target	1.0	10.0	50.0	100.0	150.0	200.0	500.0	1000.0
2×10^{-9}	54,356	58,166	52,459	62,415	63,126	66,495	65,060	73,834
2×10^{-8}	52,174	63,045	71,291	74,431	75,157	79,061	78,647	77,666
2×10^{-7}	50,933	70,617	82,465	70,402	81,857	78,917	78,190	82,183
2×10^{-6}	51,039	79,490	87,673	91,086	91,408	88,717	92,086	87,677
2×10^{-5}	51,329	90,120	93,153	93,048	92,711	51,103	51,512	51,173
2×10^{-4}	50,880	93,853	50,778	51,195	51,145	51,130	51,235	51,202

Table 5.11: Neural Network vs. Random on a 4×5 board with 50 hidden nodes, win rate for $100,000$ games.

Learning rate \ Target	1.0	10.0	50.0	100.0	150.0	200.0	500.0	1000.0
2×10^{-9}	51,535	51,762	55,565	63,346	60,305	68,028	64,353	74,395
2×10^{-8}	51,809	56,991	70,946	69,141	72,523	72,060	73,789	68,217
2×10^{-7}	51,270	73,359	73,170	79,685	83,323	80,375	84,059	82,357
2×10^{-6}	51,130	79,394	90,348	90,105	92,349	91,980	92,285	91,376
2×10^{-5}	50,916	92,391	90,357	94,443	51,386	51,364	51,314	50,829
2×10^{-4}	50,654	94,772	51,059	51,006	51,242	51,226	51,383	50,755

Table 5.12: Neural Network vs. Random on a 4×5 board with 60 hidden nodes, win rate for $100,000$ games.

Learning rate \ Target	1.0	10.0	50.0	100.0	150.0	200.0	500.0	1000.0
2×10^{-9}	56,759	56,480	59,620	62,892	74,012	68,493	70,660	75,766
2×10^{-8}	53,657	50,602	71,404	72,571	71,696	73,969	68,197	87,000
2×10^{-7}	50,515	76,373	82,640	82,002	79,576	84,434	83,420	87,000
2×10^{-6}	51,052	79,895	89,286	86,750	92,268	92,250	96,091	94,633
2×10^{-5}	52,326	93,859	95,182	94,922	51,119	51,225	51,048	50,979
2×10^{-4}	58,901	51,096	51,290	51,129	51,271	51,131	51,073	51,026

Table 5.13: Neural Network vs. Random on a 4×5 board with 100 hidden nodes, win rate for $100,000$ games.

These results show that having 100 hidden nodes for a 4×5 board has the best results. As opposed to the Neural Network with two hidden layers we also observe that increasing the number of hidden nodes does not result in the network dying more often. After this we increased the number of hidden nodes to 150, 250, 350, 500 and 750 with a learning rate of 2×10^{-5} and a target of 50.0 this resulted in win rates of $\sim 96\%$, $\sim 97\%$, $\sim 99\%$, $\sim 98\%$ and $\sim 51\%$ respectively. These results show that increasing the number of hidden nodes does not necessarily result in a better win rate and could also result in the network dying. The win rate of $\sim 99\%$ with 350 hidden nodes does show a significant improvement over a $\sim 90\%$ win rate with only 20 hidden nodes. Another drawback of increasing the number of hidden nodes is that the number of games being played per second is lower.

Now we again move onto the 6×6 boards with an increased number of hidden nodes. The results of this are shown in Table 5.14.

Learning rate \ Target	50.0	100.0	150.0	200.0	500.0	1000.0
2×10^{-9}	46,992	47,774	44,389	45,641	47,909	49,634
2×10^{-8}	34,647	53,042	33,809	62,172	63,860	62,099
2×10^{-7}	33,983	66,530	34,033	62,289	67,702	74,517
2×10^{-6}	67,415	33,509	69,044	74,439	33,526	33,734
2×10^{-5}	33,532	33,471	33,698	33,583	33,612	33,549
2×10^{-4}	33,892	33,674	33,541	33,667	33,665	33,510

Table 5.14: Neural Network vs. Random on a 6×6 board with 100 hidden nodes, win rate for $100,000$ games.

These results show a slight improvement over the previously best win rate on a 6×6 board by the Neural Network but was able to learn to play at this level in only a fraction of the games it took before. This was at the cost of playing less games per second but still resulted in less time spend training. We still see that the network dies often. Using this result we tried different numbers of hidden nodes with 2×10^{-7} as the learning rate and 1000.0 as the target; we used 50, 200, 300 and 400 hidden nodes and let them play for $5,000,000$ games which resulted in win rates of $\sim 78\%$, $\sim 89\%$, $\sim 33\%$ and $\sim 83\%$ respectively. Increasing the number of hidden nodes beyond this would slow down the network by a very large amount taking more than a minute to play $10,000$ games, it should already be noted that playing $5,000,000$ games with 200 hidden nodes took less than 4 hours, while having 400 hidden nodes took around 12 hours. Some weird behavior was observed for 50 and 400 hidden nodes where the win rate would fluctuate downwards at times and then recover towards the better win rate; this behaviour did stop after enough games were played and the win rate slowly increased over time.

5.3.1 Temporal Rate

The results so far show that learning Clobber using a Neural Network on smaller boards results in very high win rates but on 6×6 boards only results in a win rate of $\sim 89\%$ against a random opponent after $5,000,000$

games played. Since for this experiment we will only train for $1,000,000$ games it should be noted that the network that achieved $\sim 89\%$ win rate only had a $\sim 77\%$ win rate after $1,000,000$ games played. The win rate on a 6×6 board could be improved by weighing moves later in the game as more important; this would result in slightly more random play for the first few moves of the game but would result in the network winning more games overall. We do need to be careful about raising the learning rate too high, otherwise the network could die very fast. We will use values that were shown before to already provide good results. The results are shown in Table 5.15.

Parameters	Temporal Rate	Neural Network wins
hidden nodes: 100 target: 200.0 Learning Rate: 2×10^{-6}	0.1	33,154
	0.01	33,477
	0.001	65,884
	0.0001	71,005
hidden nodes: 100 target: 1000.0 Learning Rate: 2×10^{-7}	0.1	66,204
	0.01	32,811
	0.001	69,374
	0.0001	63,321
hidden nodes: 200 target: 1000.0 Learning Rate: 2×10^{-7}	0.1	33,505
	0.01	33,746
	0.001	75,464
	0.0001	72,779
hidden nodes: 200 target: 1000.0 Learning Rate: 2×10^{-8}	0.1	62,609
	0.01	63,209
	0.001	64,263
	0.0001	66,771

Table 5.15: Neural Network vs Random using different Temporal Learning rates, win rate for $100,000$ games on a 6×6 board.

Comparing these values to the win rate that was obtained without using a temporal factor we see that using a temporal factor lowers the win rate the network is able to achieve at the cost of being slightly slower to train.

5.3.2 Leaky ReLU

The dying ReLU problem is a problem that can be seen in the results so far in many cases. We tried to combat this by lowering the learning rate and changing the target. Another approach to combat this problem is changing the activation function to Leaky ReLU. we start with 4×5 and 6×6 boards with 100 hidden nodes and one hidden layer. The results are shown in Table 5.16.

Learning rate \ Target	1.0	10.0	50.0	100.0	150.0	200.0	500.0	1000.0
2×10^{-9}	56,171	58,413	60,585	64,290	64,518	64,176	64,339	70,395
2×10^{-8}	56,789	61,928	70,331	66,931	70,245	67,858	73,635	76,002
2×10^{-7}	59,084	63,246	76,351	74,063	74,953	75,798	70,311	71,778
2×10^{-6}	59,484	64,448	71,668	74,620	81,820	74,090	72,565	76,593
2×10^{-5}	61,049	71,433	70,383	74,374	75,656	79,461	70,749	55,308
2×10^{-4}	62,476	73,569	64,703	55,291	55,328	55,590	55,246	55,225

Table 5.16: Neural Network vs. Random on a 6×6 board with 100 hidden nodes, using Leaky ReLU, win rate for $100,000$ games.

The first thing that should be noted is that using Leaky ReLU takes slightly longer when compared to using ReLU, in the order of magnitude of 3%. The results shown are actually the best we have been able to achieve so far on a 6×6 board, none of the parameters resulted in the network dying. We got the best win rate yet after $1,000,000$ games played on a 6×6 board of $\sim 82\%$.

5.3.3 Neural Network vs. Monte Carlo

Training the Neural Network against the Monte Carlo player would take an incredibly long time, so instead we will be training against a random player for $1,000,000$ games with parameters that were shown to be very good against the random player. After this training phase we will stop training and let the resulting Neural Network play against the Monte Carlo player with different number of playouts.

Parameters	Number of playouts	Neural Network wins
hidden nodes: 100 target: 150.0 Learning Rate: 2×10^{-6} Leaky ReLU	5	502
	10	543
	20	560
	50	509
hidden nodes: 100 target: 1000.0 Learning Rate: 2×10^{-6} Leaky ReLU	5	556
	10	498
	20	543
	50	509
hidden nodes: 200 target: 1000.0 Learning Rate: 2×10^{-7} ReLU	5	528
	10	599
	20	509
	50	537

Table 5.17: Neural Network vs Monte Carlo on 6×6 board size, win rate for $1,000$ games.

We see in Table 5.17 that our Neural Network has an edge over Monte Carlo after training for only $1,000,000$ games of playing against Random. We should note that our Neural Network plays much faster than the Monte Carlo player. The number of play outs that the Monte Carlo player does did not seem to matter too much against the Neural Network.

Chapter 6

Conclusions

We have shown that a Neural Network based approach for playing Clobber was able to beat a Monte Carlo player slightly more than 50% of the time on 6×6 boards, and was able to win most games against a Random player. We observed that increasing the number of hidden layers resulted in the Neural Network dying much more often while increasing the number of hidden nodes and having only one hidden layer did not cause the Neural Network to die much more often.

Using Temporal Rate learning with Rectified Linear Units did not result in a better win rate for the network on 6×6 boards, but did cause the network to die in some cases because of the higher learning rate. A much more successful variant was changing the activation function from ReLU to Leaky ReLU, which prevented the network from dying and gave us better results against both Random and comparable results against Monte Carlo.

6.1 Future Research

There is still quite some work for future researchers to do. Clobber has not seen a great amount of research yet. The Neural Network for example can always be improved by using more advanced techniques to make it learn better, as well as scaling the Neural Network up. The Neural Network could also be trained against the decision making of expert human players (if they exist) like was done for Go.

From the last results we can see that Leaky ReLU was a good improvement over ReLU but we only looked at the problem with a very narrow scope. This should be expanded to include different parameters like different numbers of hidden nodes with multiple hidden layers, and could also be combined with a temporal learning rate on larger boards.

Finding the correct parameters was one of the hardest parts of this research especially when the board size was increased. This could be solved by using natural computing techniques to find and optimize the different

parameters of the Neural Network. This could also allow the Neural Network to play at a high level on even larger boards.

Another approach would be to use the AlphaZero algorithm, which is a general reinforcement learning algorithm and was used to achieve superhuman play in chess, shogi and Go after only 24 hours of learning, see [SHS+17].

Altogether, there is still much research that can be done related to Clobber and Neural Networks.

Bibliography

[AGNW05] Michael Albert, J.P. Grossman, Richard Nowakowski, and David Wolfe. An introduction to Clobber. *Integers*, 5, 2005.

[Alt17] I. Althöfer. Computer olympiad 2011 — Clobber. *http://www.althofer.de/clobber/clobber-2011-pan-10x10.jpg*, [accessed 18/12/2017].

[BPW+12] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo Tree Search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:1:1–43, 03 2012.

[DDF02] Erik D. Demaine, Martin L. Demaine, and Rudolf Fleischer. Solitaire Clobber. *CoRR*, cs.DM/0204017, 2002.

[Dic18] Dicksonlaw583. Mcts (english) - updated 2017-11-19 - monte carlo tree search - wikipedia. *https://en.wikipedia.org/wiki/Monte_Carlo_tree_search*, [accessed 30/01/2018].

[Eto17] Teddy Etoeharnowo. Neural Networks for Clobber. *Bachelor thesis, Leiden University*, 2017.

[Gri17] R. Grimbergen. Reijer grimbergen's research pages. *https://www2.teu.ac.jp/gamelab/RESEARCH-/ResearchPix/clobber.png*, [accessed 19/12/2017].

[GU16] Janis Griebel and Jos Uiterwijk. Combining Combinatorial Game Theory with an α-β solver for Clobber. 2016.

[JKRL09] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *Proceedings of the 2009 IEEE 12th International Conference on Computer Vision*, pages 2146–2153, 2009.

[Kar18] Andrej Karpathy. Cs231n convolutional Neural Networks for Visual Recognition. *http://cs231n.github.io/neural-networks-1/#actfun*, [accessed 21/01/2018].

[MHN13] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve Neural Network acoustic models. In *Proceedings of the 30th International Conference on Machine Learning*, volume 30, 2013.

[Mor17] T. Morris. Next price predictor using Neural Network indicator for MetaTrader 4. *https://www.forexmt4indicators.com/wp-content/uploads/2014/10/NN1__1.gif*, [accessed 19/12/2017].

[NE10] Hinton Nair, Vinod and Geoffrey E. Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, 2010.

[SB18] Richard Sutton and Andrew Barto. Reinforcement learning: An introduction. *The MIT Press*, 2018.

[Sha18] Sagar Sharma. Activation functions: Neural Networks — towards data science. *https://cdn-images-1.medium.com/max/1600/1*A_BznoCjUgOXtPCJKnKLqA.jpeg*, [accessed 30/01/2018].

[SHS+17] David Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *ArXiv e-prints 1712.01815*, 2017.

[Sie13] A.N. Siegel. Combinatorial game theory. *AMS*, 2013.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550:page 354, 2017.