A Machine Learning to Play Go
TUM Data Innovation Lab WS 2017/18

Yushan Liu
yushan.liu@tum.de

Nathanael Bosch
nathanael.bosch@gmail.com

Karthikeya Kaushik
karthikeya.kaushik@gmail.com

Benjamin Degenhart
benjamin.degenhart@tum.de

February 13, 2018

Project Lead: Dr. Ricardo Acevedo Cabra
Supervisor: Prof. Dr. Massimo Fornasier
Advisor: M. Sc. Bernhard Werner

Abstract

Because of its high complexity, programming a machine which can play Go on a high level has always been a challenging task. Recently, by applying Monte Carlo tree search combined with artificial neural networks, AlphaGo has succeded in defeating professional Go players, and its successor AlphaGo Zero became the best Go player in the world.

In our project, we want to examine how well neural networks can perform on a 9×9 Go board without using tree search, which is not competitive for a 19×19 board. Experimenting with various network architectures including non-convolutional networks, convolutional networks, policy bots, value bots, different data preprocessing methods and training settings, we tried to recreate moves played by humans in real games. Our best performing bot uses a convolutional residual network and has an approximate winning probability of 66% against GNU Go, which plays at an average amateur level (ranked about 5 to 7 kyu).

Contents

1 Introduction

For the longest time, computer programs looking to beat human players in board games relied on heuristics designed to mimic human moves. The more complex and broader the heuristic, the better the computer player. A good analogy for heuristic methods is that of a puppet, handled by an intelligent puppeteer (the programmer). But there are a few issues with this method. Firstly, the computer is only as good as the programmer designing the heuristic. Adaptability becomes an issue here because the computer player is only conditioned to respond to situations necessary for the heuristic to work and is therefore not flexible. Secondly, (although in a more abstract sense) the computer loses out on all the common knowledge available from previously recorded plays.

More recently, the Monte Carlo tree search (MCTS) method, which is based on exploring the search space with random sampling and expanding the search tree based on the analysis of the present state, gained prominence. Simply put, the machine makes use of its immense computational power to look ahead into the future and determines which move produces the best possible result! Here is where things get tricky. For the 19×19 Go board, Tromp and Farnebäck give the number of *theoretically* possible games to be $10^{10^{48}}$ [Tromp and Farnebäck, 2007]. It is very unlikely that our computers will reach the ability to brute force the search space in the near future. (Skynet on steroids, perhaps.) This effectively rules out a pure MCTS approach, because the search space is mind-numbingly large.

The simplicity and elegance of the rules of Go only serve to make the gameplay all the more intricate, yet possessing a common, straightforward goal of capturing territory. It is interesting to explore Go because from it we can hope to progress to more difficult problems and apply the ideas we learn in this context.

This project is an attempt at understanding neural networks and applying it to develop a bot that plays Go on a 9×9 board, which is a smaller subset of the larger 19×19 board.

2 The Go Game

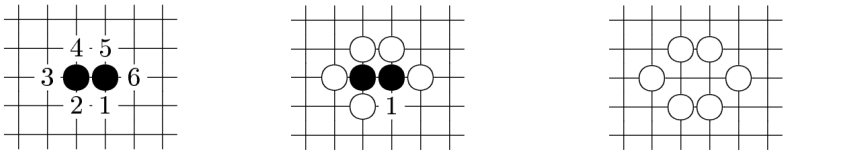
Before we start talking about our implementation of computer Go, we will first introduce the game of Go and the basic rules.
Go is an ancient Chinese board game, invented 3000-4000 years ago, with a simple two player turn based play. The players are called Black and White, corresponding to the colors of their stones. Games are usually played on a 19×19 board, with each player taking alternate turns in placing one stone on the grid points. The intention of each player is to cover as much territory as possible, without being surrounded by enemy stones.
In our project, we focus on a smaller 9×9 board to reduce the complexity of the problem.

2.1 The Rules of Go

Basic rules
Black plays first and places a stone on an empty grid point of the Go board. The two players take turns and can either place a stone or pass. A stone on the board cannot be moved, except for when the stone is captured after which it will be removed from

the board.

The adjacent empty points of a stone are called *liberties*, and a stone is captured if the number of its liberties is zero. Neighboring stones of the same color which are connected vertically or horizontally are called a *group* or a *chain*. The liberties of a group are the union of the liberties of each of its stones so that a group can also be captured if none of its stones has any liberties left (see Figure 1).

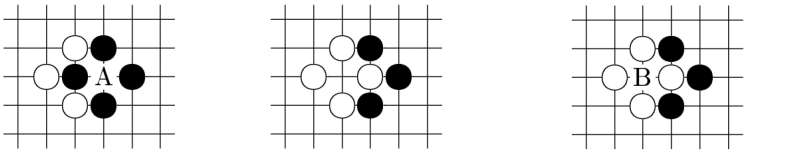


This black group has 6 liberties. | Now, the black group has only 1 liberty. White plays at 1. | The black group has 0 liberties left and is removed from the board.

Figure 1: Capture of a group

Players are not allowed to commit suicide which means to play a move that decreases the liberties of one of her own stones or groups to zero. However, such a move can be played if opponent stones will be captured after this move.

Another important rule is the *ko rule* which prevents repetitive board positions. As one can see in Figure 2, White has just played a stone on A capturing a black stone. Now Black is not allowed to directly play on B since an endless loop could result from it, but she has to play somewhere else or pass.



White can capture a black stone by placing on A. | The captured black stone is taken from the board. | Black cannot immediately capture the white stone by placing on B.

Figure 2: Ko situation

The game ends if both players have passed consecutively or if one player resigns.

Scoring
At the end of the game, the amount of territory of each player is counted. Since Black begins the game, White is given a compensation called *komi* which is usually between 5.5 and 7.5 points.
There are two types of scoring systems: area scoring (Chinese rules) and territory scoring (Japanese rules). In area scoring, the number of each player's stones on the board and the surrounded empty points are counted, while in territory scoring, the number of surrounded empty points and the number of captured opponent stones are counted. Both methods usually lead to the same result.

Our bot can score according to both Chinese rules and Japanese rules. In official bot tournaments usually Chinese rules are used, as they allow for easier automated score evaluation.

2.2 Computer Go

The game of Go has such a high state space complexity compared to other complete information games (such as chess) that is was long believed impossible to build a computer which can defeat humans in Go. The events in

- 2015: AlphaGo defeated Fan Hui 2-dan 5-0 in 19×19 even games,
- 2016: AlphaGo defeated Lee Sedol 9-dan 4-1 in 19×19 even games, and
- 2017: AlphaGo Zero defeated AlphaGo 100-0 only after three days of training

proved us otherwise.

We have come a long way since the first Go program, probably written by Albert Zobrist in 1968 [Zobrist, 1970]. Some advances were made in earlier times, but the fact that merely the endgames of Go are PSPACE-hard (at least as hard as NP) [Wolfe, 2002] slowed down the rate of improvement quickly. However, in 2006, Monte Carlo tree search was introduced [Kocsis and Szepesvári, 2006], which could be efficiently implemented for computer Go, and led to a rapid increase in strength of computer Go programs. After combining MCTS with artificial neural networks or even reinforcement learning, as can be found in the different AlphaGo architectures, computer Go programs successfully managed to surpass the best ranked human players.

More recently, AlphaGo Zero became the most highly rated player in the world [Silver et al., 2017]. It used the concept of reinforcement learning, in which by self play the computer learns to play Go without any human knowledge. This represents a significant change in how machine intelligence problems are approached. Essentially, AlphaGo Zero managed to learn all of human knowledge in a few days of continuous self play and in the process also generated novel strategies that human experts now study with high interest.

For our project, we played against GNU Go, a free software that is rated between 5 to 7 kyu (average amateur player), so as to get some understanding about the performance of our bot.

3 Artificial Neural Networks

Our main goal is to create a machine which learns to play Go only using articifial neural networks without tree search. Neural networks are currently one of the most popular models and, while being more demanding regarding training effort and providing results that are often not very interpretable, are one of the strongest tools to solve some problems that were long regarded as very hard.
Since the first introduction of neural networks in the 1950s, there were many advances that helped making them more usable and popular:

- Efficient training of neural networks is only possible since the introduction of the *backpropagation algorithm* by Paul Werbos [Werbos, 1974].
- Different techniques were developed to train deep neural networks without overfitting and vanishing gradients, like e.g. “dropout” and “batch normalization”.
- The advances in hardware and the explosion of computational power, especially in GPUs, made training more efficient.

- The large availability of data allows very complex networks to be trained without overfitting.

There are countless other improvements. Building on the basic idea of a neural network, many other architectures have evolved, often motivated by a concrete problem, but sometimes also by biological examples which researchers tried to recreate.

3.1 Neural Networks

A *feedforward neural network* consists of one input layer, a specific number of hidden layers, and one output layer (see Figure 4). Each layer is connected with the next layer by certain parameters (*weights* and *biases*), and the information only moves forward in the direction of the output layer. Given the inputs or observations $I_1, ..., I_D$, a neural network is a function f with $f(I_1, ..., I_D) = (O_1, ..., O_K)$ where $(O_1, ..., O_K)$ is the output vector. The goal is to find weights and biases such that the network can predict the outcome of unseen data with high accuracy.

The value of each neuron is determined by an *activation function*, where each layer can have a different activation function. This function takes as input the sum $\sum_{i=0}^{n} a_i w_i + b$, where the a_i's are the activations of the neurons from the layer before, the w_i's the corresponding weights, and b the bias of this neuron. Common activation functions are the sigmoid function $h(x) = \frac{1}{\exp(-x)}$, the hyperbolic tangent $h(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$, and the recitfied linear unit (ReLU)

$$h(x) = \begin{cases} 0 & \text{if } x \leq 0, \\ x & \text{if } x > 0. \end{cases}$$

For a plot of these functions, see Figure 3.

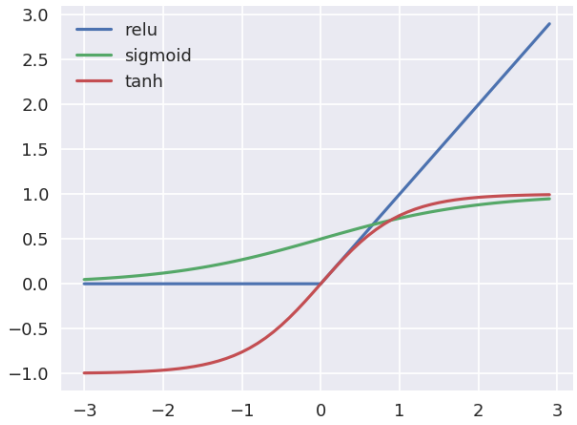


Figure 3: Plot of the three different activation functions we mentioned

For a classification task, the activation function of the output layer often is the softmax function $h(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{K} \exp(x_j)}$, $i \in \{1, ..., K\}$, where K is the number of classes, i. e. the softmax maps the output vector to a probability distribution over the classes.

At the beginning, the weights and biases of the network are initialized randomly. For each input x_i, $i \in \{1, ..., N\}$, we calculate the network output $\hat{y}_i$, $i \in \{1, ..., N\}$, and

compare it to the desired output y_i, $i \in \{1, ..., N\}$. We want to increase the accuracy of the network and thus want to minimize the error which is measured by a *loss function*. Typical loss functions are the mean squared error $E = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2$ and the cross entropy $E = -\sum_{i=1}^{N} p(y_i) \log p(\hat{y_i})$, where $p(x)$ is the probability distribution of x. Then we use the backpropagation algorithm to update the weights and biases. Usually we divide the training set into smaller batches and update the weights after processing each batch. Going through all training samples is called an *epoch*. We repeat this training process for a fixed number of epochs or until some stopping criterion is fulfilled.

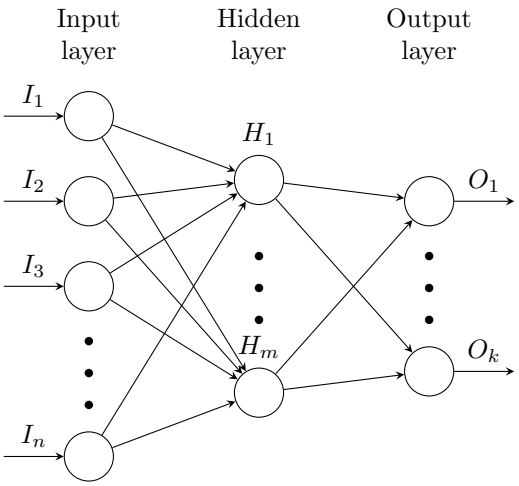


Figure 4: Example of a neural network with one hidden layer

For the Go problem, one could for example take the current board position as input for the network which calculates the best possible next move as output.

To see how a neural network works in detail, we consider a simple example, the well-known XOR problem. This problem is not linearly separable so that we need at least one hidden layer.

Example
We want to find a neural network that produces the results from Table 1 which consists of four observations.

I_1	I_2	Output O (I_1 XOR I_2)
0	0	0
0	1	1
1	0	1
1	1	0

Table 1: The XOR problem

A neural network solving the XOR problem is displayed in Figure 5 (of course, there are many other possibilities). Let the activation function of the hidden layer and the output layer be

$$h(x) = \begin{cases} 0 & \text{if } x \leq 0, \\ 1 & \text{if } x > 0. \end{cases}$$

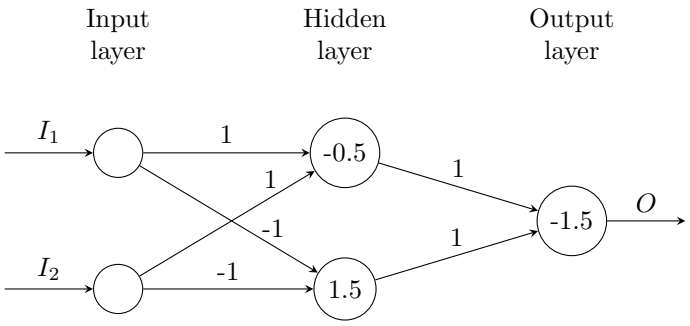


Figure 5: A neural network solving the XOR problem. The numbers on the edges specify the weights and the numbers in the neurons the biases.

We can prove that this neural network indeed solves the XOR problem and predicts each outcome correctly:

$$
\begin{aligned}
f(0,0) &= h(1 \cdot h(1 \cdot 0 + 1 \cdot 0 - 0.5) + 1 \cdot h(-1 \cdot 0 - 1 \cdot 0 + 1.5) - 1.5) = h(-0.5) = 0 \\
f(0,1) &= h(1 \cdot h(1 \cdot 0 + 1 \cdot 1 - 0.5) + 1 \cdot h(-1 \cdot 0 - 1 \cdot 1 + 1.5) - 1.5) = h(0.5) = 1 \\
f(1,0) &= h(1 \cdot h(1 \cdot 1 + 1 \cdot 0 - 0.5) + 1 \cdot h(-1 \cdot 1 - 1 \cdot 0 + 1.5) - 1.5) = h(0.5) = 1 \\
f(1,1) &= h(1 \cdot h(1 \cdot 1 + 1 \cdot 1 - 0.5) + 1 \cdot h(-1 \cdot 1 - 1 \cdot 1 + 1.5) - 1.5) = h(-0.5) = 0
\end{aligned}
$$

The XOR problem is only about classifying four different situations. Usually, we have a larger training set consisting of samples with their corresponding labels (desired outputs). The network should learn the essential features based on the training set so that it can then make generalized predictions for new data.

3.2 Convolutional Neural Networks

A *convolutional neural network* (CNN) is a type of feedforward neural network which is very often used for images, but also for natural language processing or continuous responses. Let us take an image, which is usually represented by a pixel array, as an example. The main idea is that the image consists of several subregions, also called receptive fields, from which we can extract interesting features that help us to assign the image to a certain category.

Every convolutional layer has a filter (a matrix containing the weights) which is applied to each receptive field, and the output is passed on to the next layer. One can visualize this filter as a window that is sliding over the image and detects relevant features, such as edges or certain patterns. Each filter, which can be of various size, corresponds to a particular feature. It is important to note that we have shared weights for one layer, i. e. we use the same filter while sliding over the image, thus reducing the number of parameters drastically compared to fully-connected layers. At the final layer, the learned features contribute to the right classification.

A convolutional network often also includes pooling layers to discard unnecessary information. A pooling layer maps each receptive field to a single neuron in the next layer. An often used method is max pooling which takes the maximum of the neuron activations in each receptive field. Pooling decreases the required memory space and also prevents overfitting.

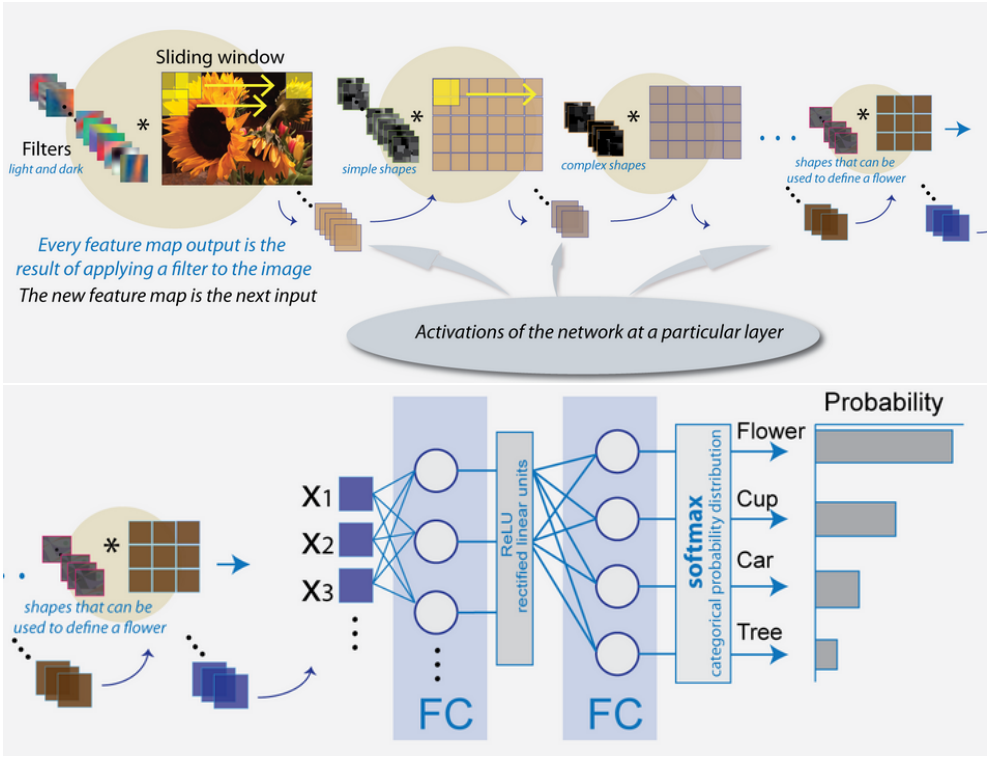


Figure 6: Architecture of a convolutional neural network[1]

The typical architecture of a CNN for classification is shown in Figure 6. It consists of convolutional layers, pooling layers, and fully-connected layers.
Convolutional layers are a reasonable choice in the context of Go, as it enables our network to learn and recognize certain patterns on the board. To close the gap to its regular field of application, *Computer Vision*, the board can be interpreted as a 9×9 image, and while convolutions are sometimes criticized for their viewpoint-dependence, this problem does not arise in Go as the underlying information of the image is actually a two-dimensional board, not a three-dimensional object. Our empirical results also confirm the choice of current literature in using convolutional neural networks for computer Go (see 5.4.3).

4 Implementation

4.1 Overall Setup and Code Architecture

4.1.1 Playing

We decided to use Python[2] as our programming language, a natural choice these days for all things machine learning related. As a version control system we chose *git* and GitHub as hosting service for our repository[3]. Thanks to the ability of Python's package manager *pip* to automatically install dependencies listed in a file called `requirements.txt`, it was quick and easy to clone our repository to new local or virtual machines, set up a virtual Python environment, install the dependencies, and

[1] https://de.mathworks.com/help/nnet/ug/introduction-to-convolutional-neural-networks.html
[2] Version 3.6
[3] `https://github.com/nathbo/GO_DILab`

run the code.
The first task was to implement the rules of the Go game. We went with a NumPy[4] matrix that allows for stones to be placed on a 9×9 board. It forbids illegal placements and removes captured stones. As convenience for bots we added a method to get the currently playable locations for a color.
The next step was to conform with the Go Text Protocol[5] (GTP), a text based protocol for communication with computer Go programs. Most Go clients and servers are based on this protocol and bots plugged into them need to be able to follow GTP commands.
To follow the terminology used by the protocol's authors, we called the class wrapping our `Game` class into GTP `GTPengine`. Upon execution, this engine writes to and reads from standard streams (*stdin*, *stdout*). In that way it becomes instantly usable as GTP engine from either the command line or by any client acting as GTP controller. We implemented `GTPengine` in a way that it can either be predefined which bot is going to respond to *genmove* commands (the GTP way of asking an engine to generate a move), or alternatively the bot type can be passed in as an argument when starting the Python program. The latter was particularly useful when running automated bot tournaments using the tool *ringmaster*, which is part of Gomill[6], a tool suite for Go developers.

To be able to play against our bots we also implemented a `GTPcontroller` ourselves. It handles the communication between two GTP engines and keeps track of the board state to verify stone placements. In that way we were able to set up games between `HumanGui` (using PyGame[7]) and any one of our bots; see a screenshot in Figure 7. Having a GUI makes playing against bots, or watching them play against each other, much more convenient compared to watching pure GTP commands and the responses to them.

To be able to collect feedback from (skilled) humans playing against our bot, we set up a virtual machine on the LRZ Compute Cloud[8] that allowed anyone to connect to it and start a Go game against one of our bots. Utilizing *X11 forwarding*, it was possible to invoke our GUI over the *ssh* connection.

4.1.2 Learning

The degree of freedom is quite high in various parts of this project; which aspects to train a neural network on (policy vs. value, liberties, other aspects or combinations of multiple ones), how to select the input data (more about that in Section 4.2), and which architecture to choose for the neural network - including the number of layers, their types, number of neurons, types of activation functions, number of training epochs, dropouts, convolutions etc. To allow ourselves to experiment with different approaches but still keep as much code as possible in common, we decided to employ a basic pattern from object oriented programming. Namely extending a base class with specialized classes. In that manner we have a `BaseLearn` class that gets extended by various `Learn` classes, each implementing a different approach. `BaseLearn` provides the mechanisms to connect to the **SQLite** database containing all replayed games (see Section 4.2). Furthermore, it has a method that creates the 8 symmetries (dihedral group D_4) from a given board and controls the training process, including the storage of the network topology as *json* file and the model weights as *HDF5* file.

[4] `www.numpy.org`
[5] `https://senseis.xmp.net/?GTP`
[6] `https://mjw.woodcraft.me.uk/gomill/doc/0.8/ringmaster.html`
[7] `www.pygame.org`
[8] `www.lrz.de/services/compute/cloud_en/`

Figure 7: Our Go GUI

The command to query the database, the processing of the input data (resulting in a `X` and `y` *NumPy* array) before starting the training, as well as the configuration of the network topology is, however, left for the `Learn` classes that extend `BaseLearn` to specify.

We chose to work with existing neural network libraries instead of implementing that functionality ourselves. For the most part we used *Keras*[9] which acts as a convenience-wrapper around libraries like *TensorFlow*[10] and *Theano*[11]. Making use of the backend-interchangeability of *Keras*, we used *TensorFlow* as backend during the training phase and then *Theano* in the playing phase (in which only *predict*-calls are made to the trained model). The reason for this were threading issues between *TensorFlow* and *PyGame*, the library used for our GUI.
The final version of our bot, however, uses *PyTorch*[12] for both the training and for choosing moves during a game.

Depending on the size of the input data and the hardware, training neural networks can take a lot of time. We outsourced our training operations to virtual machines running on the LRZ Compute Cloud (as mentioned above) or on the Google Cloud

[9] `https://keras.io`
[10] `www.tensorflow.org`
[11] `https://github.com/Theano/Theano`
[12] `http://pytorch.org`

Platform[13]. The final bot was trained on the Machine Learning System Nvidia DGX-1 of the LRZ Data Lab[14]. This outperformed our previous training setups by a large degree due to *PyTorch*'s automatic utilization of available GPUs.

4.2 Training Data

We used SGF files from two sources. SGF stands for "Smart Game Format" and is a common format to store Go game records.

1. Our advisor Bernhard Werner provided us with 76440 SGF files of 9×9 games (one game per file) from the Dragon Go Server[15]. The games on this server are publicly accessible.

2. Nathanael downloaded another 344374 SGF files of 9×9 games from the Computer Go Server[16].

Inspecting the data

Inspection of these data sets revealed the following distributions. Figure 8 shows the distribution of number of turns in both datasets. Where dataset 1 refers to source 1 from above, whereas dataset 2 is source 2. In figures 9 and 10 the datasets were merged together and analyzed. Figure 9 shows the scores extracted from the result-field in the SGF files. *B+10* for instance means that the black player won by +10. Consequently, the white player lost by −10 in that example.
Figures 11 and 12 show the distribution of the elo ratings of both players across all games. Elo rating was the first rating system with probabilistic underpinnings. Arpad Elo originally developed it for the game of chess. Note that dataset 1 contained no elo integers and instead a rank string. This was converted, when possible, to an elo rating following the conversion presented on Sensei's Library[17]. That explains the number of missing ratings. The difference of minimum and maximum in the two distributions are therefore to be treated with a critical eye as the conversion from rank to elo is not cleanly standardized.

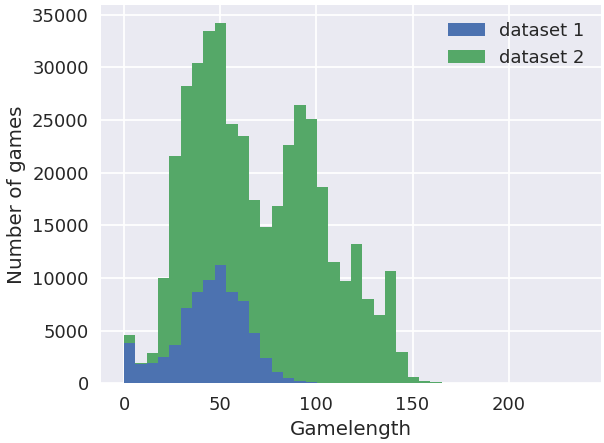


Figure 8: Distribution of number of turns in both datasets

[13] https://cloud.google.com
[14] www.lrz.de/services/compute/special_systems/machine_learning/
[15] www.dragongoserver.net
[16] www.yss-aya.com/cgos/9x9/archive.html
[17] https://senseis.xmp.net/?EloRating

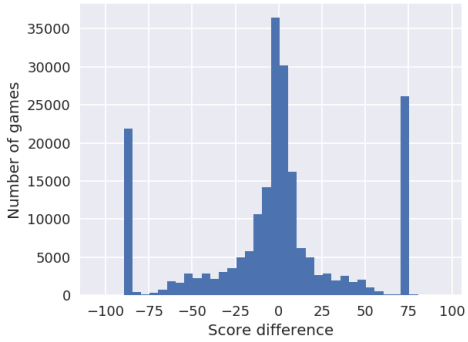


Figure 9: Distribution of game results

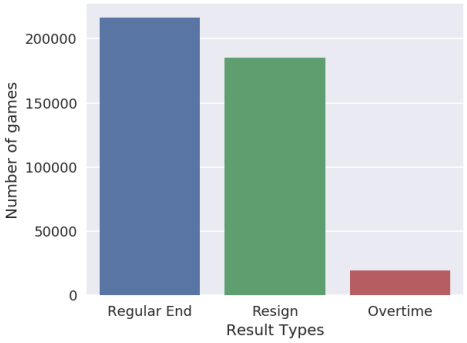


Figure 10: Distribution of result types

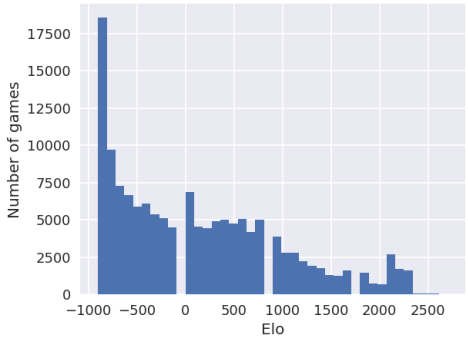


Figure 11: Elo distribution in dataset 1

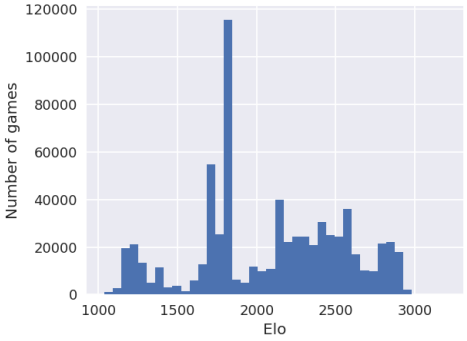


Figure 12: Elo distribution in dataset 2

We wrote code that replayed all of the 420814 games from these two sources using our own Go game implementation described in Section 4.1.1. In each step the board state was recorded and fed into a **SQLite** database. Replaying the games is necessary because the SGF format stores only the coordinate of each stone placement, but not which effect these placements might have in terms of capturing stones. The resulting table of all board states contains $\sim 29.2M$ rows. So, on average one game has ~ 69 moves. For faster retrieval of information from the database we included the index of the next move in each entry of the boards. Not including this would have required a lookup in the *meta* table for each board state; very infeasible. The table structure of our databased looked like the following.

→ A **meta** table containing meta-info about each game and also its SGF content in full - in case a training-approach wants to include information that isn't captured in the columns we already extracted.

→ A **games** table containing all board states of all games.

Once we started training only on games played by strong players (based on their elo rating), it turned out to improve data access time to add another table that has the same contents as the *games* table, but contains only boards from games where both players were above a certain elo threshold. Furthermore, we stored these board states in random order to counteract correlation issues described in 5.1.3.

5 Approaches to Building a Go Playing Bot

Our goal was to create a computer program that plays Go well, only using neural networks, without any form of tree search. But a neural network cannot play completely on its own, there has to be a form of *player logic* that computes the *input* for the network out of the current game state, and which then uses the *output* of the network in order to decide its next move.

As an example, one possible idea might be to have a network use the current board as the input, and output the best move to make in that position. But "best" is not very well defined, and there is no way for us of really having this information. Instead we might want to approximate the behavior by players in our training data, assuming that all of them play somehow reasonable. Approximating this underlying function, using only a partial observation of our whole board state space, is then what we want our network to learn: Given a board, output the probability distribution for the next move. This information can then easily be used in our player logic (bot) in order to decide on the next move.

But there are many possibilities of encoding the board as input, and we might also be interested in *feature engineering* (the hand-crafted addition of input features), and the proposed output type is also only one of multiple possibilities. For the training, *data preprocessing* and even the *data selection* can be varied in order to create different results, and the *training* process itself is also strongly modifiable, using different learning rates, optimizers, losses, number of epochs, and batch sizes. Last but not least, the *network architecture* itself is another crucial aspect of our learning and playing algorithm, and there are many parameters to be chosen (number of hidden layers, layer size, activation functions, initialization, layer type, and many more).

In the following we will explain more about those individual customization possibilities, followed by some empirical results of our project over the course of the semester.

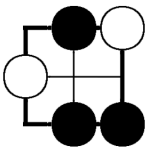
5.1 Variations

5.1.1 Network Input

As we want our network to output relevant information, we want it to know everything about the current *game state*. This game state consists of the current board, but also of additional information like which color is currenty at play and the game history (in order to know about *Ko* situations). Depending on the ruleset even the captured stones should be considered, but as we play with chinese rules this was not relevant for us.
The first and most obvious part of the game state is the current board position, consisting of the locations of black stones, white stones, and empty grid points. We explored ways of representing the board states in understandable ways, in order to use them as input for our neural networks. Some possibilities could be

- $\mathcal{B} \in \{-1, 0, 1\}^{81}$, an 81-dimensional vector, where 1 and -1 represent the black and white stones, and 0 the empty gridpoints, or
- $\mathcal{B} = \mathcal{B}_{\text{black}} \times \mathcal{B}_{\text{white}} \times \mathcal{B}_{\text{empty}} \in \{0, 1\}^{3 \cdot 81}$, binary representations of the board, where 1 stands for a black stone, a white stone, or an empty gridpoint, respectively.

See Figure 13 for an example.



(a) Go board to encode

$$\begin{bmatrix} 0 & 1 & -1 \\ -1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

(b) Simple matrix representation

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

(c) Binary encoding (black, white, empty)

Figure 13: Example Go board with two different representations. Here shown as matrices, but depending on the network they can be transformed into vectors.

As Go has the *ko* rule, the board history is also necessary to define the current game state. But since we implemented the *superko* rule where any reccurence of an earlier board position is forbidden, and since our bot has access to this information in order to only ever consider moves that are legal, we do not use the previous board positions as input.

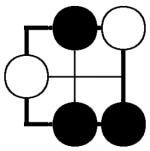
The color of the current player, B or W, is also a suitable input since given the board position, the best move in this situation depends on whose turn it is, and the outcome of the game depends on the color of each player in order to consider *komi*. The information about the current color at play can also be included in a different way: Instead of encoding the board like above, as (*black stones*, *white stones*, *empty locations*), we could also encode them as (*player stones*, *opponent stones*, *empty locations*). The resulting board representation then contains also information about the current color at play.

Another approach we tried is to input the liberties of each position which is encoded as

- $\mathcal{L} = \mathcal{L}_1 \times \mathcal{L}_2 \times \mathcal{L}_3 \in \{0,1\}^{3 \cdot 81}$, where $\mathcal{L}_i, i \in \{1,2,3\}$, is a binary representation of positions with 1, 2, or ≥ 3 liberties, respectively.

$\mathcal{L}_{\text{black}} \in \{0,1\}^{3 \cdot 81}$ and $\mathcal{L}_{\text{white}} \in \{0,1\}^{3 \cdot 81}$ denote the number of liberties of the black stones and the white stones, respectively. See Figure 14 for an example.

Other possible inputs are values describing the game state (number of captured stones, territory counting at this point, result of the game etc.) or the board state (number and size of groups, number of eyes – single empty spaces inside a group etc.). Of course, it is possible to take any combinations of these values as input. However, it is computationally expensive to calculate these numbers, and it is not clear if they are beneficial for the performance of the network.



(a) Go board to encode

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

(b) Binary encoding of black's liberties (1, 2, and ≥ 3).

Figure 14: Example Go board and binary encoding of black's liberties. Again shown as matrices, but depending on the network they can be transformed into vectors.

5.1.2 Network Output and Player Logic

We had two main approaches on the desired functionality of our neural networks: Directly output a good move, or output the win probability at a certain board position for each player. These two types of networks are also commonly used in the literature and are often called *policy network* and *value network*.

The policy network outputs a probability distribution over all possible moves:

- $\mathcal{O} = (p_1, ..., p_{82}) \in \mathbb{R}^{82}$, where p_i, $i \in \{1, ..., 82\}$, is the probability for a move on position i. Move 82 stands for pass.

The value network evaluates the current board position and states the win probabilities for Black and White, according to the game results of this board in our training set:

- $\mathcal{O} = (\text{win}_{\text{black}}, \text{win}_{\text{white}}) \in \mathbb{R}^2$,
  where $\text{win}_i = \frac{\text{number of times } i \text{ wins after this board position}}{\text{number of times this board appears in the training set}}, i \in \{\text{black}, \text{white}\}$.

These two types of networks directly lead to two player logics which we called *policy bot* and *value bot*.
The policy bot uses the output of the network to choose the available legal move with the highest probability, which means in our case the move that humans are likely to play.
The value bot basically performs a depth-1 tree search and evaluates the resulting board for each available move. It then chooses the move with the highest resulting win probability.

5.1.3 Data Selection

Our data consists of 420814 games, or $\sim$ 29.2 million board states. At no point during the semester have we had the chance to use all of the available data for training (the motivation behind this is debatable, but our experience suggests that the networks need a certain complexity in order to play well and this complexity then needs a large amount of data to prevent it from overfitting), so we tried different ways of selecting the data we use for training.

One idea of a "smarter" data selection might be to use only the "best" games, where we need to introduce some notion of what makes a game good. We did so by generating the minimum elo of both players in a game, and we then selected the games with highest minimum elo. The idea seems very reasonable, as we assume that we learn better moves from proficient players than from bad players. The outcome was

surprising: The performance of the trained bots was largely inferior to bots with the same architecture and training size, but that were trained with randomly sampled data. The reason for it is actually quite simple: If we take boards from few games, they will be a lot more similar to each other than if we take boards from completely different games. The data is therefore too correlated, and while the training size itself is the same, the amount of interesting information is lower than when choosing boards randomly.

Something obvious, yet notable, about the game of Go is that it is independent of the viewpoint. This translates to 8 inherent symmetries (dihedral group D4) of the board: Four rotations, and four flipped rotations. We can exploit this property in order to create "free" training data. For each board we choose to select, we can compute the eight symmetries, along with the symmetries of the following move, to then effectively train with 8 times as much data. We used symmetries, although not always, as we were most often limited by the available RAM, which means that we had the choice of including symmetries or increasing our training data size by the factor 8.

5.1.4 Architecture

Our choice of architecture evolved over the course of the semester. In the beginning, we had very small networks with one hidden layer in order to test our whole training pipeline. During most of the semester we worked with sequential neural networks, consisting of fully connected layers with some choice of *activation function*. While neural networks are often introduced with activation functions like the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$ or the tangens-hyperbolicus $f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, the rectified linear unit (ReLU) is currently one of the most used activation functions for deep neural networks: $f(x) = \max(0, x)$. The reason for this evolution is that as networks get deeper the gradients tend to vanish, while the derivative of ReLU is easy to compute as either 0 or 1.
Other than that choice there are the obvious hyperparameters like *number of hidden layers* and the *hidden layer size* that directly influence the number of learnable parameters in a network, i. e. the weights and the biases. Those weights are often initialized with a slightly modified uniform distribution, but we can also choose our own way of initialization, which gives us another additional way of customizing the model.

Apart from the classic sequential neural networks, the deep learning literature developed many advanced network architectures and layer types, often for some specific tasks. While the choice is very large, the most relevant options applicable to Go are networks normally employed for computer vision, notably image recognition. As introduced in 3.2, convolutions are a reasonable concept in relations to Go. Layers like *dropout* (randomly deactivating neurons during training), *maxpool* (only keeping the highest activations in a specific region from the previous layer) or *batch normalization* (on-line normalization of the current batch by using the running mean and variance of the previous observations) are often used in deep convolutional neural networks, often to prevent overfitting or to enable networks with high depth to learn well.

All together, the internal architecture of a neural network leaves many choices to us, and many hyperparameters to select.

5.2 Empirical Comparison of Our First Approaches

We empirically compared some of the variations presented above. We chose to fix the network architecture and the data selection, and wanted to evaluate the performance

results for different input and output choices for the network. The choices of training size and architecture were motivated by our prior explorative work.

5.2.1 Architecture

The architecture was the same for all models: A sequential neural network with three hidden layers, of size $(200, 400, 200)$. We used the *rectifier non-linearity* (ReLU) as the activation function in each layer, and a *softmax* on the last layer.

Input: We selected three different board representations for this comparison (see also 5.1.1):

- "Naive board" - Board as a matrix: $\mathcal{I} = \mathcal{B} = \{-1, 0, 1\}^{81}$
- "Encoded board" - Board as three binary feature planes (black, white, empty), encoded separately: $\mathcal{I} = \mathcal{B} = \{0, 1\}^{3 \cdot 81}$
- "Encoded board with liberties" - Same as above, but additionaly binary encodings of the available liberties (1, 2, or ≥ 3 liberties as binary features): $\mathcal{I} = \mathcal{B} \times \mathcal{L}_{\text{player}} \times \mathcal{L}_{\text{opponent}} = \{0, 1\}^{3 \cdot 81} \times \{0, 1\}^{3 \cdot 81} \times \{0, 1\}^{3 \cdot 81}$

Output: We evaluated both of the output possibilities presented in 5.1.2, the policy and the value approach.

This gives us six different models and bots.

5.2.2 Training

The training data consisted of 400000 board positions, sampled from the full data we have available. We chose not to compute the symmetries, as we were severely limited by our RAM. If we used symmetries, we would have been forced to use less training data, and our assumption was that the information contained in our data selection should increase by choosing completely unrelated boards over the computed symmetries. As a loss function we used the categorical cross entropy, as both the policy and the value approach consist of a classification problem.

5.2.3 Results

We played 200 games against random for each of the six variations. The results can be seen in Table 2.

	Value Bot	Policy Bot
Naive board	79.75%	25.50%
Encoded board	99.25%	27.50%
Encoded board & liberties	93.25%	50.50%

Table 2: **Empirical win percentage** against an opponent that plays randomly (200 games each).

Some unusual findings from this comparison:

- All value approaches perform better than the policy approaches
- Adding liberties to the input seem to have a negative impact on the value bot, but a very positive impact on the policy bot

The results strongly suggest to use the "encoded board". Computing the liberties is computationally very expensive, and the performance gain is only relevant for a policy approach. When evaluating these results it is important to keep our setup in mind: All of these approaches used the same neural network architecture, and the training size was limited compared to later approaches. We are aware that higher performance can be achieved if the architecture is tuned for each approach, but we wanted to get an initial evaluation of the different performances, so that we could then further focus on tuning the architecture for a specific choice input and output.

5.3 Improved Policy Bot

As all the bots mentioned in 5.2 did not manage to win against GNU Go level 1, we focused on optimizing one bot which should perform better.

5.3.1 Architecture

The input is $\mathcal{I} = \mathcal{B} \times \{B, W\} = \{-1.35, 0.45, 1.05\}^{81} \times \{B, W\}$, a naive board (black = -1.35, empty = 0.45, white = 1.05) including the information about the current player, and the output is the move with the highest probability, which means that we have a policy bot. We chose slightly different values for the encoding so that the weights are not multiplied by zero for an empty location.
We still use three hidden layers of sizes (200, 400, 200), but now with a modified weight initialization, where the weights are initialized according to a normal distribution with mean 0 and standard deviation $\frac{1}{\sqrt{n_\ell}}$, where n_ℓ for layer ℓ is the number of neurons from the layer before, which act as the input neurons for this layer.

5.3.2 Training

We trained this bot on more data, namely 1 million board positions, since using more data seems to increase the output accuracy of the bot. The 8 symmetries for every board position are also included. The number of epochs were set to 200 while batch sizes of 1000 were used. The loss function is again the categorical cross entropy.

5.3.3 Results

We evaluated this policy bot against random, GNU Go level 1, and GNU Go level 10. The results are shown in Table 3. It is an improvement compared to the policy bots in Table 2, but it is still possible sometimes to win against this bot by playing randomly. Interestingly, it performs worse than the value bots against random, but a lot better against GNU Go level 1, which the value bots could not win against once.

Opponent	Win percentage
Random	85.00%
GNU Go level 1	21.25%
GNU Go level 10	7.25%

Table 3: **Win percentage** of our improved policy bot against GNU Go level 10, GNU Go level 1, and against random, over a 200 game evaluation.

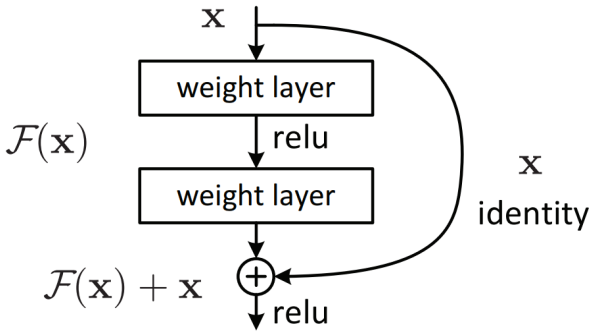


Figure 15: Basic building block in residual networks [He et al., 2015]. In our case the "weight layer" is the 3×3 convolutional layer.

5.4 Final Version: Convolutional Residual Network

5.4.1 Architecture

We tried out many different architectures, but the first results where underwhelming. The number of parameters feels even larger than when working with regular neural networks, so design decisions are even more difficult to make.

Neural network architecture often comes down to experience, research, and a lot of trial and error. Current literature specific to deep learning approaches to Go, but also more general deep learning and computer vision related articles, present many examples of neural network architectures. We decided to implement a modified version of the network used by AlphaGo Zero [Silver et al., 2017].

Input to the neural network is a $2 \times 9 \times 9$ tensor consisting of two 9×9 binary feature planes. The first plane consists of binary values indicating the presence of the current player's stones, the second plane represents the opponent's stones.

Architecture: The input is processed by the neural network, which consists of a single convolutional block, followed by 5 residual blocks (see also Figure 15 for a schematic), and the policy head. Each part will be explained in the following.
The convolutional block consists of:

1. Convolution: 256 filters, kernel size 3×3, stride 1, padding 1
2. Batch normalization
3. ReLU: Rectifier non-linearity

Each residual block consists of:

1. Convolution: 256 filters, kernel size 3×3, stride 1, padding 1
2. Batch normalization
3. ReLU: Rectifier non-linearity
4. Convolution: 256 filters, kernel size 3×3, stride 1, padding 1
5. Batch normalization
6. Skip connection back to the input of the current block

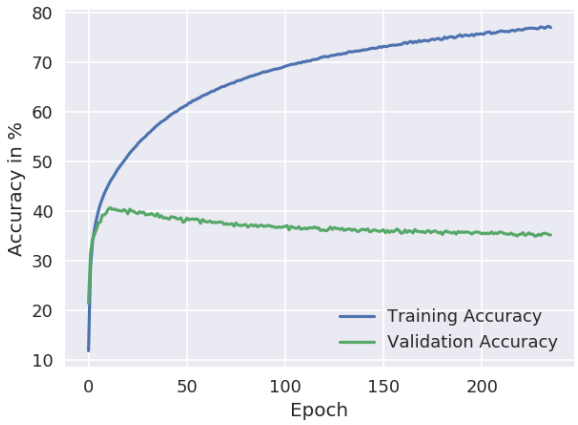


Figure 16: Training of the final convolutional neural network

7. ReLU: Rectifier non-linearity

The policy head consists of:

1. Convolution: 2 filters, kernel size 1×1, stride 1, padding 0
2. Batch normalization
3. ReLU: Rectifier non-linearity
4. Fully connected linear layer of size $9^2 + 1$

Output: The output of the network is a vector of size $9^2 + 1$, which corresponds to the logit probabilities for all grid points and the pass move.
The bot logic: The resulting bot is a classical policy bot. The neural network outputs move probabilities, and the bot plays the legal move with the highest probability.

5.4.2 Training

We trained the model on 5 million board states, randomly sampled from our available data, using categorical cross entropy as the loss function. The training for our final version took three hours on the Machine Learning System Nvidia DGX-1 of the LRZ Data Lab [18]. Figure 16 shows the training progress over time. Note how for the first few epochs both the training accuracy and the validation accuracy increase steeply, but at epoch 9, the validation accuracy has already decreased compared to epoch 8. Epoch 11 gives the best validation accuracy at 49.36%. Note how even in later epochs the training accuracy still strongly increases, while the validation accuracy slightly decreases, as our model is able to overfit the training data while losing its generalization capabilities. Further trainings with more training data might circumvent this problem and allow for longer training, but we were limited by the required training time.

5.4.3 Results

Using this network architecture, together with an increased size of training data, drastically changed the performance, both evaluated by its win percentage against

[18] `www.lrz.de/services/compute/special_systems/machine_learning/`

other Go programs and from our human perspective on the gameplay. It seems to play less randomly, and it is easier to understand the ideas and mistakes of the bot. We played 1000 games against random, GNU Go level 1 and GNU Go level 10. The results can be seen in Table 4.

Opponent	Win percentage
Random	100.00%
GNU Go level 1	67.50%
GNU Go level 10	63.30%

Table 4: **Win percentage** of our convolutional bot against GNU Go level 10, GNU Go level 1, and against random, over a 1000 game evaluation.

5.5 Other Experiments

Reinforcement Learning

Reinforcement learning is a concept in behavioral pyschology, where an agent learns to perform a task by maximizing some idea of a reward. In broad terms, the computer intitally starts off by making random moves, and then slowly tunes itself to make moves that result in a higher reward. The trade off between the exploration of available options and exploitation of known moves is controlled by the Epsilon Greedy Algorithm (in this case). In order to demonstrate a proof of concept, we worked on Atari Go, a subset of the Go game, in which the point is to capture at least one enemy stone. The simple Q learning algorithm in combination with a neural network was used.

The Q learning algorithm is a method by which the computer learns to recognize states, keeps a record of these states, and computes the best course of action to take in order to maximize the reward. For this trivial case of surrounding one stone, we chose a reward of +50 for a surrounded stone, and -1 for any other move leading to a non-terminal state. It is obvious that even for the trivial case of one stone, the computer requires a lot of training due to the large search space for Go.

After 200,000 games, the computer learned to surround one stone placed randomly. This training took approximately 4 days to complete. Due to time constraints and the absence of fast code, this course of action had to be abandoned.

6 Conclusion and Outlook

The goal of this project was to explore ways of implementing a bot that plays Go reasonably well using neural networks. We have built a Go bot that wins approximately 66% of the times against GNU Go, which is a fairly strong Go playing bot. The final bot can be described as a human player who has learned the rules of the game, and is beginning to grasp the major ideas of Go.

It was essential that in order to find reasonable approaches to machine learning in Go, we had to first understand the game of Go itself. We spent a few weeks familiarizing ourselves with the the basic ideas of Go, “Think globally, play locally”, the various rules, and so on.

A major part of the project was spent building an engine that can handle a game of Go and who's code is readable and modular. To make the game handling more visually appealing, we added a GUI. Go client software uses a communication protocol called GTP that lets us communicate with our bot and other bots like GNU Go, which we had to implement ourselves.

During the course of this project we understood some of the essential ingredients that make for an intelligent Go playing machine. We were clear about the route we wished to pursue, namely using one artificial neural network. The neural network we trained using data from Go servers resulted in a bot that played better than random, but bad overall. It was concluded that in order to obtain the best performing bot, we needed large amounts of data, and better approaches at deriving useful (training) information from it - i. e. value and policy approaches. We chose to optimize the policy bot and it gave us good results. Finally, we chose a convolutional neural network approach for our bot, and that provided by far the best results.

Literature relating to computer Go suggests trying Monte Carlo tree search as well as reinforcement learning as the next logical step. Due to lack of time and code inefficiency, we were unable to fully implement these methods.

We thank the Leibniz Supercomputing Centre (LRZ) of the Bavarian Academy of Sciences and Humanities (BAdW) for the support and provisioning of computing infrastructure essential to this project.

References

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, abs/1512.03385.

[Kocsis and Szepesvári, 2006] Kocsis, L. and Szepesvári, C. (2006). Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg. Springer-Verlag.

[Silver et al., 2017] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. (2017). Mastering the game of go without human knowledge. *Nature*, 550:354–.

[Tromp and Farnebäck, 2007] Tromp, J. and Farnebäck, G. (2007). Combinatorics of go. In *Proceedings of the 5th International Conference on Computers and Games*, CG'06, pages 84–99, Berlin, Heidelberg. Springer-Verlag.

[Werbos, 1974] Werbos, P. (1974). *Beyond regression : new tools for prediction and analysis in the behavioral sciences*. PhD thesis, Harvard University.

[Wolfe, 2002] Wolfe, D. (2002). Go endgames are PSPACE-hard. *More Games of No Chance*.

[Zobrist, 1970] Zobrist, A. L. (1970). *Feature Extraction and Representation for Pattern Recognition and the Game of Go*. PhD thesis, The University of Wisconsin - Madison. AAI7103162.