April 2016

Improving MCTS and Neural Network Communication in Computer Go

Joshua E. Keller
Worcester Polytechnic Institute

Oscar Perez
Worcester Polytechnic Institute

Improving MCTS and Neural Network Communication in Computer Go

Joshua Keller

Oscar Perez

Worcester Polytechnic Institute

a Major Qualifying Project Report
submitted to the faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements
for the Degree of Bachelor of Science

April 24, 2016

Abstract

In March 2016, AlphaGo, a computer Go program developed by Google DeepMind, won a 5-game match against Lee Sedol, one of the best Go players in the world. Its victory marks a major advance in the field of computer Go. However, much remains to be done. There is a gap between the computational power AlphaGo used in the match, and the computational power available to the majority of computer users today. Further, the communication between two of the techniques used by AlphaGo, neural networks and Monte Carlo Tree Search, can be improved. We investigate four different approaches towards accomplishing this end, with a focus on methods that require minimal computational power. Each method shows promise and can be developed further.

Acknowledgements

We would like to acknowledge:

- Levente Kocsis, for his advice and guidance throughout the project
- Sarun Paisarnsrisomsuk and Pitchaya Wiratchotisatian, for their implementation of the neural networks we used
- MTA-SZTAKI, for providing excellent facilities for the duration of our project
- Gabor N. Sarkozy, our WPI advisor
- Worcester Polytechnic Institute, for providing us with this opportunity

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

1.1 A New Era in Go Knowledge

The game of Go has existed for centuries. In fact, it is probably the oldest known strategy game in the world. As a result, Go theory has had an exceptionally long time to grow and develop. Over time, people have noticed patterns and techniques and given them colorful descriptions, for example: "two headed dragon", "tiger's mouth", "throwing star shape", etc.

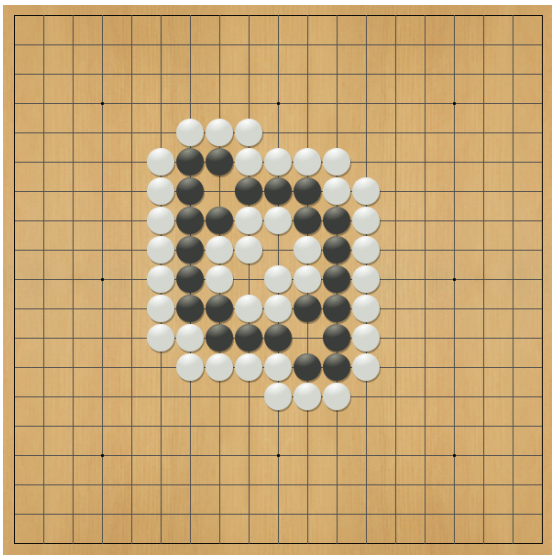

Figure 1.1: A Two-Headed Dragon - taken from [1]

Entire sequences of moves have become customary in certain situations as an agreed-upon "fair trade" (these are termed "joseki"). For instance, from a particular joseki, one player might gain a more secure territory in the corner, while the other obtains better central influence. The idea is that these advantages balance each other out. A new Go player can study these techniques, learn when to apply them in games through practice, and very quickly become a much better player.

Until recently, Go knowledge has always been added to by the top players and theoreticians. Computer Go programs did not have much to teach us, consistently playing at a level far below that of the best humans. All of this changed in March 2016. A program developed by Google DeepMind, called AlphaGo, challenged Lee Sedol to a 5-game match, one of the strongest Go players, if not the strongest, in the world. The outcome of this match marked the beginning of a new era for Go, one in which we can learn from computers as well as humans.

1.2 AlphaGo vs. Lee Sedol

The match itself was widely publicized. It was televised throughout South Korea. It had 60 million viewers in China. There was an international array of commentators analyzing each game live [2]. Most of the viewers were rooting for Lee to win. He himself was quite confident he would win, at first.

Lee apparently underestimated AlphaGo in the first game. In their paper [3], the AlphaGo team had provided the games of AlphaGo's recent 5-game match with European champion Fan Hui. AlphaGo had defeated Fan Hui in a landslide 5-0 victory, but Fan Hui was ranked much lower than Lee Sedol. Lee looked at the games and suspected that AlphaGo's playing style was too defensive, and he shouldn't have too much trouble winning. However, AlphaGo had been training itself in the 5 months

since that match. It exhibited a dramatic improvement in playing strength in their first game.

In the end, Lee Sedol lost the match 4 games to 1. This was an incredible victory for AlphaGo. It had conquered what is often termed the "holy grail of artificial intelligence," a feat that was thought to be more than a decade away.

However, AlphaGo did not come away unscathed. It did lose the fourth game of the match. Interestingly, it was playing as Black in that game. The only other game that Lee Sedol came close to winning was the second game, in which AlphaGo was also playing as Black. In Go, Black moves first, which gives that player an advantage. To compensate for this, White is given extra points at the start of the game, called komi. Some speculate that AlphaGo was more comfortable (whatever that can mean for a computer program) when playing White, because then equality on the board would be enough to secure a win [4]. As Black, AlphaGo would need an 8-point advantage or more on the board for a win (the komi was 7.5 points to avoid ties). Apparently it preferred the komi to the first-move advantage.

The game that Lee Sedol did win was an exciting one. He played a very tactical style that turned the game into an all-or-nothing fight, instead of a slow-moving incremental buildup of advantages for both sides that played into AlphaGo's superior calculation abilities [5]. On move 78, he played a brilliant move, a close-range tactical move that put him back in the game just as it seemed he might be losing. Gu Li, one of the commentators for game 4 (and a top professional player himself), referred to this move as the "hand of God." The "hand of God," or "divine move," is something many professional Go players aspire to achieve at least once in their lives. Essentially, it is a move so startlingly original and powerful that it is as if it were divinely inspired. Certainly Lee's move 78 was not foreseen by commentators, and apparently not even

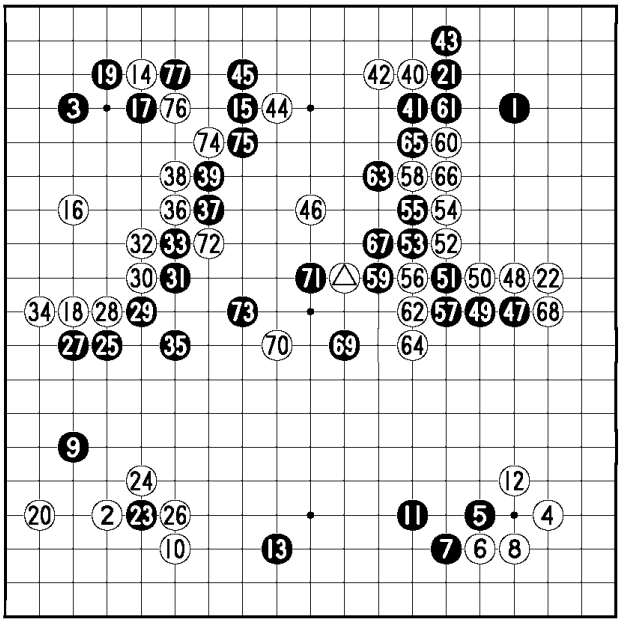

Figure 1.2: The Hand of God - Lee Sedol's "hand of God" move is marked with a triangle.

by AlphaGo. It is a move he can be proud of for years to come, and in a way, it makes up for the losses he had in the other games of the match.

The reader is strongly encouraged to watch the game at [8].

1.3 Two Powerful Techniques

Go is a very hard game for computers to play. The traditional approach in similar games, such as chess, is to construct a tree and look at all the possible move sequences of a certain length. Even in chess the full tree of all complete games is much too big, so the tree is cut off at a certain point, and the positions are evaluated using some evaluation function. In chess, the material count (i.e. 9 points for a Queen, 5 points for a Rook, etc.) serves as a useful and practical evaluation function. It can be made more subtle by introducing positional attributes, such as -0.2 for each pair of doubled

pawns.

One problem for Go is that the search tree has to be much bigger in both width and depth: Go games last about 5 times longer than chess games, and each turn, there are roughly 5 times as many possible moves in Go compared to chess. Another, perhaps more serious problem, is that there is no good simple evaluation function for Go positions (see Section 2.1.3 for a good example of why the "territory function" fails). All of this makes AlphaGo's recent victory all the more surprising. AlphaGo's use in particular of two groundbreaking techniques allowed it to face these difficulties and win.

The first is an ingenious trick to **replace the evaluation function by simulations**. In its simplest form, this is called **Monte Carlo Tree Search**. The essential idea is this: instead of evaluating positions by a function when the tree gets too deep, play an (intelligently) random game from that position, and record the result. Positions with more wins are considered better, and those parts of the tree can be explored further. This results in a somewhat unbalanced tree, but one that is hopefully unbalanced towards the good moves.

AlphaGo actually uses a variant of MCTS that includes an exploration bias. This is to encourage looking at moves that haven't been explored as much, to help balance the tree and make sure a good move is not overlooked. Many theorems have been proven about this technique, called **Upper Confidence Bounds on Trees**; we give some of them in Section 2.2.2.

The second is a radical departure from the idea of simple, hard-coded heuristic functions designed explicitly by programmers. The key is that a good evaluation function can be approximated by an automated procedure that learns over time how to recognize good moves. This approximation is stored in a structure of layers, weights

and connections, called a **neural network**, so named because it was originally inspired from the study of neuron structures in the human brain.

Neural networks are trained over time by sending positions to them, evaluating their output, and changing them slightly in different ways depending on whether the output was correct or not. At the end of training, the neural network often provides a good approximation for what it was designed to measure; however its developers do not have the same insight into it that they would have for a heuristic function they coded by hand.

The output of neural networks can be evaluated in several ways during training. One is by starting with an existing data set (for instance, the set of all Go games played on KGS Go Server [6]), and sending positions to the neural network. If it predicts the move that was actually played, it is correct. If not, it is wrong. This is called **supervised learning**. Another possibility is **reinforcement learning**. In this case, the neural network plays games against an opponent (possibly a previous iteration of the same network). If it loses, it is altered in one way. If it wins, it is altered in a different way. AlphaGo made use of both of these types of training.

AlphaGo also took advantage of a recent innovation in neural network structure (and also inspired by biology, this time by the study of the visual cortex). This innovation led to the development of **convolutional neural networks**. Convolutional neural networks take advantage of the near translation-invariance present in Go (that is, if all the stones in a position are shifted by one row, the best move will also shift by one row). These are discussed further in Section 2.2.3.

1.4 Next Steps for Computer Go

This is an exciting time for computer Go.

Let us return to the AlphaGo vs. Lee Sedol match for a moment. In game 2 of that match, AlphaGo played a surprising unconventional move 37.

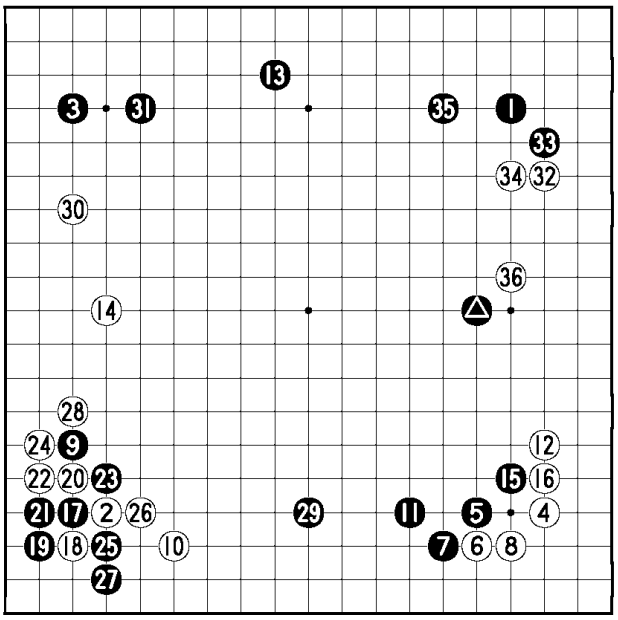

Figure 1.3: AlphaGo's "computer-style" move - AlphaGo's unconventional shoulder hit at move 37 of game 2, marked with a triangle

At first, the commentators thought it was a mistake in the move relay - perhaps someone's mouse had slipped while transferring the move. Lee Sedol himself left the room for a few minutes to regain focus. Fan Hui called it "a beautiful move that no human would play." [9].

It turned out that AlphaGo had deliberately gone against the traditional human styles of play it had originally learned from. According to David Silver, (at the start of [7]), AlphaGo believed that the probability of a human playing that move in that situation was 1 in 10,000. However, the prior probability of a human playing that move is only a heuristic, a guide - it biased the search tree against the move at first, but as AlphaGo analyzed further, it found this strange move 37 performing better than the

more human-style moves it considered first.

This means AlphaGo could have much to teach us in the Go world. It could be, as Silver remarks, that if they were to train neural networks without using human games as data at first (that is, only by reinforcement learning through self-play), the computers would play in a completely unrecognizable style, one uniquely their own. Yet somehow, this style would be more correct.

Thus, there is a lot of progress still to be made. Training neural networks by reinforcement learning alone could result in a new computer-style of play. Communication between the two techniques AlphaGo used can also be improved, allowing the neural network to better communicate with the Monte Carlo Tree Search, and vice versa. Finally, there is the issue of computing power.

In the analogous situation for chess, there was a gap between when Kasparov lost to Deep Blue, and when grandmaster-level chess engines started becoming widely available. The version of AlphaGo that played against Lee Sedol was a huge distributed system running on 1920 CPUs and 280 GPUs [10]. This kind of computational power is not available to the majority of computer users today.

Our project focuses on alternatives, using faster neural networks, with the ideal of running Go programs on a normal personal computer. We explore different ways of combining neural networks with Monte Carlo Tree Search.

The rest of this paper is structured as follows. First we give some important background information that goes into more detail than our overview here. Next we explain our methods in detail. Then we give the results of our testing, and we conclude with future work.

2

Background

2.1 The Game of Go

The game of Go is one of the oldest and most popular strategy board games in the world. The rules are simple; in fact, they can be described in just a few pages. But the strategies involved in expert play are subtle and complex, and the game takes years of study to master.

2.1.1 Rules

Go is normally played on a 19×19 board, though beginners often find it easier to play on the smaller 9×9 or 13×13 boards at first.

Two players, Black and White, take turns placing a stone of their own color on an empty intersection of the board. The goal of the game is to surround as much territory (empty intersections) with one's stones as possible, while keeping one's stones safe from capture.

Stones are captured when they run out of **liberties**. In the upper left corner of Figure 2.1, Black has a stone with 4 empty spots marked a. These are liberties, free spaces that keep the stone alive (spaces diagonally next to a stone do not count as

liberties). If all 4 spaces were to be taken up by White stones, the Black stone would be captured, at which point it would be removed from the board.

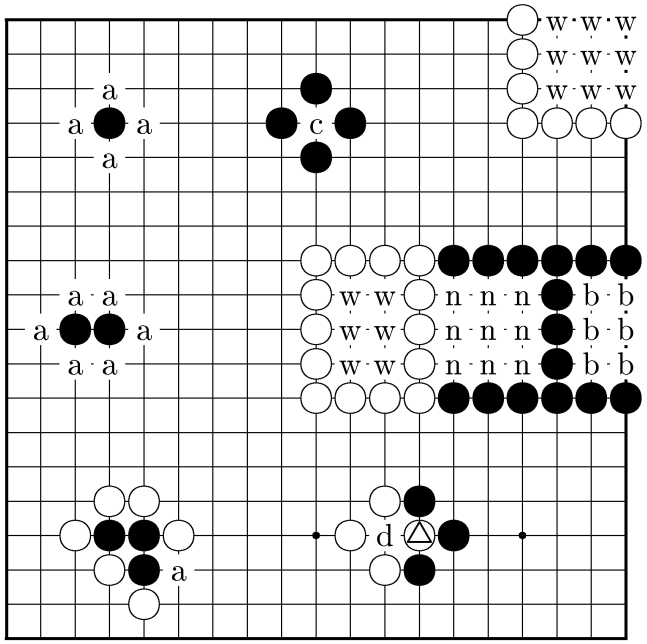

Figure 2.1: Rules of Go - Liberties at a, suicide move (illegal) for White at c, Ko at d, White territory at w, Black territory at b, neither side's territory at n

In the middle left of Figure 2.1, Black has two stones which are **connected**. Stones can only be connected orthogonally, not diagonally (as with liberties). We call connected stones a **group**. Liberties are shared among stones in a group, thus this group has 6 liberties at the points marked a.

In the lower left corner of Figure 2.1, though Black has a group of three stones, most of its liberties are already filled up by White stones. Black only has one liberty left, at a. This pattern is actually the start of a **ladder**, a common pattern in Go. It turns out, even if it is Black's move, he cannot avoid capture in the end.

In the upper middle of Figure 2.1, Black has completely surrounded the point c.

It is actually not permitted for White to play here, since White's stone would be immediately captured.

In the lower middle, we see a similar situation. White has surrounded the point d. But in this case Black is allowed to play at d, because this will capture the white stone marked with a triangle, freeing that space up for Black. Then Black's stone at d will have one liberty and survive.

It might seem that White can then immediately capture Black's stone in response, but this would lead to the same position repeated. The "Ko" rule in Go prevents this from happening. Players are not allowed to make a move that repeats a previous position. This forces them to play somewhere else first. Then on the move after that, they may recapture the stone, since the resulting position has then changed.

The game ends when both players pass their turn. The territory surrounded by each player is then counted. As mentioned in the Introduction, White receives an additional amount of points to compensate for the fact that Black moves first. These points are called komi and are normally something like 6.5 or 7.5 points, to avoid draws. For instance, if komi is 6.5, and Black has 100 points of territory to White's 95, White would still win because White's total score would be $95 + 6.5 = 101.5$. Depending on the specific ruleset used, captured stones may be added to one's score, or stones currently on the board. The player with the higher score is then declared the winner.

To be counted as one player's territory, the space must be completely surrounded by that player's stones. In Figure 2.1, spaces marked w are White's territory, spaces marked b are Black's territory, and spaces marked n belong to neither side.

2.1.2 Ranking System

Go players are traditionally ranked in the following way. A beginner starts out at 30 **kyu**, progressing through decreasing levels of kyu to eventually arrive at 1 kyu, roughly corresponding to intermediate strength. After 1 kyu, the next strongest rank is 1 **dan** amateur, continuing up to 7 dan amateur. Dan ranks can be thought of as expert ranks.

There is also a higher level of rankings beyond 7 dan amateur, the **dan professional** ranks. These range from 1 dan professional to 9 dan professional. To be eligible for these ranks one must have professional status, by fulfilling a set of strict requirements set by the professional Go association in one's country.

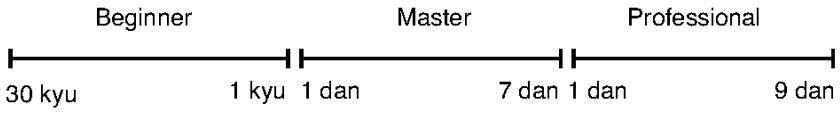

Figure 2.2: Go Ranks - Go ranks in increasing order of strength from left to right [11]

Among the amateur ranks, the difference in rank corresponds roughly to the number of handicap stones needed to give both players an equal chance of winning. For example, if one player is ranked 2 kyu and the other is ranked 5 kyu, the weaker player will start with 3 stones already on the board.

This does not apply to professional ranks, however. A 7 dan professional player and a 2 dan professional player are in general much closer in strength than a 7 dan amateur and a 2 dan amateur. In the latter case, 5 handicap stones are needed. In the former, most likely only about 2 handicap stones are needed.

2.1.3 The Role of Go in Artificial Intelligence

As discussed in [11], Go has long been thought of as a grand challenge for artificial intelligence. Recall from the Introduction that compared to chess, Go is much more difficult for a computer program to play well. In fact, there are on average 200 possible moves per turn in Go, compared to about 37 in chess. An average Go game takes 300 turns, compared to 57 turns in chess. Additionally, the combinatorial complexity of Go is not the only difficulty.

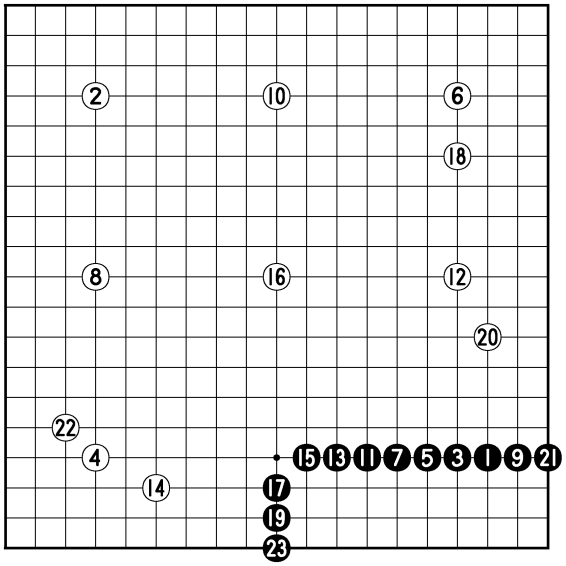

Figure 2.3: The Problem with a Territory Heuristic - White has a significant advantage in this position, but 0 confirmed points of territory. Black has 27 points of territory, but no influence in other areas of the board. (adapted from [12]

Because stones are not moved once they are placed on the board, Go moves often have very long-term effects. A stone placed on move 2 can have influence on the game during move 200, for instance. The only comparable long-term moves in chess are those which affect pawn structure, but in Go many more moves are likely to have long-term

influence. This makes it much more difficult to evaluate a move's effectiveness, if some of its effects can only be witnessed after looking more than 100 moves ahead.

Related to this issue, Go positions are much harder to evaluate without look-ahead, say, by a heuristic function of some kind. In chess, counting the material for both sides gives a reasonable rough estimate, but in Go one side can have a significant positional advantage but less territory or fewer stones captured.

For example, in Figure 2.3 above, a simple heuristic that counts territory is seen to be far less effective than the corresponding simple material-counting heuristic for chess. In this case, Black is at a signficant disadvantage, but in terms of confirmed territory Black is 27 points ahead at the moment.

These difficulties (long-term effects of decisions, combinatorial complexity, lack of good heuristic functions) are common to many real-world problems besides computer Go. For example, in healthcare, the amount of information doctors must take into account is rapidly increasing to the point where it is impossible to understand all of it thoroughly. However, intelligent decisions must be made quickly, and every case is different. These techniques can also be applied in online marketing. Making recommendations to users based on products they have expressed interest in in the past is a quite difficult problem well suited to deep learning. Progress in computer Go may be able to translate to tangible gains in these other areas as well.

In fact, as mentioned in the Introduction, a significant milestone has just been achieved in Go AI. Google DeepMind's program AlphaGo won a 5 game match against 9 dan professional Lee Sedol, one of the strongest Go players in the world. This came as a surprise to many experts, who thought that such a victory would only be possible in 10 years or more. The techniques successfully used by AlphaGo will be described in the next section.

2.2 Computer Go Techniques

We now explain the techniques AlphaGo uses in more detail. Recall from the Introduction that AlphaGo uses a combination of techniques to select its moves. The first is Monte Carlo Tree Search (MCTS). The second is convolutional neural networks. The way AlphaGo combines these techniques will be discussed in Section 2.2.4.

2.2.1 Monte Carlo Tree Search

The first computer Go technique, MCTS, combines two fundamental ideas in AI. The first is Minimax tree search and the second is Monte Carlo simulations. We first explain both of these topics in detail, then we discuss MCTS and explain the benefits of this method compared to others in computer Go.

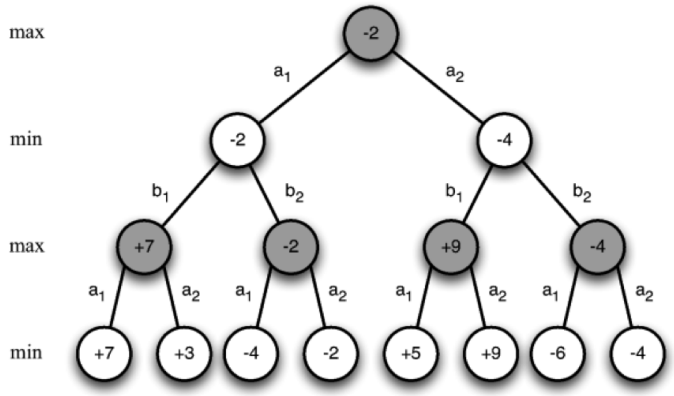

Figure 2.4: Minimax Search - from [11]

The Minimax game tree is a method used for deterministic, perfect information games. Figure 2.4 is an example of a Minimax search tree. Each node in the tree represents a game state, and the leaves of the tree are terminal states. Each node is connected by an action, and for each layer of the tree, each action alternates between the two different players. Each terminal state has a reward value associated with it,

and each node has an optimal value associated with it. The optimal values for the nodes are calculated by going down the tree where each player selects the move that will give them a maximum reward (or make their opponent receive the lowest possible reward). This method is impractical for most games. As the branching factor becomes larger, creating a tree that takes into account all possible actions and calculates all of the optimal values for each of the nodes becomes too computationally expensive. Because of this, a faster method is needed. In fact, in practice, Minimax search trees often do not go all the way to the terminal states, and instead a heuristic function is used to evaluate the leaves. However, creating a good heuristic function is a very difficult problem for the game of Go, because it is very difficult to determine who is winning based on deterministic things such as confirmed territory and stones captured. There are many other factors in play that are difficult to quantify.

A Monte Carlo simulation is a system where the probability of a certain event is calculated by running multiple trial runs. With this, it is possible to generate a "best move" policy instead of a heuristic function. A policy is a mapping from states to actions. This best move policy would find the move that has the highest probability of succeeding for each state. Using a Monte Carlo simulation could replace the need for a heuristic function in a Minimax tree and reduce the time necessary to arrive at a good evaluation of the best move, even for a game as combinatorially complex as Go. The random element of this policy would also be better than a fixed policy. This is because fixed policies introduce systematic errors, which can be exploited by opponents. However, with a randomized policy, these kinds of errors are prevented.

Monte Carlo tree search is the combination of Minimax game trees and Monte Carlo simulations. Monte Carlo Tree search starts with a root and expands the tree using a randomized policy. This process can be seen in Figure 2.5.

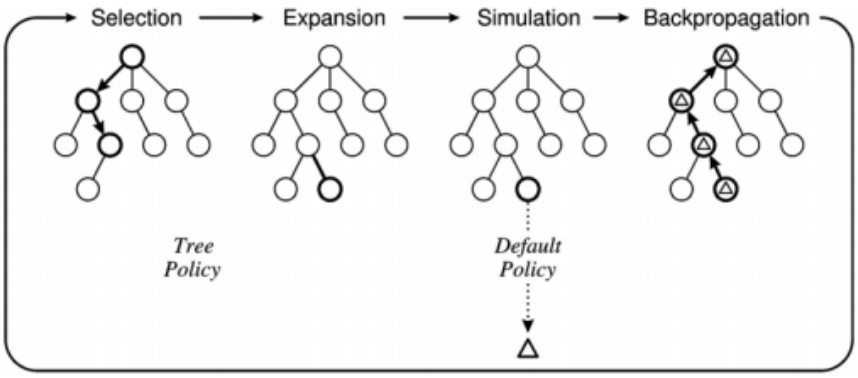

Figure 2.5: MCTS Phases - from [11]

The first phase is selection. It chooses a path on the Minimax tree, reaches a game state and decides to evaluate it. After evaluation of the state, it decides to expand the tree with another action. In the simulation phase, it decides on which action to use by finding the "best action" according to a default policy. Then, the tree back propagates to the root and repeats this process. After MCTS reaches a satisfactory number of states, the randomized policy from a Monte Carlo simulation is used to calculate the rewards of the terminal states. This Minimax tree is then used to determine the best action.

2.2.2 Upper Confidence Bounds on Trees

Before moving on to convolutional neural networks, it is beneficial to examine in more detail an important approach Monte Carlo Tree Search can use to select a path from the search tree during the selection phase.

Prior to the work done in [13], actions were sampled uniformly or using a heuristic bias on their probability of selection that had no theoretical guarantees. The problem with uniform sampling is that it is slow. The problem with heuristic biases is that the estimated values of leaves in the tree will not necessarily converge to the true

optimal values (that is, the values that would be obtained from a full minimax search), even after many many iterations. However, using the **Upper Confidence Bounds applied to Trees** (UCT) method in the selection phase, this convergence can be achieved under certain conditions. It also converges significantly faster than uniform sampling, and even if the method is stopped beforehand, the probability that it biases towards suboptimal actions is low.

Intuitively, UCT achieves these things by addressing the **exploration-exploitation dilemma**. On one hand, actions that appear optimal already should be explored more, to find the best action more quickly. This is the exploitation side of the dilemma. On the other hand, if an optimal action is mistakenly estimated as suboptimal at first, there should always be some incentive to explore it again, or it will be overlooked. This is the exploration side. To balance these competing goals, UCT uses an algorithm originally developed for **bandit problems with K arms**.

A bandit with K arms is analogous to a casino with K slot machines. Each arm (slot machine) has its own probability distribution of rewards, and at each time t exactly one machine can be selected to play. The problem is to determine an allocation policy that maximizes one's total reward.

The allocation policy that UCT adapts to MCTS is called UCB1, and it works as follows. Let $\bar{X}_i$ be the average reward obtained so far from machine i. Let s_i be the number of times machine i has been played so far. Let t be the current time. Then to select the machine to play at time $t+1$, UCB1 picks the machine j that maximizes:

$$\bar{X}_j + \sqrt{\frac{2 \ln t}{s_j}}$$

Note the second term in this expression. It is an exploration bias term. If machine i is visited more often relative to the other machines, it will be explored less. UCT

actually uses a constant multiple of this bias term instead, to account for drift in the rewards over time.

The rewards can drift in time in UCT because of the way UCT differs from UCB1. In UCT, the actions available at a given node of the tree are the "arms" of the bandit, but the key difference is that below any given node, UCT is again being used to select the actions to try. Thus the average reward of the node above could gradually increase, for instance, if the nodes below it took some time to converge to their own optimal values (if, say, they were initially underestimated).

The main theorem in [13] establishes that UCT converges to the optimal values, given enough time (here MDP refers to a "Markovian Decision Problem"):

Theorem 1 *Consider a finite-horizon MDP with rewards scaled to lie in the* $[0, 1]$ *interval. Let the horizon of the MDP be* D*, and the number of actions per state be* K*. Consider algorithm UCT such that the bias terms of UCB1 are multiplied by* D*. Then the bias of the estimated expected payoff,* $\bar{X}_n$*, is* $O\left(log(n)/n\right)$*. Further, the failure probability at the root converges to zero at a polynomial rate as the number of episodes grows to infinity.*

UCT has also performed considerably better than alternatives in practice. See [14] for some examples. There is also some theoretical analysis that is worth mentioning. This analysis shows the consistency for the whole procedure. The first result provides an upper bound for the number of plays of a suboptimal arm. The theorem goes as follows.

Theorem 2 *Consider UCB1 applied to a non-stationary problem. Let* $T_i(n)$ *denote the number of plays of arm* i*. Then if* i *is the index of a suboptimal arm,* $n > K$*, then* $\mathbb{E}[T_i(n)] \leq \frac{16C_p^2 ln(n)}{(\Delta_i/2)^2} + 2N_0 + \frac{\pi^2}{3}$

Here, Δ_i is a measure of the suboptimality of action i, C_p is equal to the constant by which the expression $\sqrt{\frac{2 \ln t}{s_j}}$ is multiplied, mentioned above, N_0 is a term that

measures how close the estimate is to the true value, n is equal to the number of plays, and K is the number of possible actions. The next result provides a bound on the bias. The theorem goes as follows.

Theorem 3 *Let* $\bar{X}_n = \sum_{i=1}^{K} \frac{T_i(n)}{n} \bar{X}_{i,T_i(n)}$. *Then* $|\mathbb{E}[\bar{X}_n] - u^*| \leq |\delta_n^*| + O(\frac{K(C_p^2 ln(n) + N_0)}{n})$

Here, $\bar{X}_i$ is equal to the average number of rewards, δ_n^* is a measure of how suboptimal the rewards are, and μ^* is the reward for the most optimal action. Also, K, C_p, n, N_0, and $T_i(n)$ are all defined as before.

2.2.3 Deep Convolutional Neural Networks

In order to better make decisions in the game of Go, professional players need to look for patterns. This helps a player learn crucial information during a game such as who owns which territories. Neither MCTS nor UCT are capable of finding patterns in Go. If a Go program were capable of recognizing patterns and reporting useful information about them, then this would allow MCTS to cut down on the moves it considers when expanding. This would save MCTS time and allow it to explore better moves more frequently. Fortunately, Deep Convolutional Neural Networks are capable of analyzing such patterns.

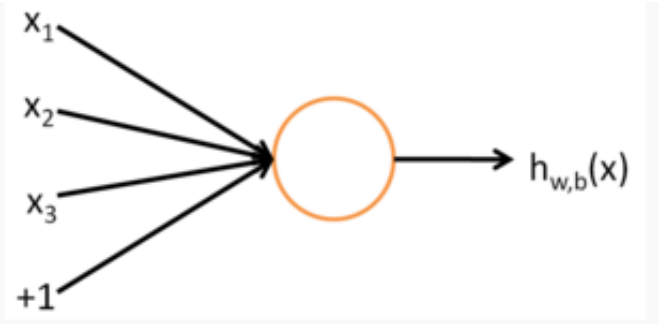

Figure 2.6: Simple Neural Network - This network consists of just one neuron. (from [15])

A Neural Network is a tool that is used to classify objects based on its features. It does this by analyzing known data and forming an activation function based on it. This function is then used to classify unknown data based on its features. An example of a simple neural network can be seen in Figure 2.6. This neural network is composed of a single neuron, which contains a single instance of an activation function. It accepts inputs x_i and assigns to each x_i a weight w_i. It then computes $\sum_{i=1}^{n} x_i w_i + b$, where n is the total number of features the object has, and b is a bias used to help determine how to classify the object. In order to determine if an object belongs to a certain group or not, we simply check whether the activation function exceeds a certain threshold. A fully connected Neural Network is many neurons stringed together, where the output of one neuron can serve as the input for another. This is demonstrated in Figure 2.7.

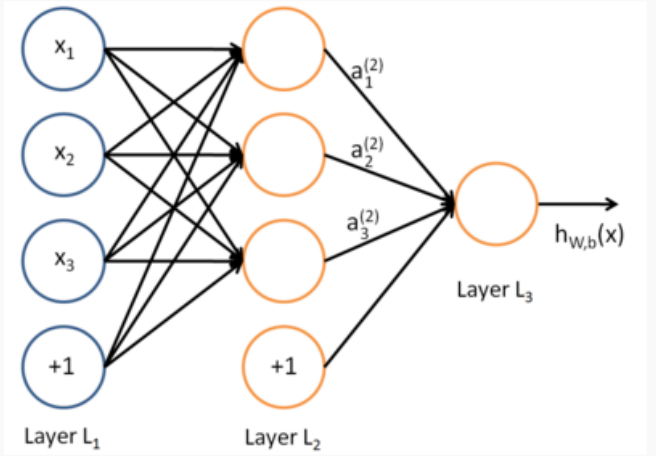

Figure 2.7: Fully Connected Neural Network - This network consists of layers of neurons such that all neurons in one layer are all connected to all neurons in the next layer. (from [15])

There are different activation functions which could be used inside of a neural network. Another type of activation function is a logistic, or sigmoid, function. A sigmoid function is a bounded differentiable real function that is defined for all real

input values and has a positive derivative at each point. An example of such a function is

$$\sigma(x) = \frac{1}{1 + e^{-\beta X}}$$

Here, β is the vector of weights used to weigh the value of each input x_i. The threshold for this activation function is 0.5. This sigmoid function is particularly nice, because it gives a value between 0 and 1 and it is odd about the point (0, 0.5). Also, it is an easy function to differentiate which is necessary when training the Neural Network.

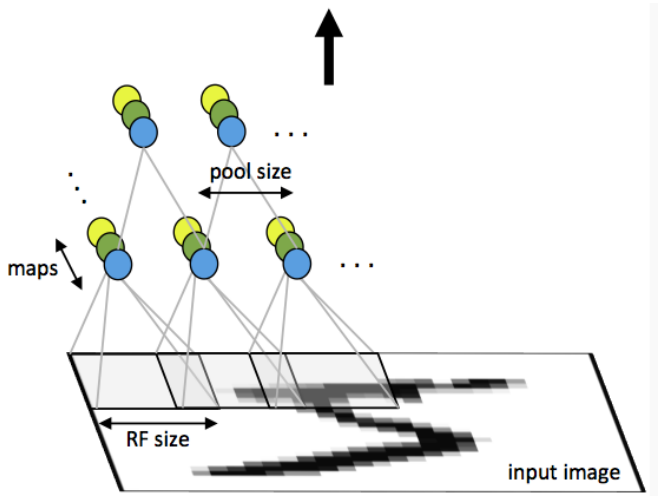

Figure 2.8: Convolutional Neural Network (CNN) - the extra steps involved in a Deep CNN (from [15])

A Convolutional Neural Network is a specific type of Neural Network. When the number of features becomes too large, the neural network begins to become slow. Also, a more important issue is that it becomes difficult, and in some cases even impossible, to even train the neural network in the first place. A Convolutional Neural Network attempts to solve this problem. It does this by creating a few more steps. These steps can be seen in Figure 2.8. The first step is to take the input and to divide it into distinct overlapping sections. Then, these are taken and put through filters to obtain

convolved maps. These maps are then split up into disjoint sections and these sections are pooled together to obtain a statistic (usually mean or max) of the group of maps. These statistics are then put into a traditional fully connected neural network.

2.2.4 How AlphaGo Combines MCTS with Neural Networks

Here we briefly summarize the way AlphaGo uses neural networks to inform MCTS that is relevant to our project. For a full description of the techniques behind AlphaGo, see [3].

Note that AlphaGo used more than simply a convolutional neural network trained by supervised learning of expert games as described above. In fact, AlphaGo also trained a reinforcement learning neural network through self-play, and then used this network to train a value network to be used as a kind of heuristic function to aid in position evaluation. We ignore these details in the following.

The two differences AlphaGo introduces to standard MCTS are in the selection phase and the expansion phase. Briefly, the neural network is queried and its output is stored in the expansion phase, and the output is used in subsequent selection phases.

More precisely, when a leaf node is expanded, its position is sent to the neural network trained by supervised learning. The output is a probability distribution over the legal moves from that position for the current color. This is associated with that leaf node as prior probabilities for those actions.

In the next selection phase, suppose this (former) leaf node has been selected. To choose an action from the leaf node, the prior probabilities are taken into account. If s is the state of this node, a is the action being examined, Q is the value estimate function from MCTS, $N(s, a)$ is the number of times this action has been taken before from this state (in this case, 0), and $P(s, a)$ is the prior probability for action a, then

action a's bias, $u(s, a)$ is a constant multiple of:

$$\frac{P(s, a)}{1 + N(s, a)}$$

The action that will ultimately be selected is the one that maximizes:

$$Q(s, a) + u(s, a)$$

AlphaGo introduces some other variations in the value function that are not discussed here. In particular, it uses a weighted average of the standard MCTS value function combined with the output of its own value network. For details, see [3].

Our project considers alternative ways of combining convolutional neural networks with Monte Carlo tree search, focusing on methods that do not require a lot of computational power. Our methods for achieving this goal are given in the next chapter.

3

Methods

Our work focused on modifying Pachi, which is one of the strongest open-source Go programs [16]. Pachi's default move selection algorithm is actually a variant of MCTS called RAVE, though Pachi can also be set to use vanilla MCTS. Pachi's move selection is discussed in more detail in the following section.

We also made use of a neural network implementation taken from last year's MQP project [17]. Their neural network implementation had the following specifications [17]:

- 1 hidden layer
- 10 kernels
- 5×5 hidden layer filter size
- no pooling layer
- rectified linear function as the activation function for the hidden layer
- softmax function as the activation function for the output layer

3.1 Move Selection in Pachi

In order to determine which move it will play, Pachi uses MCTS with a specific set of heuristics and policies [18]. In our project, we made use of Pachi's RAVE engine in particular. Pachi's RAVE engine has a way of carrying out the four phase process of MCTS that makes it unique.

The first phase in MCTS is selection of the node it wishes to expand. The way this is done is by considering all of the child nodes, and descending to the node which is found to be the most urgent.

Once it finds a suitable node to expand, it first creates child nodes for all of the possible follow-up moves. Each node is then assigned a value based on several virtual simulations and heuristics. These heuristics contribute ϵ fixed-result virtual simulations, (where $\epsilon = 20$ for a 19×19 board). There are six different kinds of heuristics which prevent the program from making poor move choices during the expansion phase.

The first heuristic is the "eye" heuristic. This heuristic makes sure that a move does not play into one's own eyes. Generally, such a move is poor, and should not be considered by the program. However, there are rare circumstances where the move is actually important. For this reason, the program cannot simply disregard the possibility; it can only strongly discourage it. The next heuristic encourages ko fights. It does this by adding virtual wins to moves that retake a ko that is no more than 10 moves old. The third heuristic is a simple one which takes effect in the very early game. It awards wins if the move in consideration is not on the edge of the board in addition to being far enough away from other stones. It also gives losses if the move is on the edge of the board. The fourth heuristic is the Common Fate Graph or CFG heuristic. This heuristic has two purposes. The first is to motivate it to focus on each individ-

ual sequence properly. This is important, because the tree should not be randomly jumping back and forth between interesting sequences. The second is to be consistent with the Go concept of "sente". The idea of "sente" is that local play is required in certain situations, so moves outside of a certain area should not be considered. The fifth heuristic focuses on playing joseki dictionary moves. These are move sequences that are guaranteed to give each player a fair outcome. These moves are given twice the default ϵ virtual wins in order to encourage joseki moves. The final heuristic comes from suggestions from the playout policy. If the program saw a particular move as good in the playouts, it would encourage exploration of that move with this heuristic.

In the playout phase, the moves made in the simulations are selected semi-randomly. The moves should be selected randomly to maintain the spirit of MCTS; however choosing moves based on realistic play proves to be highly beneficial for program performance. The way that it does this is by using a set of heuristics, and each heuristic has an opportunity to be used with a certain probability p. For a 19×19 board, which is the board size we used for our project, the default probability is $p = 0.8$. If a heuristic is chosen, it returns a set of moves. If the set is non-empty, then a move from the set is randomly selected and played. However, if the set is empty then the next heuristic is tried with probability p. In the event that none of the used heuristics matches, then a move is randomly chosen (excluding moves which fill an eye or moves that put oneself in atari). The first heuristic is one that checks if it can recapture ko. If the opposite side played a ko in the last 4 turns, then the program recaptures with probability $p = 0.2$. The next heuristic checks, with $p = 0.2$, if the liberties of the last move group form a nakade shape. If they do, then the program kills the group by playing in the middle of the eyespace. If the opposite side's last move put one of its own groups in atari, then the program captures the group with $p = 0.9$. Also, if the opposite side's

last move put us in atari, then the program tries to escape or counter-capture other neighboring groups with $p = 0.9$. The fourth heuristic puts an opponent's group into atari if their group has only two liberties. It does this aiming to give greater probability to the situations where the opposite side has low chances of escaping. Also, the heuristic notices if the current player has a group with only two liberties. If this is the case, it tries to gain more in order to avoid being put into atari. The next heuristic tries to do the same as the previous one, but with more groups of 3 or 4 liberties. It does this with $p = 0.2$ probability. For the final heuristic, any options that neighbor the last two moves and also match with 3x3 board patterns that are stored in their pattern dictionary, are played with $p = 1$. As mentioned before, some of the heuristics used here are used to influence one of the heuristics used in expansion. However, bad self-atari moves are pruned and not taken into consideration.

3.2 Our Approaches

We modified Pachi's move selection algorithm in four main ways. First, we added output from the neural network to Pachi's prior heuristics-based knowledge. Next, we optimized the algorithm by taking into account the depth of the current node in the search tree. If the depth was large, we used a faster, less accurate neural network.

The last two approaches we used involved improving communication between the neural network and Pachi's MCTS. This communication can go both ways, and we worked on improving both directions. To obtain better communication from the neural network to MCTS, we trained a neural network (based on the original neural networks from [17]) with the explicit goal of informing the search, rather than simply predicting expert moves on its own. To obtain better communication from MCTS to the neural network, we added a search-based feature to one of the neural networks, specifically:

the fraction of the playouts in which the color to move owned the given point at the end of the game.

Details of each of these approaches follow.

3.2.1 Adding the Neural Network to Pachi's Prior Knowledge

The first approach formed the basis for our other approaches. As mentioned above, Pachi's move selection incorporates prior heuristic knowledge, which is calculated for all possible moves from a node whenever that node is expanded. This heuristic knowledge includes encouragement to explore local sequences of moves, encouragement to evaluate ko fights, and discouragement from playing in one's own eyes (which, though almost always a bad idea, can only be strongly discouraged, not prohibited, because of exceptions).

All of this prior knowledge is stored as a set of virtual playouts, using the notion of **equivalent experience** from [19]. This is similar to the notion of virtual experience mentioned in [11], with the experience weighted differently depending on the size of the board. In our case, we are only interested in a 19×19 board; thus we used weights based on the weights for a board of that size.

Our implementation added the neural network's output to this prior knowledge. We attempted to do this in as unobtrusive a way as possible. First, we determined that the variation of weights was low, taken over the set of all weights used for various nodes that were about to be expanded. In other words, the most weight given to prior experience for a particular move was very similar to the least weight given to prior experience for a particular move. This allowed us to simply add the neural network's own evaluation of the position to this prior knowledge, giving it equal weight to the current weight of the prior knowledge. Thus, the neural network's output was given

the same weight as the entire heuristics-based knowledge already present in Pachi.

Another way we tried to reduce any unwanted effects of this modification was by maintaining the same total weight at the end. In this case, that meant dividing the total weight by 2 after incorporating the neural network information. This prevented the weight of the prior experience from being too high, reducing the impact that MCTS playouts would have on its value.

3.2.2 Optimizing for Current Depth

This implementation was based on the first. Like the first, it uses neural networks to help MCTS determine which moves it should explore during its exploration phase. However, this implementation used more than one neural network. During the exploration phase, it decides which neural network to use based on its depth in the tree. If the current node is relatively close to the root of the tree, then a slower but more accurate neural network is used. However, when the current node is deeper in the tree, then a faster, but less accurate neural network is used.

This arrangement was chosen since there are fewer nodes closer to the root. Additionally, the way in which nodes close to the root of the tree are expanded is more important, because they have an influence on all of the subsequent nodes. For these reasons, it is appropriate to use a slower, but more accurate neural network for these nodes. Conversely, the nodes that are deeper in the tree are in overwhelmingly greater numbers, and they are also slightly less important than the nodes closer to the root. For these reasons it is appropriate to use a faster, but less accurate neural network for these nodes.

In order to determine the best transition point (that is, the minimum depth at which the faster neural network would be used), we collected information on the amount

of times MCTS expanded nodes of each depth. This resulted in a distribution that contained the number of expansions that MCTS performed on nodes at each depth. We used this distribution to help us determine where to use the expensive neural network without using it an unreasonable amount of times.

3.2.3 Training the Neural Network to Inform the Search

The next approach addressed communication potential between the neural network and MCTS that we believe has not been investigated before. In particular, neural networks used in Go in the past were generally trained to predict moves played by experts. One exceptional case was in AlphaGo, in which a reinforcement learning neural network was trained to optimize its winrate rather than its predictive power. But even in this latter case, the neural network was trained to optimize its winrate when the neural network was used alone. In our approach (and the approach taken by AlphaGo), the neural network is ultimately used in conjunction with MCTS. Therefore, it is natural to consider training the neural network in a way that is consistent with its role as part of a bigger algorithm involving MCTS. This is the key idea of our third approach.

Specifically, we trained a neural network based on the output of Pachi with the neural network (call this Pachi$_{nn}$) rather than the output of the neural network alone. Ideally, we would train it based on the winrate of Pachi$_{nn}$ against some reference opponent. However, due to time constraints, we decided to train it based on the predictive power of Pachi$_{nn}$ instead. This is still preferable to the original method of training, in which the neural network was used alone, since the prediction rate of Pachi$_{nn}$ is more relevant to the strength of Pachi$_{nn}$ than the prediction rate of the neural network by itself.

We used simultaneous perturbation stochastic approximation (SPSA) to train the

neural network in this way. The method of training used in [17] (and in our other approaches) does not suffice here, because the Pachi$_{nn}$ system is noisy. This requires some explanation.

3.2.3.1 Why SPSA is Necessary

A neural network can be thought of as a function that maps, in our case, a set of features of a Go board to an output probability distribution. Adopting notation from [15], the function itself can be represented as follows:

$$a_i^{(l)} = f_l\left(\left(\sum_{j \in A_i^{(l)}} W_{ji}^{(l)} a_j^{(l-1)}\right) + b_i^{(l-1)}\right) \tag{3.1}$$

Here,

$$a_i^{(l)} = \text{the value of the neural network at unit } i \text{ in layer } l \tag{3.2}$$

$$A_i^{(l)} = \{j \text{ s.t. there is a connection from unit } j, \text{ layer } l-1 \text{ to unit } i, \text{ layer } l\} \tag{3.3}$$

$$W_{ji}^{(l)} = \text{ the weight of the connection from unit } j, \text{ layer } l-1 \text{ to unit } i, \text{ layer } l \tag{3.4}$$

Also, $b_i^{(l-1)}$ is a bias term (that can be equal to zero), and f_l is the **activation function** for layer l.

Since we used convolutional neural networks rather than fully connected neural networks, not all connections are present, hence our use of $A_i^{(l)}$ in the above. For the first layer, $a_i^{(1)}$ is simply the input value to that unit of the neural network. In our case, there were 361 inputs to the neural network, each corresponding to a point on the Go board. Finally, we note that in the neural networks we used, the activation function for layers 1 and 2 is the **rectified linear** function,

$$f_l(x) = \max(0, x) \tag{3.5}$$

and the activation function for the last layer is the **softmax** function:

$$f_l(x)_i = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}} \text{ for } i = 1, ..., k \tag{3.6}$$

Note the softmax function has the effect of normalizing the output so that all output is in the range $(0, 1)$, as we should expect from a probability distribution.

All of this is just as in last year's project [17]. As described there, training a neural network is just modifying the weights W_i in each layer, so that the overall function better approximates the desired output for each input in the training data. How well it currently approximates the desired function can be measured with a **cost function**:

$$J(W, b; x, y) = \frac{1}{2}\|h_{W,b}(x) - y\|^2 \tag{3.7}$$

Here, $h_{W,b}(x)$ is the output of the neural network for vector x (which in our case is itself a vector), and the pair (x, y) is one example from the training data set, where x is the input and y is its desired output. In general for training data of size m, we have the following cost function (from [15]):

$$J(W, b) = \left[\frac{1}{m}\sum_{i=1}^{m} J\left(W, b; x^{(i)}, y^{(i)}\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(W_{ji}^{(l)}\right)^2 \tag{3.8}$$

This cost function is a function of the set of weights W and biases b for the neural network, given a fixed training set. To minimize it, and thus approximate the desired behavior for the training set, **gradient descent** is a quite useful approach. However, this requires calculating the gradient of the cost function.

This function is complicated (and the second term is a weight decay term that has no bearing on our discussion here), but it nevertheless has a certain structure that makes its gradient possible to calculate efficiently. This is accomplished through the **backpropagation algorithm**, which is possible to apply due to the way in which the

function J depends upon the output of the neural network internally and the way in which the neural network itself has a certain structure.

Once the gradient has been calculated, the weights and biases can be updated in the following way:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\delta}{\delta W_{ij}^{(l)}} J(W, b) \tag{3.9}$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\delta}{\delta b_i^{(l)}} J(W, b) \tag{3.10}$$

Here, α is the learning rate.

Now we come to the difference between this approach and the other approaches. Because the output depends on the Pachi$_{nn}$ system as a whole, rather than just the neural network itself, the cost function $J(W, b)$ loses the simple structure it had before. Now instead of depending only upon the weights and biases of the neural network in a simple way, $J(W, b)$ also depends on Pachi's playout policy, for one, and several other factors. In fact it is even misleading to write $J(W, b)$ in this case, as there are other parameters involved. As a result, the backpropagation method does not apply. Instead, we turn to SPSA.

3.2.3.2 How SPSA Works

SPSA was introduced in a paper in 1992 by Spall [20] as an alternative to finite-difference methods of stochastic approximation. Both of these rely on approximating the gradient of the cost function in situations where it is too complicated or impossible to determine precisely.

The general situation is as follows. Suppose we have a cost function f and a vector θ of weights. We wish to minimize f, but we do not have an explicit formula for f. At each iteration, we perturb our current θ by a random vector Δ of the same dimension,

where each element of Δ is $\pm c$ (c is some perturbation constant). In finite-difference methods, each iteration evaluates, for each i:

$$\Delta\theta_i(t) = \frac{f(\theta(t-1) + ce_i) - f(\theta(t-1) - ce_i)}{2c} \tag{3.11}$$

Here, e_i is the vector with 1 in position i and 0 elsewhere. In SPSA, each iteration evaluates:

$$\Delta\theta_i(t) = \frac{f(\theta(t-1) + \Delta(t-1)) - f(\theta(t-1) - \Delta(t-1))}{\Delta_i(t-1)} \tag{3.12}$$

The difference is that in SPSA, only 2 evaluations of f are required regardless of the dimension of θ. It may seem surprising that this process converges, but [20] provides conditions under which $\theta(t)$ converges almost surely to θ^*, the true optimal θ. The conditions are fairly technical, however taken together they are not very restrictive and are often satisfied in practice [20].

We apply SPSA to our training in the following way. For simplicity, we take θ to be only the weights in the last layer of the neural network. We define our function f to be 1 if the move predicted by Pachi$_{nn}$ under θ is correct, 0 otherwise. In addition to adding $\alpha\Delta\theta(t)$ to $\theta(t)$ at each iteration, (where again α is the learning rate), we also keep track of the most recent nonzero $\Delta\theta(k)$, and we add $\mu\Delta\theta(k)$ as well, where μ is a momentum constant. This is especially useful for our chosen function, which is prone to have many iterations occur with a zero change in θ.

3.2.4 Search-Based Features

The other side of the communication was addressed by our fourth approach. We developed a search-based feature that gave the neural network information about the search, rather than just the position as in the case of the neural networks in the previous approaches. This information was in the form of point ownership. At the end

of a game of Go, both sides have certain points on the board considered part of their territory. MCTS playouts are complete games; thus, at the end of an MCTS playout, certain points will be owned by Black, certain points by White, and certain points will be owned by neither side. This information is not possible to obtain from the position alone - one must have a search algorithm with playouts of some kind to arrive at final board positions from an ongoing game.

We first trained the neural network to respond to this feature. To do this we generated training data as follows. We sent move data to Pachi from over 100,000 games played on the KGS Go Server (KGS) [6]. KGS is one of the largest online Go servers, and games between strong players are a common occurrence. This makes KGS a good choice for move data with which to train a neural network, and in fact this data was originally harvested in [17]. Upon receiving each move, Pachi generated a random number of playouts between 1 and 10. It then played that number of playouts, recording in each case the owner of each point of the board. This data was then written to pattern files, and these files were used for the training.

We then sent this data to the neural network in a slightly different way. In an actual game, MCTS will already have playout data for some moves, so there is no need to explicitly call the playout function as was necessary for training data generation. Instead, we sent the actual playout data as input to the neural network. Though in some cases this could be the result of a much greater amount of playouts per move than what the neural network was trained on, we focused on using the neural network's output early on after a node expansion, when the number of playouts was likely to be less. Even in the case where the number of actual playouts is greater, we suspect this can only improve the accuracy of the neural network's output.

3.3 Testing

In order to evaluate each of these implementations, we tested each of them against Fuego. Fuego is a fairly strong, open source Go program [21], and we found it to be a good match for Pachi.

Fuego was running using 180,000 iterations (playouts in MCTS) per move, while each implementation of Pachi was using 27,000 iterations per move. When vanilla Pachi was run against Fuego using these settings, they were about even. They played 100 games against each other. Fuego won 38 games as black and 19 games as white, where Pachi won 31 game as black and 12 games as white. This shows that Fuego running at 180,000 iterations against Pachi is a fair match up, and it could be used to determine how much each of our implementations improved or diminished the capability of the Pachi program. Each test was run using 100 games where Fuego and Pachi alternated colors. Both sides were given 60 min of play time. However, since Fuego and Pachi were using 27,000 and 180,000 number of iterations, respectively, this was more than enough time for them to play a game, and as such each side wouldn't have to worry about losing on time.

In order to help us evaluate each of the Pachi programs we developed a visualization tool. This tool allowed us to see how the neural network evaluated each possible move in its current position in the middle of a game. It would assign each legal move on the board a color based on how good it was. Figure 3.1 is a screenshot of how the neural network evaluated each move.

In this figure, the intensity of the color determines how good the move is. In this case, the light grey squares are considered to be better moves than the dark grey squares. In addition to the number of wins each implementation had against Fuego, we

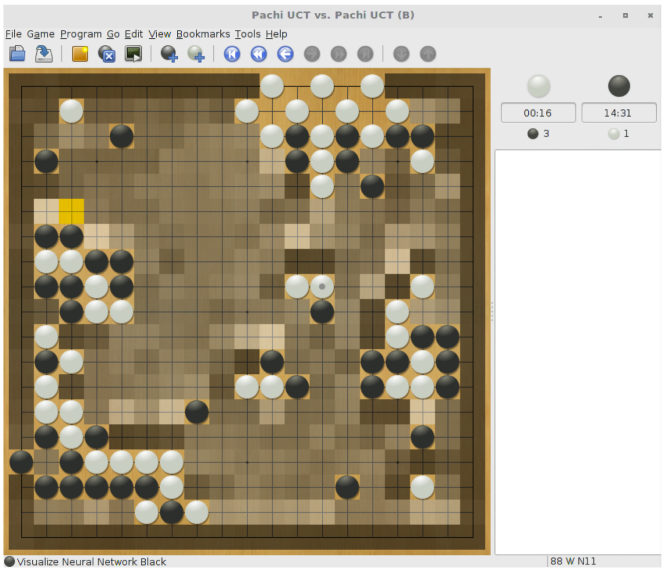

Figure 3.1: Neural Network Visualization - similar to [22]

also measured how the speed of the Pachi program was affected by each implementation. This gave us insight on how the neural network was affecting Pachi's performance. This information was crucial, because even if one of the implementations was significantly better than all of the other implementations, it would be impractical to use if it failed to perform fast enough.

4

Results & Evaluation

Here, we give the results obtained from the implementations we described in the methodology. For each implementation we devised a test which would show what kind of an impact it had on MCTS, and we measured the improvement it made. For each test, Pachi [16] was tested against Fuego [21]. In all of the tests, Fuego ran using 180,000 iterations.

For the first implementation we ran tests for the different levels of influence that the neural network had. In each of these tests, Pachi ran with 27,000 iterations. Also, in each test we gave each neural network a weight w for which it would contribute to the selection of the action the tree would be expanded with. Here, w is a number between 0 and 1, representing the influence the neural network had. The amount of influence that Pachi's heuristics had was thus $1 - w$. Below is a table of the win rates for each w we used.

In these games Pachi and Fuego alternated between White and Black, and they played 100 times. Hence, Pachi plays 50 games as White and 50 games as Black. When $w = 0.1$, Pachi won 15 games as Black and 31 games as White. When $w = 0.25$, Pachi won 13 games as Black and 31 games as White. When $w = 0.5$, Pachi won 46 games

w	Win Rate
0.1	46
0.25	44
0.5	50
0.75	50
0.9	50
1	50

Table 4.1: Pachi's Win Rate at Varying Neural Network Influence Levels

as Black and 4 games as White. When $w = 0.75$, Pachi won 0 games as Black and 50 games as White. When $w = .9$, Pachi won 0 games as Black and 50 games as White. When $w = 1$, Pachi won 0 games as Black and 50 games as White.

The ideal neural network influence appears to be 0.5. The 0.5 influence is the lowest influence with a 50% win-rate. Selecting 0.5 rather than 1 or 0.75 preserves a balance between prior heuristics used in Pachi and the neural network's output.

For the second implementation, we decided to test the win rate of the program using different boundaries for which layers the neural networks would be operating in. In order to get an idea of which layers would best serve as cutoffs, we measured the frequency with which MCTS visited each layer. Below is a graph of the results.

We decided to use two different neural networks for this implementation, both of which were provided by Levente Kocsis. We will call the two neural networks we decided to use ht19 and t18. The ht19 network used an additional feature related to the past 2-move history of the current position, while the t18 network did not. More importantly, the ht19 network was more accurate than the t18 network; however, it was also slower. In our tests, ht19 operated on nodes in the first n layers, while t18 operated on the nodes in the rest of the layers. In all situations, the neural network being used was weighted at $w = 0.5$ influence. We also ran two sets of tests, one set

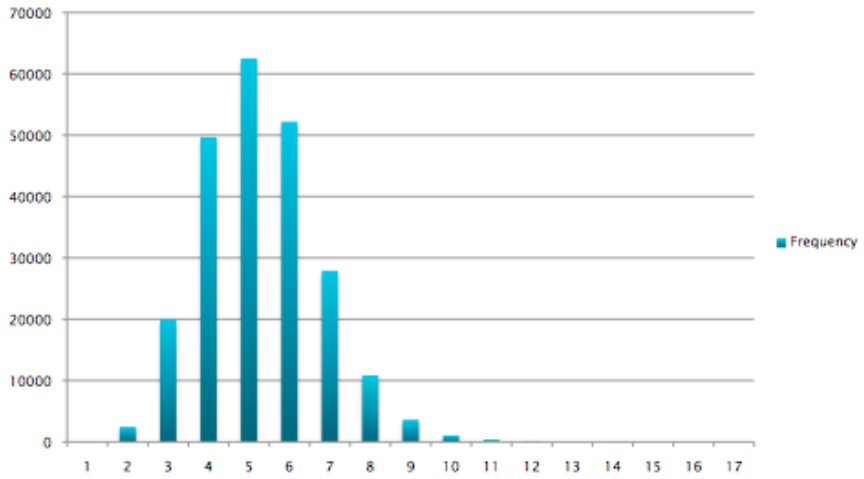

Figure 4.1: Frequency That MCTS Expanded a Node at Each Depth -

with timed games and another set with a fixed number of iterations, where Pachi ran at 27,000 iterations. For the timed games, each side had 30 minutes to play. Below is a table of the results.

Cutoff Layer	Timed	Fixed Number of Iterations
4	47	50
6	54	-
8	44	49
10	47	-
12	51	62
14	43	53

Table 4.2: Win Rate of Pachi with Different Neural Networks at Different Layers

For this implementation, we again had Pachi play Fuego 100 times, where they alternated colors. For layer $n = 4$, Pachi won 43 times as Black and 4 times as White when running with a fixed number of iterations per move, and won 45 times as Black and 5 times as White when it was timed. For layer $n = 6$, Pachi won 41 times as Black

and 13 times as White with fixed iterations. For layer $n = 8$, Pachi won 33 times as Black and 11 times as White with fixed iterations, and won 37 times as Black and 12 times as White when timed. For layer $n = 10$, Pachi won 31 times as Black and 16 times as White with fixed iterations. For layer $n = 12$, Pachi won 30 times as Black and 21 times as White with fixed iterations, and won 39 times as Black and 23 times as White when it was timed. For layer $n = 14$, Pachi won 25 times as Black and 18 times as White with fixed iterations, and won 31 times as Black and 22 times as White when it was timed.

It appears that there are two local maxima for timed games, and for games with a fixed number of iterations, the maximum win-rate is achieved at a cutoff of 12. The first phenomenon can be explained as follows. If the cutoff layer is early in the tree, then most nodes will be expanded with the help of the faster neural network. This allows more iterations to occur. If the cutoff layer is late in the tree, then most nodes are expanded with the help of the slower neural network. This allows each iteration to be more effective. However, if the cutoff is in the middle, the number of iterations is not maximized and neither is the effectiveness of each iteration. Thus it appears that it is best to either have an early cutoff or a late cutoff, or simply use only one neural network.

The second phenomenon could be due to the rarity of any node at depth 14 or greater being reached. This would reduce the program to a performance worse than a single neural network, since it almost always uses the slower neural network but it has to load an extra one that is almost never used.

For our third implementation, our testing setup was as follows. We varied the perturbation constant c, and then kept track of the current best accuracy in each case. The results are given in the following table.

Value of c	First Best Accuracy	Current Best Accuracy
0.001	0.082	0.1
0.002	0.067	0.099
0.004	0.076	0.094
0.005	0.069	0.103
0.006	0.066	0.105
0.008	0.07	0.08
0.01	0.058	0.092

Table 4.3: Accuracy of SPSA-trained Neural Network

Our third approach is quite slow. The progress appears promising, but it would be better if measured by win-rate rather than by predictive accuracy.

For our fourth implementation, we trained the neural network for 12,000,000 iterations on generated pattern files. The following table gives our results.

Top N Moves Considered	With Search-Based Feature	Without Search-Based Feature
1	0.2147	0.2057
2	0.3207	0.3125
3	0.3962	0.3863
4	0.4476	0.4402
5	0.4892	0.4842
6	0.5248	0.5171
7	0.5551	0.5475
8	0.5804	0.5714
9	0.6029	0.5925
10	0.6227	0.6146

Table 4.4: Accuracy of Neural Network with Search-Based Feature - as compared with the accuracy of the neural network with the same structure but without the search-based feature, after 12,000,000 iterations of training each

Our search-based feature appears to only marginally improve accuracy. This was not true in [17], but since the neural networks have been trained further and adjusted,

it now seems to make only a 1% difference in accuracy.

5

Conclusion & Future Work

5.1 Summary

AlphaGo's victory against Lee Sedol is a huge step forward in computer Go. We looked at four alternatives and extensions to AlphaGo's use of neural networks in informing the Monte Carlo Tree Search algorithm.

First, we varied the influence given to the neural network in the expansion phase of MCTS. We found that 0.5 appeared to be the best ratio of neural network influence to prior knowledge heuristics already found in Pachi.

Second, we experimented with using two neural networks of different strengths at different depths of the tree. We found that there are competing interests in number of iterations per move (more with the faster neural network) and effectiveness of each iteration (greater with the slower neural network). It appears that the best implementations focus on one or the other.

Third, we trained the neural network as part of Pachi through simultaneous perturbation stochastic approximation. The goal was to make the neural network better at recommending moves to explore to MCTS rather than better at predicting moves of human players on its own. There appears to be some progress; however the training is

quite slow and more work needs to be done in this area.

Fourth, we added a search-based feature of point ownership as in last year's project. This resulted in improvement last year, but after subsequent training of the neural networks this year, we saw little to no improvement when adding the search-based feature.

5.2 Future Work

There are several directions that can be pursued from here. One possibility is to train a neural network to be used with Pachi, using SPSA with improving win-rate as the goal rather than predictive accuracy. The weights of the entire neural network could be updated as well, rather than just the last layer as we did in our project.

Also, other search-based features could be explored. Though our particular search-based feature did not improve the accuracy of the neural network we used by a significant amount, other search-based features could potentially be quite useful to the neural network. Specifically, one could focus on features that in some way communicate the "state" of MCTS to the neural network, allowing it to suggest moves to explore that would benefit MCTS (this could be combined with training the neural network and MCTS as a system through SPSA as above).

Another equally interesting direction is to train a neural network initially only by reinforcement learning through self-play. As mentioned in the Introduction, this could lead to unique styles of play, unbiased by prior data.

References

[1] Sensei's Library. http://senseis.xmp.net/?TwoHeadedDragon, 2016. 1

[2] The Atlantic. http://www.theatlantic.com/technology/archive/2016/03/the-invisible-opponent/475611, 2016. 2

[3] Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *Nature* 529.7587 (2016): 484-489. 2, 23, 24

[4] Wired. http://www.wired.com/2016/03/final-game-alphago-lee-sedol-big-deal-humanity, 2016. 3

[5] Go Game Guru. https://gogameguru.com/lee-sedol-defeats-alphago-masterful-comeback-game-4, 2016. 3

[6] The KGS Go Server. http://www.gokgs.com, 2016. 6, 36

[7] Match 3 - Google DeepMind Challenge Match: Lee Sedol vs AlphaGo https://www.youtube.com/watch?v=qUAmTYHEyM8, 2016. 7

[8] Match 4 - Google DeepMind Challenge Match: Lee Sedol vs AlphaGo https://www.youtube.com/watch?v=yCALyQRN3hw, 2016. 4

[9] Medium. https://medium.com/@cristobal_esteban/move-37-a3b500aa75c2, 2016. 7

REFERENCES

[10] The Economist. http://www.economist.com/news/science-and-technology/21694540-win-or-lose-best-five-battle-contest-another-milestone, 2016. 8

[11] Gelly, Sylvain, et al. "The grand challenge of computer Go: Monte Carlo tree search and extensions." *Communications of the ACM* 55.3 (2012): 106-113. 12, 13, 15, 17, 29

[12] Bozulich, Richard. The Second Book of Go. Mountain View: Ishi International, 1987. Print. 13

[13] Kocsis, Levente, and Csaba Szepesvári. "Bandit based monte-carlo planning." *Machine Learning: ECML 2006*. Springer Berlin Heidelberg, 2006. 282-293. 17, 19

[14] Péret, Laurent, and Frédérick Garcia. "On-line search for solving Markov decision processes via heuristic sampling." *learning* 16 (2004): 2. 19

[15] Deep Learning. Computer Science Department, Stanford University. http://ufldl.stanford.edu/tutorial, 2013. 20, 21, 22, 32, 33

[16] Pachi: Software for the Game of Go/Weiqi/Baduk http://pachi.or.cz, 2016. 25, 39

[17] Wiratchotisatian, Pitchaya Student author – MA, Paisarnsrisomsuk, Sarun Student author – ID, and Sarkozy, Gabor N. Faculty advisor – CS. *UCT-Enhanced Deep Convolutional Neural Networks for Move Recommendation in Go*. Worcester, MA: Worcester Polytechnic Institute, 2015. Web. 25, 28, 32, 33, 36, 43

[18] Baudiš, Petr, and Jean-loup Gailly. "Pachi: State of the art open source Go program." *Advances in Computer Games*. Springer Berlin Heidelberg, 2011. 24-38. 26

[19] Gelly, Sylvain, and David Silver. "Monte-Carlo tree search and rapid action value estimation in computer Go." *Artificial Intelligence* 175.11 (2011): 1856-1875. 29

[20] Spall, James C. "Multivariate stochastic approximation using a simultaneous perturbation gradient approximation." *Automatic Control, IEEE Transactions on* 37.3 (1992): 332-341. 34, 35

[21] Fuego. http://fuego.sourceforge.net, 2015. 37, 39

[22] Computing Elo Ratings of Move Patterns in the Game of Go. http://www.remi-coulom.fr/Amsterdam2007, 2016 38