Deep Learning for Go

Bachelor Thesis

Author(s):
Rosenthal, Jonathan

Publication date:
2016

Permanent link:
https://doi.org/10.3929/ethz-a-010891076

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Deep Learning for Go

Bachelor Thesis

Jonathan Rosenthal

June 18, 2016

Advisors: Prof. Dr. Thomas Hofmann, Dr. Martin Jaggi, Yannic Kilcher
Department of Computer Science, ETH Zürich

Contents

Contents **i**

1 Introduction **1**
1.1 Overview of Thesis 1
1.2 Jorogo Code Base and Contributions 2

2 The Game of Go **5**
2.1 Etymology and History 5
2.2 Rules of Go 6
2.2.1 Game Terms 6
2.2.2 Legal Moves 7
2.2.3 Scoring 8

3 Computer Chess vs Computer Go **11**
3.1 Available Resources 11
3.2 Open Source Projects 11
3.3 Branching Factor 12
3.4 Evaluation Heuristics 13
3.5 Machine Learning 13

4 Search Algorithms **15**
4.1 Minimax 15
4.1.1 Iterative Deepening 16
4.2 Alpha-Beta 17
4.2.1 Move Ordering Analysis 18
4.2.2 History Heuristic 18
4.3 Probability Based Search 19
4.3.1 Performance 20
4.4 Monte-Carlo Tree Search 21

5 Evaluation Algorithms **23**

5.1 Heuristic Based . . . 23
5.1.1 Bouzy's Algorithm . . . 24
5.2 Final Positions . . . 24
5.3 Monte-Carlo . . . 25
5.4 Neural Network . . . 25

6 Neural Network Architectures **27**
6.1 Networks in Giraffe . . . 27
6.1.1 Evaluator Network . . . 27
6.1.2 Probability Estimating Network . . . 29
6.2 Networks in AlphaGo . . . 29
6.2.1 Input Channels . . . 29
6.2.2 Policy Network . . . 30
6.2.3 Value Network . . . 31
6.3 Jorogo . . . 31
6.3.1 Activation Functions . . . 31
6.3.2 Stride and Pooling . . . 32
6.3.3 Search Network . . . 32
6.3.4 Evaluation Networks . . . 32

7 Results **37**
7.1 Internal Matches . . . 37
7.1.1 Time Scaling . . . 39
7.2 Search Network . . . 39
7.3 Evaluation Networks . . . 40

A Derivation of Score Expectations **47**
A.1 Territory Score . . . 47
A.2 Area Score . . . 48

Bibliography **49**

Chapter 1

Introduction

Since the creation of computers, scientists have been writing game playing engines to play games as placeholders for intelligence. Games often offer nontrivial problems where humans develop their own strategies based off logic. Moreover, different entities can be compared directly thanks to the fact that the difference between players (human or programmed) being objectively measurable.

For the standard approach to creating strong engines for complete information, two player games, a tree based search with an evaluation function has to be applied, utilizing domain knowledge to return a particular score. This approach is extremely successful in chess where engines have eclipsed top humans on commercial hardware for about a decade, but much less so in Go, mainly due to the difficulty of creating effective heuristics for evaluation functions.

In this thesis we created a Go engine, called *Jorogo* [36], which develops on ideas from Matthew Lai's MSc thesis *Giraffe: Using Deep Reinforcement Learning to Play Chess* [32] as well as ideas from DeepMind's *AlphaGo* [39]. The Giraffe chess engine used reinforcement learning to train two networks which control the programs search and evaluation functions. Especially of interest is Lai's probability based search idea, which is described in Section 4.3. AlphaGo made headlines all over the world by first becoming the first program to beat a professional human player under tournament settings without handicap and then by becoming the first engine to beat a top professional player under the same settings.

1.1 Overview of Thesis

Chapter 2 and Chapter 3 are general chapters about Go, computer chess and computer Go. They are intended to give context to readers unfamiliar

with the topic of this thesis and to readers interested in working on similar projects.

Chapter 4 explains the minimax and alpha-beta algorithms for context before getting into Lai's probability based search. Probability based search was successfully implemented in Jorogo and proved to be extremely effective in the context of Go engine programming. This is an important result as regular depth first tree search algorithms are not considered effective in Go. At the end of the chapter a short explanation to Monte-Carlo Tree Search is given for context.

Chapter 5 explains the difficulties of creating effective evaluation algorithms in Go. It describes some key concepts that posed problems or were implemented in Jorogo, but removed before the final iteration.

Chapter 6 is arguably the most important chapter. Neural network architectures used in Giraffe and AlphaGo are described before reaching the primary contribution of Jorogo. The last part of the chapter describes the network architecture which was favored in Jorogo over the network architecture used in Google. The primary idea is to try to remove a degree of abstraction from the network and try to split up the network into three parts. The first part of the network consists of a series of convolutional layers which are which are activated by *ReLU* activation functions except for the last layer which is activated by hyperbolic tangent function and describes probabilities of intersections. The second part of the network is a translation unit which translates probabilities of intersections into an expected score dependent on rule set. The final part of the network is not strictly necessary but translates the expected score into a winning probability.

Chapter 7 contains an overview of some of the data obtained during the development of Jorogo. Many different features and ideas were tested over the course of development and very few of these are showcased in the final version of the code.

1.2 Jorogo Code Base and Contributions

During development we were able to confirm the effectiveness of Lai's *probability based search* (cf. Section 4.3), as well as come up with a novel architecture for the evaluation network. The architecture (described in Subsection 6.3.4) has several advantages over Google's *Value Network*. Although considering the size networks used in Jorogo the performance of this network was superior to a network with a similar (but much smaller) architecture to AlphaGo's value network, it is unclear how well the performance would scale to network sizes used in Google's paper.

Both temporal difference learning, as used in Giraffe, and supervised learning on games generated by self play, inspired by AlphaGo's reinforcement learning approach, were tried over the course of the project. Unfortunately, neither of the two approaches proved to be successful. Empirically, this may have been due to the size of networks generating games and variations of too low quality to match what is possible with supervised learning on expert quality games.

Creating the code for Jorogo was a great deal of work, but very rewarding work. The several thousand lines of original code in the final project do not describe the whole project. Hundreds of hours went into trying out different network architectures, bugfixing, brainstorming for new ideas and discussing algorithms with colleagues. Over the course of the project the author even ended up learning to play the game fairly well!

To give an example which implies some of the work which took place behind the scenes one can take a look at the generated neural network training snapshots. Towards the end of the project, when the code was running on an *Nvidia GTX 970* GPU, around 350′000 training iterations could be performed per hour. A snapshot of the network was taken once every 250′000 training iterations. By the end of the project, despite sometimes reusing name prefixes and sometimes deleting snapshot files, around 1′500 different snapshots were able to accumulate! Only a very small fraction of these are part of the final submission since each snapshot is several MB in size, however they proved useful in enabling comparisons such as the graphs in Chapter 7.

Jorogo is written in C++ and relies on the Caffe [30] deep learning framework. As such it can use the full power of a high performance GPU or alternatively run on a slow laptop processor[1]. Furthermore two files available online [27] from *GnuGo* were used in order to simplify the implementation of the GTP [28] protocol.

As GTP is implemented in Jorogo, it can be used in third party GUIs. On Linux a nice GUI to test the engine in is Quarry [15]. Jorogo also implements several commands not part of GTP. The perft command [6] is a useful command to either debug Jorogo or help debug other Go engines. To watch a demonstration game a useful command is `play_game_time` which takes as argument the number of milliseconds Jorogo should spend per move and plays a game from the current position until the end of the game.

[1] Which it did during the majority of the project's duration.

Chapter 2

The Game of Go

This section is intended for readers unfamiliar with Go, a two player board game primarily played in Asia. Most of the chapter describes Go specific terms and rules.

2.1 Etymology and History

The name *Go* is a simplification of the Japanese name for the game *igo*. The capitalization is used to differentiate the game from the equivalently named verb. Aside from *Go*, the Korean and Chinese names, *baduk* and *weiqi*, are also commonly used to refer to the game. The Japanese name originated from the Chinese name and both have the same meaning: "the game using pieces to surround" [12].

Go's origins can be traced back to China in the 4th century BCE, with historical works of the time referencing an event from 548 BCE[45]. At the time, the game was played on a 17×17 grid as opposed to the standard 9×9, 13×13 and 19×19 board sizes of today. It took another millennium for the game to reach Japan, where it rapidly gained popularity within the Japanese imperial court.

While Go has flourished in Asia, it did not catch on to the same degree in the West. Among those who have tried to popularize the game outside of the East, especially Edward Lasker[1] was instrumental for Go's development in the United States.

[1]Though Edward Lasker was an excellent chess player and won the US chess championships five times, he is not to be confused with the even more famous and distantly related second chess world champion and mathematician Emmanuel Lasker.

2.2 Rules of Go

Despite Go's long history and widespread popularity, there is no single set of rules. Rule sets used by different regions differ primarily on whether *area* or *territory scoring* is used to decide the winner at a game's conclusion. Most other differences between rule sets are fairly minor and tend to deal with edge cases such as whether to allow suicide (which is rarely a good idea) or position repetitions. An overview of the most common rule sets can be found in Table 2.1 at the end of this section.

2.2.1 Game Terms

In practice a Go board, called *goban*, is either a 9×9, 13×13 or a 19×19 grid as shown in Figure 2.1. While these sizes are by far the most common, other sizes have been used and there is no rule specifying the exact size of the grid[2]. From a mathematical perspective, the game need not even be quadratic or two dimensional. Indeed the rules can be easily defined for any graph[3]. In the present work we restrict ourselves to the most standard professional size of 19×19. Jorogo has been written in a way that it could be retrained to play any other quadratic board size given training data and changing just a few lines of code.

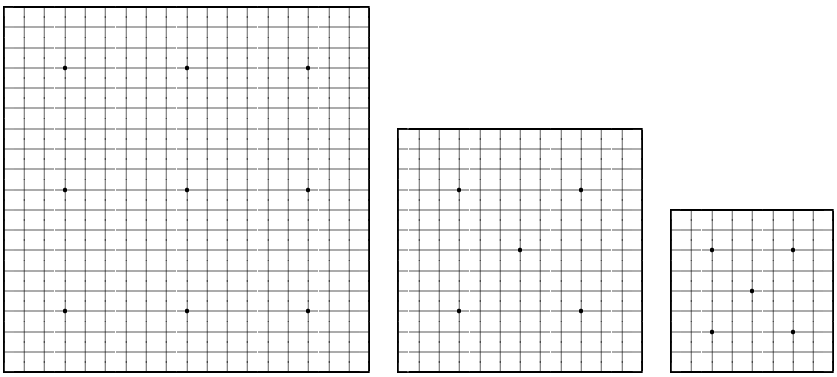

Figure 2.1: Side by side comparison of the three most standard goban sizes. The largest one is used most often in professional play and is the size supported by Jorogo

Vertices of the goban are called *intersections*. Pieces, called *stones*, may be placed on intersections. Contrary to chess, the squares of the goban are irrelevant, only the intersections and their connecting lines are important. Connected intersections are *neighboring intersections*. A *group* refers to a set

[2]The go text protocol[28] explicitly allows both larger and smaller sizes
[3]which do not even have to be connected!

of stones of the same color whose intersections form a connected subgraph of the goban.

A *freedom* or *liberty* is an empty intersection neighboring a group of stones. A single empty intersection may be a freedom for any number of groups. An *eye* is a freedom which is completely surrounded by stones of a single color. A group is *dead* if it cannot create two eyes without the opponent's cooperation. A group is *alive* if it is not dead.

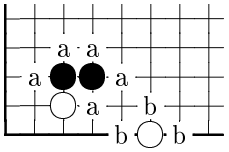

Figure 2.2: The black group has 5 liberties marked by "a" whereas the the bottom white stone only has 3 liberties marked by "b"

Instead of making a regular move, as described below, a player may *pass*. If both players pass during subsequent turns the game proceeds to the *scoring phase*. During the scoring phase the players decide how much of the board is controlled by each player[4]. The intersections each player controls is part of his *area*. Furthermore, each intersection that does not contain one of the player's own stones belongs to his *territory*. Depending on the rule set being used, intersections belonging either to the player's territory or area give *points* which are added to a player's final score. To offset the first player's advantage of getting to initiate play, compensation points, called *komi*, are added to the second players final score.

2.2.2 Legal Moves

In Go, stones are never moved. Instead they are placed onto empty intersections, with black initiating play. While in chess the number of legal moves tends to increase over the course of the game until numerous pieces have been exchanged, the maximum number of legal moves in Go[5] is in the starting position and the minimum number tends to get reached at the end of the game with a fairly linear decline in between.

If a stone is placed resulting in enemy groups having no freedoms, then these groups are removed from the game and give the capturing player points. If placing a stone results in neither an immediate capture nor in the placed stone's group having any freedoms itself, then the move is *sui-*

[4] to understand this decision process, refer to Subsection 2.2.3

[5] $361 + 1$ on a 19×19 goban and $81 + 1$ on a 9×9 goban. The $+1$ is due to the fact that passing is also a legal move.

cide which is illegal under most rule sets (see Table 2.1) and results in the immediate capture of the stone's group under all other rule sets.

Due to the *ko* rule, a player may not make a move which results in the exact same position as after his own previous move. This can happen if a player captures a stone by placing a stone whose only liberty is the newly freed intersection of the captured stone. The opponent may not play in the location of the captured stone unless it captures more than one stone. More complicated situations may result in *triple-ko* or other positions, however these are rare. In such an event the game may be annulled or the move resulting in the position repeating may be made illegal, depending on the rule set.

2.2.3 Scoring

During the scoring phase of the game the players must first agree on which groups are dead (i.e. which groups could not create two eyes without the opponent making a mistake) and which stones are alive. This part of scoring is very difficult for computers since the *life and death problem* is NP-complete [24].

While disputes on life and death are rare between humans, rule sets clearly define how to handle such cases. Chinese rules use area scoring as described below. In the event of a dispute the players simply keep on playing until the status of all remaining groups are agreed upon. One trick that an engine can utilize is to always claim that all groups on the board are alive, until the opponent agrees. In this case Chinese rules degrade to using Tromp-Taylor scoring which is much easier to implement in a Go engine and identical to area scoring with the assumption that all groups on the goban are alive.

After having decided which stones are dead, these are removed from the board and added to their opponent's respective score. Each stone that was captured during the game's play also gives the capturing player a point. Every empty intersection to which there is a path of empty intersections from a player's surviving stones belongs to that player's territory as well as his area. Every living stone belongs to the owner's area. Depending on the rule set, each player either gets a point for every intersection in his area or for every intersection in his territory. Finally, the komi value, agreed upon before the game, is added to the score of the white player. The player with the higher total score wins.

Table 2.1: Overview of the most common Go rule sets. Information is from the British Go Federation [10]

	Japanese, Korean	AGA	Chinese	SST (Ing)	New Zealand
Scoring	Territory	Area or Territory	Area	Fill-in	Area
Does it cost to make a move inside my territory?	Yes	No	No	No	No
Can you count points in a seki?	No	Yes	Yes	Yes	Yes
Is suicide allowed?	No	No	No	Yes	Yes
What happens if a position repeats?	"No Result"	Forbidden	Forbidden or drawn game	Restricted by the SST ko rule	Forbidden
Status of "bent four in the corner"	Dead	Play it out	Play it out	Play it out	Play it out
Komi	$6\frac{1}{2}$	$7\frac{1}{2}$	$7\frac{1}{2}$ ($3\frac{3}{4}$ by Chinese counting)	8	7
Placement of handicap stones	Fixed	Fixed	Free	Free	Free
Compensation for handicap stones	No	$n-1$ points assuming area scoring	n points	No	No

Chapter 3

Computer Chess vs Computer Go

As we shall see, a lot of factors contribute to making writing a strong chess engine easier than writing a strong Go engine. After having written a chess engine in my spare time and now a Go engine as part of my BSc thesis, I would like to shed some light on the difficulties in each case.

3.1 Available Resources

The *Chess Programming Wiki* [4] is easily the best resource for writing a game playing engine for any two player, perfect information game. While its primary focus is on computer chess, there are excellent pages explaining generally useful topics such as the Alpha-Beta search algorithm (see section 4.2) or transposition tables [8]. Chess specific state of the art algorithms such as magic bitboards [5] are also well explained. However, getting your hands on information for computer Go tends to be much harder. Most of the important algorithms are contained in papers, but there is no single resource available for Go as the CPW is for chess.

3.2 Open Source Projects

While Go has several notable strong open source projects such as *Fuego* [26], the top programs are generally all commercial or private engines such as Google's *AlphaGo* [39] and Facebook's *Darkforest* [42]. Hence, many state of the art ideas are not getting exchanged amongst one another. On the other hand, the highest rated chess engine in the world[1], *Stockfish* [23], is completely open source. Since 2005 computer chess has made huge leaps and bounds since *Fruit 2.1*[2], when the number two engine in the world [9] went open source. Stockfish is over 750 ELO stronger than *Fruit 2.1* on identical hardware [1].

3.3 Branching Factor

It is natural to assume that the primary difficulty of writing a strong Go playing engine is the immense branching factor[1] of full 19×19 Go. In 1999 Go playing engines faired no better on 9×9 boards than they did on 19×19[22]. It was not until the advent of Monte-Carlo search techniques in Go[2] that depth first search techniques proved to be truly effective. As such, the branching factor was not issue until 2006.

It is important to understand that it is not the branching factor that is relevant for an engine's search, but the effective branching factor, that is to say the number of reasonable moves in each position. That is illustrated in the following hypothetical game: The object of the game is to be the first person to have greater than a single point, which would result in the game being finished. The initial state implies both players having no points. The legal moves consist of giving the opponent any number of points $p \in \mathbb{N}$ with $p \geq 0 \wedge p < n$ where n is an chosen large number. It is a perfect information, two player game. It is easy to see that the only move that does not lose immediately is the move $p = 0$, in other words, there is only one reasonable move. While the branching factor of this game is n, the effective branching factor is 1! This means that we could write an engine that looks at all reasonable moves up to a large depth despite theoretically the branching factor of this game being arbitrarily large.

Stockfish in its most recent game in the TCEC tournament[41] reached a depth of 51 in roughly 800 million search nodes. This implies an effective branching factor of 1.5, which is much smaller than the average branching factor of chess, which about 35[3].

It is hard to estimate what the effective branching factor is for Go, however for special cases, such as ladder sequences, it should be close to 1: If the ladder is broken, then the capturing player should generally avoid it; however, if the ladder is not broken then the defending player should generally not waste time defending the lost stones.

[1] The branching factor is the average number of legal moves for any given legal position

[2] Despite the algorithm being used for the first time for go in 1992[19] it was not until 2006 that it gained any serious popularity after the creation of the state of the Monte-Carlo Go engine MoGo[29].

[3] In fact, Stockfish's branching factor is even smaller, since the given depth is simply the number of iterative deepening steps and does not contain quiescent search steps or various extension techniques which would further increase depth. The unique search node count is also much smaller due to researches from iterative deepening, aspiration windows, singular extensions and many other features result in significantly higher node counts. For an in depth overview of selected Stockfish features, see the chess programming wiki.[7]

3.4 Evaluation Heuristics

Chess has an extremely well researched set of heuristics, which can be used to efficiently predict the winning probability of each player. The value of individual pieces can for example be estimated by giving each piece a base value and multiplying the number of legal moves with the given piece by another fine tuned variable. While state of the art position evaluation is much more in depth, this method already allows a chess engine to play on a strong amateur level. Furthermore, this traditional approach is extremely fast: the number of legal moves for a bishop or rook can for example be calculated with a single AND operation, one shift operation, one multiplication and a population count function, all of which are typically implemented on a hardware level.

In Go, on the other hand, it is significantly harder to find good positional evaluation heuristics. Even solving the NP-complete life or death problem would not necessarily make it easier to evaluate most positions because often, a dead stone is still more valuable than no stone and can still have a significant effect on a games outcome. When analyzing games, strong Go players will often claim a move to be "inefficient" or a stone "not working hard enough," which, while making sense in the context of a game analysis is not easy to formalize mathematically[4]. The most notable algorithm in this area is Bouzy's algorithm which gives an estimate of the concept of influence[18]. Bouzy's algorithm is implemented in Jorogo, however it is not used. The algorithm is outclassed as an evaluation function by machine learning techniques, but it may be a reasonable idea to use it to generate feature planes.

3.5 Machine Learning

Both Monte-Carlo Tree Search and neural networks have been used in chess, however they have not nearly shown the same degree of success as in Go. Presumably the reason is that traditional evaluation functions are much more effective in chess than in Go and much faster, as previously mentioned. There are particular areas where some degree of pseudo-machine learning has been used effectively in chess. History heuristics, for instance, are tables that adjust their variables online based on previously successful search results. History heuristics are used for move ordering and a variant is implemented in Jorogo, but no longer used since its task is performed by the search network. Other examples include various King safety algo-

[4]For good examples of such Go game analysis I refer the reader to gogameguru. The two quotes are taken from An Young-gil's analysis of the first game between Lee Sedol and AlphaGo[48]

rithms, which often involve fairly abstract functions taking different scaling parameters and returning non-linear values from tuned arrays of variables.

Variables in chess are typically trained either via SGD or via hill climbing. For simple engines defining parameter values by hand is possible, but as an engine grows, the correct values for different evaluation heuristics interact making it very hard to do parameter fine tuning manually.

Chapter 4

Search Algorithms

The reason evaluation algorithms are necessary for games like chess is due to the fact that the search space is inexhaustible. In order to come up with reasonable moves within a certain time constraint, a game playing engine must therefore either decide what move it believes to be optimal simply based on the position at hand, or traverse a game tree by some set of rules. Historically game engines have been categorized as either Shannon A or Shannon B type engines, referring to whether all moves in a position are analyzed or moves are filtered based on some rules. Though state of the art chess engines use type A algorithms such as alpha-beta search, they use a lot of heuristics to recursively reduce the depth of improbable moves. In the following we will take a look at mini-max and alpha-beta as stepping stones to explain Giraffes probability based search which is also utilized in Jorogo.

At the end of this chapter we will take a brief look at Monte-Carlo tree search which was used in AlphaGo but not in Giraffe or Jorogo. Monte-Carlo tree search is more of a type B algorithm than a type A algorithm.

4.1 Minimax

Mini-max is a basic search algorithm. It is often explained and important to understand in order to implement strong game playing engines for a variety of games, but it is not generally implemented, since, as we will see, alpha-beta is simply better.

To understand how mini-max works, consider the following hypothetical game: Despite being a two player game, the only decision that takes place is by you, specifically, you have the choice on n different coins, all with a different probability of giving heads. For each coin you have an estimate for the probability of getting heads when flipping each coin. You win if the chosen coin flips heads whereas your opponent wins if the coin flips

tails. It is trivial to see that the optimal strategy is to examine each choice of coin, and pick the coin with the *maximum* probability of returning heads. If your opponent could however decide which coin is chosen, he would go through each coin and choose the one with the *minimum* probability of flipping heads.

Now consider that instead of having a choice of coins we have a choice of coin choices that we can give the opponent. In other words, we have a new game where we can choose which variation of the previous game to give the opponent. Yet again, the decision presents itself fairly easily, since we know the opponent will, regardless of the given set of coins, the opponent will choose the coin with the minimum probability of returning heads. In other words we can assign a probability of flipping heads to each *set of coins* that the opponent may choose from instead of just to each coin and interpret the result as the result of a coin flip in itself. Via depth first search we can find the optimal decision with absolute certainty.

It is easy to see that this game can be extended recursively to any depth and number of coins and in each case we can perform a depth first search on the game tree to find the optimal choice at the root. Since this depth first search alternates between a *minimizing* and a *maximizing* player, it is called the minimax algorithm.

4.1.1 Iterative Deepening

It is easy to interject here that minimax cannot be done exhaustively for a game like chess due to its large state space. This is of course true, instead in practice the search is only done up to a certain depth before the winning probability is estimated.

The question presents itself to what depth the calculation should be done before estimating the score. To solve this problem chess engines use a technique called iterative deepening. The engines search first with depth $:= 1$, followed by depth $:= 2$ and so on until the available time runs out. This may seem like a very large investment of time at first, especially since we might know in advance that the problem is easy enough that we are guaranteed to reach a certain depth. There are, however, a number of benefits, such as being able to use previous results to improve the order in which search is performed (which we will see is important), and the cost is in fact not so important due to the exponential cost of minimax algorithms relative to their depth.

4.2 Alpha-Beta

Alpha-Beta is an improvement over the minimax algorithm. A pure alpha-beta algorithm with no extensions will always return the same result as a pure minimax algorithm without extensions[1], but in much less time expected time as we will see.

Consider yet again the game described in the minimax description above. Consider the situation where we are deciding which set of coins to allow our opponent to choose from. Assuming we are using the minimax algorithm from above, we are performing a depth first search. Before having analyzed any choice from the root, we must assume that in the worst case we will have no probability of winning, and in the opponents worst case he will have no chance of winning. In our case it is thus said $\alpha := \text{MinScore} = 0 \wedge \beta := \text{MaxScore} = 1$. Assuming we have finished analyzing the first choice and come to the conclusion that it returns some probability of winning p_i we can update $\alpha = max(\alpha, p_i)$. Now, consider we continue the depth first search and take a closer look at the second choice at the root. In this case we find that the first coin the opponent could choose from has probability $p_j < \alpha$. Even if we assume that all the other coins are no better for the opponent, we would still not choose this option as our first option returned a better value for us! In other words we can save ourselves time and avoid looking at all the other coins of this second choice and move on to the next choice in the root. When a search node returns early like this it is said to *fail-high* or to produce a *beta cutoff*. Logically if the tree is deeper the opponent can follow the analog logic with β. Doing this recursively returns to the *alpha-beta* algorithm.

There are two things of note here. The first is that actually, alpha-beta is rarely implemented, instead symmetric evaluation functions are used and the *negamax* [31] algorithm is implemented. Negamax is a slight simplification of alpha-beta since the same function can be used for both the minimizing and maximizing players, thus simplifying code. Code listing 4.1 shows an example of how to implement negamax. The second, more important detail is that the order in which moves are looked at is very important. In the worst case where moves are looked at in order from worst to best, alpha-beta degrades to minimax, since no pruning can be done.

```
Negamax(Score alpha, Score beta, int depth) {
```

[1]The "without extensions" part is necessary, an advanced reader may consider for example the effect of adding only transposition tables. If in a sub-variation of a line ignored by alpha-beta contains a position searched to depth n, then the result may get saved and in a later part of the closer to the leaves it may get searched again due to some transposition. In this case the alpha-beta algorithm will return a less precise result then minimax. However, minimax, due to having to search more nodes up to a certain depth, may also overwrite valuable table entries, resulting in alpha-beta paradoxically returning better search results for a search to the same depth.

```
  if (depth <= 0) { return position.get_score(); }
  list<Move> moves = position.get_moves();
  for (Move move : moves) {
    move.make();
    Score score = -negamax(-beta, -alpha, depth - 1);
    move.unmake();
    if (score >= beta) {
      return beta;
    }
    alpha = max(alpha, score);
  }
  return alpha;
}
```

Listing 4.1: Sample C++ for the side invariant implementation of alpha-beta, called negamax

4.2.1 Move Ordering Analysis

To what degree move ordering in alpha-beta is important actually varies on quite a few factors. For example if we have an evaluation function returning 0 for every position (or only have equal coins in our previous example) we encounter in our alpha-beta or negamax searches, then trivially the order in which we search moves is completely irrelevant. To analyze the effect we will assume that only a single move is good enough to prune part of the search tree in any variation. In the optimal case with perfect move ordering we can calculate the number of leaf nodes in the search and compare them to the baseline minimax algorithm. This problem was already solved in the 1960's[25] to be $b^{\lceil \frac{n}{2} \rceil} + b^{\lfloor \frac{n}{2} \rfloor} - 1$ where b is the branching factor and n is the depth of the search tree. As can be seen this is in the order of $\mathcal{O}\left(b^{\frac{n}{2}}\right)$ however there is the interesting effect *odd-even effect* where the size increases differently depending on whether the current depth even or odd. This can be observed in Table 4.1.

4.2.2 History Heuristic

While most move ordering schemes in chess are domain specific (eg: Look at moves that capture the enemy queen first) the history heuristic is one which is more general. There are different variations of the idea but in principle it works as follows. At initialization we prepare a table with which we can index all legal moves in some way. The moves do not strictly have to be unique, in chess for example the table could be a 2-dimensional array referencing all possible origins and destinations[2] and in Go this could be

[2]So the size of this table would be 64

Table 4.1: Shown below are the minimum and maximum number of leaves of an alpha-beta search tree with a constant branching factor of $b := 250$ which should be a reasonable value for a mid game position in Go. The odd-even effect can be observed in that the node count only really explodes from even to odd search depths.

n	parity	$b^{\lceil \frac{n}{2} \rceil} + b^{\lfloor \frac{n}{2} \rfloor} - 1$	b^n
0	even	1	1
1	odd	250	250
2	even	499	62500
3	odd	62749	15625000
4	even	124999	3906250000
5	odd	15687499	976562500000
6	even	31249999	244140625000000
7	odd	3921874999	61035156250000000

just an array with one entry per intersection. Now, whenever we have a beta cutoff, we make the assumption that the move that was good enough to result in a fail high must also be good in other positions of the search and we give some bonus to the value associated to the move in the history table. The bonus given should somehow depend on the depth as there are typically many more nodes closer to the leaves than closer to the root. In chess a typical value would be depth^2. Finally whenever we are deciding what moves to search in what order, we first sort the move list in descending order according to their history values.

In Jorogo the history heuristic was implemented in order to have a better baseline alpha-beta algorithm. As we shall see however, this baseline version was outclassed by Matthew Lai's probability based search.

4.3 Probability Based Search

An issue that is inherent in alpha-beta is that it treats all variations equally. Regardless of how improbable it is that a variation is good, the variations always get searched up to a full depth, and regardless of how risky they are we don't search a line any deeper than the alternatives. There are a number of issues that arise from this most notably the horizon effect[16] which has been researched since the late 1960s.

In order to deal with this problem a number of ideas have been tried and tested for a number of different games and problems. Some ideas are very domain specific, such as having separate searches to deal with *ladder sequences* in Go, and some are more general, such as the *quiescent search*[3] ex-

[3]Quiescent search is essentially a method of searching through forcing moves to look for

tension in chess. In state of the art chess a whole cocktail of them tend to be used in each engine, eg: check extensions, late move reductions, recapture extensions, singularity extensions, razoring and so on. It is not an easy task to understand how these different extensions and reductions interact and many of them do not generalize to other games or problems.

Instead Lai suggests a much more general solution. His idea is to spend time on each subvariation that is proportional to the probability of the move being part of the games actual sequence. Of course, this is not a probability we know in general, so what he suggests instead is to train a neural network to predict the probabilities of different sub-variations.

Since in Go pieces never move but are placed, probability based search is easier to implement in Go than in chess. Specifically, in chess a call to the neural network which predicts move probabilities must be made separately for each legal move. On the other hand a single neural network with 362 outputs[4] can be trained in Go to predict the probability of all moves in a given position at once.

4.3.1 Performance

Probability based search was successfully implemented in Jorogo and compared to baseline versions in two separate matches.

In each case the respective versions of Jorogo were identical, using "Chinese" evaluation neural network to evaluate winning probabilities unless encountering successive pass moves in the search tree, in which case Tromp-Taylor scoring was used to determine either a max or min score depending on whether the engine would be winning or losing if both versions passed.

The first baseline version used a regular alpha-beta search function and ordered moves according to the history heuristic. The second baseline version also used a regular alpha-beta search function, but called the same neural network which is used in the standard Jorogo version to predict move probabilities. This baseline version ordered the moves according to their probabilities, but only searched $\sqrt{n}$ of the respective n legal moves in each position to full depth in order to save time. The remaining moves were searched at depth $n - 1$.

The two matches consisted of 50 openings each. Each opening consisted of two random moves before relinquishing control to the engines. Each opening was played twice, with the respective engines alternating colors. The games we played at a speed of 1 second per move and adjudicated in the event of games taking more than 600 moves, which rarely happened.

tactically stable positions.

[4] one output per intersection plus one output unit for the pass move

Both matches ended in one-sided victories for the probability search based engine. Against the history heuristic based search version, the main version won 97-3 whereas against the version using a neural network to order moves the score ended 98-2 in the main versions favor. Closer empirical analysis suggests that the probability search based engine was superior due to the higher depths that it reached compared to the other two versions of the engine. Early in the game the probability search based engine reached depths of around 10 while the other two versions did not manage to reach higher depths than 2 within the same time constraints.

4.4 Monte-Carlo Tree Search

From 2006 until the announcement of AlphaGo's results against Fan Hui, state of the art Go engines were dominated by the Monte-Carlo tree search algorithm. Monte-Carlo tree search was not implemented in Jorogo, but it was an important part of AlphaGo. The idea behind it is to estimate the winning probability in a given position by simulating games at ultrasonic speeds and then using these evaluations to guide search and determine what positions to do more simulations.

Monte-Carlo tree search follows the following steps[21]

1. *Selection*: Starting from the root, recursively go through the current search tree to some leaf node. The way this is done may be defined differently.
2. *Expansion*: New leaf nodes are created according to some rules and added to the tree.
3. *Simulation*: A game is played from the selected position. Depending on the result of the game, a score is updated.
4. *Backpropagation*: The result of the simulated game is propagated back up to the root according to some rules.

These steps are repeated some number of times before the engine picks the move with the best score.

Chapter 5

Evaluation Algorithms

Most perfect information, 2 player games interesting enough to create a game playing engine typically have game trees which are far too large to search exhaustively. This includes both Go with over 10^{170} legal positions [43] as well as chess, whose game tree complexity was lower bounded to 10^{120} by Claude Shannon [38]. This means that at some point, any reasonable search algorithm must return an estimate as to how good a position is for either side. Functions which return such estimates are *evaluation algorithms* or *evaluation functions*.

The following sections attempt to discuss why heuristic based approaches are better in chess than in Go, discuss the special case of evaluating final positions, describe Bouzy's algorithm to determine influence, briefly discuss Monte-Carlo evaluation functions and finally take a look at using neural networks for evaluation functions.

5.1 Heuristic Based

Heuristic based evaluation functions are extremely good in chess. Since a small difference in the number of pieces each side has tends to be decisive, most games have roughly equal material, so it makes sense that comparing the pieces on the board is a good way of evaluating the position.

In Go however things are very different. There is exactly one type of piece, a stone. Pieces do not have any mobility either, so individual stones are more difficult to compare. The biggest problem however is that not stones are important, but territory, coming up with a mathematical measure of how much territory belongs to one player or the other is non-trivial.

The most common trade-off in Go is territory vs influence. The idea is that one player allows the other to set up his stones in a way where he is guaranteed to make some number of points. As compensation these stones

are typically surrounded to a degree, meaning they have little effect on the game outside of giving the player said points. Unfortunately "territory" and "influence" are rather abstract terms. Unless an area is completely enclosed it is nontrivial to define what parts belong to territory and which do not, and influence is an even more abstract concept essentially meaning "effect on part of the board". Bouzy came up with a heuristic to estimate influence, as we shall see.

5.1.1 Bouzy's Algorithm

Of the few heuristic based evaluation parameters known in Go, of special note is Bruno Bouzy's algorithm for evaluating influence [18]. Bouzy combined previous work from Zobrist [49] with the morphological operations [37] of dilation and erosion used in visual computing.

Implementations of Bouzy's algorithm were successfully used in GnuGo [20] as well as in Indigo [17]. The algorithm was also implemented in Jorogo, but not used in any of the more recent algorithms. It may be a good idea to try using the results of the algorithm to generate features for a neural network, however this is outside of the scope of this work.

5.2 Final Positions

Instead of being able to evaluate any position, in Go it is a necessary subproblem to be able to objectively evaluate final positions where both players have passed. Contrary to in general positions, influence is irrelevant, only territory. If it is known which groups of stones are alive and which groups are dead then the final result can be easily evaluated. Unfortunately, while humans rarely dispute over the life and death status of stones, solving final positions algorithmically is much more challenging.

According to Müller [34] using a combination of static analysis and search results in 75% of board points remaining unproven on a set of standard human final positions. Using a neural network a team of researchers was able to improve this result massively, correctly scoring 98.9% of positions [44]. In order to achieve this latter result researchers had to manually analyze thousands of games themselves, which was not feasible for the scope of this bachelor thesis.

Somewhat of a hack to get around the life and death problem is to use Chinese rules[1] and claim every stone on the board is alive. Since under Chinese rules disputes over life and death are solved by resuming play and stones are alive by default, Chinese rules degrade to using Tromp-Taylor

[1] Which are very commonly used anyways. AlphaGo exclusively plays with Chinese rules.

scoring instead of regular area scoring in this case. This is not possible under Japanese or Korean rules, since they use territory scoring which penalizes placing superfluous stones in your own territory.

5.3 Monte-Carlo

As noted briefly earlier a Monte-Carlo approach may be used to evaluate a position. This means a lot of ultra high frequency games are played from a given position and the winning probability of each side is estimated to be the same as the proportion of games each side has won. One problem with using Monte-Carlo functions to evaluate positions is that final positions still need to be evaluated somehow in order to decide who won each individual simulation. While in chess this is trivial, things are much harder in Go, as noted in the previous section.

5.4 Neural Network

Neural networks can be used very effectively as evaluation algorithms, as demonstrated by AlphaGo. It is much harder to train a neural network to evaluate positions then to suggest good moves however. Not using an evaluation function at all, instead simply accepting the search network's suggestion turned out to be a viable alternative.

Comparing the main version of Jorogo to one that completely foregos evaluation based searched resulted in an engine that was weaker than the main version but was still able to win about one in four games at 250ms/move.

Chapter 6

Neural Network Architectures

While neural networks are indeed complicated black box algorithms at runtime, there are many structural details that can be adjusted and fine tuned before training even begins. Whether to use convolutional neural networks, which activation functions to use where, what loss function to optimize, how many layers to use and how many units to use in each layer are all important questions that need to be answered when designing a neural network solution. In the following we discuss design decisions as well

6.1 Networks in Giraffe

Matthew Lai used two different neural networks in Giraffe. The first neural network, the *evaluator network*[32, p.17-26], was used to estimate the winning probability of each player. The other network, the *probability estimating network*[32, p.29-32], was used to estimate the probability of each move being the optimal move in any given position. The Lai did not use machine learning libraries for his engine, instead he implemented his own relying on linear algebra libraries.

Convolutional neural networks are not mentioned in the thesis at all. It was not specified why convolutional neural networks were not used. While the reasoning can only be speculated upon it is important to note that convolutional networks benefit disproportionately from GPU programming, which would have been a fair bit of extra work for an MSc thesis.

6.1.1 Evaluator Network

The evaluator network uses a piece slot system to define its inputs. Each piece has 3 basic features, one indicating its x-coordinate, one indicating

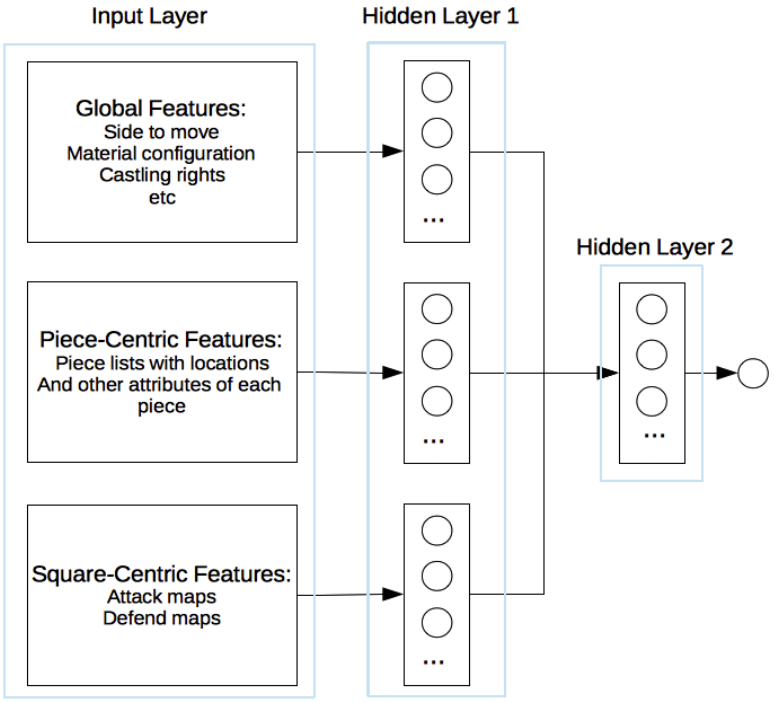

Figure 6.1: Illustration of the network architecture used in Giraffe. Source is the associated MSc thesis [32]

its y-coordinate and the last one indicating whether or not the piece exists[1]. Each piece also has some additional features such as whether or not the piece is defended and how many legal moves a piece may have. Finally the input contains more general position information, such as the castling rights, whose move it is and the material configuration.

Lai notes that his board representation has the flaw of not being able to represent every legal position due to promotions allowing more of certain pieces than in the starting position, but correctly notes that such positions are very rare and shouldn't usually matter.

After the input layer, the evaluator has two fully connected hidden layers activated via ReLU activation functions, followed by a linear output unit activated by a hyperbolic tangent(TanH) function. Figure 6.1 is taken directly from Lai's MSc thesis and illustrates the network architecture used in Giraffe.

[1]Ie if a piece has been exchanged or a pawn promotes, then the piece no longer exists. Logically this last variable is not needed for the two kings

Training

The in order to train the evaluator network Lai built a training set by taking 5 million random positions from games available from CCRL [1]. One random move was made in each position before running the TD-Lambda algorithm [40] on the result.

6.1.2 Probability Estimating Network

The probability estimating network uses the same features as the evaluator network plus some additional features describing move source and destination, the moving piece's type and the move's *rank* which is a feature describing how the move relates to other legal moves in the position. The output unit is activated with a logistic activation function resulting in a value in $[0, 1]$. This was done to simplify gradient computation. Aside from that the probability estimating network has the same structure as the evaluator network.

Training

In order to generate training samples, 5 million positions were taken from CCRL. Training positions were then sampled from the internal nodes of searches on these positions. Training samples were labeled by performing a time limited search the training positions and then saving the best move. Finally, gradient descent was performed on the training samples.

6.2 Networks in AlphaGo

DeepMind trained several networks in different ways for different tasks. Three different network architectures were used. Just like Giraffe's *evaluator network*, AlphaGo's *value network* is used to predict the probability either side will win the game in a given position. The decision which moves to analyze and in what order is made by a *policy network* in the main game tree. Since a call to a large neural network is very expensive, DeepMind implemented a faster, less precise neural network, *fast policy network*, to sample actions during Monte-Carlo simulations. The fast policy network is outside of the scope of this thesis.

6.2.1 Input Channels

AlphaGo's networks used features structured as 19×19 binary bitboards. For example a feature describing how many liberties each group of stones has was split into 8 bitboards. The first bitboard had a value of 1 for every stone that was part of a group with only a single intersection and 0 everywhere else. The second bitboard was used to describe groups with

Table 6.1: AlphaGo's feature planes [39]. The player color feature plane was only used in the value network.

Feature	# of planes	Description
Stone color	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (number of adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether the move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

two liberties and the last bitboard was used to describe groups with eight or more liberties.

Two constant channels were added in order for the network to be able to recognize features related the the goban's edge.

Instead of describing features by stone color, features were defined by the player to move. For example instead of having three channels to described black, white and empty intersections respectively AlphaGo has three channels describing intersections with the stones of the player to move, intersections containing his opponents stones and empty intersections respectively. An overview of all input feature channels is given in Table 6.1.

AlphaGo's networks used identical inputs for both policy and value networks with exception of the a single channel. This channel describes whether the player to move places black or white stones and is necessary to take komi into consideration.

6.2.2 Policy Network

Abstractly, the policy network's architecture can be described as having a set of convolutional layers which are activated by rectifier activation functions [35] followed by a softmax function. More specifically there are three types of convolutional layers. For the first layer, the input is padded with 2 zero collumns along the edges to create an image with a height and width of 23. Then filters of dimension 5×5 are convolved with stride 1 over the image to create a new image of size 19×19 before the activation function is called. The next kind of layer pads the image to 21×21 and convolves a filter of just 3×3 with stride 1 to create a new 19×19 image, before again

calling the activation function. This second kind of layer is used for layers $2 - 12$. Finally one single 1×1 filter is convolved over the image to create a final two dimensional image output from the convolution. Different numbers of filters were tried for the first two kinds of convolutional layers, however the match version used 192 filters.

Training

Policy networks were trained using the *REINFORCE* [46] algorithm in order to generate self play games to train the value network with. The version of AlphaGo that faced Lee Sedol however used a policy network trained via regular supervised learning on expert quality human games.

6.2.3 Value Network

The value network has the same convolutional architecture as the Policy Network with exception of the last layer getting activated via rectifier function instead of a softmax function. Following the convolutional part is a fully connected layer with 256 rectifier units. The final layer is another fully connected layer with a single unit activated by a hyperbolic tangent function.

Training

DeepMind explained that training their value network directly on games from the *Korean Go Server* [14] lead to very large over-fitting with the network essentially memorizing the games. They argued that this was due to the high correlation between different positions in the dataset. To reduce this bias they created a dataset of 30 million positions, each from different games, via self play of different versions of the policy network.

6.3 Jorogo

Two different neural networks are at the heart of Jorogo. The *search network* is similar to AlphaGo's policy network and guides the search. The *evaluation network* is similar to Giraffe's evaluator network and AlphaGo's value network. Its job is to predict winning probabilities.

6.3.1 Activation Functions

Based on recent work [47] leaky rectifier functions were chosen over regular rectifier activation functions. Earlier in development both parametric rectifier activation functions as well as regular activation functions were tried with comparable results to the final version.

6.3.2 Stride and Pooling

State of the art convolutional neural networks often utilize pooling layers to reduce the size of hidden layers. While pooling layers were tried early in development they did not prove to be successful. While 19×19 is a very large size for a traditional board game, it is small for an image. The benefits of using pooling layers is thus diminished somewhat. Furthermore in a game like Go whether or not a set of stones is one row to the left or right can mean the difference between winning or losing a game, whereas in image processing, if a cat's eyes are one pixel further left or right is fairly irrelevant.

Out of the same reason a stride value of 1 was chosen over larger values.

6.3.3 Search Network

Both the search network and evaluation network were heavily inspired by AlphaGo's neural network architecture. Both networks start with the same convolutional layers. Compared to Google's convolutional network architecture the main differences are that there are fewer filters, fewer layers and, as previously mentioned, the use of leaky rectifier activation functions as opposed to regular activation functions.

Contrary to AlphaGo's policy network, Jorogo's search network was designed to also give the probability of a player passing. The output of the convolutional layers is activated just like the other layers with a leaky rectifier unit. These 361 outputs are first fully connected with a single unit which is also activated via leaky rectifier function. This single unit is used to give the pass probability. The aforementioned 361 units plus the pass probability unit are activated via softmax function to get the probabilities of each move.

Training

The search network was trained via supervised learning on a dataset combined from games from the *Korean Go Server* [14] and the *Go4Go* [33] game database. Gradient descent was run with a cross-entropy loss function. Whenever learning stalled for a prolonged period of time the learning rate was lowered until the validation error stalled.

6.3.4 Evaluation Networks

Two (or three, depending on how you count) architectures were in the final version of Jorogo. One was specifically trained to be similar to googles architecture while the other was designed in a more creative way that allows Jorogo to play with any Komi as well as with both area or alternatively territory scoring.

Google Inspired Evaluation Network

After the convolutional part of the network, which is identical to the same structure within the search network, the google inspired version of the evaluation network has a fully connected layer of 256 rectifier linear units followed by another fully connected layer with a single output. This last unit is activated via hyperbolic tangent function. As can be seen the structure of this network is essentially the same as AlphaGo's policy network, with the difference that all it uses leaky rectifier linear units, less filters and less convolutional layers.

Area and Territory Evaluation Networks

Google's value network is a fairly abstract piece of mathematics. It combines features like whether or not the side to move gets a bonus from komi with a concrete board representation. The output of the convolutional part of the network is not really interpretable and the only thing that is truly known is that there is a strong correlation between a positions winning probability and the value network's output.

Jorogo tries to improve on that by introducing a two in one network architecture. Specifically, while the convolutional layers have the same structure as in the google inspired evaluation network, the last convolutional layer is interpreted as predicting the probability that a given intersection will be part of either side's area at the end of the game. These probabilities will be referred to as *intersection probabilities* for the rest of the thesis. These probabilities can be taken and used to calculate the expected score from each individual intersection. The calculation is different depending on whether area or territory scoring is used[2]. Summing up the individual score expectations from each intersection with komi as well as with the difference of previously captured stones (stones the current player captured minus stones his opponent captured) leads to the expected final score. This final score is what we are actually optimizing and it would be reasonable to simply use this as the evaluation output. Instead, two fully connected layer followed by a connected single hyperbolic tangent activated unit was added to translate the expected score into an expected winning probability taking values between $[-1, 1]$. This has the benefit of defining minimal and maximal output values, making results comparable with the Google inspired network and being able to reuse existing code.

This structure has a lot of benefits over the Google inspired evaluation network. The network takes into account and can handle different komi values

[2]In the code these two related networks are referred to as Chinese and Japanese evaluation networks respectively since the primary difference between these two most popular rule-sets is that the former uses area scoring whereas the latter uses territory scoring. In this text we will refer to them as area scoring network and territory scoring network.

as well as the number of previously captured stones[3]. The network could be trained on a set of games using either area or territory scoring (Chinese or Japanese rule games for example) and would be able to play games with the alternate method of scoring equally well. Empirically the area and territory network architectures needed much fewer games to train without over-fitting at all. Specifically the network was able to use a set of around $70'000$ games with Japanese rules to train with only slight over-fitting whereas the Google inspired network was already clearly over-fitting on a dataset with well over twice the amount of games.

A fair interjection is that the area and territory evaluation networks do not optimize winning probability but only expected final score. Indeed there are cases where the wrong choice may be made, for example, given two moves, one guaranteed to result in a final score of $+0.5$ and the other giving $+1.5$ with a probability of 0.9 and -0.5 with a probability of 0.1 the area and territory networks would choose the second move even though there is a 10% chance of losing! There is also the other side of the coin however in that what we as humans interpret as good play is also not strictly moves increasing winning probability. Eg given two options, each without any chance of winning, we consider the move better which loses by a smaller margin! It is of importance to note that assuming the network could reach a state where it is perfect, meaning it always calculates the exact final score with perfect play, it will never lose a position which is winning.

While the area and territory evaluation network structure makes sense and is fairly intuitive if the convolutional layers do indeed predict the probability of intersections belonging to each respective player, there is an issue in that it is hard to guarantee the network is doing exactly that. One way to try to ensure that the network is indeed returning such probabilities and not something more abstract is to train for this directly. Labeling can be done by evaluating final positions and defining which intersections belong to what area. These labels may then be used for any training position taken from the connected game. Compared to directly training the network to predict winning probabilities this has a major benefit in that each training sample has a label for each individual intersection instead of just a single label. This approach was done as a pre-training step in Jorogo. Since labeling final positions is nontrivial, as is discussed in Section 5.2, it is natural to use a reinforcement learning approach of evaluating final positions with a pre-trained network and then training again with the new labels. Many different options are possible, but analyzing these to greater depth was outside of the scope of this thesis.

In Jorogo the most pragmatic approach was settled on, training to predict

[3] AlphaGo instead can only play with Chinese rules and a komi of 7.5. In order to use other rules the neural networks would have to be retrained.

the winning probabilities and then testing whether the result made sense. Testing was done in a fairly creative and indirect way. The point was to train a territory network on a set of games which used territory scoring and reusing these weights in an area network. If the network were would be returning something abstract then it is unlikely the area network would be returning the probability of winning under area scoring rules, so a simple way to test it is to have Jorogo play against itself with one version utilizing the territory scoring network and one version utilizing the area scoring network. Since final positions are evaluated via Tromp-Taylor scoring the entire match should be in favor of the area network if everything functions as expected. 250 different openings of two moves each were played with alternating colors at a rate of 250ms/move. The final score was a convincing victory in favor of the area evaluation network, winning $299 - 201$ which is statistically significant at the 99.9% level[4].

As for the performance of the area and territory evaluation networks: As we will see in the next chapter the networks were able to learn much faster than a network using the Google inspired network architecture and were able to outperform it in a match $453 - 47$ despite being worse at the result prediction task.

Area and Territory Translation Functions An important detail of the area and territory evaluation networks is the math to transform intersection probabilities into an expected score. Equations 6.1 and 6.2 show how to get the expected territory and area scores respectively given the intersection probability $p(x)$ for some intersection x. The y in these equations is a variable that takes the value 1 if it has the same color as the player evaluating, 0 if it is empty and -1 if the stone belongs to his opponent. The derivation to get these equations can be found in the appendix.

$$T(x, y) := 2p(x) - 1 - y \tag{6.1}$$

$$A(x, y) := 2p(x) - 1 + y^2 \cdot \big(p(x) - 0.5 \cdot (y^2 + y)\big) \tag{6.2}$$

[4]Given the null hypothesis that the winning probability is even, the probability of this result is a simple binomial function: $\sum_{n=0}^{201} p^n * (1-p)^{500-n} * \binom{500}{n} = \sum_{n=0}^{201} \frac{1}{2}^{500} * \binom{500}{n}$. Wolfram Alpha evaluates this to be smaller than 10^{-5}

Chapter 7

Results

7.1 Internal Matches

During the development of Jorogo most testing was informal and based intuitively on the validation test results or the subjective impression different versions of the engine gave when playing. For example at one stage there was a baseline version of Jorogo which relied on a simple negamax search implementation with history heuristics and used the Tromp-Taylor score as its evaluation function. This naive engine was weak, but was able to keep pace with other versions at the time. Eventually there were some improvements and a new version was strong enough to very convincingly defeat the former baseline in a game with a 9 stone handicap[1]. Despite a single data point not being formally significant, this result was significant enough to create a new baseline.

Closer to the project deadline different features were tested more formally by playing internal matches of $100 - 500$ games and analyzing the results. For these matches only a single variable was altered at a time to compare with the main version of Jorogo. The main version of Jorogo uses a neural network to run a probability based search and another neural network as an evaluation function, unless there are subsequent passes in the search. In the case of subsequent passes the program will return the constant score for a win or loss respectively. As a result of this, a losing engine will not pass unless it believes itself to be losing anyways or the search is not deep enough to consider the pass reply after its own pass move. The latter case is unlikely to be an issue unless no actual search is performed. The main version's evaluation network is an area evaluation network which was originally trained as a territory evaluation network on games played with Japanese rules in a supervised learning fashion.

[1] This is the largest handicap that is usually given between human players.

Matches were usually played at a rate of 250ms/move with a couple of exceptions. When the search function was adjusted to use a more traditional alpha-beta search algorithm the time control was increased 1s/move to try and increase the amount of searches that reached at least depth 2. A set of matches, shown in Figure 7.1, specifically tested the effect of different time controls on playing strength. This is an important test as one of the primary motivations for any tree based search is that it should be able to scale with time!

For each match openings consisted of two moves sampled uniformly from all legal moves. Since the openings are so short, the probability of any opening being far enough out of balance that one side cannot win is not very large. To further equalize chances, each engine plays both sides of each opening once.

Table 7.1 contains an overview of various match results. The matches with varying search functions and evaluation functions are examined in Subsection 4.3.1 and Subsection 6.3.4 respectively. We will briefly take at look at the effect of time on performance here.

Table 7.1: Overview of select internal match results. The reinforcement learning based evaluation networks were trained as described in Subsection 6.3.4

Variable changed	Version 1	Version 2	Result	Time control
Time	network suggestion	200ms/move	$25-175$	see versions
Time	100ms/move	200ms/move	$78-122$	see versions
Time	200ms/move	200ms/move	$106-94$	see versions
Time	300ms/move	200ms/move	$117-83$	see versions
Time	400ms/move	200ms/move	$130-70$	see versions
Time	500ms/move	200ms/move	$139-61$	see versions
Time	600ms/move	200ms/move	$138-62$	see versions
Search Algorithm	probability based search	alpha-beta with history heuristics	$97-3$	1 second/move
Search Algorithm	probability based search	alpha-beta with neural network	$98-2$	1 second/move
Evaluation Algorithm	area evaluation network	territory evaluation network	$299-201$	250 ms/move
Evaluation Algorithm	area evaluation network	google inspired evaluation network	$453-47$	250 ms/move
Evaluation Algorithm	area evaluation network	RL google inspired evaluation network	$132-68$	250 ms/move
Evaluation Algorithm	area evaluation network	RL area evaluation network	$142-58$	250 ms/move

7.1.1 Time Scaling

Based on the results in Figure 7.1 we can see that the strength of the engine seems to scale slightly less well than the proportion of time it spends. The most important thing here is that the engine scales at all with time. This was not necessarily the case before the widespread usage of Monte-Carlo Tree Search and is arguably one of the reasons for the misconception that Go engines are not brute force engines compared to chess engines[2].

If we take a look at Figure 7.1, then we can note that a doubling of the time allocated to the engine from 200ms/move to 400ms/move results in a winning probability of 65%. This winning probability is expected from two players with an Elo [11] difference of about 108 points. Alternatively we can notice that since the probability of beating the base version of 200ms/move with 100ms/move is 39% we can infer a winning probability of about 61%. This would be expected with a rating difference of 78 Elo points. According to a forum comment [13] by HG Muller [3], a doubling of time per move is usually worth about 70 elo[3], so empirically Jorogo scales at least as well with time as an average chess engine.

7.2 Search Network

Shown in Figures 7.2 and 7.3 are the probabilities of moves played by expert players as estimated by the search network. While Figure 7.3 shows how the accuracy of this estimation changed over the course of 10 million iterations, the more interesting information is contained in Figure 7.2.

Intuitively, since the number of legal moves drops, one would expect it to be easier to predict moves later on into the game. The result in Figure 7.2 implies that this is not the case. Instead the effective branching factor actually seems to *increase* over the course of the game and moves become less predictable. The figure also shows that the effective branching factor to be much much smaller than the actual branching factor in any case.

The exception is before any moves have been played at all. This makes sense since without any stones to orient the play is a lot more free and due to symmetries if certain moves are good then at least 3^4 other moves must be equally good due to symmetries.

[2]Curiously Monte-Carlo algorithms are arguably the most brute force approach imaginable aside from trying to evaluate a game's entire game tree. An algorithm having a brute force component is not inherently something negative.

[3]I have had several discussions about this with engine developers. Top chess engines scale by about 50 Elo per doubling of time. In chess the drawing margin grows with the level of play, so this may have an effect on this number.

[4]7 if the move is not on one of the boards diagonals. Admittedly the center of the board has no symmetries, but that is an exception.

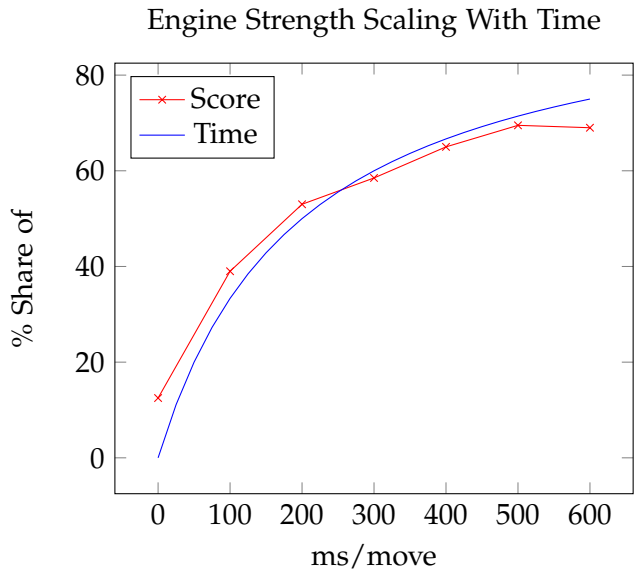

Figure 7.1: Graph of match results with Jorogo playing against itself with different time limits. 200 games were played for each data point. The blue line represents the total percentage of time allocated to the engine and the red line represents the percentage of games it was able to win at that time control.

In any case the fact that a move played in a game on turn 10 is given an average probability of around 40% is an interesting result in itself. The fact that after the opening this probability never drops far below 20% is a good sign that a probability based search approach to Go can be successful.

7.3 Evaluation Networks

The evaluation networks described in Subsection 6.3.4 do not have surprising results on their own. When compared however there are some interesting aspects. First off, the territory evaluation network was trained on a subset of the of the data that the google inspired evaluation network was. Despite this, the territory evaluation network actually over-fit *less* than the google inspired network. Testing larger networks was outside of the scope of this work, however the territory evaluation network's architecture may be an excellent choice for larger networks for this reason alone.

As can be seen in Figure 7.4 and Figure 7.5, both the territory network and google inspired networks can more accurately predict a winner as time progresses. This is natural as the probability of winning should be close to equal for both players in the beginning of the game but as mistakes are made the balance can shift towards the eventual winner. While the territory evaluation network already learned a great deal after 250′000 iterations

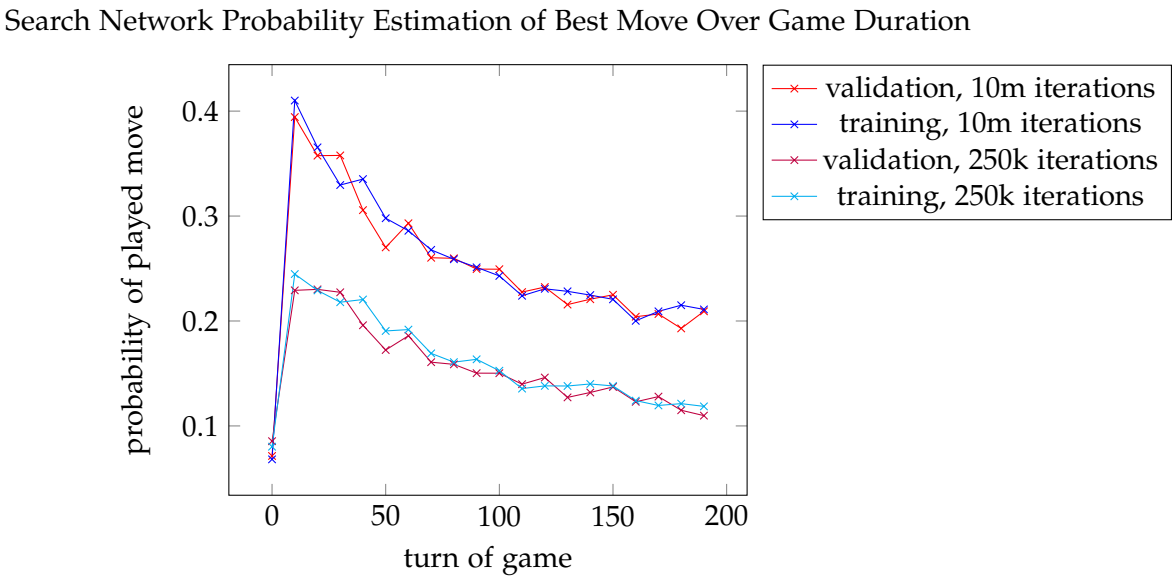

Figure 7.2: Shown above are the probabilities estimated by the search network for moves played in human expert games. The lines at the top are the values after 1 million training iterations while the values at the bottom are after 250 thousand training iterations. The training and validation set errors are based on sets of 960 games each. If a game lasted less than 200 moves it is ignored for the moves that did not get played. Pass move probabilities are also estimated.

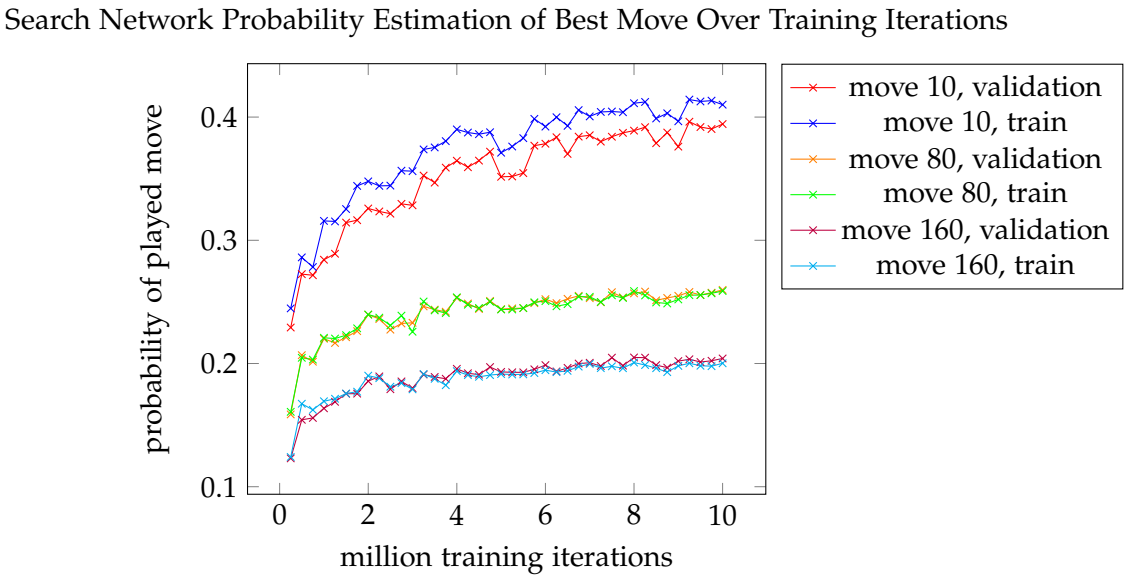

Figure 7.3: Shown above are the probabilities estimated by the search network for moves played in human expert games. The training and validation set errors are based on sets of 960 games each. If a game lasted less than 200 moves it is ignored for the moves that did not get played. Pass move probabilities are also estimated.

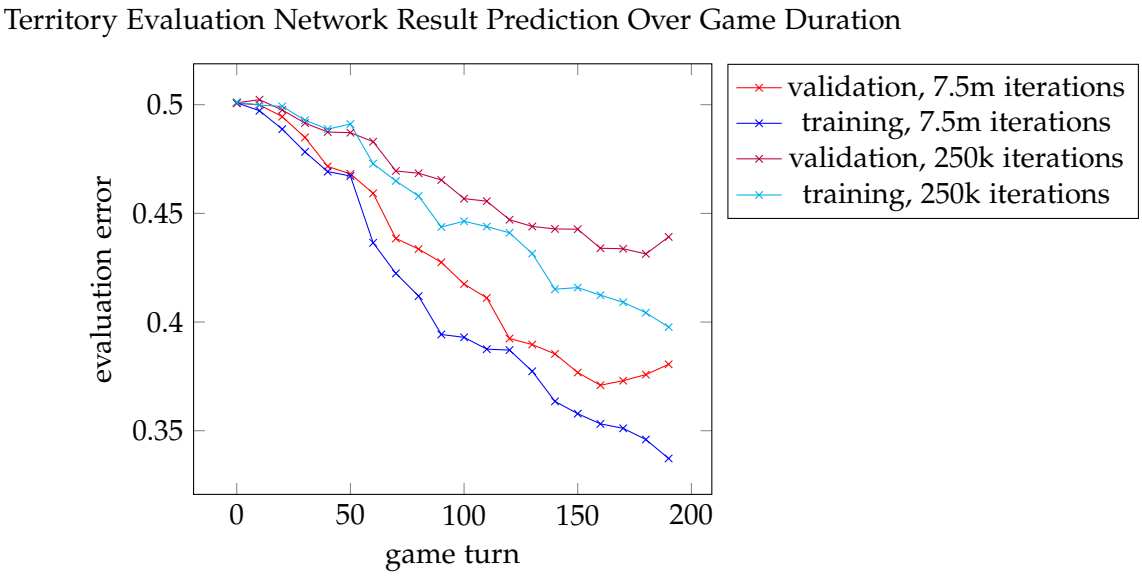

Figure 7.4: Shown above are the root mean squared errors of winning probabilities over the course of an average game estimated by the territory evaluation network. In the beginning of the game it cannot be predicted who will win, hence it makes sense that that the RMSE starts at 0.5 and drops from there. The results are based on 960 training and 960 validation games. Here a quick note that a similar graph can be found in Google's paper [39] but labeled with the mean squared error instead of the RMSE. It is probable that this is simply a typo since the error also starts at 0.5.

(when the first snapshot was taken) the google inspired evaluation network took a while longer to learn. By the one million iteration mark, the network seems to have made good progress.

It is interesting to note that the form of the training and validation errors in Figure 7.4 look very similar after 250 thousand and 7.5 million iterations with the exception that the errors have gotten smaller.

Figure 7.6 and Figure 7.7 show how quickly the territory evaluation network and google inspired evaluation network were able to learn over time. The axises are important to note as the territory evaluation function was trained for much shorter a duration. To get a better idea of what is going on take a look at Figure 7.8. Despite starting out less accurately the google inspired evaluation network catches and surpasses its competition in the task of predicting a winner. It is interesting to note however that despite doing better at this task, when used in Jorogo it is weaker than the territory evaluation network. This may be, because the territory evaluation network is rather artificially translating an expected score into a winning probability which makes it harder to match the google inspired evaluation networks performance however does not effect how well it plays since the move that

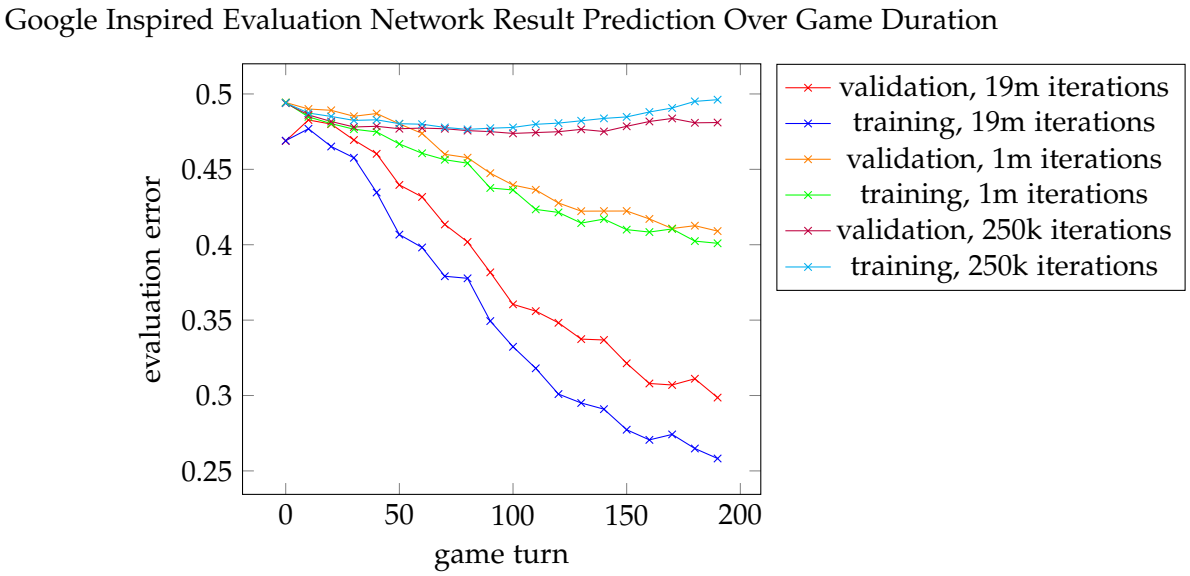

Figure 7.5: Shown above are the root mean squared errors of winning probabilities over the course of an average game estimated by the territory evaluation network. In the beginning of the game it cannot be predicted who will win, hence it makes sense that that the RMSE starts at 0.5 and drops from there. The results are based on 960 training and 960 validation games.

maximizes the expected score with perfect play also maximizes the winning probability with perfect play[5]

For a more direct comparison between the two networks consider Figure 7.8. As can be seen the territory evaluation network is much faster to train and does not keep overfitting further after having finished training. As mentioned above, despite being less accurate at this metric the territory and area evaluation networks were able to outperform the google inspired evaluation network in direct match comparisons.

[5]With perfect play the expected winning probability is always either 0% or 100%.

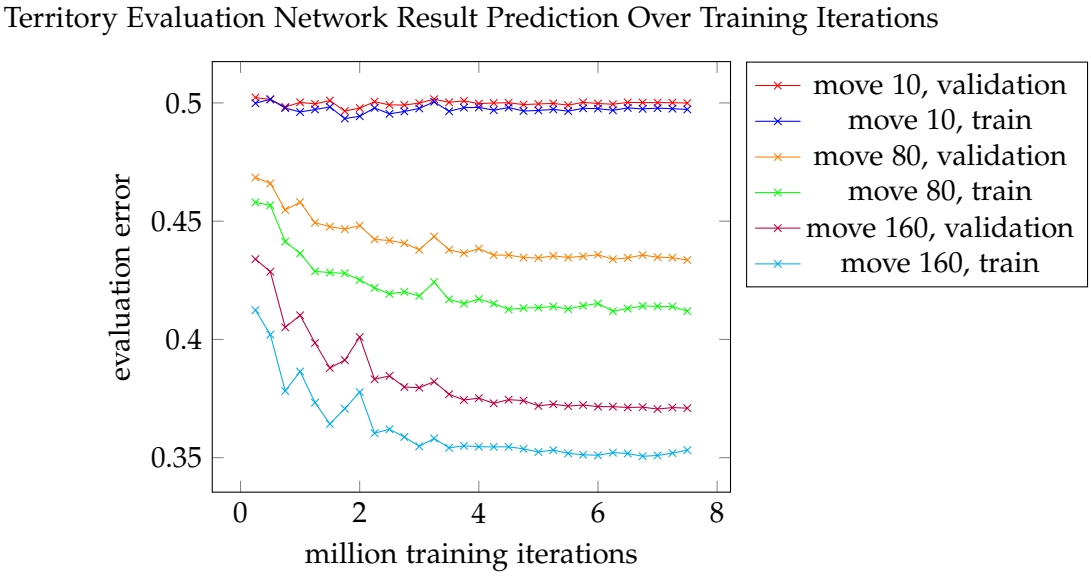

Figure 7.6: Shown above are the root mean squared errors of winning probabilities during training as estimated by the territory evaluation network. The results are based on 960 training and 960 validation games.

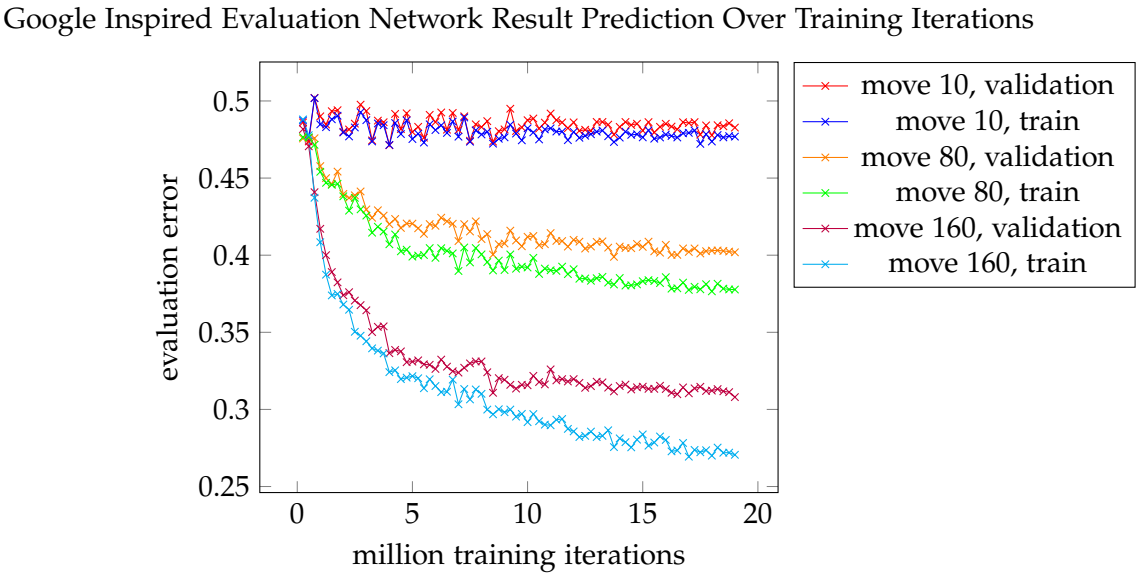

Figure 7.7: Shown above are the root mean squared errors of winning probabilities during training as estimated by the territory evaluation network. The results are based on 960 training and 960 validation games.

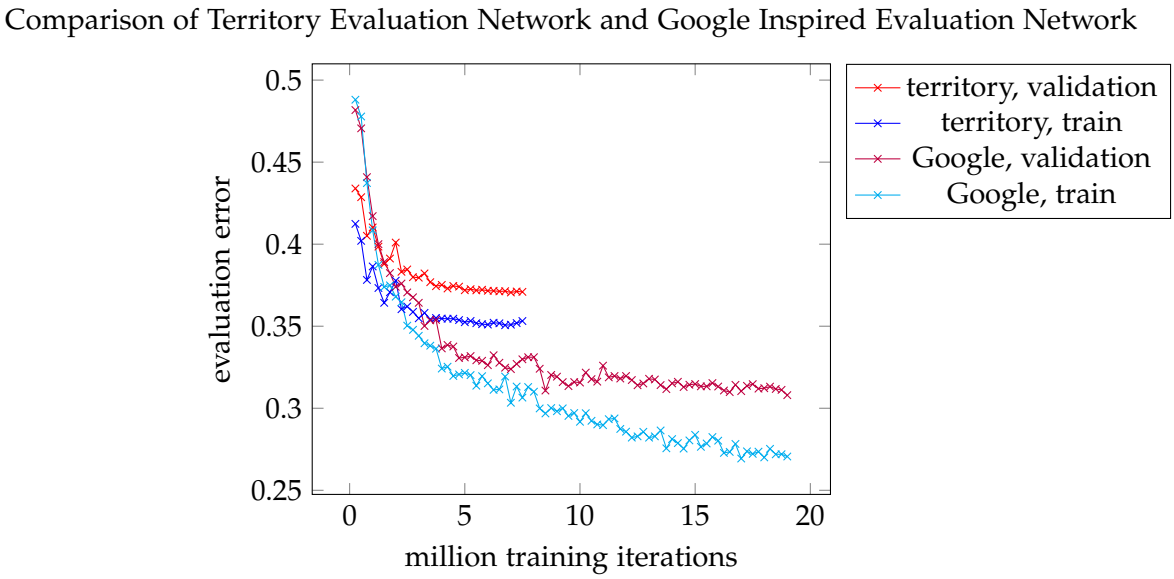

Figure 7.8: Shown above are the root mean squared errors of winning probabilities during training as estimated by the territory evaluation network. The results are based on 960 training and 960 validation games.

Appendix A

Derivation of Score Expectations

A critical result is the calculation to find the expected area and territory scores based off of the probability of an intersection belonging to a player's area or not by the end of the game. Given this probability $p(x)$ and the variable y which is defined as being 1 if the intersection contains a stone of the player to move, 0 if it is empty and -1 if the intersection contains an opponents stone we will calculate the respective expected territory and area scores for individual intersections, $T(x,y)$ and $A(x,y)$. These can be summed up over all vertices and added to the previous captures and komi to get the expected total territory and area score.

A.1 Territory Score

Under territory scoring we know the following. If an intersection contains one player's stone and ends up in that player's area at the end of the game, then it will not effect the score, however if it belongs to his opponent at the end of the game then his opponent will get one point for the stone and another point for the intersection belonging to him. If on the other hand an intersection is empty then whoever has the point within his area will get a point.

A more formal mathematical formulation of these rules follows.

$$T(x,y) := \begin{cases} p(x)\,(0) + (1-p(x))(-2), & \text{for } y = 1 \\ p(x)\,(1) + (1-p(x))(-1), & \text{for } y = 0 \\ p(x)\,(2) + (1-p(x))(0), & \text{for } y = -1 \end{cases} \tag{A.1}$$

Since y is defined to be either $1, 0$ or -1 we can simplify the equation as can easily be verified.

$$T(x,y) = 2p(x) - 1 - y \tag{A.2}$$

A.2 Area Score

Analog to territory scoring we can set up rules for area scoring. If an intersection contains one player's stone and ends up in that player's area at the end of the game, then he will get a point, however if it belongs to his opponent at the end of the game then his opponent will get one point for the stone and another point for the intersection belonging to him. If on the other hand an intersection is empty then whoever has the point within his area will get a point.

We can again formulate these rules as an equation.

$$A(x,y) := \begin{cases} p(x)(1) + (1-p(x))(-2), & \text{for } y = 1 \\ p(x)(1) + (1-p(x))(-1), & \text{for } y = 0 \\ p(x)(2) + (1-p(x))(-1), & \text{for } y = -1 \end{cases} \tag{A.3}$$

A simplification of this definition is again possible if we consider only the three possible values for y but the construction is a bit trickier. As can easily be verified for the three possible values of y we have

$$A(x,y) := 2p(x) - 1 + y^2 \cdot \left(p(x) - 0.5 \cdot (y^2 + y)\right) \tag{A.4}$$

This result looks somewhat artificial. It can be constructed as follows. As an intermediate step the three cases are expanded.

$$A(x,y) = \begin{cases} 3p(x) - 2, & \text{for } y = 1 \\ 2p(x) - 1, & \text{for } y = 0 \\ 3p(x) - 1, & \text{for } y = -1 \end{cases} \tag{A.5}$$

Now $A(x,y)$ is split into two parts with the first part being $2p(x) - 1$.

$$A(x,y) = \begin{cases} (2p(x) - 1) + (p(x) - 1), & \text{for } y = 1 \\ (2p(x) - 1) + (0), & \text{for } y = 0 \\ (2p(x) - 1) + (p(x)), & \text{for } y = -1 \end{cases} \tag{A.6}$$

What remains is to define the second part of the function in terms of the three possible values for y in order to construct equation A.4. This can be done with help of the last two equations.

$$y^2 = \begin{cases} 1, & \text{for } y = 1 \\ 0, & \text{for } y = 0 \\ 1, & \text{for } y = -1 \end{cases} \tag{A.7}$$

$$\forall z \in \mathbb{R} : \; yz + y^2 z = \begin{cases} 2z, & \text{for } y = 1 \\ 0, & \text{for } y = -1 \end{cases} \tag{A.8}$$

Bibliography

[1] Ccrl 40/4 - index. http://www.computerchess.org.uk/ccrl/404/. (Accessed on 06/05/2016).

[2] chessprogramming - fruit. https://chessprogramming.wikispaces.com/Fruit. (Accessed on 06/05/2016).

[3] chessprogramming - harm geert muller. https://chessprogramming.wikispaces.com/Harm+Geert+Muller. (Accessed on 06/16/2016).

[4] chessprogramming - home. https://chessprogramming.wikispaces.com/. (Accessed on 06/05/2016).

[5] chessprogramming - magic bitboards. https://chessprogramming.wikispaces.com/Magic+Bitboards. (Accessed on 06/14/2016).

[6] chessprogramming - perft. https://chessprogramming.wikispaces.com/Perft. (Accessed on 06/18/2016).

[7] chessprogramming - stockfish. https://chessprogramming.wikispaces.com/Stockfish. (Accessed on 06/06/2016).

[8] chessprogramming - transposition table. https://chessprogramming.wikispaces.com/Transposition+Table#cite_note-1. (Accessed on 06/14/2016).

[9] chessprogramming - wccc 2005. https://chessprogramming.wikispaces.com/WCCC+2005. (Accessed on 06/05/2016).

[10] Comparison of some go rules — british go association. http://www.britgo.org/rules/compare.html#comp. (Accessed on 06/05/2016).

[11] Elo rating system - wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Elo_rating_system. (Accessed on 06/16/2016).

[12] Etymology of go at sensei's library. http://senseis.xmp.net/?EtymologyOfGo. (Accessed on 06/04/2016).

[13] How much does time control influence chess engines strength? http://rybkaforum.net/cgi-bin/rybkaforum/topic_show.pl?tid=20910. (Accessed on 06/16/2016).

[14] Kgs go server. https://www.gokgs.com/. (Accessed on 06/15/2016).

[15] Quarry homepage. http://home.gna.org/quarry/. (Accessed on 06/18/2016).

[16] Hans J Berliner. Some necessary conditions for a master chess program. In *IJCAI*, pages 77–85. Citeseer, 1973.

[17] Bruno Bouzy. *MODÉLISATION COGNITIVE DU JOUEUR DE GO*. PhD thesis, UNIVERSITÉ PARIS 6, 1995.

[18] Bruno Bouzy. Mathematical morphology applied to computer go. *International Journal of Pattern Recognition and Artificial Intelligence*, 17(02):257–268, 2003.

[19] Bernd Brügmann. Monte carlo go. Technical report, Citeseer, 1993.

[20] Daniel Bump, G Farneback, A Bayer, et al. Gnu go, 1999.

[21] Guillaume M JB Chaslot, Mark HM Winands, H Jaap Van Den HERIK, Jos WHM Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

[22] Ken Chen and Zhixing Chen. Static analysis of life and death in the game of go. *Information Sciences*, 121(1):113–134, 1999.

[23] Marco Costalba, Joona Kiiski, Gary Linscott, and Tord Romstad. Stockfish: Uci chess engine. https://github.com/mcostalba/Stockfish.

[24] Marcel CrâŞsmaru. On the complexity of tsume-go. In *Computers and Games*, pages 222–231. Springer, 1998.

[25] Daniel James Edwards and TP Hart. The alpha-beta heuristic. 1961.

[26] Markus Enzenberger, Martin Müller, Broderick Arneson, and Richard Segal. Fuego—an open-source framework for board games and go engine based on monte carlo tree search. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4):259–270, 2010.

[27] Gunnar Farnebäck. Gtp - go text protocol. https://www.lysator.liu.se/~gunnar/gtp/. (Accessed on 06/18/2016).

[28] Gunnar Farnebäck. Gtp—go text protocol.

[29] Sylvain Gelly, Yizao Wang, Olivier Teytaud, Modification Uct Patterns, and Projet Tao. Modification of uct with patterns in monte-carlo go. 2006.

[30] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[31] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1976.

[32] Matthew Lai. Giraffe: Using deep reinforcement learning to play chess. *arXiv preprint arXiv:1509.01549*, 2015.

[33] Mace Li. Go4go. http://www.go4go.net/go/. (Accessed on 06/15/2016).

[34] Martin Mueller. Playing it safe: Recognizing secure territories in computer Go by using static rules and search. 1997.

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

[36] Jonathan Rosenthal. dalab/bsc_jonathan: deep learning for chess & go. https://github.com/dalab/bsc_jonathan. (Accessed on 06/18/2016).

[37] Jean Serra. *Image analysis and mathematical morphology, v. 1*. Academic press, 1982.

[38] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[39] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel,

and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[40] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[41] Martin Thorensen. Tcec - top chess engine championship. http://tcec.chessdom.com/. (Accessed on 06/06/2016).

[42] Yuandong Tian and Yan Zhu. Better computer go player with neural network and long-term prediction. *arXiv preprint arXiv:1511.06410*, 2015.

[43] John Tromp and Gunnar Farnebäck. Combinatorics of go. In *Computers and Games*, pages 84–99. Springer, 2006.

[44] Erik CD van der Werf, H Jaap Van Den Herik, and Jos WHM Uiterwijk. Learning to score final positions in the game of go. In *Advances in Computer Games*, pages 143–158. Springer, 2004.

[45] Burton Watson. *The Tso chuan: selections from China's oldest narrative history*. Columbia University Press, 1992.

[46] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[47] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[48] An Young-gil. Go commentary: Lee sedol vs alphago - game 1. https://gogameguru.com/go-commentary-lee-sedol-vs-alphago-game-1/. (Accessed on 06/06/2016).

[49] Albert L Zobrist. A model of visual organization for the game of go. In *Proceedings of the May 14-16, 1969, spring joint computer conference*, pages 103–112. ACM, 1969.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Deep Learning for Go

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Rosenthal	Jonathan

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich 2016-06-17	J. Rosenthal

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.