Bachelor's Thesis

New approaches in Go game AI with Deep Convolutional Neural Networks and Tree search

Falk Heuer
Supervised by Leon René Sütfeld and Gordon Pipa

12.9.17

UNIVERSITÄT OSNABRÜCK

I hereby certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other university.

Osnabrück, 12.9.17

Signature:

Table of contents

1 Introduction **3**
1.1 Abstract 3
1.2 Introduction 4

2 The Game of Go **5**
2.1 Rules 5
2.2 Tactics 7
2.3 Ranks 8
2.4 Handicap 9
2.5 Go and Tradition 10

3 Computer Go **11**
3.1 A Brief History of Computer Go 11
3.2 Deep Neural Networks 12
3.2 a) Architecture 12
3.2 b) Training 13
3.3 Convolutional Neural Networks 16
3.4 Monte Carlo Tree Search 17
3.4 a) Introduction 17
3.4 b) UTC 18
3.5 The AlphaGo Player 19
3.5 a) Policy and Value Networks 19
3.5 b) Fusing the Algorithms 21
3.5 c) Strength 22

4 Methods **24**
4.1 Environment 24
4.2 Data preparation and augmentation 25
4.3 Network Structure 27
4.3 a) Policy Network 27
4.3 b) Reinforcement Learning 28
4.3 c) Value Network 29
4.3 d) Meta Network 29
4.3 e) Player 30

4.4 Results....31
4.4 a) Policy Network....31
4.4 b) Value Network....32
4.4 c) Meta Network....33

5 Conclusion....34
5.1 Summary....34
5.2 Further Work....34

6 Appendix....36
7.1 Tensorboard Graphs....36
7.2 Go in Korea....39

7 List of Abbreviations....43

8 Bibliography....44

Abstract

Go has long been considered a hard game for computer agents. Recent advances have pushed development to new heights and in 2016, a computer player firstly beat a top human player in Go. In this thesis, I portray the history of Go and computer Go and elaborate the core algorithms that operate in modern computer Go agents, namely Monte Carlo Tree Search and Deep Convolutional Neural Networks.
Further, I present my computer Go player '*Entropy*' and examine 3 neural networks. A 'policy net' for move selection and a 'value net' for move evaluation (terms coined by Silver et al. [1] and since used in the computer Go scene consecutively). The policy network is trained to predict moves of professional players and can thereby play itself. It reaches a playing strength of an average human club player without adding a tree search. The value network predicts the outcome of a game given a certain game position and can compare different positional branches in a tree search. It partially works but succumbs to overfitting, an issue owed to the computational limitations of the hardware setup.
Finally a novel, merged network called '*meta network*' is examined, which trains both neural networks simultaneously and shares variable numbers of hidden layers, proposing a more biologically inspired and technically simplistic approach. This network performs well and delivers very promising results for implementation in a neural network + tree search bot.

Chapter 1

Introduction

Games portray a perfect playground for Artificial Intelligence Research. With their finite, well defined set of rules and clear definition of states they represent a suitable simulation of the real world with varying levels of difficulty for computer agents[2].
For many games, simple backtracking or minimax algorithms deliver good results. Most board games fall into this category[3], even chess with a relatively high branching factor[3] can be approached in this fashion. For Go however, the branching factor is of such a high magnitude that classic Minimax fails entirely[4]. To put things into perspective, the Go Board has 19^2 fields, creating a grid with 361 possible moves and 10^{170} legal positions – more than the number of atoms in the universe.
Two Algorithms have been found that are capable of handling Go games: Monte Carlo Tree Search (MCTS)[5] and Deep Convolutional Neural Networks (DCNNs)[6]. MCTS generates games by random play in the search space and evaluates moves by the number of wins these games result in, hereby creating a probability matrix over the board. Neural networks can be trained to predict moves played by strong players and also generate a probability matrix for move prediction over the board. When combined properly, these methods can achieve superhuman play in Go[1].

In late 2015, Silver et. al published an agent that was capable of beating a professional Go player, the first of its time [1]. In March 2016 their agent AlphaGo beat Lee Sedol, considered by many as one of the strongest Go players of our time.

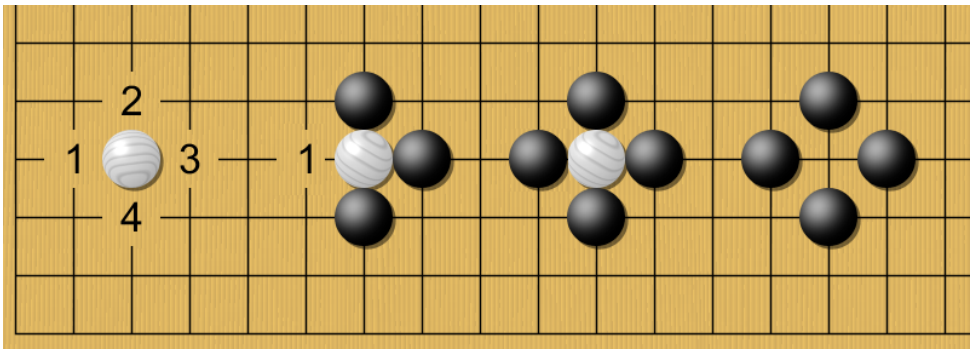
Chapter 2

The game of Go

2.1 Rules

Go is one of the oldest games in history, dating back as far as 2,500 years[7]. It is believed to have originated in China, but quickly spread to Japan, Korea and other Asian countries, having a long tradition in all of these countries. Go is played on a 19 by 19 grid, where two players place a stone of their colour alternating on the intersections. The goal is to make more points than the opponent, which are gained in two ways: By surrounding empty areas on the Goban (board) and by capturing opponent stones.

Stones that touch each other through a line (neighbouring stones along the 4 cardinal directions) form groups. Every group has a certain amount of lines touching fields that are still empty, which are called liberties of that group. Opponent stones that touch the group along the lines remove its liberties. If a player manages to reduce the liberties of an opponent group to 0, this group is considered dead and taken off the board. Every captured stones counts as one point. It is illegal to play suicidal moves by reducing the liberties of an own group to 0. On the borders of the Goban, fields have only 3 or even 2 liberties, unlike the 4 they posses in the centre.



Ill. 1: Liberties | Atari | Capture | Result

A special situation is a 'ladder': A group is in 'Atari', i.e. has only one last liberty before it dies, player A extends the group at that place to get 2 liberties, but player B plays in such a way that it is in Atari again and a chain of Atari sequences will follow. Unless if the player that is being 'laddered' has a stone along the path of the ladder to break the ladder (henceforth called ladderbreaker), he will eventually touch the border and the group dies. Ladders play a significant role in local patterns, and despite them not being played out, their result must be known to evaluate the status of the group in question.

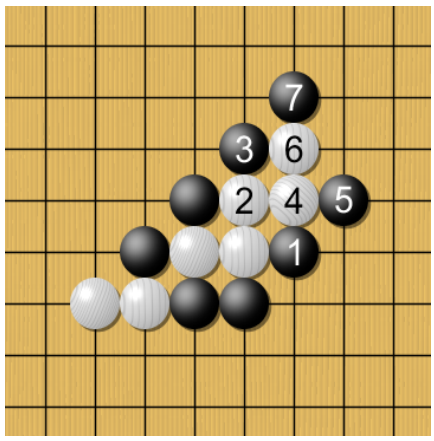


Illustration 2: A ladder

As the game progresses, stones will surround areas in which it is impossible to survive for an opponent stone, since a superiority of one colour makes it easy to capture any opponent stone played in this area. Such area is considered territory, and every empty field counts as a point for the player[1].

Black always plays the first move of the game. This has a traditional origin[2]. Because of the advantage to play first, white is granted a certain amount of points in advance, called 'Komi', usually ranging between 6,5 and 7,5 points, occasionally being set to even 8 or 10.

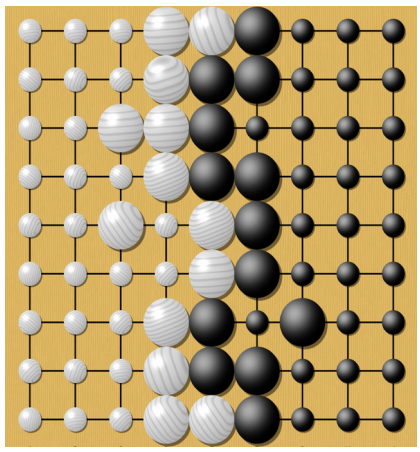


Illustration 3: Counting territory. Every small coloured dot represents a point for the respective player.

There is a special situation in which one player is able to capture a stone, and in the next move the other player can recapture his stone, as both stones have exactly one liberty. To prevent the game from becoming infinitely long, repeating board positions are prohibited and such a situation is called 'Ko'. Before recapturing the stone, a player has to play

[1] Under Japanese rules. Chinese and Ing rules count the empty areas plus the own stones on the board and disregard captured stones, which effectively results in the same score ±1 (depending on the player that plays the last stone). Since computers that require counting points such as MCTS bots are easier to program on the latter, disregarding capturers, many computer players play under Chinese rules, while most European players learn Japanese rules.

[2] Black and white symbolize Jing and Jang, and the game is supposed to symbolize an alternating flow between the two. Initially, white was the weak colour and black the strong one, as it is often seen in the oriental culture like in Judo belts. But since the white stones are – also traditionally – made from shells, a difficult and expensive procedure, while the black stones are made from slate, the stronger player should have the honour to use the more valuable stones.

elsewhere, only then (when the global board position has changed) is he allowed to capture the stone in the Ko again. If the Ko is an important place to play, he will make a Ko threat: He threatens to do harm elsewhere, and if the opponent answers at that place, he can play the Ko again. Now it is the opponents turn to play a Ko threat in order to continue fighting over the Ko. Eventually a Ko threat will be ignored and one player gets to finish the Ko while the other player gets what he threatened.

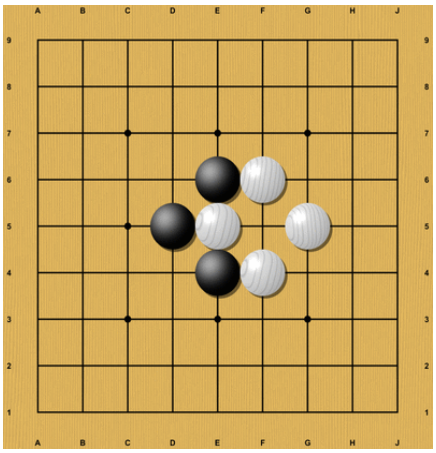


Illustration 4: A Ko

2.2 Tactics

So far, the basic rules have been covered. In order to understand the game more profoundly, a lot of higher-level concepts build on these rules, some of the basics will be explained here.

At the beginning of the game, the whole board is empty, and players will try to make territory as fast as possible. To surround an area, the highest amount of stones is required in the middle, and due to the borders, the lowest amount is required in the corners (Ill. 5).

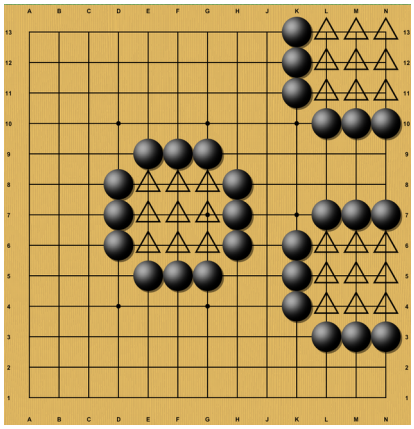


Illustration 5: Territories

This is why in the beginning of the game, players will almost always first play close to the corners, followed by the sides and the centre. This early stage of the game is called 'Fuseki', with local corner patterns being 'Joseki'[1].
When stones of different colour touch, fights will emerge, and players will need to smartly handle their groups to try and dominate the game. The goal here is not limited to capturing stones alone: Having stones "facing" empty areas can be a good basis to either make territory or be able to attack harder when the opponent tries to conquer the area. Splitting opponent groups is just as important as managing own groups, preventing them from becoming too many to handle and whenever possible, connecting important groups. Often it is better to sacrifice single stones (or even small groups) to strengthen the remaining groups and keeping track of the bigger picture.

When a single stone is touched (played next to it), the first intuition is to touch the other colour yourself and weaken it thereby: Its stone now has only 2 liberties left, while the own stones each have 3. Another option is to extend to the attacked stone, making it a

[1] Japanese terms are being used here due to their popularity in the western world.

larger, stronger group: This group now has 5 liberties. The player that initiates such a fight will always have the disadvantage of one stone, since the first opponent stone is already there. Thus, starting fights should be well planned. At the beginning of the game, both players will play away from the opponent stones, and only when space becomes sparse or local patterns come close fights will emerge. Often, stones will lightly dance around each other until a player touches the opponent (e.g. to cut stones apart) and a fighting sequence follows. When a player has many stones in one area it can also be helpful for the intruder to touch stones in order to achieve traction: A single stone is selected and attacked with the intention to put importance into this stone, such that the own, outnumbered stones can build on that attack and get stronger. Of course, not all own stones will survive this intrigue, but its single purpose is to get one, *any* living group in this area of opponents strength. This technique is called 'Sabaki'.

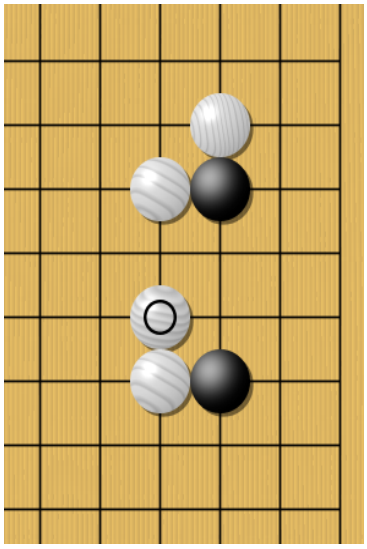


Illustration 6: Attachments

Living groups are the counter part to dead ones (being either actually captured or so weak that it is futile to try and save them, hence also called 'dead' or 'prisoners'). A living group has 2 individual 'eyes': Empty fields inside of the group. If there is one (or none) eye in a group, it will be surrounded and eventually captured. A one-eyed group can be captured by playing in the eye last, making this move valid by taking off the opponent stones. Even if the eye is bigger than 1 field, playing inside will either kill the group with the next move or let the stone inside be captured (a 'throw-in') to capture the whole group again by playing at the very same spot. In living groups, this is different: There will always be an eye remaining that cannot be simultaneously filled as the other one. Once groups surpass a certain threshold of inner size ('eyespace') it is not necessary to define 2 separate eyes anymore because the area can be divided into at least 2 eyes at any time should it be required. This is why big groups don't have well defined eyes. When eyespace becomes too big though, it can get possible for the opponent to place an entire living group inside of the group.

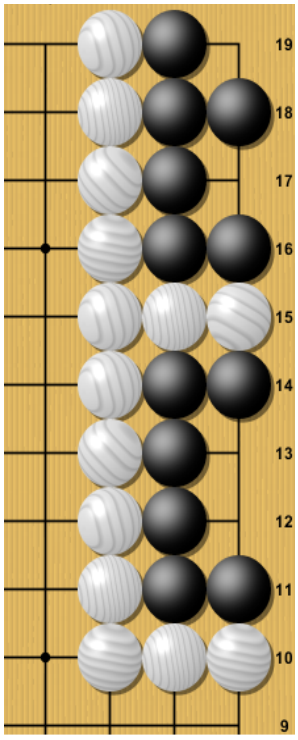


Illustration 7: Living group (top) and dead group (bottom)

Another kind of life is 'Seki': A situation in which two opposing groups share 2 or more liberties (or one eye each and 1 or more liberties), yet no other liberties. The player who plays first reduces a mutual liberty and allows the opponent to capture his group when he plays. Henceforth, no player reduces the liberties and both groups are considered alive. There are more forms of life, like eternal life, life with 2 false eyes and life in double Ko, yet they occur less frequently and would go beyond the scope of this thesis.

2.3 Ranks

Like in Chess, Go players strength can be rated with an ELO System. Higher values mean stronger players, and ranks change after playing games. The further an opponents rank is away from the player, the bigger the change. Traditionally, Go players have Kyu (Pupil) and Dan (Teacher) ranks, ranging from 30k (beginner) to 1k (fairly strong club player) via 1d to a value between 6d and 9d[1]. Parallel to amateur ranks, many countries have professional ranks which lie within top amateur strength and above. Professional ranks are gained by professional teaching, examinations, and increased only via special events like winning tournaments. For that reason a 9p (9. professional Dan) does not relate to a 1P in playing strength as a 9d (9. amateur Dan) relates to a 1d. The offset between 2 amateur ranks has a practical background: One rank equals one handicap stone in a game.

Illustration 8: A traditional Japanese Setup. Source: [27]

2.4 Handicap

Since playing strengths variate vastly and amateurs rarely find a similarly strong opponent, a handicap system facilitates fair games. In official games, particularly professional games, handicap is not used, but in order to make fair and interesting amateur games it is a widely used method. If the players ranks deviate 1 stone, the Komi is removed[2]. For 2 ranks or more, the difference of ranks is put on the board in the form of black stones and Komi is set to 0,5. A 6d could for example give a 2d 4 handicap stones. These stones give the weaker player an advantage from the beginning of the game. The place for the handicap stones for black follows strict rules in the Japanese system, in Chinese rules black can choose the place of the stones.

[1] In Korean and Chinese ranking systems, 6d is the strongest conventionally achievable rank, while 7d players need to win special tournaments to rank up. European ranks peak at 7d, online Go servers often set a maximum rank of 9d. Upper boundaries for ranks can lead to a skewed ranking system when many players on one rank push weaker ones downwards, as it sometimes happens in China with 6d or in Holland with 1d. The other effect, few players taking a broad range of ranks resulting in easier ranking up can happen too as in higher Dan ranks on some Go servers.
[2] Instead of 6,5 or 7,5 points it is lowered to 0,5 points. The half point is a common practise to evite draws called 'Jigo'.

2.5 Go & Tradition

Over the course of more than two Millennia, Go has manifested itself deeply in the culture of various oriental countries. China and Japan both claimed to be the origin of the game for a long time and it is now believed to have been invented in China and soon after brought to Japan.

In China, Go was first played with two black and two white stones in each corner, and only later did the empty board (and with it a more fee opening) get introduced. Early Go boards also varied between 15, 17 and 19 fields in length, while today there are 9 and 13 fields in length for beginners and 19 for normal games. The Ko rule joined the basic set of rules later to prevent Jigos (draws). In the 18th century, Honinbo Shusaku won almost all of his games with black, which did not only lead to a new view on the opening of the game (the Shusaku-Fuseki) in which a lot of low stones[1] were played in such a way that black keeps trying to maintain his advantage points-wise while white needs to play very active to break through it. It also led to the invention of Komi later, a compensation for white not to play the first move. In the 19th century Go Seigen, a Chinese Go prodigy revolutionized the opening by introducing the Shin-Fuseki and focussing more on 'Moyos'[2] opposed to making territory along the sides. It is believed by many that with the breakthrough of computer Go to professional level, another Go theory revolution will follow, and indeed professionals already intensively study some new moves that AlphaGo came up with.

[1] Stones on the 3rd line counting from the border or closer to the border.
[2] Big areas that hold potential territory.

Chapter 3

Computer Go

3.1 A Brief History of Computer Go

Go programs date back as far as to 1968 [8] when Albert L. Zobrist published a paper about pattern recognition. Until the 21st century, Go engines were below amateur Dan-level. Mostly, Go programs deployed hand-crafted pattern databases to recognize local good moves in combination with opening dictionaries for global moves among other heuristics. A popular Go program is GnuGo [9] which plays at a level of 5k and is often used as a reference player for stronger, contemporary programs.

Due to the combinatorial explosion of board positions in Go, it is impractical to tackle the game with classical tree search algorithms. In the 00s, Monte Carlo Tree Search (MCTS) bots were the first to reach amateur Dan level. The popular open-source bot 'Patchi' is such a program. Patchi reaches 4d on a distributed machine [10]; commercial programs such as Zen and CrazyStone reached up to 6d amateur level around 2010-14. MCTS bots create search trees by selecting moves randomly and generating whole games this way which can be evaluated in terms of win-ratios. Their strength heavily depends on the effectiveness of their heuristics for initially selecting moves. Patchi uses a large pattern database to detect shapes. Despite this, global shapes can be very complex and a pure MCTS bots style looks "ugly" to a human player because they often select suboptimal shapes. By simulating thousands of games, their look-ahead is what compensates this.

Neural networks have been successfully used since 2015[6]. At the European Go Congress in Liberec, Check, in July 2015, Deep Convolutional Neural Networks (DCNNs) were discussed in a presentation as promising players with a then-strength of 2k. Convolutional networks are particularly well suited for pattern recognition with large amounts of input data and abstract feature extraction. They are mainly used for image recognition techniques. For Go, they solved the problem of search breadth, replacing all heuristics for both global as well as local patters by outputting a probability distribution over the Go

board for move candidates. Pure DCNN bots will generate "pretty" moves to a Go players eye, generating nice shapes and following Go proverbs about positional judgement. However due to their lack of a lookahead, some moves may simply not work in the long run.
When using a DCNN as a heuristic for MCTS trees Go bots can reach professional level. AlphaGo did so in October 2015 by winning a 5-game match 5-0 against European Champion Fan Hui [1] and later, after improving the ANNs precision with self-play, 4-1 against one of the Top Professionals, Lee Sedol, in March 2016 [11].

The game AlphaGo lost against Lee Sedol was discussed controversially among amateurs and professionals as AlphaGo seemed to have made a mistake when marching towards a solid win. In move 78, Lee played a complicated unusual move to try and catch up a game he was about to lose. Due to its unusual nature, AlphaGo didn't consider this move in previous simulations and when it was played, AlphaGo misinterpreted the situation and started a series of bad moves leading to a defeat [12]. Demis Hassabis[1] later revealed in a talk at NIPS that AlphaGo's neural networks were specifically trained with adversarial learning against such moves.

After the publication of the AlphaGo player, additional Go Bots achieved Professional Level and beat human Pros, such as Zen in its Deep-learning variant DeepZen [13], CGI[14], and the Chinese Chat-company Tencent's "Fine Art" [15].

3.2 Deep Neural Networks

a) Architecture

Artificial neural networks (ANNs) are an area of the field of machine learning. They are motivated by biological neurons and have some structural similarities, intended to learn complex relations within data semi-autonomously. An ANN consists of a layer of units ("neurons") x representing input features and another layer y representing the output, connected with weights w ("synapses"). Input and output can be any type of data as long as it can be transformed to numerical values which the network can process. An ANN can be viewed as a black box connecting input and output with its weights in such a way that the function that maps $X \rightarrow Y$ is optimally resembled.

[1] CEO of AlphaGo's creator company DeepMind

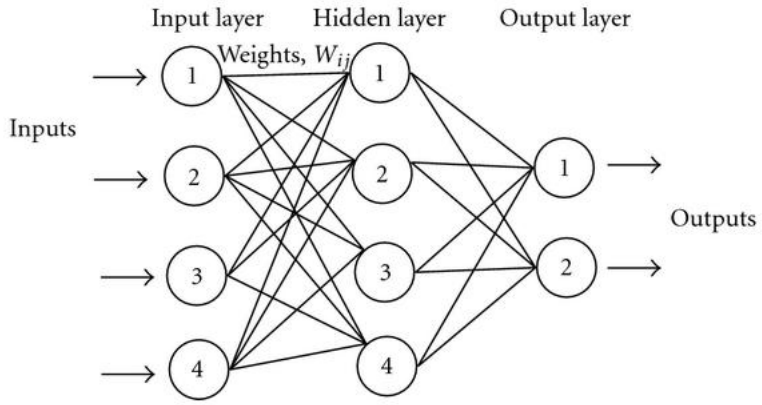


Illustration 9: Architecture of a fully connected deep neural network

Between input and output layers there can be arbitrarily many hidden layers. Layers will be sparsely or fully connected to one another, classically to the following (n+1) layer and only rarely within itself (e.g. in reservoir computing) or backwards (i.e. in recurrent neural networks). Every weight will later be learned and adjusted in accordance with the data. Additionally, a bias b can be added that linearly shifts the neuron in a direction. In such feedforward networks, the value of an individual neuron is calculated by:

$$y_i = \sum_{j=1}^{I} (W_{i,j} x_j) + b_i \tag{1}$$

Here, the ith neuron y is calculated by the sum over the values of all previous neurons X each multiplied with their respective weight that connects to y_i from the weight matrix W plus the respective bias b_i. In order to compute non-trivial problems with multilayer networks, a non-linear activation function is required after each layer of neurons. Else, the network will be mathematically equivalent to a single layer network or linear transformation [16]. For example, a biologically plausible activation function is the rectifier linear units (ReLU), defined as:

$$f(x)=\max(0,x) \tag{2}$$

Just like in biological neural networks, a ReLU solely regards positive output values (Illustration 10). This also aids preventing problems like the vanishing and exploding gradient[17][1]. However, the neglected negative values will be lost. One approach to maintain negative values are leaky ReLUs, which severely downscale negative values instead of nullifying them.

[1] Issues originating from gradient descent learning during training, see there.

b) Training

Once the network is set up, its weights needs to be trained. They will be initialized with random values scaled by a for the specific network suitable factor. It is important to note that initialization with 0 or most non-random values would prevent the network from learning due to its inability to differentiate between neurons in the training phase.

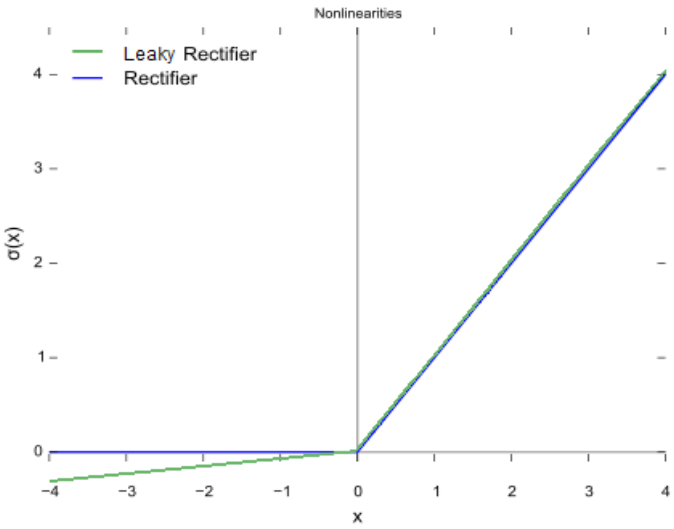


Illustration 10: ReLU compared to Leaky ReLU. ReLUs null negative values while Leaky ReLU transform them. Original image credit: [17]

The network will then be shown input-output pairs either in a supervised manner, i.e. with previously labelled pairs, or unsupervised, in which the labelling of the input with output states is done internally.

In the ***forward pass***, values for neurons will be calculated according to (1) starting from the input going through all layers up to the output until an output value y' is reached. It will then be compared with the goal value y and an error E is generated as part of the ***backward pass***. In order for the network to be more likely to output the correct value for this training sample by hopefully learning the correct transformation function, the weights in the last/nth layer will be adapted slightly in the direction of y. To get this direction, the gradient from the current output towards the goal output is calculated. The gradient is a multi-variable generalisation of the first-order derivative of the weights. To this end, the weights are updated according to:

$$a_{m+1} = a_m - \gamma \nabla F(a_m) \tag{3}$$

with a_m denoting the current position of values in the n-dimensional weight space, a_{m+1} the position after updating the weights, γ the learning rate and $F(x)$ the function the layer represents. The result of the weight update yields updated values for the neurons in the n-1th layer, which themselves can be compared to their former states, a gradient is generated, weighs adapted and further back-propagated up to the input layer. This process is called ***gradient descent***.

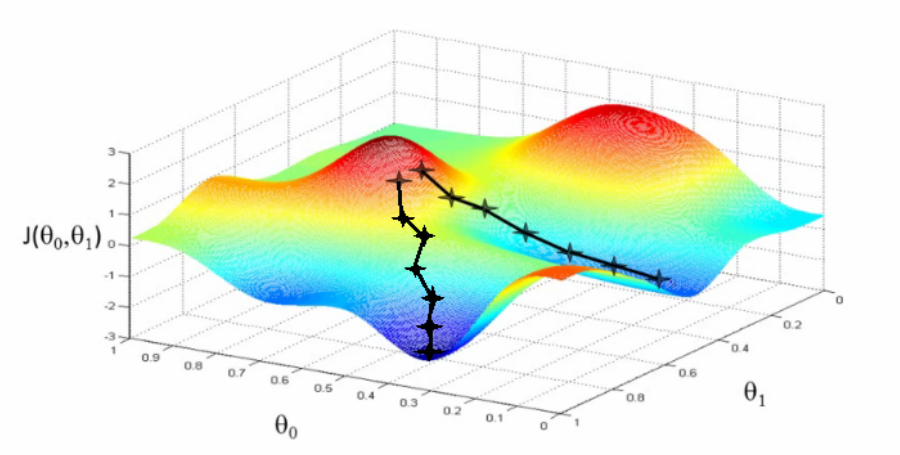


Illustration 11: A graphical representation of gradient descent. Every x denotes a value of the weights (in this example 2) that are moving towards a minimum in the function space (the coloured plane). Depending on training examples and initialisation, different local minima (blue valleys) can result.

The higher the learning rate γ, the faster will it approach the current training example and move towards convergence of the function, but with a too fast learning rate, the network will *overshoot* the minimum and possibly never converge. For this reason, initially relatively high learning rates are chosen (typically not more than a few percent) and later gradually decreased in fine-tuning until convergence.

As can be seen in Illustration 11, the function space can have many possible local minima. To avoid converging to a non-optimal local minimum, the gradient descent algorithm can be modified to incorporate a term for momentum. This term modifies the learning rate in such a way that: when past gradients were high, learning rate is increased, and it lowers it when training stalls. One implementation for this method is the Adam (Adaptive moment estimation) optimizer. Besides Adam and Standard Gradient Descent, there are many optimizers to choose from for different network tasks.

In supervised learning, the quality of the network output can be directly measured by the prediction accuracy of the network, whereas unsupervised networks will require a different parameter of goodness due to its architecture. In Go, this parameter can be the players strength. As in all function approximation tasks, it is necessary to set aside a small test set from the data before training since neural networks are prone to overfitting. The test set is not used for training and can give an unbiased estimate of the goodness of the model.

3.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) use convolutional filters instead of fully connected (FC) weights between layers. There are two main motivations for convolution and against FC layers: shapes in images as well as on the Go board consist of local patterns and only in higher level abstractions relate to global patterns. Convolutional filters connect only spatially close elements, while further elements will be connected in later layers. And second, FC layers with high dimensional input as they are in object recognition have an exponential, FC intractable computational cost. For instance, a 300 times 300 pixel image with 3 colour channels will have 7,3E10 weights in the first layer if layer 1 is chosen of the same size as layer 0. Convolutional networks can be much sparser by focussing on local connectivity.

A convolutional layer learns a set of N_k filters $F = f_1, \ldots. f_{Nk}$, which are convolved spatially with input layer x and create N_k feature maps y:

$$y_k = f_k * x. \tag{4}$$

with * as the convolutional operator. Each filter is convolved over the whole input tensor and uses the same weights to compute a single patch of the tensor. This way, shapes are learned in equivariance, or in other words will be recognized irrespective of their location in the input tensor.

At the edges of the Go board or picture, filters will reach only up to the filter size, but not at and over the edge to recognize (partial) shapes here. Zero-padding enlarges the input tensor with zeros for filters to reach over the edges. For other applications, other forms of padding can be used.

Another parameter for convolution is its stride. Classically, a filter will have a stride of 1 and generate a layer of the same breadth and height as the input ± the filter size depending on padding. With a stride of 2 the filter will jump every second neuron and the output tensor will be 1/4th the size.

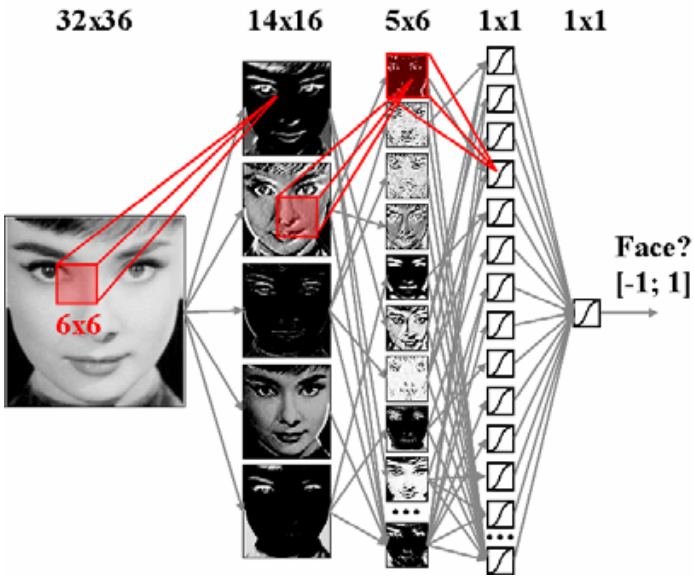


Illustration 12: A deep convolutional neural network with various filter sizes for face recognition. Source:[18]

3.4 Monte Carlo Tree Search

3.4 a) Introduction

Neural networks reduce the search breadth to a handful of candidates for suitable moves. Many moves, particularly in local fights, need to be evaluated in depth by "reading" follow-up moves of the opponent and oneself. While humans can evaluate follow-up positions directly, for a computer agent the board position and captured stones do not suffice to immediately deduct a sensible evaluation, for local fights are usually of much more importance than an estimate of current points alone.

Therefore, the agent will evaluate games by creating a tree search and evaluate positions depending on final states of the branch. Since, as discussed previously, the tree is too big to be generated wholly, only some branches can be explored and expanded up to a terminal leaf node. This is done with Monte Carlo simulations.

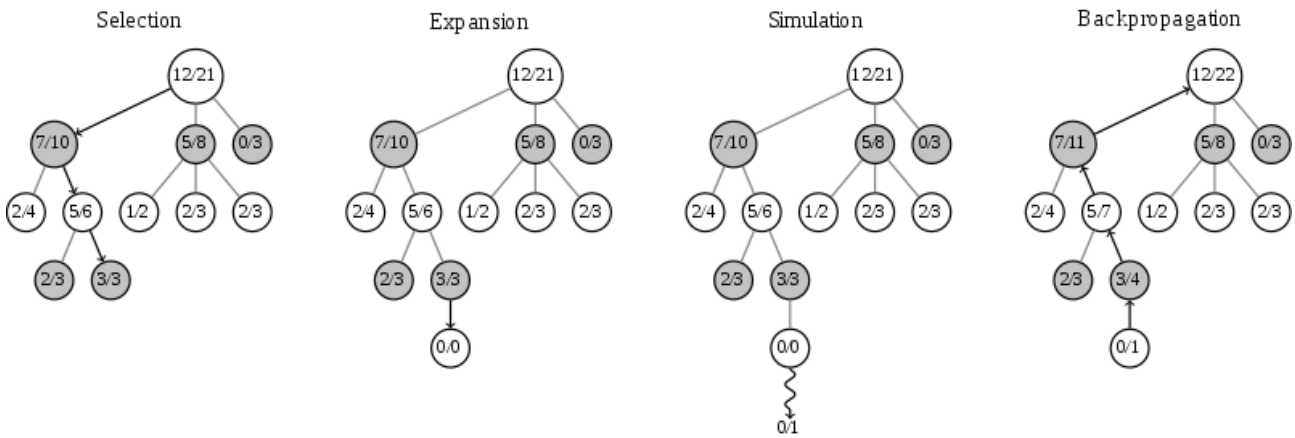


Illustration 13: Steps of MCTS. Source: [19]

Monte Carlo simulations consist of 4 parts: Selection, Expansion, Simulation and Backpropagation. In Selection, nodes will be selected starting from the root node down until a leaf node is reached. Then, a number of possible follow-up moves will be expanded unless if the node is a terminal node already and one of the expanded nodes will be selected. In simulation random or pseudo-random moves will be played until the branch reaches a terminal node and a result is gained. To this end, the result will be propagated up the tree and nodes will be updated with win rates.

When the simulations are done, the node with the highest number of simulations is chosen. It is important to note that not the node with the highest win rate is chosen, since that would potentially be a yet unexplored and therefore uncertain node. The number of simulations can be adjusted in accordance with a suitable runtime. An advantage of MCTS over other tree searches is that it can be stopped anytime and still yield a reasonable result and converges to optimal play as time proceeds.

3.4 b) UTC

At the core of the algorithm stands the selection procedure. Usually, one will want to select the node with the highest win rate, however this will quickly get the algorithm stuck in a certain branch of the tree and disregard exploring other moves. This is where UTC (Upper Confidence Bound 1 for trees) will come into play. UTC selects the move with the highest value for

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}} \tag{5}$$

where w denotes wins, n simulations, and t total number of simulations in node i. c is the exploration parameter. For increasing values of c, the algorithm will increasingly prefer to explore rarely visited nodes over those with the best winning probabilities.

Additionally, heuristics can be added to UTC to improve move selection. They can be hand-crafted pattern searches or, more recently, neural networks. Since heuristics are usually computationally heavy, they are often not used in the Simulation Stage, occasionally reduced to only the root node[1]. Further improvements exist, but a more thorough examination of UTC is outside the scope of this thesis.

Algorithm 1 MCTS Pseudocode:

```
function MCTS_SIMULATIONS(current position s_0)
    create root node l_0 with state s_0
    for number_of_simulations:
        if not CHILD:
            create nodes l with states s
            attach CHILDREN l to current node
            select CHILD according to UTC [+ heuristics]
            while not final_state:
                expand and select random position s
            generate score w
            while parent node:
                update wins/simulations in current node
                goto parent node
            break
        select CHILD according to UTC [+ heuristics]
    return most visited CHILD(l_0)
```

[1] Such as in Hiratuka bot, 3d version. The root node expands a DCNN, further nodes don't use heuristics.

3.5 The AlphaGo Player

3.5 a) Policy and Value Networks

AlphaGo is not the first bot to deploy DCNNs for Go [6] and combine them with MCTS [21], however it is the most successful and first program to push Go bots into the professional regime[1]. Its architecture is used in all current top-ranking bots [1, 13, 14, 15]. AlphaGo effectively combines MCTS with DCNNs. In practice, it contains two networks: A policy network and a value network, used as a heuristics for suitable moves in a given board position and as evaluation of a board position, respectively.

The policy network receives state-action pairs *(s, a)* from human professional games and is trained to maximize the likelihood of the professional move *a* in state *s*. As described in 3.2 a), its building blocks consist of alternating convolutional layers and ReLUs. It is finished up with a softmax layer. Softmax layers serve to map the K-dimensional input tensor *z* of arbitrary real values into a K-dimensional tensor *σ(z)* that consists of values in the range [0, 1] which add up to 1[22]:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K. \tag{5}$$

This way, the output becomes interpretable as a probability distribution instead of arbitrarily high values.

The number and size of filters and number of convolutional layers were found empirically. The first layer is a 5 * 5 convolution, followed by 12 layers of 3 * 3 convolutions and the softmax. The input is augmented via heavy preprocessing to aid convergence of the model. Besides the black and white stones on the 19 * 19 board, the following features were extracted and encoded in additional input planes: Empty fields, Legal fields, A plane filled with ones and one with zeros (set for the network to recognize the beginning of padding), 8 planes for previous moves, 8 for liberties of the group, 16 for opponent and own group size, 8 for liberties after a move is played and 2 for ladder recognition. These sum up to a 48 * 19 * 19 tensor.

The output is represented by a 19 * 19 matrix using the professional move on the board in a one-hot encoding.

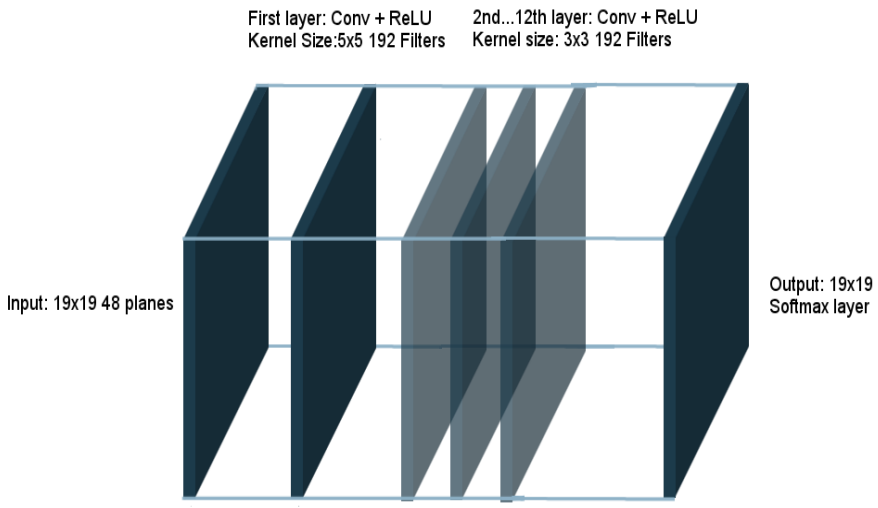


*Illustration 14: The policy network. The state input is represented by 48 planes for the 19 * 19 go board, capturing information such as stone colour, liberties or size of the group for every field. It is processed by 13 convolutional layers followed by ReLUs each and finished with a Softmax layer. The output is a 19 * 19 tensor for the correct professional move.*

Training with Standard Gradient Descent, this model reaches 57% convergence, i.e., 57% of the top-1 predictions are in accordance with the actual professional moves. It has a playing strength of roughly 3d amateur [1]. Additionally, an unspecified smaller and therefore faster but less precise network was trained analogously to be later used in MCTS simulations. This network only has an accuracy of 24,2%, yet it only takes 2µs to process a position compared to 3ms for the main network. Computation time is critical in the huge amount of positions MCTS simulations generate.

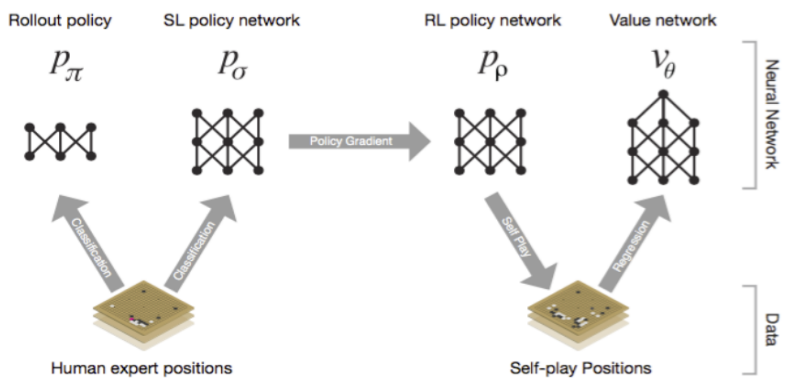


Illustration 15: Starting from state-action pairs of human expert games, the SL network is trained to predict expert moves. The rollout policy is shallower and faster and will predict moves in MCTS simulations. When the SL policy network is set up to play games against itself, it is trained on the winning player and further improves to the RL policy network. The vastly bigger amount of self-play positions can be used to train a stable value network, which predicts the outcome of a game in the current state. Source: [1]

So far, the network is trained in a supervised manner and therefore called SL (supervised learning) policy network. A new policy network with the same architecture is initialized with the SL policy weights and plays games against itself by using top-1 predictions. The positions of the game are stored and at the end of the game labelled with +1 for the winning side and -1 for losing. The network is then trained again with these positions and a new game is initiated. In order to prevent overfitting to the current policy network, an opponent is chosen from a pool of former networks. The now created RL (reinforcement learning) policy network is significantly stronger than the SL policy net, winning about 80% of games against it. However, in the main AlphaGo algorithm the SL policy network is still stronger despite its weaker stand-alone performance, likely because "humans select a diverse beam of promising moves, whereas RL optimizes for the single best move." [1]

Besides the policy network, a so called value network is trained in a very similar manner. It uses the same input tensor as the policy network and the same 13-layer deep architecture for the network, but only a binary output. The output is the winning colour, such that the value network learns to predict the winner of a game given a certain position. It has to learn the same shapes as the policy network, but unlike the policy network, it can evaluate the position instead of suggesting a move. Due to the output being so sparse, overfitting is a much bigger problem for the value network than it is for the policy network (a position is quickly mapped to its respective game and outcome such that when remembering this game positions, positions a few moves later or earlier can be predicted easily). Therefore the value network is trained on moves from the long series of self-play games created in the process of training the RL policy network.

3.6 b) Fusing the Algorithms

While neural networks are great for recognizing shapes, pure DCNN players have no ways to study followup moves and are strategically weak. Strong DCNNs can recognize shapes deep enough to avoid moves that result in bad positions a few moves ahead, but they can never read out complex problems ('Tsumego').

On the other hand pure MCTS players can create deep search trees with many followup moves, but they stand and fall with the quality of the heuristics they sample moves from to generate a tree. When efficiently using a DCNN as such a heuristic, strong bots can be created. AlphaGo builds a tree by using the value network and simulations jointly as reward for the MCTS algorithm plus a bonus from the SL policy network which helps selecting promising moves:

$$a_t = \underset{a}{argmax}\left(Q(s_t, a) + u(s_t, a)\right) \tag{6}$$

At each step t the best action a_t is selected based on the reward $Q(s_t, a)$, which itself is a composition of the returns from the value network and the winning ration MCTS simulations give. This composition is found empirically and chosen to be 50/50. Evaluation cannot be done with the policy network since its outcome always sums up to one and is not comparable throughout branches, while the outcome of the value network depends on comparable game result estimations. $u(s_t, a)$, which is returned form the SL policy network, is added on top and particularly helpful at the beginning of the creation of the tree when few or no Q-values exist. It will point at the nodes to observe the most and later be more and more replaced by the reward values from $Q(s_t, a)$. It decays with increasing visit count just like formulated in UTC.

$$u(s,a) \propto \frac{P(s,a)}{1+N(s,a)} \tag{7}$$

Here, $P(s, a)$ depicts the prior probabilities and $N(s, a)$ the visit count of the node. When traversing the tree, the algorithm will chose the best node in accordance with (6) starting from root until it reaches a leaf node, expand and store a new node with the policy network and perform a simulation as well as a value network evaluation. Then, the total simulation scores and visit counts for all parenting nodes are updated as described in MCTS and another simulation can begin. Note that only the new node, but not the simulation itself, is stored in the tree.

When a time threshold for the algorithm is reached, the most visited move from root is selected.

3.6 c) Strength

To get the most out of the AlphaGo algorithm, it is provided with a heavy hardware setup. The first publicly playing version that won against the human professional Fan Hui in autumn 2015 used 176 Tensor Processing Units (TPUs) and 1202 Central Processing Units (CPUs) [1]. It was a distributed system and is shown in Ill. 16. There was also another single machine version with 8 TPUs and 48 CPUs. MCTS requires a lot of computing power while going through the many paths of the game and will improve with stronger CPUs. Neural networks on the other hand will work efficiently on a large scale only when using heavy parallel processing, which is usually done with Graphical Processing Units (GPUs). Google revealed in 2016 that it developed specific chips for parallel computing purposes that set an end to hacking

Illustration 16: The AlphaGo TPU Cluster, source: [24]

GPUs for neural networks instead of graphical tasks. These TPUs have been deployed in AlphaGo [23].
As can be seen on in Ill. 17, the Fan Hui AlphaGo version was roughly 4 stones stronger than then-top Go AI CrazyStone or Zen which relied on MCTS without DCNNs. Only a few month earlier, in July 2015, neural networks had just been discovered for Go and discussed at the European Go Congress in Liberec, at that time with a playing strength of 2kyu and completely unfamiliar ground to computer Go. One year later, in March 2016, AlphaGo beat Lee Sedol 4-1, a top human professional with the highest achievable rank, 9. professional Dan. David Silver revealed that this version was 3 stones stronger than the previous AlphaGo [28] and many professionals agreed that AlphaGo now has a deep understanding of Go [29]. In early 2017, AlphaGo played on 2 popular Korean Servers (Tygem and FoxGo) under the alias "Master" 60-0 against human top-professionals from various nationalities. This version improved another 3 stones compared to the one Lee Sedol played against [28]. It appeared again in May 2017 to play against the currently strongest professional, Ke Jie, 3-0. This version ran on a single machine.

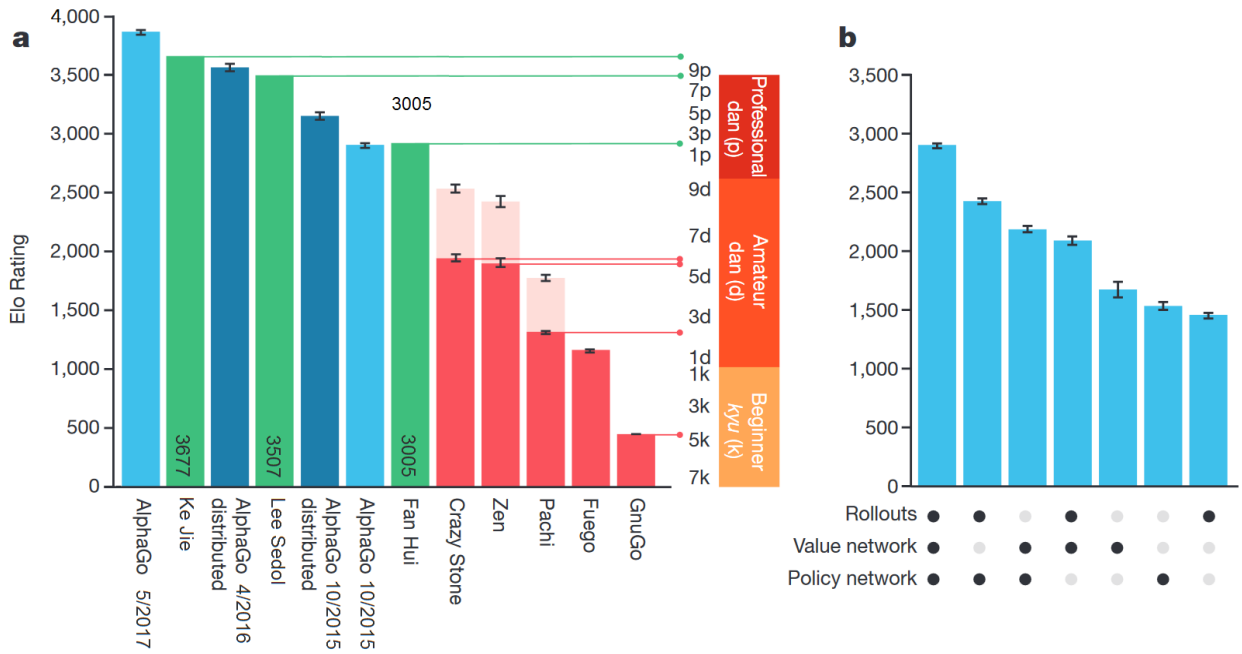


Illustration 17: a) Strength of various AlphaGo versions compared to other bots and human professionals. Professionals strength were retrieved from [30]. b) Strength of decompositions of AlphaGo. Original image source: [1]

In a decomposition of AlphaGo the usefulness of both networks and simulations can be demonstrated (Figure b). Not only is it critical to include the policy network as prime heuristic for move selection, but also a version with value network and simulations for the reward scores much higher than having only either value network or simulations in AlphaGo. This demonstrates that these two completely different approaches to evaluating a board position can sensibly complete each other.

Chapter 4

Methods

The objective of the programming part of this work is to both create a reasonably strong agent that can play with humans and experiment with alternative and new approaches in Go game AI. A policy and value network was written as well as a novel, combined *"meta network"*. To get a full grasp of the topic, all scripts were written by the author from first principles.

4.1 Environment

Due to its scientific orientation and broad availability of deep learning libraries Python 2.7 was chosen as programming language for the bot '*Entropy*'. Tensorflow 1.2.0 was used for creating the neural networks and flanked with TfLearn on top. Tensorflow accesses the GPU via CUDA 8.5 and CUDNN. This way, number crunching for weight calculation can be done efficiently in C in the background on the GPU while the programmer comfortably writes in pure Python. Other dependencies include: multiprocessing 0.70a1 for parallel game processing, random, glob.

The training machine consists of an Intel Core i7™ 4820K Hashwell-E (6-Core) CPU and a Nvidia GeForce™ 1080 GPU, 8TB HDD and 32GB RAM. This machine processed the data more than an order of magnitude faster a previous machine with a 3rd Generation Core i7™ and a GeForce™ 660M. It becomes evident here that this computationally gigantic task is eased significantly with GPU Grids like those that DeepMinds AlphaGo possesses.

To train the networks, a large database of games from strong players is required. Various databases were tested, among them:

- The KGS dataset. 180.000 games on a 19 * 19 board from the Kiseido Go server, played by strong amateur players (6d or higher) with mixed but mostly medium time settings. Download link: [25].
- The 13 * 13 Dataset. 50.000 games played by professionals under fast time settings (5 minutes main time) on a 13 * 13 board by Go-Quest author Tanasa. Download link: [26].
- The Pro dataset. 70.000 games on a 19 * 19 board played by professional players over many centuries of Go history and slow or very slow time settings (games last between few hours and days). Download link: [27]

4.2 Data preparation and augmentation

Most Go game records are written in standard go format (.sgf). Sgf encodes moves as a tuple of letters from a-s (field 1-19) and allows for many other optional information to be stored, such as player data or handicap stones. To convert a sgf file into training data for a neural network, every board position needs to be converted to a 19 * 19 array and paired with the next move (for the policy network) or the game result (for the value network). Rules such as the formation of groups, taking stones and groups off the board when captured, Ko, and handicap follow. In addition to black and white stones each represented by a layer in a 19 * 19 * X tensor for the DCNN input, the final composition of features for *Entropy* were:

- own stones
- opponent stones
- empty fields
- the colour of the player to move
- Last 4 moves
- legal fields (including consequences from the KO-rule, suicide)
- 3 planes for liberties (groups with 1, 2, 3+ liberties)
- 2 planes for liberties after a move was played (groups with 1, 2+ liberties)
- Working ladders (ladders that work for own groups and ladders that don't for opponent groups)
- Non-working ladders (ladders that don't work for own groups and ladders that work for opponent groups)

Summing up to 13 feature planes. A previous version included more planes for liberties and several planes for group stone sizes and had 30 feature planes. However this led to gigantic amounts of data (many TB) and was impossible to process. The task here is to minimize the amount of feature planes while maintaining as much relevant information as possible, with e.g. AlphaGo using a total of 48 planes. Even the final 13-plane numpy

tensors summed up to 1,7TB of data for the KGS dataset and took more than 10 days to compute with multiprocessing enabled on 12 threads/6 cores. In order to not over-exhaust CPU and RAM, data was stored in blocks of 1-5GB and loaded indicidually for training.

During training it became eminent that beside the straightforwardly relevant 2 planes for the player's stones, the last 4 played moves had a relatively big effect on the convergence of the model. They may help the player identify the "heat" of a game: Often, many moves are exchanged in one area of the board where a fight is occurring, until the size of the move (meaning: the points it is worth to play) drops below that of a move elsewhere. In this way one could picture a heat map of move size over the board where recently played moves in one area help identify its activity and make it more likely to play there than in another area with other possibly good shape moves.

Ladders are a special kind of feature extraction as they solve a rather specific task (whether a group with only 1 liberty can be captured or will escape) for an often long read. Feeding this information into the network does not raise convergence very much, but in actual gameplay, a falsely read ladder can lose the entire game.

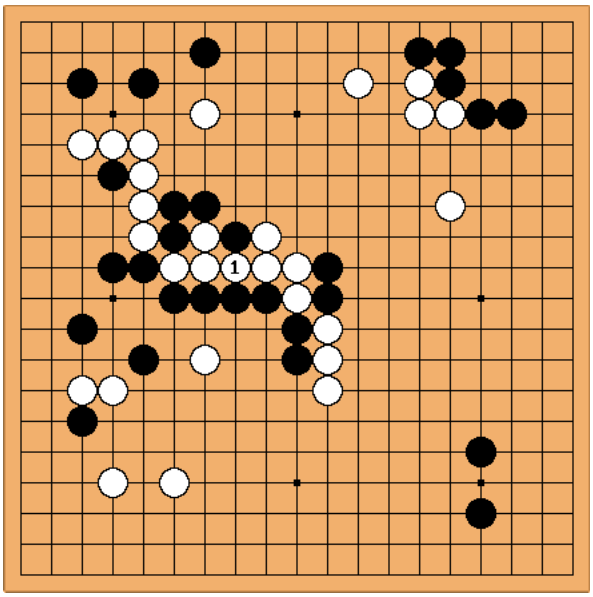


Illustration 18: Can black capture the 3 stones in the centre? Source: [28]

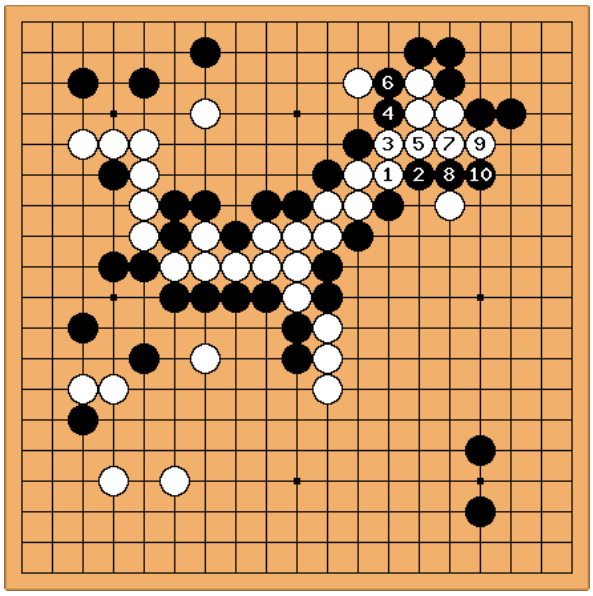


Illustration 19: He can. White (O Meien, 9P) resigned. Source: [28]

Liberty planes were encoded such that any group with exactly x liberties will appear in the plane for liberty x. They help the network in fighting situations, where own liberties and the opponent's liberties need to be compared.

Before handing the input over to the network, training data needs to be shuffled in and between games to distribute similar board positions as broadly as possible and prevent overfitting. Since the dataset was so large, chunks of data were loaded, shuffled, and stored again during training to distribute the games.

Another augmentation done to the games is to flip and turn the board. Since a Go board is symmetric, it doesn't matter whether it is played upside down. The data can hereby be increased eightfold (2^3 for horizontal flips, vertical flips and turns). A random flip/turn function did this before training, to both input as well as the output in the case of the policy network.

4.3 Network Structure

4.3.1 Policy Network

The policy network is written in Tensorflow via TfLearn. It receives an N * 19 * 19 * 13 tensor for the board positions as input and the respective next moves played in the game one-hot encoded as N * 19 * 19. One of the blocks of board positions created by the preprocessor is separated and used for testing. The network consists of 12 convolutional layers followed by ReLU activation functions and a final fully connected layer followed by a softmax activation function similar to the AlphaGo [1] and Darkforest [21] player architecture. All convolutional layers have 192 filters. The first layer has a filter size of 5, followed by 11 layers with filter size 3. The optimizer is Standard Gradient Descent (SGD). Adam has been tested and performed well, but most other successful networks were trained with SGD [1, 6 ,21], possibly because SGD has slightly better testing values after long training periods [29]. The batch size is 256, the learning rate was set to 0,03 and halved every 500.000 training steps until training stalled, with the smallest usable learning rate being around 0,003.

4.3 b) Reinforcement Learning

Reinforcement learning of the policy network was tested but aborted due to runtime reasons. Here, the policy network was once applied to an empty board, the move with the highest likelihood was extracted and played. Then features for the input tensor were extracted from this position and again shown to the policy network. Another prediction was made, this time for the other colour, and the process repeats until the game ends.

Now there are two obstacles to overcome: Determining the end of the game and scoring the game to identify a winner. A policy network will always return its best guess for the next move, but it cannot suggest not to move. Therefore it was set not to move at places that fill own eyes or that are illegal to play. When there are no places left to play, it will pass and the game ends.

It is a non-trivial task to recognize eyes: A large group may include so much space for eyes that the player does not need to play inside and defend an area that is already his. But an even larger group, or yet not clearly defined group, does need to. The same question on whether or not an area is settled arises when scoring the game. In order to calculate the points a player makes it must be clear which group is alive, in Seki, or just a lump of dead stones inside of an opponent group. Other, more rare situations include Eternal Life[1], Life with 2 false eyes[2], Triple Ko and more. In fact, the on-site scoring function of the KGS Go Server occasionally misinterprets group status and players are always required to mutually decide life and death at the end of the game since using the scoring function would not be secure.

To avoid this hassle, eyes up to a certain size (7 fields were chosen) were recognized as eyes and simply marked as illegal moves for the player, and bigger areas would be divided and filled until they were clearly recognizable as eyes. A side effect of this method is that games last a lot longer and include many superfluous moves at the end. The relatively long runtime of python scripts (extracting all features from the board for every move) contributed to this.

Despite the RL algorithm working in principle, its long runtime for creation of game data ultimately led to the dismissal of this project.

4.3 c) Value Network

The value network has the exact same architecture as the policy network except for the output, which is binary. Instead of a softmax activation in the last layer, a tangens hyperbolicus is chosen (softmax does not work since the output is not a probability matrix). It is trained and tested likewise and uses the same hyperparameters. Game results for the output are simply extracted from the sgf file. Training data is now threefold: Heavyweight storage tensor blocks for the input and binary/matrix blocks for the output of the value/policy networks respectively.

[1] Eternal life is a very rare condition in which both players need to play several moves in an area in order not to lose their stones.
[2] This form of life does not require 2 “real” eyes as in the usual case, but instead eyes that include too few stones and usually need to eventually be filled to not lose part of the group. Here however, two “false” eyes are connected in a large chain of stones and need not be filled because each eye prevents the other from being removed.

4.3 d) Meta Network

Since policy and value networks share a big part of their architecture and are trained on the same data, it makes sense to create a joint network that mutually trains on both objectives. This will make an agent simpler and lighter in structure, and since in both networks shapes need to be recognized and analysed, they might even mutually train faster than showing the same data to first one network and then the other. It is hypothesized that a joint network can make use of mutually learned shapes in this setup. There is another valuable benefit of this structure: Since the AlphaGo algorithm calls the

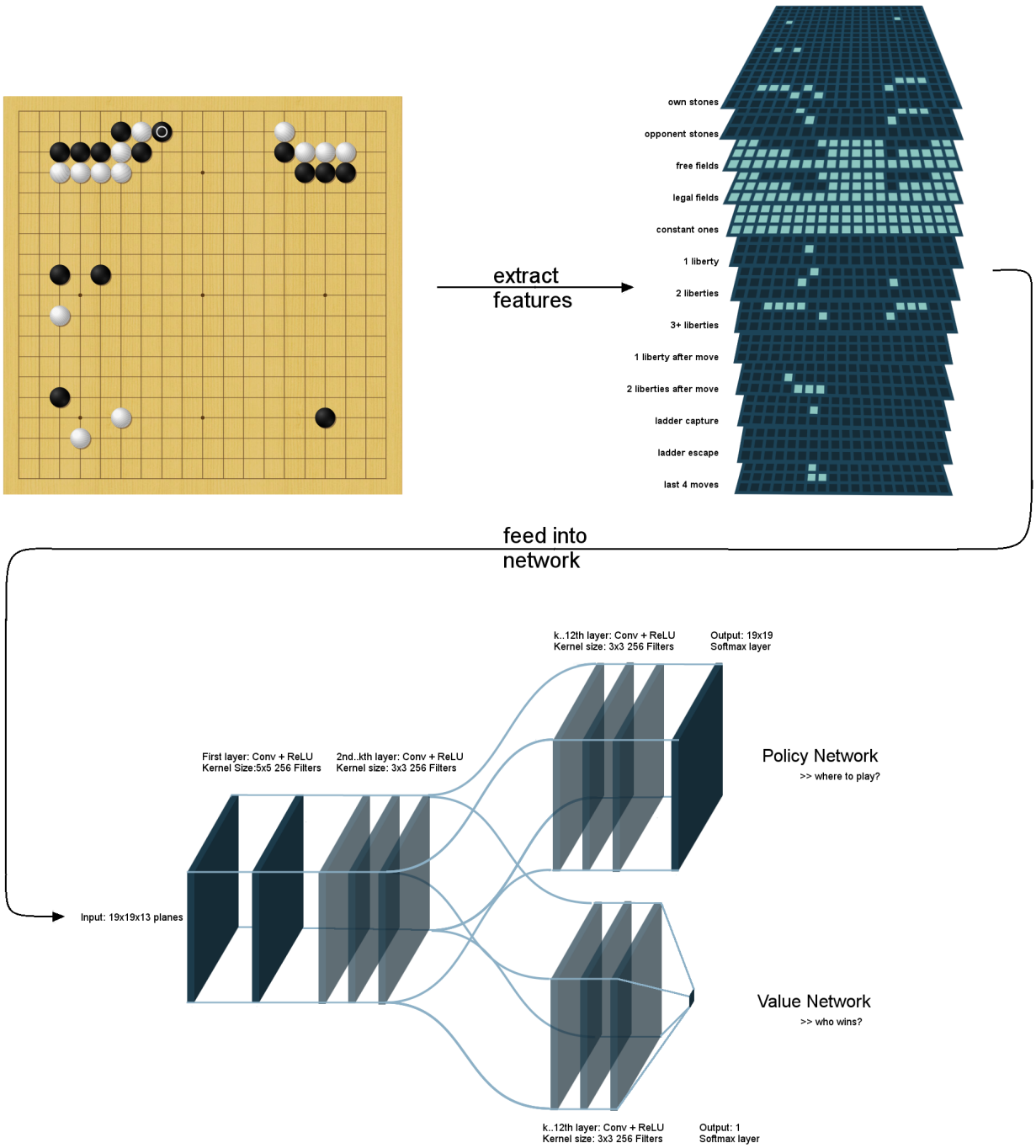


Illustration 20: Pipeline of the meta network

value network and the policy network once on every node within the search tree, computing time will be saved when instead calling only this one network.

Therefore, a novel invention called “meta network” was built. The meta network receives the board position tensor as input, begins with a layer of filter size 5 and then splits layers, training two networks, after x layers of filter size 3. There were three models tested: a split directly after the first layer, a split after 5 convolutional layers and a split after 7 convolutional layers. After the network is split, each branch continues with the respective missing layers (see Ill. 20).

This way it can be tested whether in a lightly joint model the two different objectives already disturb each other, or whether it is possible to share a maximal amount of weights in a medium or highly joint model.

4.3 e) Player

To play against the policy network, a simple python player was written. It can initiate a game as white or black or with handicap. It receives board coordinates from a human opponent and outputs the policy network’s top-1 prediction for the bot in the form of a board and the probability matrix of suggested moves along with it.

```
E 17
9 . . . . . . . . . . . . . . . . . . .   9 . . . . . . . . . . . . . . . . . . .
8 . . . . . . . . . . . . . . . . . . .   8 . . . . . . . . . . . . . . . . . . .
7 . . . . X . . . . . . . . . . . . . .   7 . . . . 8 . 1 . . . . . . . . . . . .
6 . . X . O . . . . . . . . . . X . . .   6 . . . . . . . 1 . . . . . . . . . . .
5 . . . . . . . . . . . . . . . . . . .   5 . . . . . . . . . . . . . . . . . . .
4 . . . . . . . . . . . . . . . . . . .   4 . . . . . . . . . . . . . . . . . . .
3 . . . . . . . . . . . . . . . . . . .   3 . . . . . . . . . . . . . . . . . . .
2 . . . . . . . . . . . . . . . . . . .   2 . . . . . . . . . . . . . . . . . . .
1 . . . . . . . . . . . . . . . . . . .   1 . . . . . . . . . . . . . . . . . . .
0 . . . . . . . . . . . . . . . . . . .   0 . . . . . . . . . . . . . . . . . . .
9 . . . . . . . . . . . . . . . . X . .   9 . . . . . . . . . . . . . . . . . . .
8 . . . . . . . . . . . . . . . . . . .   8 . . . . . . . . . . . . . . . . . . .
7 . . . . . . . . . . . . . . . . . . .   7 . . . . . . . . . . . . . . . . . . .
6 . . . . . . . . . . . . . . . . . . .   6 . . . . . . . . . . . . . . . . . . .
5 . . X . . . . . . . . . . . . X X . .   5 . . . . . . . . . . . . . . . . . . .
4 . . . . O . . . . . . . . . O . X O . .   4 . . . . . . . . . . . . . . . . . . .
3 . . . O . . . . . . . . . . . . . O . .   3 . . . . . . . . . . . . . . . . . . .
2 . . . . . . . . . . . . . . . O . . . .   2 . . . . . . . . . . . . . . . . . . .
1 . . . . . . . . . . . . . . . . . . .   1 . . . . . . . . . . . . . . . . . . .
. A B C D E F G H J K L M N O P Q R S T   . A B C D E F G H J K L M N O P Q R S T

>>>
```

Illustration 21: Playing against the policy network. On the left one can see the board and the last bot move on top. On the right, its probability matrix in values from 1-9 is displayed.

4.4 Network Results

4.4 a) Policy Network

The KGS Dataset.

The KGS Dataset was tested most intensely because of its high quality. It offers a huge amount of training data (180.000 games with over 30 million positions). Preprocessing took around 10 days with 12 threats on the core i7™ 4820K and generated 1,7TB of data in numpy arrays. After 1,8 million iterations (2 weeks training on a GTX 1080) it reached 47,5% accuracy on the test set and slightly higher values on the training set.

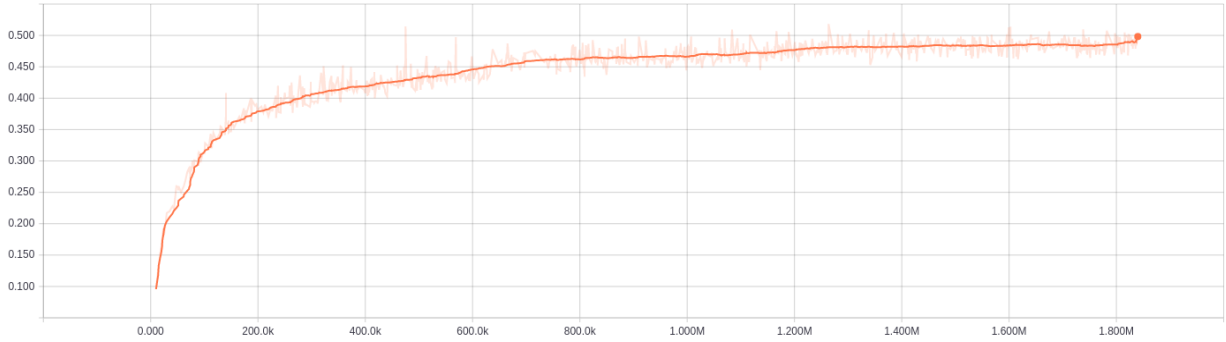


Illustration 22: Training accuracy with smoothing of the policy network in TensorBoard

This network was tested on the KGS Go server against several human players and bots. It reached a stable rank of 7k after 10 games against roughly equal opponents. Games were played either even (no handicap, proper Komi) or without Komi if one player was a rank stronger. The time settings were 10 minutes main time plus 5 times 20 seconds. A rank of 7k is around the average rank of a Go club player [30].

Its playing style can be described as calm (it does not actively search fights) and light (it prefers to spread single, individually less important stones on the board rather than getting invested into bigger 'heavy' groups that have clear strategic intentions). Its main flaw is its bad reading (due to its lack of a tree search). It often has a better evaluation of the board than its opponents but fails when it comes to Life and Death of groups and local fights. An example game can be found in the supplementary documents.

The 13 * 13 Dataset.

This Dataset consisted of games played by professional players in a fast time setting (5 minutes time per player plus 3 seconds per additional move). The network's input tensor was adjusted to the smaller board size. It reached an accuracy of 47%[1] on the test set after 80.000 iterations and about 13 days training time on the GTX 660M. It was tested on the KGS Go server by playing a few games against humans and has a rank of around 15k. The game conditions were the same as in the KGS dataset. On this smaller board, less moves are available and classification rates need to be higher to achieve the same rank on a 19 * 19 board. Since the quality of the games is a lot worse than the one from the KGS dataset, no further training was performed (the time settings can be considered "blitz" or "ultra-blitz" and such is the quality of moves despite professionals making them). It is interesting studying 13 * 13 neural networks since to the authors knowledge no other publicly available networks were trained on this board, but the lack of suitable training data makes it hard to do so.

The Pro Dataset.

This Dataset is a collection of professional games, many of them played on real boards, dating back to as early as 1062DC. It also includes recent games and even some games by FineArt, Zen, AlphaGo and other computer agents that play on Pro level. Go Theory changed a lot over the centuries, and in order not to confuse the network with ancient styles only games from after the 19th century were used (which includes the majority of the dataset). It reached a convergence of 46% after 1.300.000 iterations. The training accuracy fluctuated a lot more than in the KGS dataset (see appendix). This might be due to the games being from over 100 years of Go history and the learned style changing over time, leading to wrong classifications in different eras. Overfitting was also an issue in this dataset due to its size with training accuracies occasionally surpassing 60%. Its playing strength is 9kyu and was tested the same way as the other networks.

4.4 b) Value Network

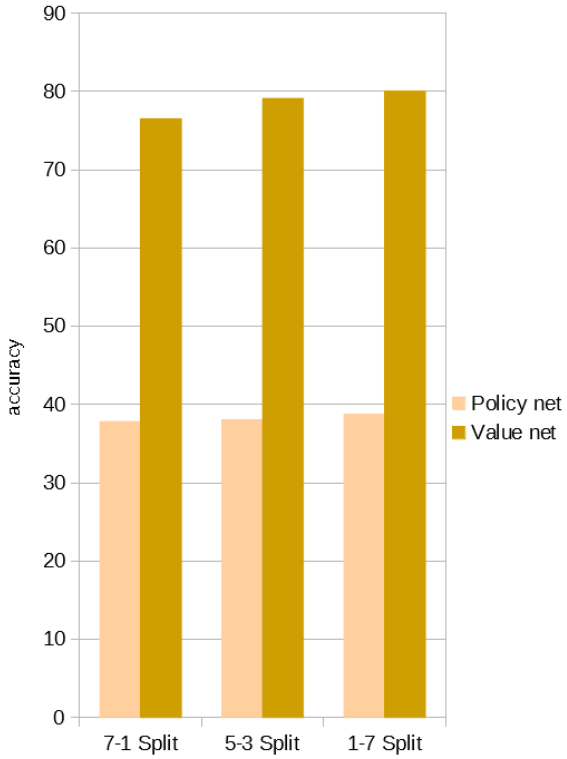
The value network was trained on the KGS Dataset for 800.000 iterations on the GTX 1080 (6 days training). It reached a training accuracy of 93% and a testing accuracy of 59%. Note here that 50% accuracy is equivalent to chance level. A value network needs huge amounts of training data / games as it is very prone to overfitting by remembering game positions. This is why the value network shows such bad results. In order to achieve a better converging value network, more games will be needed, of which only a fraction of the

[1] The accuracy of networks on different board sizes lying so close to one another is purely coincidental.

positions will be shown such that similar positions from the same game don't occur. This is what Silver et. al did by using the data generated in the reinforcement learning (RL value network) process. But as noted, creating games in this setup is very slow. To give an idea, the network in Silver et. al used 30 Million positions from the RL pipeline. If we assume that a maximum of 4 positions per game can be used to maintain positions distinct this leads to a game dataset 50 times the size of the KGS dataset.

4.4 c) Meta Network

The 3 variants of the meta network were all trained for 200.000 steps (2-3 days training on the GTX 1080 depending on the meta network variant). It shows a surprisingly good accuracy for all variants in both networks, confirming the hypothesis that shape knowledge can be shared between networks. Even the 7-1 split, in which both parts of the network have only 1 convolutional layer for themselves, trained well (achieving only 2% less accuracy in the policy network than the 1-7 Split). However it can be assumed that accuracies diverge further after more training and more knowledge is gained.
More data from the meta networks can be found in the appendix.



Training accuracy after 200k steps. Either 7, 5 or 1 Conv+ReLU Layer are shared, the following layers exist in both nets. The value network begins training at 50% (chance)

Chapter 5

Conclusion

5.1 Summary

MCTS and later DCNNs were milestones in the history of computer Go. Both of these algorithms have counterparts in human play and yet are different in their execution. In the past 2 Years computer Go agents went from fairly average to superhuman in terms of performance. Both the Go world as well as the Deep Learning Community benefited form these developments and gained recognition.

In this thesis the core algorithms were described starting with the basic building blocks of neural networks up to the complete algorithm that is used in modern computer Go players. The neural networks described were then built, tested, modified and extended. A novel network was introduced, tested and is suggested to be implemented in a MCTS bot. A majority of time was invested in the development of the DCNNs and I learned a lot about programming modern neural networks. Finally, during the process of researching literature on Go and computer Go, I had the pleasure to visit a Go Academy in South Korea for 3 month and improve my own Go level thanks to the Cognitive Science Program abroad. In the appendix there can be found some impressions from the Academy and the life of the Yunguseng (Korean Go students) as well as professionals.

5.2 Further Work

There are a lot of areas to improve the Go bot *Entropy*. First off, a tree search would significantly improve the bot. There are MCTS implementations in python [31] but they are, compared to state of the art bots, very slow. A faster programming language such as C or Go is of need, which could also improve preprocessing for the neural networks and

make reinforcement learning possible in a reasonable speed. Then, the value network network can be trained with more data and find application in evaluating moves.

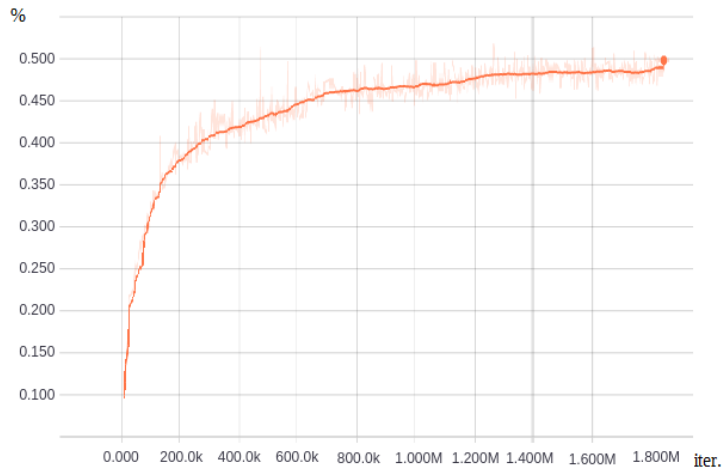
Another area to improve the speed of the bot is the preprocessing runtime in itself. The code for *Entropy* was written as a proof of concept and later improved to create a maximally efficient policy DCNN for a stand-alone player. As such, it takes much less than 1 second to generate a move. In combination with MCTS, improving preprocessing runtime will improve the quality of the whole bot. In such a bot, the meta network can find application too and demonstrate its benefit of having to use only half the DCNN passes that are necessary in the current setup.

Chapter 6

Appendix

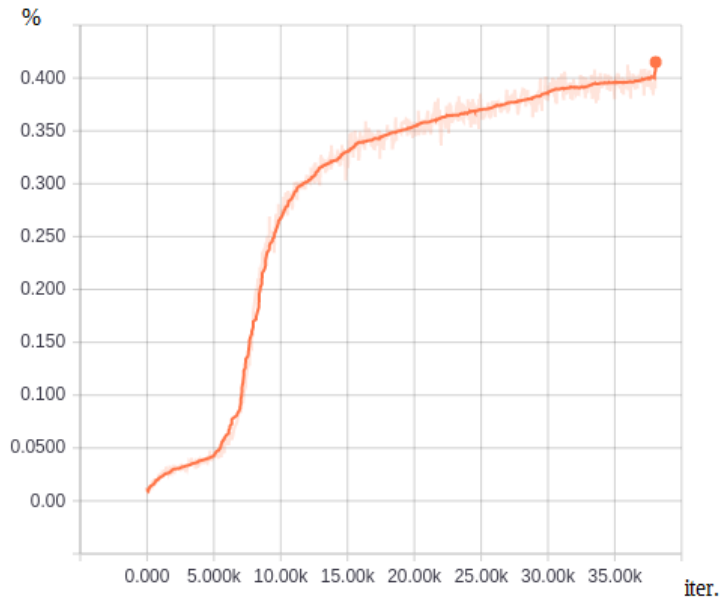
6.1 Tensorboard Graphs

Policy network, KGS dataset

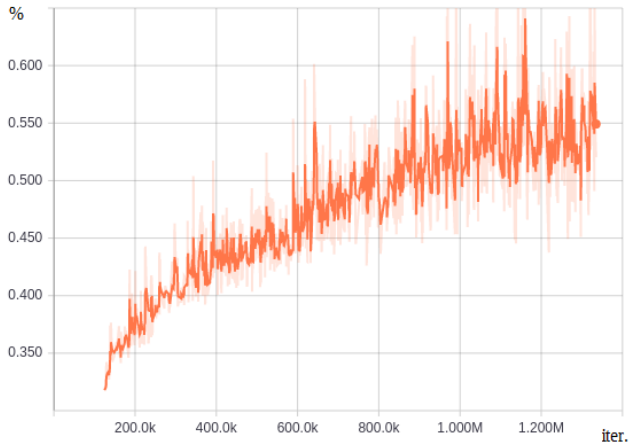


Policy network, 13 * 13 dataset

Zoom into first 35k iterations

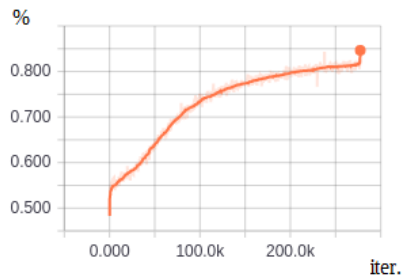


Policy network, Pro dataset

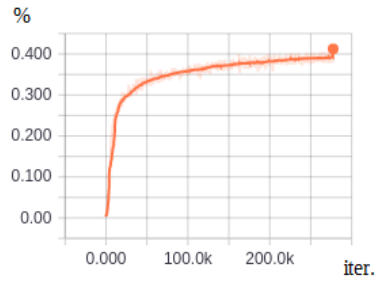


The meta network, 1-7 split

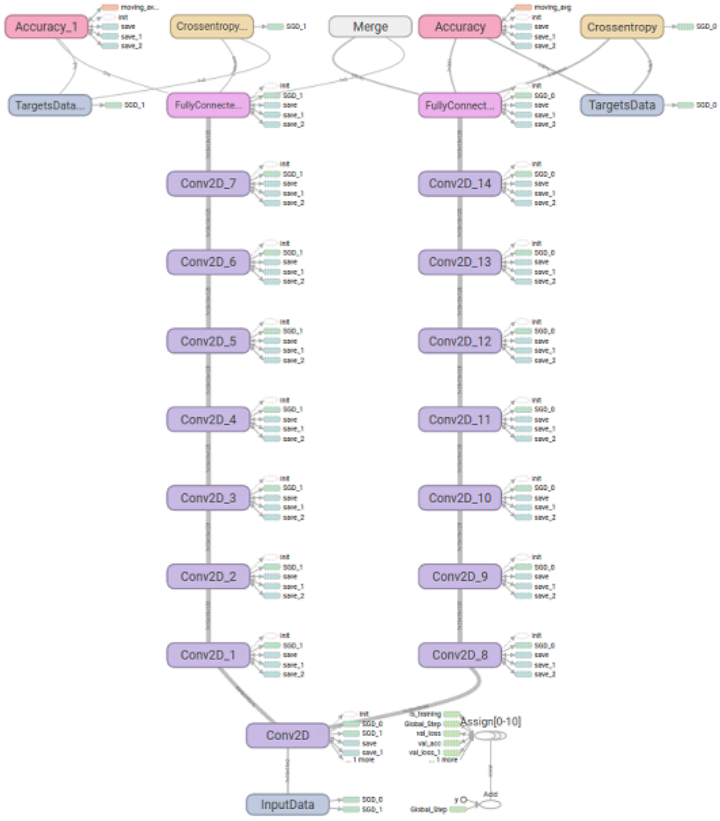
Accuracy of the value network split.



Accuracy of the policy network split.

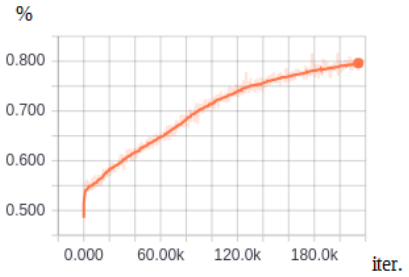


Training graph of the network.

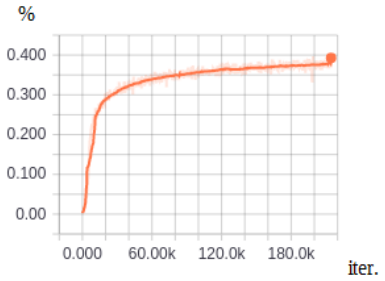


The meta network, 5-3 split

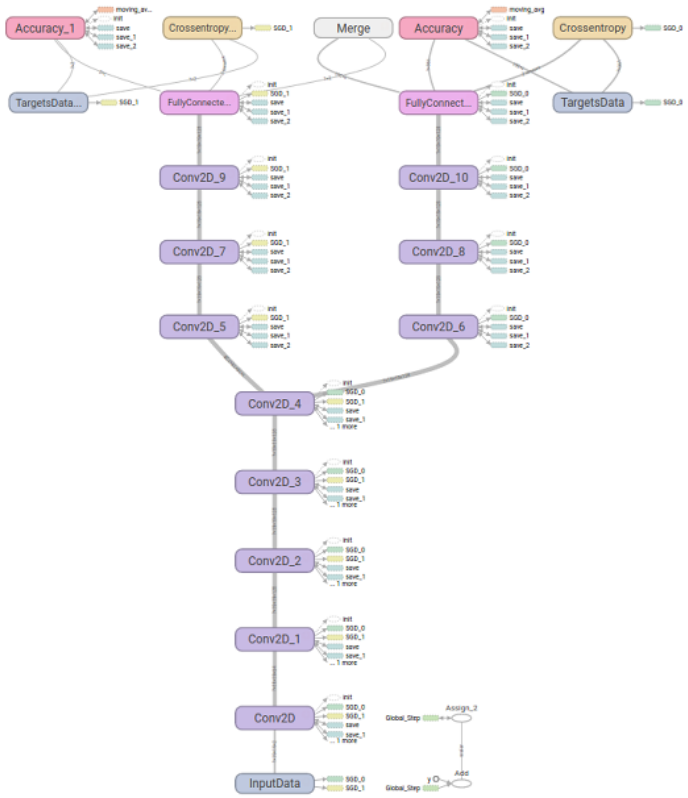
Accuracy of the value network split.



Accuracy of the policy network split.

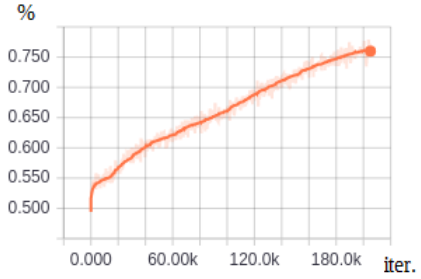


Training graph of the network.

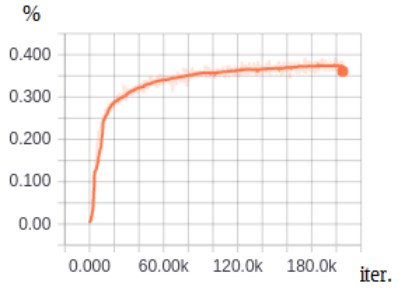


The meta network, 7-1 split

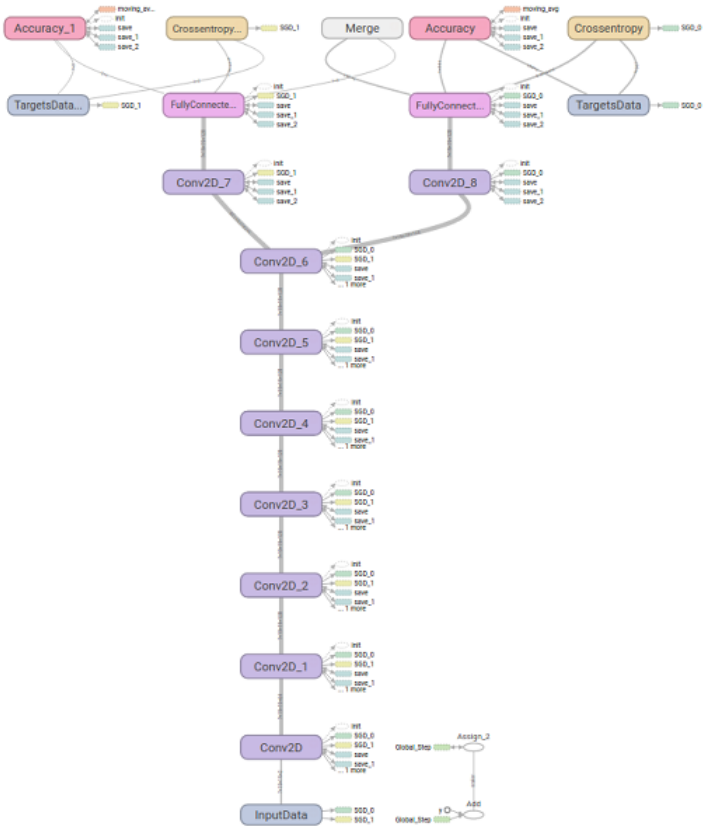
Accuracy of the value network split.



Accuracy of the policy netwrok split.



Training graph of the network.



6.2 Go in Korea

Some impressions from the BIBA Go Academy, located in Sanbon, South Korea. It is led by Kim Seung-Jun, 9 Dan Professional. During my 3-month stay I learned a lot about playing Go (studying time was from 9AM to 10PM on weekdays). In the following pictures one can see the professionals and international students from all over the world as well as Younguseng play.

Illustration 23: International Students and Koreans in the Academy in Sanbon, Korea

Illustration 24: Young Professionals studying a live game...

Illustration 25: A tournament room in the Korean Baduk Association, KBA, Seoul

Illustration 26: The Teacher studying with his students.

Illustration 27: Inside a Korean Baduk TV Channel

Illustration 28: A Go ceremony honouring top players of the year, some of them are very famous.

Illustration 29: Koreans start playing very, very young. The boy is in the centre is 4D strength.

Chapter 7

List of Abbreviations

Adam.....Adaptive Moment Estimation (Optimizer)

ANN.....Artificial Neural Network

D.....Dan (Master) Rank

DNN.....Deep Neural Network

DCNN.....Deep Convolutional Neural Network

FC.....Fully Connected

K.....Kyu (Pupil) Rank

KGS.....Kiseido Go Server

MCTS.....Monte Carlo Tree Search

Pro.....Professional Player

RL.....Reinforcement Learning

sgf.....Standard Go Format (File format)

SL.....Supervised Learning

UTC.....Upper Confidence Bound 1

Chapter 8

Bibliography

1. *Silver, David; Huang, Aja; Maddison, Chris J.; Guez, Arthur; Sifre, Laurent; Driessche, George van den; Schrittwieser, Julian; Antonoglou, Ioannis; Panneershelvam, Veda (2016). "Mastering the game of Go with deep neural networks and tree search". Nature.* ***529*** *http://www.nature.com/doifinder/10.1038/nature16961*
1. *Paul Gee, James (2006). "Are Video Games Good for Learning?". Keynote address at Curriculum Corporation 13th National Conference. http://cmslive.curriculum.edu.au/verve/_resources/gee_paper.pdf*
2. *Korf, Rickard E, Chickering, David Maxwell (1996). "Best-first minimax search". https://doi.org/10.1016/0004-3702(95)00096-8*
3. *Silver, David; Gelly, Sylvain (2008). "Achieving Master Level Play in 9 x 9 Computer Go". 23rd AAAI Conference on Artificial Intelligence. http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Applications_files/master-level-go.pdf*
1. *Silver, David; Gelly, Sylvain (2011). "Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go". http://www.cs.utexas.edu/~pstone/Courses/394Rspring13/resources/mcrave.pdf*
1. *Clark, Christopher; Storkey, Amos (2015). "Teaching Deep Convolutional Neural Networks to Play Go". https://arxiv.org/pdf/1412.3409.pdf*
2. *Potter, Donald L. (1985). "Go in the Classics (ii): the Tso-chuan". Go World. Tokyo: Ishi Press (42).*
3. *Zobrist, Albert L.(1970). "A New Hashing Method with Application for Game Playing". Technical Report. www.cs.wisc.edu/techreports/1970/TR88.pdf*
4. *https://www.gnu.org/software/gnugo/download.html*
5. *http://pachi.or.cz/*
6. *https://gogameguru.com/tag/deepmind-alphago-lee-sedol/*
7. *Hui, Fan (2016). "Game 4: 'Endurance'". https://deepmind-media.storage.googleapis.com/alphago/pdf-files/english/ls-vs-ag4/LS%20vs%20AG%20-%20G4%20-%20English.pdf*
8. *https://senseis.xmp.net/?ZenGoProgram*
9. *https://senseis.xmp.net/?CGI*
10. *https://senseis.xmp.net/?FineArt*

11. *Bengio, Yoshua; Glorot, Xavier (2010). "Understanding the difficulty of training deep feedforward neural networks". AISTATS'10* http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf
12. https://en.wikipedia.org/wiki/Rectifier_(neural_networks)#/media/File:Rectifier_and_softplus_functions.svg
13. *Grosse, Roger (2017). "Exploding and Vanishing Gradients".* http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/readings/L15%20Exploding%20and%20Vanishing%20Gradients.pdf
14. https://www.researchgate.net/figure/255798161_fig1_Fig-2-Convolutional-neural-network%27s-structure-for-the-combined-cascade-of
15. https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
16. *Tian, Yuangdong; Zhu, Yan (2016). "Better Computer Go Player with Neural Network and Long-term Prediction".* https://arxiv.org/pdf/1511.06410
17. *Bishop, Christopher M. (2006). Pattern Recognition and Machine Learning. Springer.*
18. https://www.heise.de/newsticker/meldung/Google-I-O-2016-Tensor-Prozessoren-halfen-beim-Go-Sieg-3210953.html
19. https://www.nextplatform.com/2016/05/19/google-takes-unconventional-route-homegrown-machine-learning-chips/
20. https://www.u-go.net/gamerecords/
21. https://groups.google.com/forum/#!topic/computer-go-archive/A8AjDVWQu3g
22. https://badukmovies.com/pro_games
23. https://denisfeldmann.fr/bestiary3.htm
24. https://shaoanlu.wordpress.com/2017/05/29/sgd-all-which-one-is-the-best-optimizer-dogs-vs-cats-toy-experiment/
25. https://www.britgo.org/press/faq.html
26. https://github.com/pasky/michi
27. https://gafferongames.com/post/introduction_to_virtual_go/.
28. http://www.usgo.org/news/2017/05/new-version-of-alphago-self-trained-and-much-more-efficient/
29. *Deutsche Go Zeitung, Heft 01/2016. Hebsacker Versag. Pp. 22-26*
30. http://www.goratings.org

All URLs were retrieved on 10/09/2017.