Technische Universität München
Faculty of Mathematics: M15

A Machine Learning playing Go

Utilizing neural networks to generate an artificially intelligent Go program without tree search

Project Report by
Benedikt Mairhörmann,
Stefan Peidli,
Faruk Toy,
Patrick Wilson

Supervisor:	Prof. Dr. Massimo Fornasier
Project Lead:	Dr. Ricardo Acevedo Cabra
Advisor:	M.Sc. Bernhard Werner
Submission Date:	February 14, 2018

Contents

1 Project Goal and Organization

The initial goal of this project was to develop a Go playing computer program for the small 9×9 Go board using supervised learning techniques.

Here the small board was chosen because the computational cost and the amount of training data needed to develop a reasonably good Go engine was estimated too high for the 19×19 board. Furthermore, the challenge was to train only an artificial neural network without using tree search techniques, because using a clever tree search technique on the small board was likely to suffice for a good Go engine. The underlying question behind this goal was: how *good* can a neural network based Go engine become at playing a game of Go on the small board?

Also, the objective was not to build an engine similar to DeepMind's *AlphaGo*. This research task encouraged to be creative in the design of the neural network, because one can not rely on the power of the – for almost all Go engines – crucial role of tree search. The initial task was to see, how good the decisions of a Go engine could become after training a neural network with training data. After achieving this initial goal, we tried to enhance the performance of our engine in different ways, see section 4.

For the organization of the team we chose to use a flat hierarchy, so that every member was able to work in his current field of interest. We were not concerned, that such an organization structure could lead to leaving unpleasant work undone. In fact we were excited to see fast progress. In the weekly group meetings with our mentor Bernhard Werner we recieved very valuable input which helped to steer the project in a constructive direction. Furthermore, we organized weekly group meetings and subgroup meetings by ourselves. For this project it turned out to be most effective to work together in pairs, permuting through all combinations depending on the task. A special thanks goes to Dr. Ricardo Acevedo Cabra for supporting us in technical questions for instance in topics concerning the *LRZ*.

2 The Game of Go

"...the rules of go are so elegant, organic, and rigorously logical that if intelligent life forms exist elsewhere in the universe, they almost certainly play Go."
– Edward Lasker, chess grandmaster

"Gentlemen should not waste their time on trivial games – they should study Go."
– Confucius, The Analects, ca. 500 B.C.E.

In this section we will give a brief overview of the history of Go and Computer Go as well as outlining the basic rules of the game Go, inspired by [AGA], [BGA] and [Sen].

2.1 Go from East to West

Go is one of the oldest board games in the world and the oldest game still played in its original form. It is believed that it originated in China 2,500 - 4,000 years ago, but the game's true origins are unknown. Today we are only left with various myths about its creation: for example that the legendary emperor Yao invented Go to discipline his son, Dan Zhu.

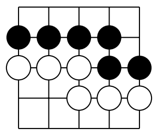
The most important Go playing countries in the Far East are Japan, China, Korea and Taiwan, all of which maintain communities of professional players. Major tournaments in these countries attract sponsorship from large companies and have a huge following comparable to big sport events in Europe and America. There are approximately 50 million Go players in the Far East and many people who do not play still follow the game with keen interest.

Go was not played in Europe until 1880 when the German Oskar Korschelt wrote a book about the game. However, the game was slow to spread and it was not until 1957 that the first regular European Championship was held. Nowadays in European countries, the standard of play is significantly below that of the top players in the Far East, but the gap is steadily closing as more of the top European players spend time studying the game in Japan, Korea and China.

2.2 Basic Rules of Go

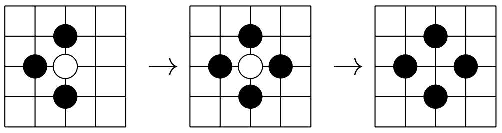
Go is a two player board game. It begins with an empty board that consists of n vertical and n horizontal lines that intersect in $n \times n$ intersection points for $n \in \mathbb{N}$. A normal game is played on a 19×19 board, but for beginners and quick games 9×9 or 13×13 are common choices. Each player has an effectively infinite supply of either black or white stones and the player using the black stones makes the first move. The players take turns, on each turn deciding to place one of their stones on a vacant intersection point on the board or passing.

The main objective of the game is to gain more *area* than the opponent by surrounding intersection points with your stones. These surrounded points are called *territory* and the territory including the border stones is called *area*. The player with the most area wins. In Chinese rules, the area is counted towards your final score. In Japanese rules territory is counted towards your final score. In the context of this paper we will use Chinese rules.



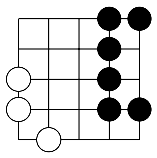
In this game white wins by: $13 - 12 = 1$ points.

It is also possible to *capture* the opponent's stones which results in these stones being taken from the board. Stones are captured, if they do not have any *liberties* left: a stone's liberties are the vacant intersection points connected to a stone by horizontal or vertical lines.



The white stone is captured.

Stones of the same colour that are connected by lines share their liberties. If all liberties of a stone or a line-connected group of stones of the same colour are occupied by the opposing colour, the stones are removed from the board and the player loses area. These line-connected groups are often called *chains* to differentiate them from more general groups. A *group* can only be defined informally as "a functional unit on the board, occupying and influencing a certain area"[1].



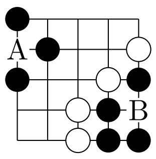
A white and a black group. The black group is a chain.

Each turn the player can choose to pass, instead of placing a stone, if there is no sensible move. The game ends, when both players pass consecutively. At the end of the game,

[1] https://senseis.xmp.net/?Group

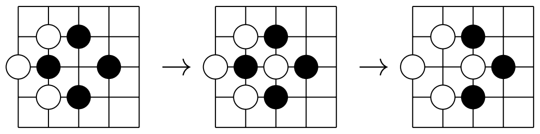
the player with the most area wins[2].

There are slight restrictions on playing stones onto vacant intersections. A stone can not be placed in such a way that it or the group it belongs to has no liberties. This is known as the suicide rule. The only exception of the suicide rule is when by such a move an enemy stone is captured.



It is illegal for white to play in the marked intersection A. White could play in B and capture the four black stones.

This exception however is restricted by the rule of Ko, which does not allow repeatedly recapturing a single stone back and forth.



White captures black in a Ko situation. Now black can not immediately capture white.

The player using the black stones begins and therefore has a slight advantage, so the white player gets bonus points at the beginning of the game called *komi* which are mostly set at +7.5 points. Here the .5 is given especially at tournaments to decide a draw for white. If players have different strength, they can equal out the difference by giving the weaker player *handicap stones*. In Chinese rules these are distributed on the board before the game wherever the weak player desires. In Japanese rules the handicap stones have fixed positions.

There is an official ranking system of the strengths of players that goes for amateurs from 7th dan (highest) over 1st dan to 1st Kyu to 30th Kyu (lowest) and for professional players from 9th dan professional (highest) to 1st dan professional[3].

[2]For the exact Chinese area counting rules, see https://senseis.xmp.net/?ChineseCounting
[3]For a more elaborate explanation of the rules, see https://senseis.xmp.net/?ChineseRules

2.3 Computer Go

"Go on a computer? – In order to programme a computer to play a reasonable game of Go, rather than merely a legal game – it is necessary to formalise the principles of good strategy, or to design a learning programme. The principles are more qualitative and mysterious than in chess, and depend more on judgment. So I think it will be even more difficult to programme a computer to play a reasonable game of Go than of chess."

– I. J. Good, Mathematician, 1965[Goo]

One could say, that the game Go is more dependent on intuition than on calculus and computing power. But is it possible to teach a computer human intuition?

History of Computer Go

After the first Go computer program had been written in 1968, the field of computer Go progressed only slowly. Up until the early 2000s these programs were not able to beat Go players even with a large amount of handicap stones. Only in 2008 the first Go program (*MoGo*), using *Monte Carlo tree search* (MCTS), was able to beat a professional Go player (8th dan) with a handicap of 9 stones. Up until 2015 no Go program was able to beat a professional with less than 4 handicap stones. Then in 2016 – to the surprise of many experts in the fields of Go and AI – *AlphaGo* defeated the strongest Go player at the time, Lee Sedol 9-dan, using two neural networks to support the MCTS.

Comparison to Chess

The development of Go playing programs differs from that of Chess playing programs. Already in 1997 *Deep Blue*, developed by IBM, was the first computer to beat a reigning world champion in a chess match under regular time conditions. No machine learning techniques were used, but a clever tree search with strong computational power was sufficient.[Dee]

In comparison with chess, the Go rules are simple. But as simple as they are, Go is a game of great complexity. The search space in Go is enormous: it is more than 10^{100} times larger than the search space of chess (a number greater than the number of atoms in the universe)[TF16]. Go has the highest state space complexity of any game which is why Go is seen as the hardest information complete game for a computer program to master. Traditional methods, such as constructing a search tree over all possible sequences of moves, become quickly intractible even with the immense computing power of today.[4]

So to beat Go it appears to be inevitable to create more sophisticated heuristics to guide the tree search. It seems necessary to create artificial intuition for the choice of moves that should be evaluated and the value that is assigned to board positions.

The Value of a Board Position in Go

Determining the value of a board position in Go is not easy. Many aspects of the game are hard to measure quantitavely. It is often difficult to evaluate whether a group is dead or alive

[4]For more insights on Computer Go, see `https://senseis.xmp.net/?ComputerGo`

and it can be ambiguous what the sphere of influence of a group is[5]. Even worse: sometimes it is not even clear which stones are a “functional unit that influence a certain area”, i.e. form a group. Moreover most of the time there are several “best moves” depending on the current strategy, which leads to complex trade-offs. While a move strengthens your group in the best way possible, the very same move could lead to a weak position of another group or to losing stones to the enemy. In Go it is important to evaluate local situations in the context of the entire board. This “think globally act locally” strategy can be seen algorithmically as trying to find a balance between depth-first search and breadth-first search. A local move can have an effect on the global situation and the best local move is maybe on the global context a poor move.

It is a big problem to implement heuristics that capture the correct value of moves and board positions, if humans can not quantitavely measure and decide these themselves. So to find suitable heuristics that can tame the tree search in the vast search space of the game, researchers applied techniques of artificial intelligence to make the program find good heuristics itself.

State of the Art – AlphaGo

The company *DeepMind* started developing the Go playing program *AlphaGo* in 2014. In the version described in [S+16] the researchers used two neural networks in combination with the Monte Carlo tree search (MCTS) method. The *Policy Net* is a neural network that takes a Go board position and the current player as input and outputs a ranking of the *best* moves for this specific board. Note that *best* here is not well-defined. In the supervised learning framework the *best* moves are the most popular moves in the training data in the hope that these approximate the moves that lead to a win. The *Value Net* is a neural network that also takes a Go board and the current player as input and outputs a winning probability for this player with this board position. The MCTS then iteratively looks at the top moves for one board position given by the Policy Net and evaluates these with the Value Net with the final goal of maximizing the winning probability. To make the Policy Net predict *best* moves given a board the researchers used a database of 30 million board positions to train the network using supervised learning as a first step. The second step of training improved the Policy Net by using reinforcement learning. The Value Net was trained using data produced by games between the Policy Net and itself. Furthermore, AlphaGo makes use of *convolutional neural networks* which have shown great results in visual domains, such as image classification. Similar to finding shapes in pictures, these turn out to be useful to find shapes and forms in Go board positions.

The above outlined version of AlphaGo defeated the number one Go player in the world in 2016. After reaching this milestone the team of DeepMind continuously improved their Go program and newer versions could beat older versions by a margin. In 2017 the newest version named *AlphaGo Zero* [S+17] defeated the champion-defeating version from [S+16] in 100 out of 100 games, surpassing it by far. In contrast to previous versions, AlphaGo Zero was trained solely by self-play using refined techniques of reinforcement learning.

[5]The Chinese rules define life and death this way: “At the end of the game, stones which both players agree could inevitably be captured are dead. Stones that cannot be captured are alive.”, see `https://senseis.xmp.net/?Death`

3 Data and Implementation

3.1 Data Preprocessing

At the beginning of the project we were given 76439 random games from the Dragon Go Server[1]. Go games are usually saved in the text file format *.SGF*[2] (SmartGameFormat). One *SGF*-file stores information about one game between two players. It mainly contains which moves were played, but a lot of other information can also be stored, such as: who won, the names of the players, what handicap and komi was used, etc. We decided to use *Pandas* dataframes to analyze and process the data.

Firstly, we checked the basic information such as board size and the rule set. We came to the conclusion, that all games were played on a 9×9 board using Japanese rules. We found out that most of the games had a komi between -6.5 and 6.5. The most interesting properties to us were number of moves, rank of the players, result as well as the komi and handicap. While analyzing these properties of the game files we encountered some outliers, for example extreme cases of very high komi (up to 120.5) or handicap. Games like this most likely consist of moves that are not good or optimal, because white wins very easily and that can have an effect on the way of playing. Furthermore, these games tended to have a very small amount of moves before one player resigned.

It turned out to be most beneficial to sort the games by the rank of the players. Hence, we created lists of GameIds for the top n best dan games, $n \in \mathbb{N}$. Here game A is better than game B iff the rank of the black player of game A is higher than the rank of the black player of game B. This makes sense, as the white player is usually the higher ranked player. There were 295 games where both playes were of dan rank. These high ranked games all had regular amounts of moves and no extreme komi or handicap. We also created lists of GameIds for the top n games, categorized by the minimum rank of both players. These datasets contained some curiosities at first but we could easily filter these out by simply considering only games with a minimum game length of 10 moves and with a result. A 9×9 Go game usually takes around 40-50 moves, for example the 295 dan games consisted of about 12,600 moves.

3.2 Implementation of the Game Go

As a programming language we chose to use *Python* because of its popularity for Machine Learning tasks. Even though none of us had worked with *Python* before, we were eager to take on the challenge. This turned out to be a great opportunity to elevate our programming knowledge and skills to a new level. Before being able to train a neural network to learn to play Go, we first needed to create a computer program that is able to play legal moves in a

[1]http://www.dragongoserver.net/
[2]https://senseis.xmp.net/?SmartGameFormat

game of Go, a so called *Go engine*. Therefore we firstly created a program, that can output legal move coordinates while receiving move coordinates as an input.

Go Clients

A *Go client* is a computer program that provides a graphical user interface for Go programs. Most Go clients, e.g. *CGoban*[3], are able to read *.SGF*-files and produce a graphical display of a Go board which enables the user to follow the move sequences in that game step by step. Other Go clients such as *GoGui*[4] serve as an interface that communicates with a Go engine. This means one only has to implement the Go engine to be able to obtain a working visualized Go playing program. Another advantage of using clients is that the scoring/counting at the end of the game is also taken care of by these programs.

The Go Text Protocol

A very convenient and the most used way of communication between Go engines and Go clients is by the *Go Text Protocol*[5] (*GTP*) created by Gunnar Farnebäck. This protocol defines specific commands that are used for sending information from the engine to the client and vice versa. It is easy to implement and enables the use of a lot of clients as well as being able to test your Go engine on to a public server such as the popular *Kiseido Go Server* [6]. On these servers human players or other Go engines can play against your Go bot which can be useful to test its strength.

Random Bot

Our first milestone was the implementation of a Go engine that communicates via GTP and can play legal moves. This first prototype selected moves completely random. For our long term planning it was very important, because it left us with just one big task to do, namely replacing the move generating function of this engine with one that suggests good moves. We also managed to compile the engine into an executable program using *Pyinstaller* which made our program portable and attachable to a Go interface such as *GoGui*.

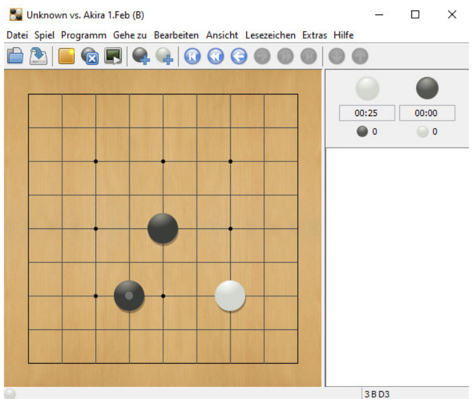

Unknown vs. Akira 1.Feb (B)
Datei Spiel Programm Gehe zu Bearbeiten Ansicht Lesezeichen Extras Hilfe
00:25
00:00
0
0
3 B D3

Figure 3.1: The Graphical Interface of the Go client GoGui with an attached Go engine.

[3]https://www.gokgs.com/download.jsp
[4]https://senseis.xmp.net/?Gogui
[5]http://www.lysator.liu.se/~gunnar/gtp/
[6]https://www.gokgs.com/

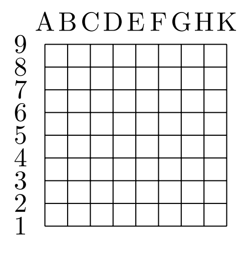
3.3 Implementation of the Neural Network

We decided to implement a *Policy Net*, i.e. a neural network with the learning goal of outputting a ranking of the best moves for one player given a board position. We wanted to find out how *clever* a Go engine based on this neural network can become. Furthermore, we decided to implement this neural network from scratch without using any machine learning specific programming library. We followed the approach of implementing the whole neural network including backpropagation ourselves, to gain a deeper understanding of the mathematical concepts behind neural networks. For mathematical operations we used the *Python* package *NumPy*. This helped us to gain a deeper understanding of the mathematical concepts behind neural networks and it also enabled us to directly implement our creative ideas into the neural network. Finally, we used the package *Pandas* to manipulate the data and the packages *Matplotlib* and *Seaborn* for data visualization.

3.4 Training Data

As we will see more elaborately in section 4 the Policy Net is a function that takes a board position as an input and outputs a move distribution. Our hope was that training this neural network with board/move pairs from our *.SGF*-files will generalize to a function that outputs *good* moves for arbitrary boards. To enable this training we needed to create labeled training data. For this we represented 9×9 boards as 81-dimensional vectors with black $\backsimeq -1$, white $\backsimeq 1$ and vacant intersection point $\backsimeq 0$. A move is represented as a 82-dimensional vector with exactly one 1 where the move is made and zeros elsewhere, the 82nd entry corresponding to passing.

We followed two approaches in creating labeled training data out of the *.SGF*-files. The first approach aggregated a key-value data structure where the key is a board position and the value is a move distribution. For example the distribution corresponding to the empty board consists of all first moves in the analyzed game files, e.g. Figure 3.2 for the 295 dan games.



$$\backsimeq \begin{pmatrix} 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0 \end{pmatrix} \mapsto \begin{pmatrix} 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&2&0&0&0&0\\ 0&0&48&52&60&52&48&0&0\\ 0&0&52&52&92&52&52&0&0\\ 0&2&60&92&920&92&60&2&0\\ 0&0&52&52&92&52&52&0&0\\ 0&0&48&52&60&52&48&0&0\\ 0&0&0&0&2&0&0&0&0\\ 0&0&0&0&0&0&0&0&0 \end{pmatrix}$$

Figure 3.2: The empty board and the corresponding move distribution in the first approach.

The second approach for labeled training data was to simply collect all board/move pairs from the game files. In this case the move of one board/move pair is a distribution with exactly one non-zero entry corresponding to where the move was played on the board.

We realized that every Go game is a representative of an equivalence class of eight games considering four 90° rotations and two reflections of the board. Therefore we applied these rotations to each board position in the training data and were able to artificially enlarge the training data by a factor of 8. In both approaches we added an additional element to the distribution to realize the option of passing instead of playing a move, because this yielded better results than training an empty distribution. We did not find useful mathematical foundations on what style of training data to use, so we tested both approaches.

Finally, we decided to train the network to always "think" from the perspective of the black player. This can be accomplished by inverting the board whenever it is white's turn, let black play instead and then inverting the outcome again. Our motivation for doing this was that the constellation of stones on the board is not evaluated color dependent in general. We figured out that identifiying colour symmetric boards would let our network learn from all the information whereas otherwise the network would build two color seperated ways of playing using only half of the information for each of these ways. For the sake of completeness it is important to say, that a player's color does influence the choice of moves in some cases considering komi and the fact that black starts the game. Nonetheless, the aspect of computational efficiency was more important to us.

3.5 Workflow Management

At the beginning of the project we used *Trello* as a Scrum-like system to manage our planning, short- and longterm goals and active tasks. Once we started the implementation, we used *GitHub*[7] as a version control system. Later we also used *GitHub* instead of *Trello* for our task management. Throughout the project we also used a *Google Drive* Folder to manage and share files. This project documentation is written in LaTeX.

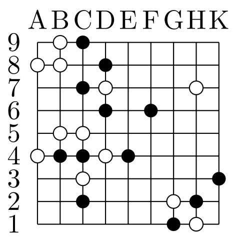
[7]Our GitHub repository for this project can be found at:
https://github.com/stefanpeidli/GoNet

4 Neural Network

In mathematics, a neural network is a combination of large linear functions used to approximate a highly complex function, particularly often for classification problems.
In our case we want to approximate the function

$$V\colon \mathbb{B} \to \Delta^{81}, \qquad \textit{Board} \mapsto \textit{Distribution},$$

where $\mathbb{B} \subseteq \{-1, 0, 1\}^{9^2}$ is the space of possible board configurations that can appear in a game of Go with the naive black $\backsimeq \bullet \backsimeq -1$ and white $\backsimeq \circ \backsimeq 1$ encoding exemplified by



$$\backsimeq \begin{pmatrix} 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \end{pmatrix}$$

The 81-dimensional simplex is denoted by $\Delta^{81} \subset \mathbb{R}^{82}$. Its elements are relative distributions of best moves (including passing, hence 82 entries) to be played on a particular board. We call this network **Policy Net**.

4.1 Introduction to Neural Networks

The fundamental building blocks of neural networks are artificial neurons, which are simplified models for real neurons. Each of them takes some input $x = (x_1, ..., x_n)$ and multiplies it by a weights vector $w = (w_1, ..., w_n)$. The resulting vector is then summed up into

$$s = \left(\sum_{i=1}^{n} x_i w_i \right) + w_b \tag{4.1}$$

The neuron then produces an output $y \in \mathbb{R}$ by applying an activation function $f : \mathbb{R} \to \mathbb{R}$ to the weighted sum:

$$y = f(s) = f(\langle x, w \rangle + w_b) \tag{4.2}$$

In figure 4.1 one can see an artificial neuron compared with a diagram of a real neuron cell. The Dendrites collect signals from other cells, weight them, and if a threshold is surpassed the neuron starts firing electric impulses through the axon towards other cells. The key simplification is modeling the impulse firing by just a real valued number (frequency), allowing for much faster simulation.
The activation function usually is some kind of sigmoid shaped function and models the stimulus response. From a biological point of view, this should be modeled by a quite sharp

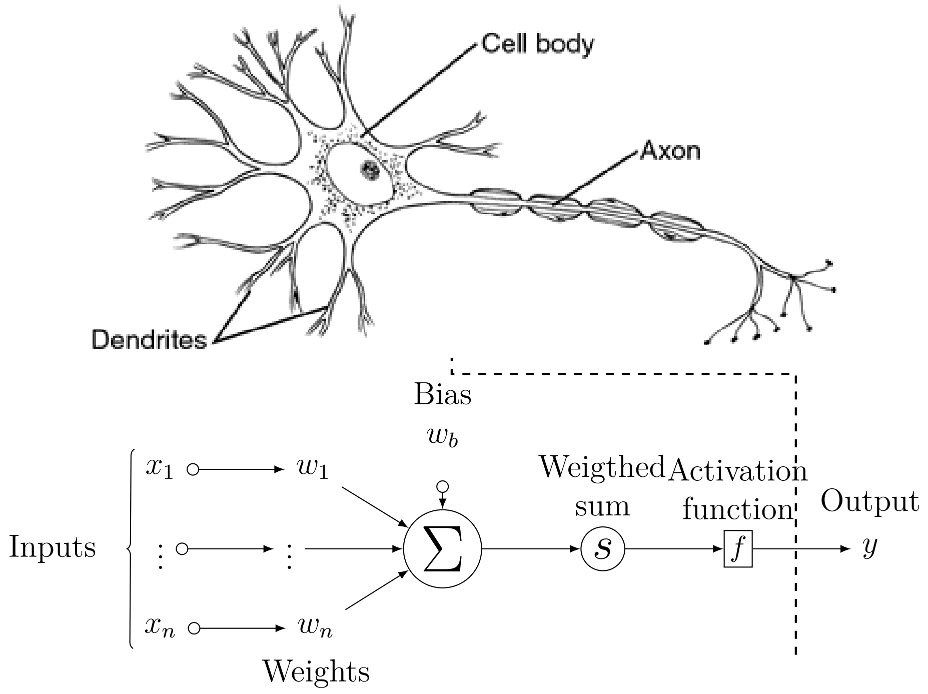


Figure 4.1: Comparison of a real neuron scheme (taken from [Dor90]) with a diagram of an artificial neuron.

signum function translated by a threshold for the required stimulus to hyperpolarization of the membrane potential. Due to the fact that we are dealing with bounded precision and need high smoothness in order to apply certain learning methods to the neuron, sigmoidal functions seem to be a reasonable choice. The bias or offset w_b corresponds to the resting potential of the neuronal membrane.
To make use of the emerging qualities of neural networks a large number of neurons is needed. For this, multiple neurons are collected in a parallel stack called layer. Consequently writing equation (4.1) in matrix form for the whole layer we obtain

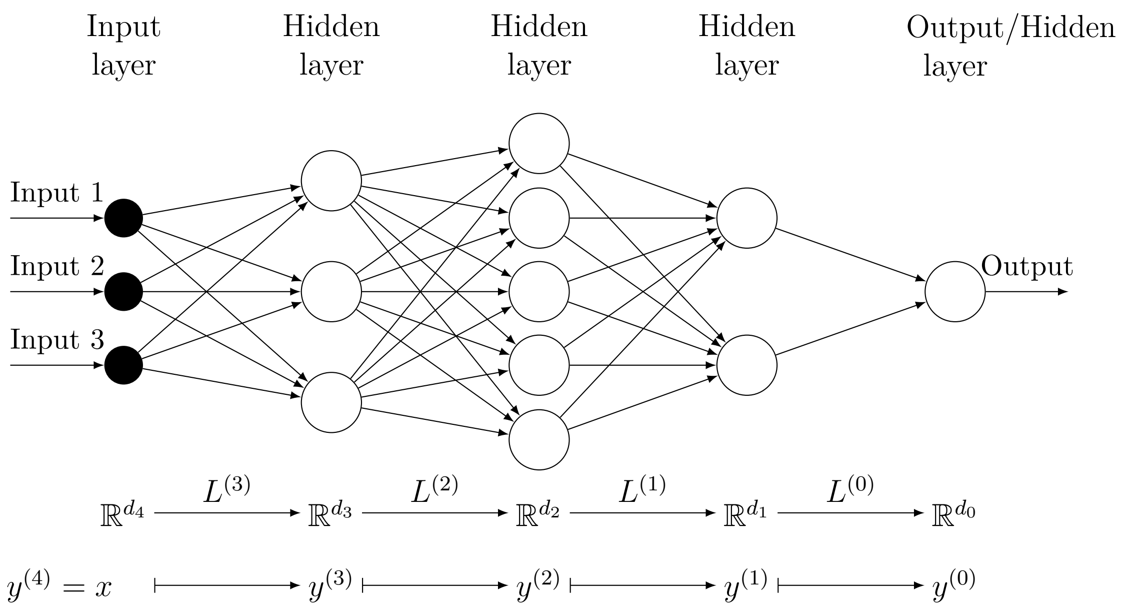
$$s = Wx + W_b = \begin{pmatrix} W & W_b \end{pmatrix} \begin{pmatrix} x \\ 1 \end{pmatrix} =: \hat{W}\hat{x} \tag{4.3}$$

where now $s \in \mathbb{R}^n$ and $\hat{x} \in \mathbb{R}^m$ are vectors and $\hat{W}$ is the weight matrix. The additional entry in the last row of $\hat{x}$ together with W_b (the last column of $\hat{W}$) corresponds to the offset terms w_b which form a vector in the layer formulation. In the same manner (4.2) reads

$$y = F(s) = F(\hat{W}\hat{x}) \tag{4.4}$$

where F has to be understood as component-wise application of the scalar activation function f, i.e. $F(s)_i = f(s_i)$ such that F matches the dimensions of the respective layer.

Apart from parallel stacking neurons into layers, we also chain the layers serially to form a so called Multi-Layer-Feedforward neural network. Such a multilayer neural network consists of layers $L^{(i)}$ with $i = 0, ..., m$, each described by its own weight matrix $\tilde{W}^{(i)} \in \mathbb{R}^{d_{i+1} \times (d_i+1)}$. By



Input layer
Hidden layer
Hidden layer
Hidden layer
Output/Hidden layer
Input 1
Input 2
Input 3
Output
$\mathbb{R}^{d_4} \xrightarrow{L^{(3)}} \mathbb{R}^{d_3} \xrightarrow{L^{(2)}} \mathbb{R}^{d_2} \xrightarrow{L^{(1)}} \mathbb{R}^{d_1} \xrightarrow{L^{(0)}} \mathbb{R}^{d_0}$
$y^{(4)} = x \mapsto y^{(3)} \mapsto y^{(2)} \mapsto y^{(1)} \mapsto y^{(0)}$

Figure 4.2: Example of a multilayer neural network with four hidden layers.

first extending equation (4.1) to the whole layer $L^{(i)}$ containing $d^{(i)}$ single neurons one obtains

$$s_j^{(i)} = \sum_{k=0}^{d^{(i)}} y_k^{(i-1)} W_{kj}^{(i)} + W_{bj}^{(i)} \tag{4.5}$$

or similar to (4.3) in matrix form

$$s^{(i)} = \tilde{W}^{(i)} \hat{y}^{(i-1)}. \tag{4.6}$$

As usual, $\hat{y}$ denotes y extended with a single scalar 1 for the offset. Since the output of one layer is understood as input of the next one, we formally define the i-th layer

$$L^{(i)} \colon \mathbb{R}^{d_{i+1}} \to \mathbb{R}^{d_i}, \tag{4.7}$$

$$L^{(i)}(y^{(i-1)}) = y^{(i)} = F^{(i)}(s^{(i)}) = F^{(i)}(\tilde{W}^{(i)} \hat{y}^{(i-1)}) \tag{4.8}$$

Note that the activation function $F^{(i)}$ can differ from layer to layer. This way we can express the neural network as a composition of functions formed by the layers. Thus a neural network with k hidden layers of sizes $\{n = d_k, \ldots, d_0 = m\}$ approximating a function $V : \mathbb{R}^n \to \mathbb{R}^m$ looks like

$$N \colon \mathbb{R}^n \to \mathbb{R}^m, \tag{4.9}$$

$$N = L^{(k)} \circ L^{(k-1)} \circ \ldots \circ L^{(1)} \circ L^{(0)} \tag{4.10}$$

The total process of subsequently propagating the input through the neural network is called **Forward Propagation**.

Algorithm 1 Forward Propagation

```
1: Set $y^{(m)} = x$
2: for all layers $L^{(i)}, (i = m-1, \dots 0)$ do
3:     $s^{(i)} = \hat{W}^{(i)} \hat{y}^{(i+1)}$
4:     $y^{(i)} = F^{(i)}(s^{(i)})$
5: end for
```

The learning process consists of several steps. The essential goal is to compute $\frac{\partial E(W(t))}{\partial W(t)}$. This goal is achieved by multiple application of the chain rule and using backward propagated (hence the name **Backward Propagation**) signals though the layers. The whole process to obtain the gradient reads

Algorithm 2 Backward Propagation

```
1: Use Forward Propagation to obtain all the $y^{(i)}$
2: for all layers $L^{(i)}, (i = 0 \dots m)$ do
3:     Compute the gradient $DF^{(i)}(y^{(i)})$
4: end for
5: Set $\delta^{(0)} = DF^{(0)}(y^{(0)})$
6: for all layers $L^{(i)}, (i = 1, \dots m)$ do
7:     $\delta^{(i)} = \delta^{(i-1)} W^{(i-1)} DF^{(i-1)}(y^{(i-1)})$
8: end for
9: for all layers $L^{(i)}, (i = 0, \dots m)$ do
10:     $\delta_{err}^{(i)} = \frac{\partial E(y)}{\partial y}|_{y=y^{(0)}} \delta^{(i)}$
11:     $\frac{\partial E}{\partial W^{(i)}} = \delta_{err}^{(i)} \odot y^{(i)}$
12: end for
```

Here "$\odot$" denotes the outer vector product. By this algorithm we obtain the error gradient $\frac{\partial E}{\partial W}$ which is then used for learning via Gradient Descent Method

$$W(t+1) = W(t) - \eta \frac{\partial E(W(t))}{\partial W(t)}, \tag{4.11}$$

where $\eta \in \mathbb{R}_+$ is a learning parameter. It can be seen as a factor scaling the step size of the descent step in the direction of the gradient.

4.2 Design of the Neural Network

4.2.1 The Input

The most basic input for a neural network playing Go on a $n \times n$-board is a $n \times n$-dimensional vector containing the current board with colored stones encoded as -1 or 1 respectively and empty fields as zero. Because a neural network can hardly cope with zeros as inputs - as multiplying zero with a weight is essentially not very effective - the input needs to be prepared accordingly by choosing a different value for empty fields. In addition, considering M. Heining's work [Hei17], symmetric input values should be avoided. He proposed for a related problem (tic-tac-toe) a scale mapping of

$$f_{scale}(x_{i,j}) = \begin{cases} \text{-1.35} & \text{if stone at } (i,j) \text{ is black} \, (\Leftrightarrow x_{i,j} = -1) \\ 0.45 & \text{if field at } (i,j) \text{ is empty} \, (\Leftrightarrow x_{i,j} = 0) \\ 1.05 & \text{if stone at } (i,j) \text{ is white} \, (\Leftrightarrow x_{i,j} = +1) \end{cases}$$

which we have adapted for Go.

4.2.2 Activation Functions

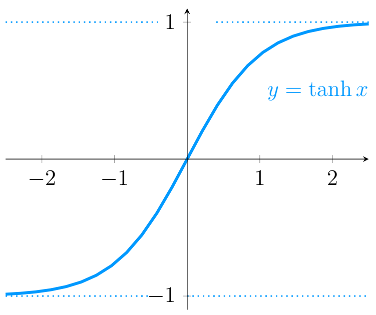
For the Policy Network defined in the introduction of this chapter, it is practical to use the softmax function in the last layer, because it yields a normalized distribution as output of the network, which is what the Policy Network requires. It is given by

$$F_{softmax} : \mathbb{R}^n \to [0,1]^m$$
$$F_{softmax}(x)_j = \frac{e^{z_j}}{\sum_{k=1}^m e^{z_j}}$$

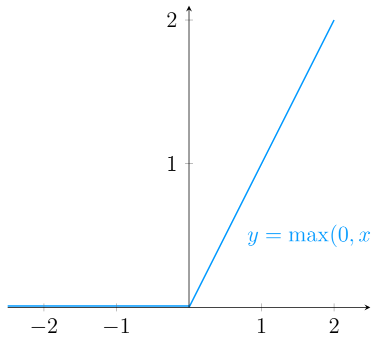
As stated by [C. Bishop Neuronal Networks for Pattern Recognition, pages: 126ff], very commonly used activation functions are sigmoid functions like *tanh* and the logistic sigmoid. Note that these two only differ by two linear transformations:

$$\frac{1}{2}(\tanh(x)+1) = \frac{1}{2}\left(\frac{e^x - e^{-x}}{e^x + e^{-x}} + 1\right) \overset{y=2x}{=} \frac{1}{2}\left(\frac{e^{\frac{y}{2}} - e^{-\frac{y}{2}} + e^{\frac{y}{2}} + e^{-\frac{y}{2}}}{e^{\frac{y}{2}} + e^{-\frac{y}{2}}}\right)$$
$$= \frac{e^y \cdot e^{-\frac{y}{2}}}{(e^y+1) \cdot e^{-\frac{y}{2}}} = \frac{e^y}{e^y+1} = \sigma(y)$$

This class of functions is characterized by being bounded and differentiable, having only a single turning point and a monotonous first derivative. Bishop also suggests using *tanh*, as



(a) The tanh function.



(b) The *ReLU* function.

Figure 4.3: Commonly used activation functions.

it shows faster convergence than logistic functions. Another suggestion we found that was commonly used were Rectified linear units, also called *ReLU*s. They promise much faster learning than sigmoid functions. In practice it performed a lot worse than *tanh*. In our opinion the high complexity of the underlying problem combined with the distribution format of the desired output might account for a disadvantage of just regular but not differentiable functions like the *ReLU*. Thus we decided to further work with *tanh* and softmax.

4.2.3 Error Functions

By error function we denote a function measuring the distance (i.e. ”error”) between the output of the neural network and the supposed value, which we will henceforth call target. The particular error function used for the Gradient Descent Method (4.11) in learning is of fundamental importance. One of the most famous error estimates is the mean squared error (MSE), which is a so called sum-of-squares error estimate. The MSE between two vectors $v, w \in \mathbb{R}^n$ is defined by

$$\mathrm{MSE}(v, w) := \frac{1}{n} \sum_{i=1}^{n} (v_i - w_i)^2 \tag{4.12}$$

As hinted by Bishop in [Bis95] on page 195, estimates from this class tend to have problems distinguishing non-Gaussian distributions with similar mean and variance. In our case targets and outputs are highly non-Gaussian, raising doubts in the suitability of the MSE for this problem.
A more thoughtful approach can be derived by looking at the space of outputs as a discrete metric space of probability distributions. The distance between distributions, i.e. elements in this space, is quantified by a statistical distance. Examples for statistical distances we found suitable for the problem were the Kullback-Leibler divergence (KLD) and the Hellinger distance (HELD).
The Kullback-Leibler divergence is defined by

$$\mathrm{KLD}(P||Q) = \sum_{i=1}^{n} P(i) \log \frac{P(i)}{Q(i)}$$

The Hellinger distance is defined to be

$$\mathrm{HELD}(P, Q) = \frac{1}{\sqrt{2}} \|\sqrt{P} - \sqrt{Q}\|_2$$

In the literature, one of the most commonly used error functions is the Cross-Entropy error which is given by

$$H(P, Q) = -\sum_{i=1}^{n} P(i) \log(Q(i)).$$

It is closely related to the KLD by

$$\mathrm{KLD}(P||Q) = H(P, Q) - H(P, P).$$

By thorough testing we found that the KLD gives best learning results, so we used it for our network most of the time. Even though the Cross-Entropy error is more commonly used in practice, we regarded the pure Entropy term as redundant, as our criterion for a error function was also to give a more naturally understandable measure of distance.
We also stumbled upon a theoretical problem regarding the error function to choose. Imagine a 2×2 board (then $n = 4$, we have four fields to play on and we can pass, resulting in an output size of five with output vectors living in Δ^4) and consider the three vectors

$$y_{dirac} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad y_{uniform} = \begin{pmatrix} 0.2 \\ 0.2 \\ 0.2 \\ 0.2 \\ 0.2 \end{pmatrix}, \quad y_{displaced} = \begin{pmatrix} 0 \\ 0.2 \\ 0 \\ 0.8 \\ 0 \end{pmatrix} \quad \in \Delta^4.$$

Now let the first vector y_{dirac} be the correct value of $V(b)$ for a certain board $b \in \mathbb{B}$. The other two vectors are two possible outputs a certain neural network N would give for the board, i.e. the result of $N(y_{dirac})$.

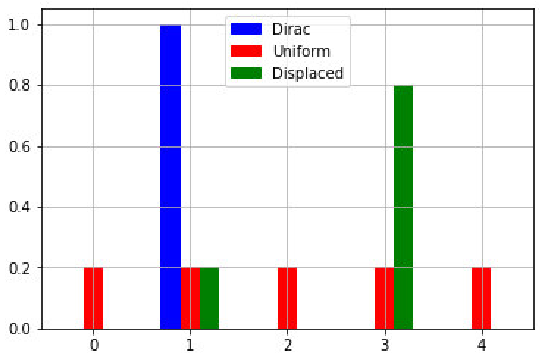


Figure 4.4: Visualization of the Dirac-Distinction problem.

Although they share the same mean absolute error, the outputs differ drastically. For our purposes $y_{uniform}$ is by far a better output in this case than $y_{displaced}$. The main reason for this is the certainty of a choice the neural network suggests. Here an output of $y_{displaced}$ would relate to a high certainty to play the wrong move number 3, whereas $y_{uniform}$ reads as a relative uncertainty about which move would be the most suitable to assess as best.
A question arises from this example: For which error function can a neural network distinguish the two vectors $y_{uniform}$ and $y_{displaced}$ in relation to the correct y_{dirac}?

	KLD	MSE	HELD	MAE
$y_{dirac} \leftrightarrow y_{uniform}$	1.6094	0.4	0.7435	1.6
$y_{dirac} \leftrightarrow y_{displaced}$	1.6094	0.64	0.7435	1.6

Figure 4.5: Errors respectively distances for chosen functions.

We see that only the MSE seems capable of solving the distinction problem correctly. Here MAE denotes the mean absolute error.
There is a way to incorporate both KLD and MSE into a error function via affine combination

$$E_\alpha(P, Q) = (1 - \alpha) \cdot \mathrm{KLD}(P||Q) + \alpha \cdot \mathrm{MSE}(P, Q)$$

for some combination parameter $\alpha \in \mathbb{R}$. The gradient of this function is then simply the affine combination of the separate gradients. With a reasonably chosen α this combined error function can solve the distinction problem while yielding a fast learning speed.

4.2.4 Initializing Weights

Why is the particular initialization of weights important? As already noted by Bishop in [Bis95] page 260, putting work in a good initialization can speed up learning and avoid negative influence on sensitive algorithms like the stochastic gradient descent.
In order to achieve that, one has to keep the propagation signals near the regions where the derivatives are maximal. For sigmoid functions this means the signals should optimally lie in a

narrow interval close around zero.

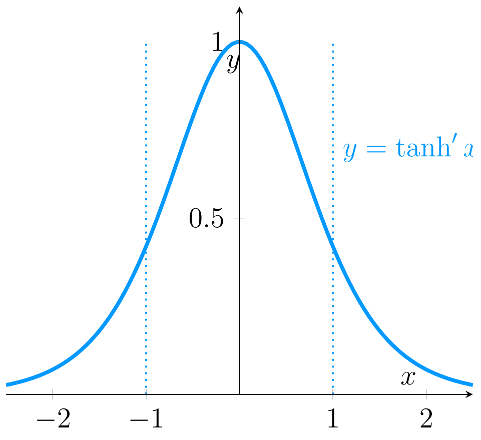


Figure 4.6: The graph of the tanh gradient.

The previous consideration requires the signal $s = \sum_{i=1}^{h} w_i x_i$ in a layer of size h to approximately fulfill

$$E(s) = 0 \qquad \text{and} \qquad Var(s) = 1.$$

Using that h is finite, w_i and x_i, respectively, are independent and identically distributed for all $i = 1, \dots, h$ and w, x are uncorrelated, we obtain

$$E(s) = E\left(\sum_{i=1}^{h} w_i x_i\right) = \sum_{i=1}^{h} E(w_i)E(x_i)$$

The easiest case for this to vanish would be $E(w_i) = E(x_i) = 0 \quad \forall i = 1, \dots h$.
For the variance it holds

$$\begin{aligned} Var(s) = E(s^2) - E(s)^2 = E(s^2) &= \left(\sum_{i=1}^{h} E(w_i)E(x_i) \cdot \sum_{j=1}^{h} E(w_j)E(x_j)\right) \\ &= \sum_{i=1}^{h} E(w_i^2 x_i^2) + \sum_{i \neq j}^{h} E(w_i w_j x_i x_j) \\ &= \sum_{i=1}^{h} E(w_i^2)E(x_i^2) + \sum_{i \neq j}^{h} E(w_i)E(w_j)E(x_i)E(x_j) \\ &= \sum_{i=1}^{h} E(w_i^2)E(x_i^2) = h \cdot E(w_i^2)E(x_i^2) \end{aligned}$$

by the algebraic formula for the variance and using $Cov(w_i^2, x_i^2) = 0$ and again that both w_i and x_i are i.i.d. for the last equality. If now the input is scaled such that $Var(x_i) = 1$ while also requiring $E(w_i) = E(x_i) = 0$ from above to hold, we would ultimately get

$$Var(s) = Var(w_i) \cdot h$$

leading to the condition $\sigma = \sqrt{Var(s)} = \frac{1}{\sqrt{h}}$. So, finally, we derived that one should choose the weights initially from a normal distribution $\mathcal{N}(0, \frac{1}{\sqrt{h}})$. There also exist approaches with different distributions like the uniform distribution. But due to its simplicity, we further stuck with $\mathcal{N}$.
It is yet to check if the assumptions on well-scaling of the input hold for the case of Go-Games. An analysis of around ten sample Go Games after applying f_{scale} on the boards yields the following:

$$E = \begin{pmatrix} 0.002 & 0.005 & 0.001 & -0.001 & 0.01 & -0.001 & 0.001 & 0.005 & 0.002 \\ 0.005 & 0.005 & 0.006 & 0.014 & 0.004 & 0.014 & 0.006 & 0.005 & 0.005 \\ 0.001 & 0.006 & 0.005 & 0.005 & 0.012 & 0.005 & 0.005 & 0.006 & 0.001 \\ -0.001 & 0.014 & 0.005 & 0.005 & 0.006 & 0.005 & 0.005 & 0.014 & -0.001 \\ 0.01 & 0.004 & 0.012 & 0.006 & 0.013 & 0.006 & 0.012 & 0.004 & 0.01 \\ -0.001 & 0.014 & 0.005 & 0.005 & 0.006 & 0.005 & 0.005 & 0.014 & -0.001 \\ 0.001 & 0.006 & 0.005 & 0.005 & 0.012 & 0.005 & 0.005 & 0.006 & 0.001 \\ 0.005 & 0.005 & 0.006 & 0.014 & 0.004 & 0.014 & 0.006 & 0.005 & 0.005 \\ 0.002 & 0.005 & 0.001 & -0.001 & 0.01 & -0.001 & 0.001 & 0.005 & 0.002 \end{pmatrix} \approx 0_{9\times 9}$$

and respectively for the Variance

$$Var = \begin{pmatrix} 0.57 & 0.82 & 0.8 & 0.81 & 0.85 & 0.81 & 0.8 & 0.82 & 0.57 \\ 0.82 & 0.93 & 0.97 & 0.98 & 1 & 0.98 & 0.97 & 0.93 & 0.82 \\ 0.8 & 0.97 & 1.02 & 0.99 & 1.01 & 0.99 & 1.02 & 0.97 & 0.8 \\ 0.81 & 0.98 & 0.99 & 1.02 & 1.02 & 1.02 & 0.99 & 0.98 & 0.81 \\ 0.85 & 1 & 1.01 & 1.02 & 1.03 & 1.02 & 1.01 & 1 & 0.85 \\ 0.81 & 0.98 & 0.99 & 1.02 & 1.02 & 1.02 & 0.99 & 0.98 & 0.81 \\ 0.8 & .097 & 1.02 & 0.99 & 1.01 & 0.99 & 1.02 & 0.97 & 0.8 \\ 0.82 & 0.93 & 0.97 & 0.98 & 1 & 0.98 & 0.97 & 0.93 & 0.82 \\ 0.57 & 0.82 & 0.8 & 0.81 & 0.85 & 0.81 & 0.8 & 0.82 & 0.57 \end{pmatrix} \approx 1_{9\times 9}$$

One can see that the required assumptions are in fact fulfilled by the scaled data.
This method is called *Xavier initialization* and works for, amongst others, sigmoid functions. Since non-sigmoid functions like ReLU are neither zero-centered nor differentiable at zero, they require a different method, the so-called *He initialization*. More theory on this method can be found for example in [Kum17].

Another technique to further improve the initial error includes the idea of Monte-Carlo-Simulation. This technique is often used for approximation in very high dimensions. In our case, the space of weights has a very high dimension. First an ensemble of multiple random weight initializations is sampled. Then all of them are tested with a training set and the weights with lowest error on that set are chosen for the neural network. The efficiency of this method depends on many factors like layer size and the input, making it hard to decide whether or not it is faster than just improving the weights by a learning algorithm. For the Policy Network weight ensembles stopped yielding an advantage as soon as Filters (see section 4.4.2 below) have been introduced and the learning method surpasses its efficiency. This again accounts for Filters as being an adequate tool for improving the information gained.

4.3 Finding Good Learning Parameters

4.3.1 Batches

Generally speaking, it might be a bad idea to update the weights after every single input, since this leads to higher computation times and in some cases to oscillation on the error surface. Therefore it has become common practice to split up the training data into so called batches

of same size and only actualizing the weights after each batch.
Since it seems that there existed no sufficiently good recommendations on what batch size to use, we tried to determine the best size through testing. We found that a batch size of around 1% of the total training set size was a good rule of thumb. One should keep in mind that there is a quite direct trade-off between accuracy and speed when choosing the batch size, making it hard to determine a perfect size for all situations.

4.3.2 Stochastic Gradient Descent and Learning Rate

A common method for minimizing a function in a high dimensional space is the gradient descent method. In our context this means finding optimal weights that minimize the error produced by the neural network using those weights. The method can be written as the basic update rule

$$W(t+1) = W(t) + \Delta W(t)$$
$$\Delta W(t) = -\eta \frac{\partial E(W(t))}{\partial W(t)}$$

where η is called learning rate or step size, and $\Delta W(t)$ can be equipped with several other terms, as we will see in chapter 4.3.3.
Since the computation of the gradient $\frac{\partial E(W(t))}{\partial W(t)}$ is numerically expensive, one uses the so called Stochastic Gradient Method or SDM for short. When learning a training set we simply look at just a random sample of data points instead of the whole set at once. This rather heuristic method yielded faster convergence in practice and is less prone to getting stuck in local minima. A more theoretical explanation for the performance of SDM can be found in [DNS14].

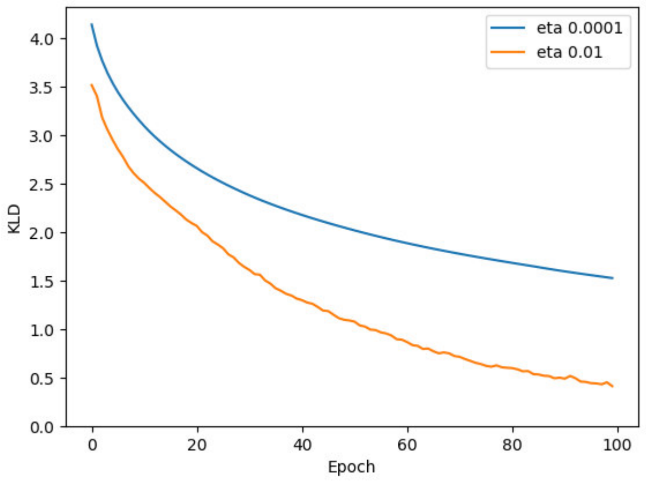


Figure 4.7: Plot of training results on a small set for different eta.

The explicit choice of η is fundamental for the learning speed and accuracy. Since the Gradient Descent Method is much more sensitive at the end of the learning process, one can save time and accuracy by letting η decay over time. This can be done with a fixed decay-time relation like

$$\eta(t) = \eta_0 \, e^{-\omega t}, \quad \omega \in \mathbb{R}_+$$

or alternatively by observing the error and reducing η stepwise if the error stagnates locally. Still, a good choice for η_0 has to be found by testing.
At the very beginning of building data structures we started with having only a single data point representing all moves played on a single board. Consequently every board would have been learned only once, even though occurring far more often than other board e.g. the empty board. To handle this, we introduced an adaptive η rule, which works as follows: Consider a board b that has appeared in the data set k times. Now instead of learning the board k times, b is equivalently learned with an adaptive η of

$$\hat{\eta} = f(k, \eta)$$

where we choose the importance weight function $f\colon \mathbb{N} \times \mathbb{R}_+ \to \mathbb{R}_+$ in different ways

$$f(k, \eta) = \eta \log(2 + k) \quad \textit{or} \quad f(k, \eta) = \eta k.$$

Though not explicitly changing the learning behavior, we achieved a better adaptation to a more promising play style by this. Overall the logarithmic rule yielded a better playing result than both linear rule and no adaptive η rule at all.

4.3.3 Regularization and Momentum Term

For most of the testing and coding phase we were limited to the use of small training sets due to computation costs and the need for flexibility and of fast response. This quickly lead to the well known problem of Overfitting of the neural network. By this, the neural network was unable to see the underlying hidden data and built up strong codependencies between its neurons, specializing it for a single particular set of games. To avoid this we used a common practice called L2-Regularization, which introduces a regularization term punishing the entries of the weight matrices for being to large.
The resulting regularized error function reads:

$$E_{\mathcal{R}}(W) = E(W) + \frac{\lambda}{2N} \sum_{i,j} W_{i,j}^2 \quad ,$$

where $\lambda \in [0, 1)$ is called regularization parameter.
Since the learning speed on very flat parts of the error surface tends to be slow, one introduces a momentum term. This term corresponds to the physical momentum in so far that it carries on a proportion of the motion on the error surface from the last update step to the next one. This way the learning rate on flat error surface regions get boosted, promising faster learning convergence. So we introduce the new term:

$$\Delta W_\mu(t) = -\eta \Delta W(t) + \mu \Delta W_\mu(t-1) \quad ,$$

with a momentum parameter $\mu \in [0, 1)$.

Combining L2-regularization and momentum term the resulting update rule is:

$$\begin{aligned} W(t+1) &= W(t) + \Delta W(t) \\ \Delta W(t) &= -\eta \frac{\lambda}{N} W(t) - \eta \frac{\partial E}{\partial W} + \mu \Delta W(t-1) \end{aligned}$$

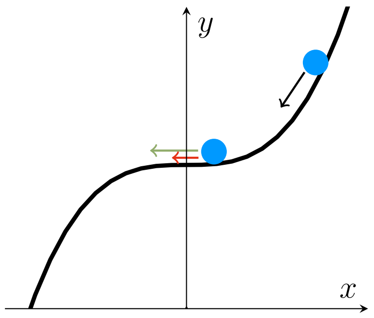


Figure 4.8: Learning step with (green) and without (red) momentum.

4.4 Architecture of the Neural Network

4.4.1 Layers

There is barely any sophisticated way to obtain the best layer architecture (size and number of layers), apart from just testing. In the following we compared different layer architectures on a small training set. As can be seen in figure 4.9 a low amount of layers, but containing a big

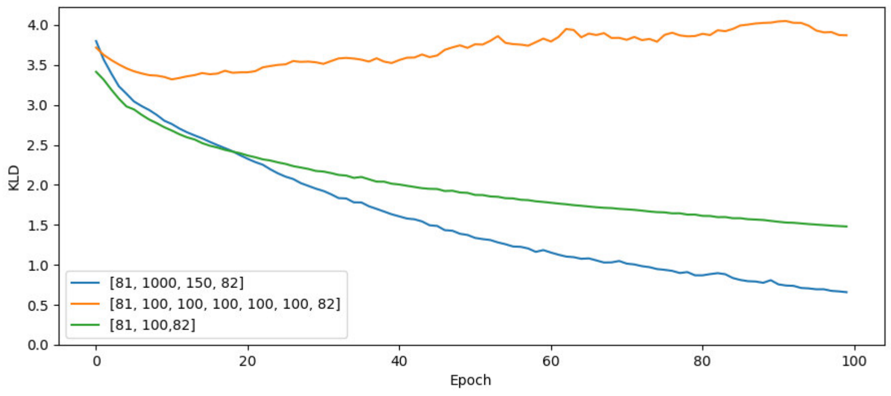


Figure 4.9: Three different layers in training result comparison.

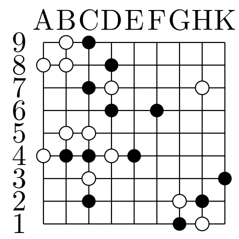
size layer yielded the best results in testing. Thus our layer architecture generally consisted of a very big layer followed by at least one smaller layer. The fluctuations and worse performance of the orange line in the figure is due to high sensitivity and to big learning rate for the many-layered network.

4.4.2 Filters

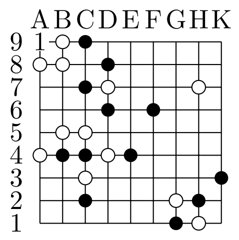
Since neural networks are very successfully being used for image recognition, methods from that particular field are worth also applying for problems like the game of Go.
Especially the concept of filtering features from the input seemed promising. Filters are an inherently natural way of extracting useful information from a given image or, as in this case, from a Go board. By this thoughtful preprocessing we intended to further increase the performance of the neural network.
One of the main difficulties for programs playing Go lies within the nature of the game itself: The inability to measure most of the strategic thinking required in a quantitative fashion. Human heuristics on Go strategy has been passed on through generations by oral traditions

over centuries and has later been written down, forming the now community-wide highly valued Go Proverbs. These old wisdoms can sometimes be translated into filters to aid the neural network in playing more strategic. This way we could incorporate game expert knowledge into the network.

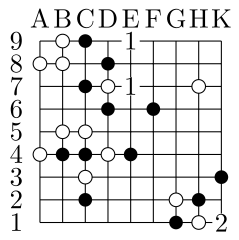
We define a Filter as a mapping from the space of Go boards $\mathbb{B}$ to some feature space V_{feat}, where most of the used Filters map to $V = \mathbb{Z}^{81}$. In that case we call the image of some board under such a filter an information matrix.
In the following we consider the particular board seen on the right. Exemplary Filters are applied to this board for the white player and presented together with the resulting information matrix.



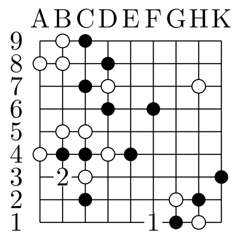
Eyes Filter This Filter searches the board for so called eyes for a player of given color. Eyes are vacant fields of size one that are totally surrounded by stones of the player. They are generally considered as valuable, as two of them in a group secure it from getting captured, making it an alive group.



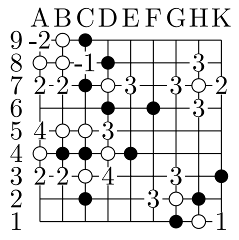
Eyes Creation Filter Quite similar to the Eyes Filter, this one checks for positions that will result in the creation or destruction of own eyes for a player.



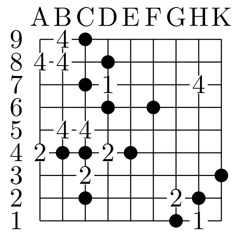
Capture Filter The Capture Filter will extract the number of stones being captured by playing a stone on a free field. This means: If a move on a field will result in n stones being taken, that field will have a value of n.



Added Liberties Filter Increasing the total cumulated liberties of your own groups is a good tactic in Go. So this Filter counts the liberties added by a move to each adjacent group. Connecting groups is strongly favored, which also leads to a better play style.



Liberties Filter This Filter locates strongly connected groups of player stones. Then the liberties of the whole strong group are counted.



5 Results

We succeded in implementing a Go engine that utilizes the *GTP* to communicate Go commands with a Go client. This means we were able to play against our Go engine in a *GUI* such as *GoGui* and test our Go engine on a public Go server. Furthermore, we successfully implemented all variants of the Policy Net seen in section 4 from scratch. We were able to train many different instances of the Policy Net by adjusting the various parameters and hyperparameters. Instead of trying different neural networks or different techniques from artificial intelligence, we concentrated on improving this Policy Net more and more with various ideas and approaches, as seen in section 4. After all of this our best Go engine had a 89% win-rate against a random Go engine and we were still able to beat it in most games.

There are reasons for this slightly disappointing result. The likely most significant reason is the restriction in the amount of training data we were able to use. Once we were able to lower the training error to a reasonable amount, we experienced overfitting: our Go engine imitated the training data well, but didn't generalize to produce good moves for unseen board positions. The natural approach to counter overfitting given a good model is using more training data. Using more training data increases the training time, so we utilized the *LRZ*-cluster resources in the hope that more computing power would make training time more feasable. This wasn't the case for our self-implemented neural network and in any constellation of CPU cores and memory it trained at the same speed. Not being able to use the amount of training samples we had available hindered the covering of the search space over which we were trying to find the optimal function. Even with our model improving and playing good moves occasionally, we were far away from covering the search space enough to produce good moves consistently. We realize now that a neural network implementation in *Python* with the package *NumPy* for mathematical operations isn't a good approach, if one cares about scalability. *NumPy* as well as the language *Python* isn't meant for implementing computation heavy programs directly. Programming libraries that are utilized by *Python*, such as *Tensorflow*, are themselves written in highly-optimized *C++* and *CUDA* code and are therefore capable of exploiting CPUs and GPUs efficiently.

Another problem with our approach lies in our assumptions. We assume the Policy Net will win the game by playing good moves. But in a game of Go a sequence of good moves doesn't make you a winner, because in the end the player with the most area wins. Additionally, we assume that we can learn the *best* moves by learning from training data that is labeled by humans. The problem here is that not every move that occurs in the training data is a good move. So one has to distinguish between good and bad moves in the training data which is a difficult task. Not every move that was played by a strong player is necessarily a good move and not every move that was played by the (at the end of the game) winning player is necessarily a good move. Because we didn't see a clean way to weight the training data, we assumed every move as equally good and the more often it occured, the better it was. An improvement to our approach would be to reevaluate moves with a second neural network that estimates a winning probability of this move, similar to the Value Net of *AlphaGo*.

We have seen that training a neural network with supervised learning techniques only takes you so far. To approximate the ideal function that always produces moves that lead to winning the game of Go, one has to go beyong supervised learning into the field of unsupervised learning. Every version of *AlphaGo* utilized reinforcement learning by self-play and with *AlphaGo Zero* it was shown that a Go engine can be realized without considering labeled training data at all.

Implementing the deep neural network including backpropagation from scratch helped us gain a deep understanding of the mathematical concepts behind neural networks, and it also enabled us to directly implement our creative ideas into the neural network. We don't regret following this approach, because it forces to thoroughly think through every little step in the theory and the achievement of making it work is very rewarding. At the same time we realize that using an established machine learning library would have resulted in a scalable neural network, a likely better overall result and a substantially smoother project progress. All in all we are very content with the progress made in this project considering we barely knew anything about Go, neural networks or the *Python* programming language at the beginning of this project.

List of Figures

The figures in this work were made using *TikZ*, *PGFPlots* and *matplotlib*. All figures were made by the authors if not explicitly stated otherwise.

Bibliography

[AGA] American Go Association, What is Go? http://www.usgo.org/what-go. Accessed: 31.01.2018.

[BGA] British Go Association. https://www.britgo.org. Accessed: 31.01.2018.

[Bis95] C. Bishop. Neural Networks for Pattern Recognition. *Clarendon Press, Oxford*, 1995.

[Dee] Deep Blue, IBM. https://www.research.ibm.com/deepblue/meet/html/d.3.3a.html. Accessed: 05.02.2018.

[DNS14] R. W. Deanna Needell and N. Srebro. Stochastic Gradient Descent, Weighted Sampling, and the Randomized Kaczmarz algorithm. *Advances in Neural Information Processing Systems 27, pages 1017-1025, Curran Associates, Inc.*, 2014.

[Dor90] W. Dorland. Illustrated Medical Dictionary. *Saunders, Philadelphia*, 1890.

[Goo] The Mystery of Go. http://www.chilton-computing.org.uk/acl/literature/reports/p019.htm. Accessed: 14.01.2018.

[Hei17] M. Heining. Dynamical Learning: Case Study on Tic-Tac-Toe (Master's Thesis). *TU München*, 2017.

[Kum17] S. K. Kumar. On weight initialization in deep neural networks. https://arxiv.org/pdf/1704.08863.pdf, 2017.

[S$^+$16] D. Silver et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 2016.

[S$^+$17] D. Silver et al. Mastering the game of Go without human knowledge. *Nature*, 2017.

[Sen] Sensei's Library. https://senseis.xmp.net/. Accessed: 31.01.2018.

[TF16] J. Tromp and G. Farnebäck. Combinatorics of Go. *Computer and Games, Springer*, 2016.