

April 2017

# Adaptive Neural Network Usage in Computer Go

Alexi Robert Kessler  
*Worcester Polytechnic Institute*

Ian Renee Shusdock  
*Worcester Polytechnic Institute*

Follow this and additional works at: <https://digitalcommons.wpi.edu/mqp-all>

---

## Repository Citation

Kessler, A. R., & Shusdock, I. R. (2017). *Adaptive Neural Network Usage in Computer Go*. Retrieved from <https://digitalcommons.wpi.edu/mqp-all/768>

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact [digitalwpi@wpi.edu](mailto:digitalwpi@wpi.edu).

# Adaptive Neural Network Usage in Computer Go

A Major Qualifying Project Report:

Submitted to the Faculty

Of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Alexi Kessler

Ian Shusdock

Date:

Approved:

---

Prof. Gabor Sarkozy, Major Advisor

# Abstract

For decades, computer scientists have worked to develop an artificial intelligence for the game of Go intelligent enough to beat skilled human players. In 2016, Google accomplished just that with their program, AlphaGo. AlphaGo was a huge leap forward in artificial intelligence, but required quite a lot of computational power to run. The goal of our project was to take some of the techniques that make AlphaGo so powerful, and integrate them with a less resource intensive artificial intelligence. Specifically, we expanded on the work of last year's MQP of integrating a neural network into an existing Go AI, Pachi. We rigorously tested the resultant program's performance. We also used SPSA training to determine an adaptive value function so as to make the best use of the neural network.

# Table of Contents

|  |           |
|--|-----------|
| <b>Abstract</b>  | <b>2</b>  |
| <b>Table of Contents</b>                                   | <b>3</b>  |
| <b>Acknowledgements</b>                                    | <b>4</b>  |
| <b>1. Introduction</b>                                     | <b>5</b>  |
| 1.1 - Artificial Players, Real Results                     | 5         |
| 1.2 - Learning From AlphaGo                                | 6         |
| <b>2. Background</b>                                       | <b>8</b>  |
| 2.1 - The Game Of Go                                       | 8         |
| 2.1.1 - Introduction                                       | 8         |
| 2.1.2 - Rules  | 9         |
| 2.1.3 - Ranking  | 11        |
| 2.1.4 - Why Study Go?                                      | 12        |
| 2.2. Existing Techniques                                   | 13        |
| 2.2.1 - Monte Carlo Tree Search                            | 13        |
| 2.2.2 - Deep Convolutional Neural Networks                 | 18        |
| 2.2.3 - Alpha Go   | 22        |
| 2.2.4 - Pachi with Neural Network                          | 24        |
| <b>3. Methodology</b>                                      | <b>27</b> |
| 3.1 - Investigating Anomalous Data                         | 27        |
| 3.2 - SPSA Analysis of Neural Network Performance in Pachi | 30        |
| 3.3 - Testing  | 32        |
| <b>4. Results &amp; Evaluation</b>                         | <b>34</b> |
| 4.1 Clarifying Last Year's Results                         | 34        |
| 4.2 Results from Optimized Neural Network Weight           | 37        |
| 4.2.1 Performance of V1                                    | 37        |
| 4.2.2 Performance of V2                                    | 39        |
| <b>5. Conclusions &amp; Future Work</b>                    | <b>45</b> |
| 5.1 Conclusions  | 45        |
| 5.2 Future Work  | 46        |
| <b>References</b>  | <b>47</b> |

# Acknowledgements

We would like to acknowledge

- Levente Kocsis, for his advice and mentorship.
- Joshua Keller and Oscar Perez, for their previous work and codebase.
- MTA Sztaki, for their facilities and hospitality.
- Gabor Sarkozy, for his guidance and support.
- Worcester Polytechnic Institute, for making this project possible.

# 1. Introduction

## 1.1 - Artificial Players, Real Results

Go is an incredibly complex game. Human players have worked for thousands of years to develop strategies and techniques to outwit their opponents. In the last few decades, however, these opponents have started to include artificial intelligences. Despite their newness to the game, artificial intelligences have grown by leaps and bounds, quickly learning the strategies and techniques developed by expert human players. As technology has improved, so too has the performance of these artificial Go players. Due to the complexity of the game and the flexibility of thought necessary to plan far ahead, computers have generally been unable to present much of a challenge to seasoned Go professionals. However, computers have been getting better. In 2016, Google's DeepMind team built an artificial Go player called AlphaGo and challenged the strongest human player in the world, Lee Sedol, to a five game match. Mr. Sedol managed to win only one game out of five against AlphaGo. For the first time in history, an artificial intelligence became the strongest Go player in the world.

AlphaGo's triumph was heralded as a shining example of what artificial intelligence can accomplish. AlphaGo used a unique approach involving both Monte Carlo Search Trees and multiple neural networks, leading to a uniquely powerful engine. While this was an innovative and well-executed approach, AlphaGo still relied on the resources of Google, a multi-billion dollar company with access to some of the best supercomputers in the world. There is a very

real gap between the best artificial intelligence an average computer can run and towering monoliths like AlphaGo. This is the problem we sought to address in our project.

## 1.2 - Learning From AlphaGo

Last year's MQP team, working on the project *Improving MCTS and Neural Network Communication in Computer Go*, had the same goal. They worked to incorporate a single neural network into an existing Go program, Pachi [14]. They gave this neural network different levels of influence over the decisions made by the program and tested how the program performed as a result.

We wanted to build off of this in our project. Our goal was to further improve the neural network integration so as to maximize performance. As last year's team had shown that Pachi could be combined with the neural network and still run on a personal computer, we wanted to take that neural network and use it in a more dynamic way. Before doing so, however, we wanted to investigate some anomalous data from last year's results. Their version of Pachi seemed to behave anomalously at different neural network weightings. We wanted to make sure that we built upon a solid foundation and did not exacerbate the strange behavior.

Once we had investigated the anomalous data, we set about improving the neural network integration using adaptive weighting. Our thinking was that one major factor working against last year's use of the neural network was how rigidly it was applied. The suggestions of the neural network were given a certain amount of influence in the move selection process, regardless of any factors such as game turn or board state. This left the combination of Pachi and the neural network unable to respond dynamically to a game that could change very drastically.

Given that Pachi is a strong artificial intelligence on its own, it wouldn't have made sense to drown out its suggestions in favor of the neural network. At the same time, we wanted to utilize the neural network in any situation where it could improve the outcome. Our goal was therefore to develop an optimal value function that used multiple parameters to determine how strongly it should weigh the suggestions given by the neural network at any given turn. We went about determining this function using a well-known algorithm called simultaneous perturbation stochastic approximation (SPSA) [19]. We describe our approach further in our Methodology section. However, before that we will present the background information necessary to understand our approach.



## 2. Background

### 2.1 - The Game Of Go

#### 2.1.1 - Introduction

Go is a board game originating in ancient China. The earliest written record of the game can be found within a historical annal from the late 4th century BCE [19]. At its heart, Go is a game about capturing territory on an  $n \times n$  grid, where  $n$  is usually 19. Each player uses either black or white stones. Territory is captured by surrounding empty areas with one's stones. Despite the seeming simplicity of its goal, Go is a game rich in strategy. The large board space combined with the complex tactics involved in outmaneuvering an opponent lead to Go being an incredibly deep game.

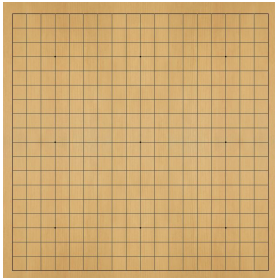
In the modern age, Go's complexity presents an interesting challenge for artificial intelligence. Go is, in some ways, is closer to the real world than other traditional games. In Go it often makes sense to sacrifice short-term gains for long-term advantages. In practical terms, this might mean trading small definitive captures for larger but more nebulous strategic gains. These concepts, forward thinking and big-picture strategic planning, are things that artificial intelligences struggle with. Advances in artificial intelligences for playing Go are, in a sense, advances that take AI closer to being able to intelligently interact with the real world. Computer scientists have spent a long time working on the problem of developing a professionally competitive Go AI. Google has recently had much success with a supercomputer running their

custom AI, AlphaGo. Despite this, there is still much work to be done in creating more efficient, less hardware intensive, artificial intelligences.

While Google's accomplishment with AlphaGo was a huge step forward, it was achieved using the vast computational resources available to Google as a company. These resources are not available for the average AI. If one's goal is to develop artificial intelligences that are better able to interact with the real world, one must take into account the very real limitations of the average computer. Thus, there is worth in trying to implement aspects of AlphaGo's structure in a way that doesn't rely on using as many resources. The ideal goal would be the creation of a powerful Go AI able to run on a household laptop. That will be the main focus of this paper, but first it is worth delving more into Go as a game.

### 2.1.2 - Rules

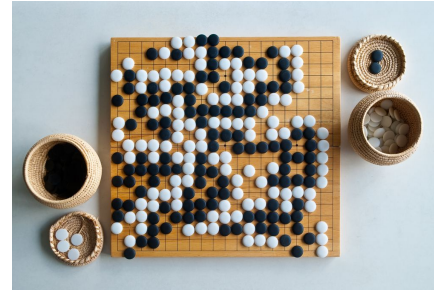
In Go, each player takes turns placing one of their stones on the game board. One player plays as Black, and one as White, with Black moving first. The standard board size is 19 x 19, as can be seen in Figure 2.1(a), though other board sizes are sometimes played. Stones are placed on the intersections of grid lines on the board as in Figure 2.1(b). Players go back and forth, each trying to outmaneuver the other and "capture" areas of the board. By the end of a game, this can lead to a very complicated arrangement of stones, such as the game depicted in Figure 2.1(c).



**Figure 2.1(a) Empty Board**



**Figure 2.1(b) Placed Stone**



**Figure 2.1(c) Full Board [5]**

A individual stone placed by itself in the middle of the board is said to have 4 *liberties*. This means that the spaces above, below, to the left and to the right of it are free. When another stone blocks one of these spaces, the original stone loses a liberty. When a stone of the same color blocks one of these spaces, the two stones are known as a *group* and the group's liberties are the sum of each stone's remaining liberties. When a stone or group loses all of its liberties, it is considered captured and is removed from the board. The black piece in Figure 2.2(a) has only one liberty left, and thus is one stone away from being captured.

Territory is considered captured if it is surrounded by stones of one color. The game is scored by a combination of the territory and enemy stones captured by each player. When this scoring is being calculated, isolated stones of one color within the territory of the other are counted towards the territory of the other. This situation can be seen in Figure 2.2(b), where the squares represent the scoring of different places on the board. Even though white has two stones in black's territory, they are isolated and are considered effectively captured.

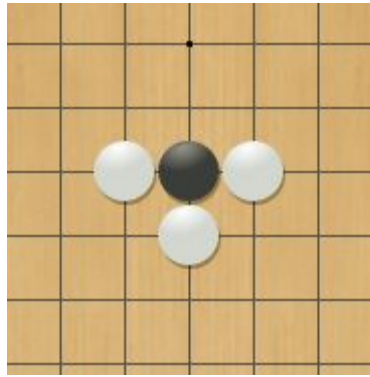


Figure 2.2(a) Nearly Surrounded Stone

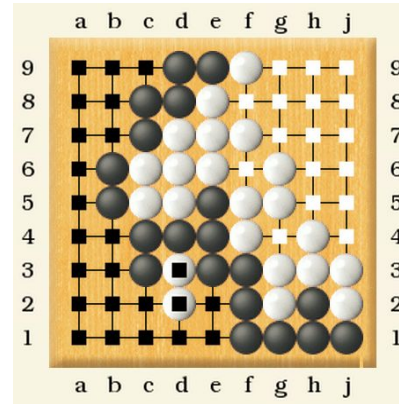


Figure 2.2(b) End Game Scoring [13]

### 2.1.3 - Ranking

The ranking of Go players is split into three tiers. These are student grades, amateur grades, and professional grades. Those at the student level are ranked by *kyu*, decreasing from 30 to 1. An absolute beginner would be a 30-kyu player, while a 1-kyu player would be a very strong intermediate player. Immediately after kyu are the amateur *dan* rankings, increasing from 1 to 7. Dan rankings are where rankings begins to be determined by official organisations and tournaments. Though this level is referred to as amateur, it takes an advanced player to reach it. Go utilizes handicaps, a penalty of a certain number of stones, to ensure players are evenly matched. The kyu and amateur dan rankings are representative of these handicaps. A 3-dan player might expect to win half of the games playing against a 1-dan player with a 2 stone handicap.

After the amateur dan rankings come professional dan rankings, increasing from 1 to 9. These ranks are available only to those players who play Go professionally. 9-dan professionals are the strongest players in the world. The handicaps in professional dan rankings are much

smaller than in the lower ranks. A 9-dan professional would likely only have a 2-3 stone handicap when playing against a 1-dan professional.

#### 2.1.4 - Why Study Go?

As touched on earlier, Go is an incredibly complex game. There are over  $10^{170}$  possible board combinations compared to the  $10^{47}$  in Chess. Go has possible  $10^{360}$  possible move sequences, much more than the  $10^{123}$  in Chess. Each move can also have a long term influence on the game. A stone placed early on can win or lose a strategic battle hundreds of moves later on. The way territory is calculated also adds complexity to the game. It can, at times, make sense to sacrifice guaranteed, immediately scorable territory in order to achieve better strategic positions in other locations. As such, simple heuristics are poor at calculating the value of a given board state. Each of these factors alone would make developing artificial intelligence for Go difficult, but the combination makes any advance in Go AI impressive.

Many real world problems, especially sequential decision based problems, face similar challenges. They have functionally infinite possible states and sequences which makes a brute force solution impossible. Actions can have long term influences which makes prediction difficult. States are not always as they appear which complicates the decision making process. Advances in Go AI could help spur advances in AI relating to these problems. As such, studying Go has many practical applications.

## 2.2. Existing Techniques

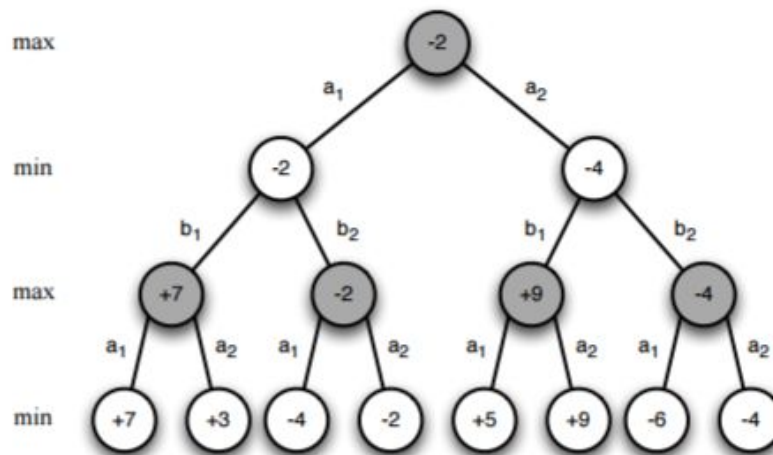
There have been numerous advances in computer Go techniques over the past few years. The introduction of Monte Carlo Tree Search followed by the use of deep convolutional neural networks has led to significant advances in computer playing ability, culminating in the win of AlphaGo over Lee Sedol.

### 2.2.1 - Monte Carlo Tree Search

The first major breakthrough in computer Go was Monte Carlo Tree Search (MCTS), explored in *The Grand Challenge of Computer Go* by Sylvian Gelly et al [8] . Until this advance, most Go programs used Minimax Tree Search. Minimax Tree Search is designed to be used on deterministic games of perfect information like Chess and Go. In other words, it is used on games with no randomness and where all information is visible to all players. An example of a minimax search tree can be seen in Figure 2.3.

Minimax Tree Search works by assigning each player a goal, to maximize or minimize the final value of the tree. Player One traditionally aims to maximize the score whereas Player Two aims to minimize it. As such, each player has opposing goals. Each node of the tree represents a given board state with the layer representing which player's move it is. The subsequent layer, if it exists, represents the possible moves that player can make. A leaf node can either represent an ending board state, with one player winning and one losing, or just an arbitrary board state to evaluate if depth is limited. These leaf nodes are then given a point value representing the given board state. A positive value represents a good board state for the player

maximizing and a negative value represents a good board state for the player minimizing. The search algorithm works by assuming both players play perfectly, with the maximizer choosing the highest values when it is her turn and the minimizer choosing the lowest values when it is his turn. Each non-leaf node is then given the best value that that player can achieve. As this recurses up the tree, the player whose turn it is can decide which move is the best to make.



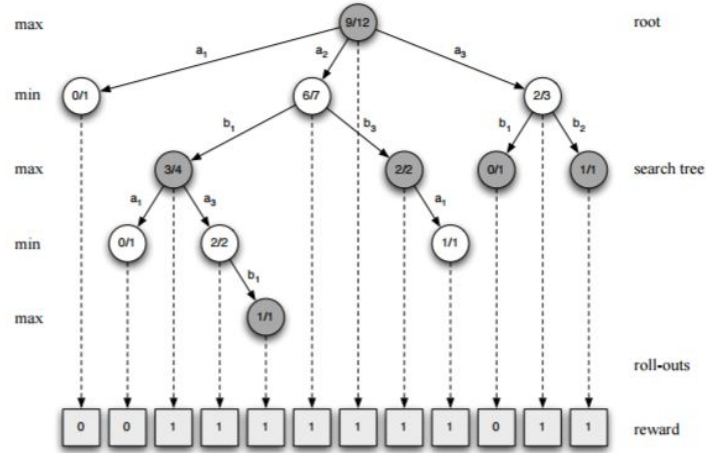
**Figure 2.3 Example of a Minimax Tree [7]**

Unfortunately, Minimax Tree Search requires every move to be fully mapped out to work. As many games have significant branching factors, Minimax takes up significant resources and computation time and is impractical for games such as Go. However, assuming the search does not go all the way to every ending state, it can be useful. This requires an evaluation, or heuristic, function to evaluate states that do not immediately result in a win or loss. For games like Chess, this can be determined by assigning each piece a point value and calculating the total point value on the board for each player. However, the rules of Go, with territory and capturing pieces, make this much more difficult to calculate.

A breakthrough solution to this problem came in the use of Monte Carlo simulations. Monte Carlo simulations work by creating a playout policy which determines what move to play in any given situation. A playout policy is created for each player and then a game is simulated from any given state  $s$  by following the playout policies for each player until the simulation ends. This makes the results of the game very sensitive to the policy. There isn't, however, a single optimal policy in any given situation. Certain policies may be good in certain situations, but not others. One way to deal with this is to allow some randomness in the generation of policies. This removes systematic errors (errors in the creation of the policy). When many simulations are run and the win-rate from state  $s$  is estimated, it is possible to tell generally if  $s$  is "easy" or "hard" to win from, regardless of the policy used. As a result, it is possible to estimate the general likelihood of winning from state  $s$ . The introduction of randomness into a deterministic game of perfect information was novel at the time of its inception [1].

Combining Minimax Tree Search and Monte Carlo simulations led to Monte Carlo Tree Search. It selectively grows a game tree and when reaching a "leaf" node (in this case a node where we stop expanding), the algorithm estimates the value of the node with Monte Carlo simulations and that information is used to perform a variant of Minimax Tree Search. MCTS favors expanding nodes with high values. In other words, it follows what it predicts to be optimal play. Figure 2.4 below shows an example of MCTS.





**Figure 2.4 Example of Monte Carlo Tree Search [8]**

One major improvement to the Monte Carlo Tree Search algorithm came with the introduction of Upper Confidence Bounds on Trees (UCT). UCT works by helping determine which nodes to select to expand. Traditionally, only “optimistic” nodes were expanded. That is, nodes that seemed to have the highest value. However, the playout policy is slightly inaccurate due to randomness and the fact that it is not optimal. As such, the estimated optimal choice and the actual optimal choice do not always align. UCT combats this by encouraging exploration. The more a certain node is visited, the less enticing to explore it becomes. This means that suboptimal nodes are forgotten once they have been thoroughly explored. Other nodes that would otherwise be ignored are explored periodically. This is guaranteed with the following:

$$Z(s, a) = \frac{W(s, a)}{N(s, a)} + B(s, a) \quad (2.1)$$

$$B(s, a) = C \sqrt{\frac{\log N(s)}{N(s, a)}} \quad (2.2)$$

where,

- $Z(s, a)$  is the node choice value (the higher the value, the more desire to simulate games from this node),

- $C > 0$  is a tuning constant,
- $N(s, a)$  is the number of simulations where move  $a$  was chosen from state  $s$ ,
- $N(s) = \sum_a N(s, a)$ ,
- $W(s, a)$  is the total reward from terminal states of these simulations, and
- $B(s, a)$  is the exploration bonus.

$B(s, a)$  provides a high bonus to the node choice value for minimally explored nodes and a low bonus for heavily explored nodes. As such, nodes with a high  $Z(s, a)$  value will either be highly optimal or poorly explored. This also prevents suboptimal nodes from being chosen. The following theorem proves this for UCB1, an implementation of Monte Carlo Tree Search with UCT:

**Theorem 1.** *Consider UCB1 applied to a non-stationary problem. Let  $T_i(n)$  denote the number of plays of move sequence  $i$ . Then if  $i$  is the index of a suboptimal move sequence  $n > K$ , then*

$$E[T_i(n)] \leq \frac{16C_p^2 \ln(n)}{(\Delta_i/2)^2} + 2N_0 + \frac{\pi^2}{3}.$$

Where,

- $\Delta_i$  is the difference in the optimal move sequence and actual move sequence,
- $C_p$  is a constant,
- $E$  is error,
- $N_0$  is a constant satisfying boundary conditions,
- $n$  is the number plays, and
- $K$  is the number of actions.

This theorem essentially states that suboptimal move sequences will eventually stop being explored. This means computation time and resources are not wasted on suboptimal moves. The following theorem guarantees that UCB1 (and UCT) never stops exploring,

**Theorem 2.** *There exists some positive constant  $p$  such that for all move sequences  $i$  and  $n$ ,  $T_i(n) \geq \text{ceil}(p * \log(n))$ .*

The two above theorems prove UCB1 never stops exploring and will eventually stop following suboptimal routes. This implies that it eventually chooses the optimal path. The following theorem proves this,

**Theorem 3.** *Consider a finite-horizon Markovian Decision Problem (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  $O(\log(n)/n)$ . Further, the failure probability at the root converges to zero at a polynomial rate as the number of episodes grows to infinity.*

In other words, UCT eventually converges to the optimal answer. Minimax Tree Search, combined with Monte Carlo simulations and Upper Confidence Bounds on Trees ultimately leads to a very effective algorithm for computer Go.

### 2.2.2 - Deep Convolutional Neural Networks

While Monte Carlo Tree Search has greatly contributed to the success of Go AIs, it is limited by the scope of its prior knowledge. Programs that utilize MCTS rely on prior knowledge to construct an evaluation function. They then use that evaluation function to calculate move preference from a given board state. This reliance on prior knowledge often leads to a bottleneck

situation in which the performance of the program is limited by the quality and quantity of prior knowledge to which it has access. One approach for addressing this bottleneck is the use of neural networks.

A neural network, at its core, is a computational model loosely structured around the functioning of neurons in the human brain. Neural networks are composed of artificial “neurons”. These neurons can be connected and layered to increase the complexity of a network. The neurons themselves are nodes that process inputs in a specific way. A diagram of what happens within a neuron is shown in Figure 2.5.

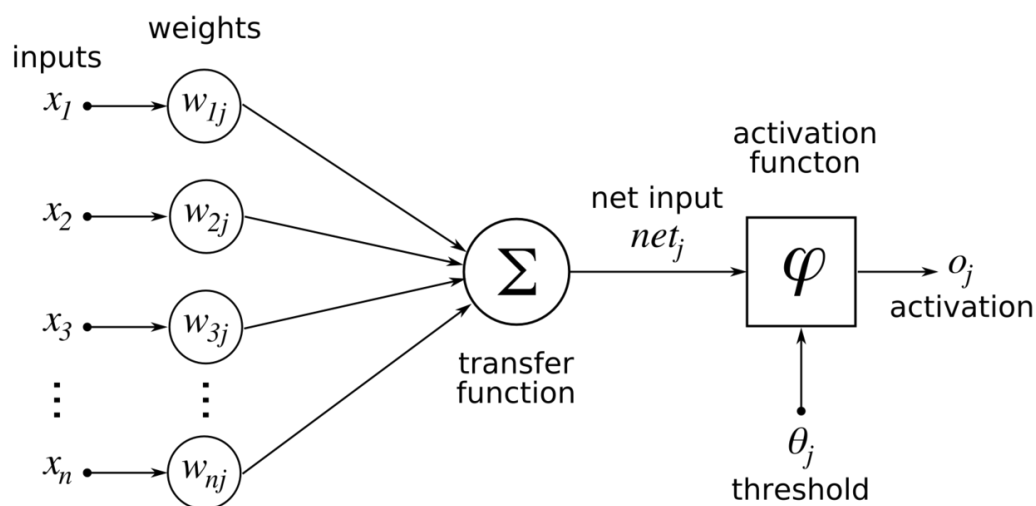
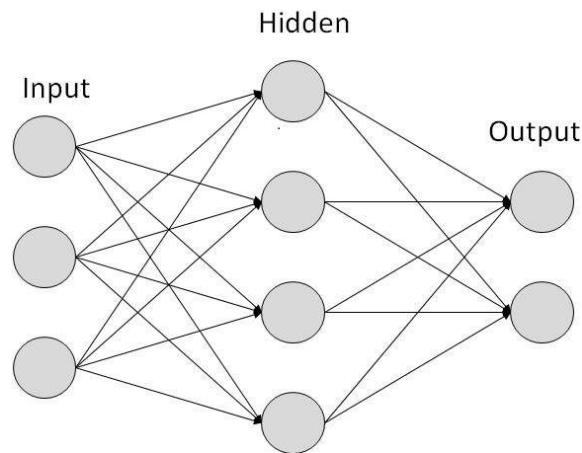


Figure 2.5 Function of a Neuron in an Artificial Neural Network [3]

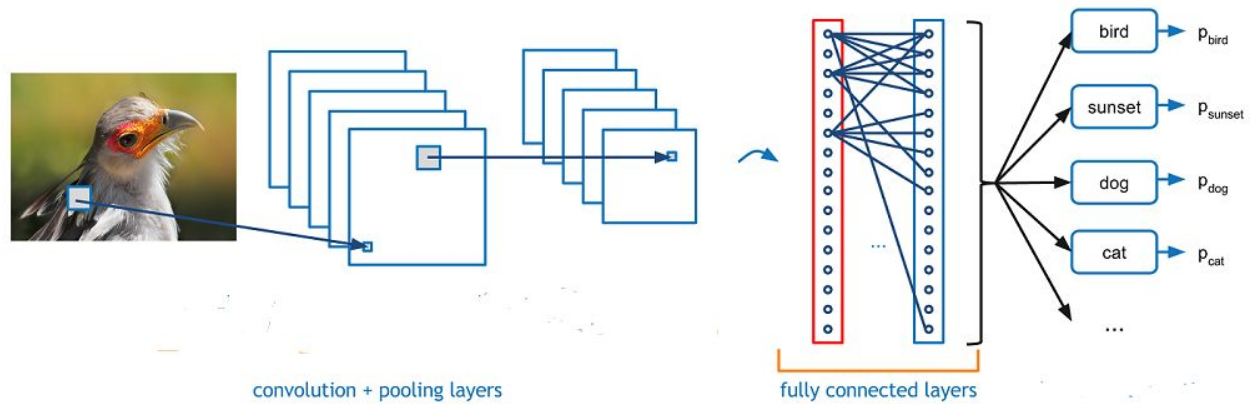
The neuron starts by receiving any number of inputs, here represented as  $x_1$  through  $x_n$ . The values of these inputs are modified by a specific weight assigned to that input. These weights are what determine how much of an impact each input has on the output of the neuron. Once the inputs have been weighted, they are fed through a transfer function. The transfer functions combines the weighted inputs, and produces a value which is sent to an activation function. The activation function processes the value it is given, and produces a result.

In this way, a neuron takes multiple inputs and produces a single output. A single neuron, however, is not very useful for solving problems. Neural networks are composed of many neurons, with these neurons most often being arranged in layers. A neural network consisting of three layers can be seen in Figure 2.6. There is the input layer, a grouping of neurons that all handle input data. The input neurons send their output to neurons in a hidden layer. This layer is “hidden” because it is not readily visible to an outside observer. Hidden layers are often stacked in practice. The outputs from the hidden layer neurons are sent to the neurons in the output layer, and it is this layer that provides the final output of the neural network.



**Figure 2.6 A Three Layer Neural Network [2]**

Normal neural networks are incredibly useful for many applications. Unfortunately, as more nodes are added, the network rapidly increases in complexity. This is because a new node must be connected to a great number of other nodes, and they in turn to it. Convolutional Neural Networks (CNNs) are an answer to this problem. They are designed to handle a large number of input features and as such excel at pattern recognition. CNNs are structured in a way similar to, but more complex than, that of a basic neural network.



**Figure 2.7 Basic Structure of a Convolutional Neural Network [4]**

The structure of a CNN is modeled after that of the visual cortex of the brain. Rather than every neuron in a layer being connected to every other neuron, the neurons are separated into small collections. These collections, or receptive fields, process overlapping sections of the input image. The processed sections are then tiled together and processed by the next layer. The overlapping of the tiles leads to a higher resolution of the processed image. After being processed by the convolutional layers, the tiled images are processed by fully connected layers. These layers identify the probabilities of the original image belonging to any of a number of categories.

Convolutional Neural Networks, being specialized to perform complex image recognition and analysis, have been used in Go engines before [6][16]. When building a CNN-centric Go engine, the CNN is used to predict, from a given board state, what move an expert Go player would play next. To this end, CNNs in Go are trained on collections of expert moves.

Recently, successful implementations of CNNs in the world of Go have focused on two methods for improving performance. These are using deeper, more complex networks, and combining CNN move prediction with MCTS rollouts. Maddison et al., using a custom feature set to describe board states, tested several networks of varying depths [11]. They found that the

deeper the network, the higher the chance that the network's predicted move would be in an  $n$ -size expected set of expert moves. These results can be seen in Figure 2.8. Correspondingly, the researchers found that the deeper the network, the better it's win rate versus an MCTS based AI, GnuGo. This can be seen in Table 2.1.

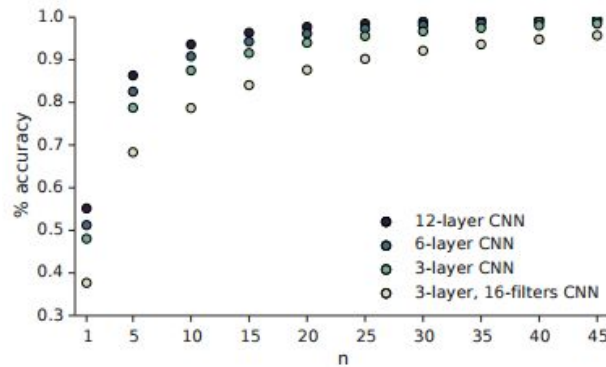


Figure 2.8 CNN testing results [12]

| Depth  | Size              | % Accuracy   | % Wins vs. <i>GnuGo</i> | stderr    |
|--|-------------------|--------------|-------------------------|-----------|
| 3 layer                                      | 16 filters        | 37.5         | 3.4                     | $\pm 1.1$ |
| 3 layer                                      | 128 filters       | 48.0         | 61.8                    | $\pm 2.6$ |
| 6 layer                                      | 128 filters       | 51.2         | 84.4                    | $\pm 1.9$ |
| 10 layer                                     | 128 filters       | 54.5         | 94.7                    | $\pm 1.2$ |
| 12 layer                                     | 128 filters       | 55.2         | 97.2                    | $\pm 0.9$ |
| 8 layer (Clark & Storkey, 2014) <sup>4</sup> | $\leq 64$ filters | 44.4         | 86                      | $\pm 2.5$ |
| <i>Aya</i> 2014                              |                   | 38.8         | 6                       | $\pm 1.0$ |
| <i>Human</i> 6 dan                           |                   | 52 $\pm 5.8$ | 100                     |           |

Table 2.1 CNN vs GnuGo results [11]

The team also tested a combination of a CNN and an MCTS based search. Playing the combined engine against a base 12-layer CNN, they found that the search-based program won in 87% of the games. This demonstrated the value in combining powerful neural networks with traditional search-based techniques, paving the way for further developments [12].

### 2.2.3 - Alpha Go

As mentioned in the Introduction, the most impressive advance in computer Go techniques occurred in 2016 in the form of AlphaGo. AlphaGo is a revolutionary new computer

Go program developed by Google DeepMind that combines Monte Carlo Tree Search, Deep Convolutional Neural Networks, and Google's massive computing resources to create the best Go program yet. When the paper was first published, Silver reported a win rate of 99.8% against other Go programs and a 5-0 win against the human European Go champion Fan Hui [17]. AlphaGo later defeated Lee Sedol, the best Go player in the world, four games to one [13]. Clearly AlphaGo is an excellent Go program.

There are three main factors that contributed to AlphaGo's success. The first, and most difficult to achieve, is the massive computing power at its disposal. AlphaGo was run with 1,202 CPUs and 176 GPUs, quantities unavailable to all but the most committed of companies. The second factor was the combination of multiple neural networks and MCTS. AlphaGo utilized a policy network that was used to select moves during rollouts and a value network that was used to estimate the winner of a game based on board position. Both of these were combined to help AlphaGo decide which move to make. The third advance was in what the policy network was trained for. Traditionally, policies were meant to predict what move an expert would make in a given state. This meant Go programs were attempting to copy experts rather than "think for themselves" and ultimately led to naïve policies that were only good in a limited scope of games. AlphaGo trained its policy network to maximize win rate. As such, it was able to make moves no one could predict.

The most astounding move AlphaGo made was move 37 of its second game against Lee Sedol[12]. AlphaGo placed a stone in a position that no human expert would have. Commentators at first thought it was a move translation mistake. Lee Sedol himself was so surprised he had to leave the room for 15 minutes to regain his focus. The move proved to be a



brilliant strategic play and allowed AlphaGo to perform strongly moving forward. AlphaGo diverged from human experts in its play strategy and benefitted as a result. This showed that it could not only copy experts, but play original moves of its own as well. AlphaGo was strong enough not just to play well, but to surprise its human opponent.

#### 2.2.4 - Pachi with Neural Network

Last year's MQP, *Improving MCTS and Neural Network Communication in Computer Go*, took inspiration from the success seen in AlphaGo's performance [9]. The team's focus was on creating a Go AI that used some of the same innovative advances as AlphaGo, but without requiring the same computational resources. To this end, they pursued four different strategies for improving the communication between neural networks and an MCTS based Go AI.

The team chose Pachi for their AI. Pachi is a powerful open-source program that uses a specialized form of MCTS called RAVE. Their project involved augmenting Pachi with a neural network. For their neural network, they chose one built and trained at the Computer and Automation Research Institute of the Hungarian Academy of Sciences, our sponsor. The group focused on integrating the neural network into RAVE's move selection process. To do this, they took four approaches. These were:

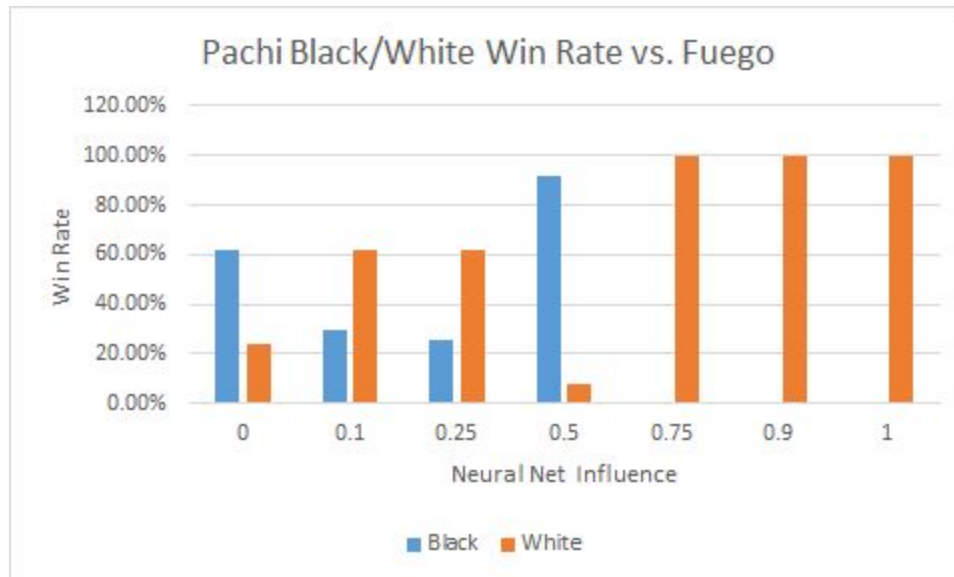
- Adding the neural network's output to that of Pachi's default evaluation,
- Using a faster, less accurate neural network at lower depths (closer to the leaves) in the search tree and a slower, more accurate one at higher depths (closer to the root),
- Training a new neural network specifically to inform the Monte Carlo Tree Search rather than help select a move, and

- Adding a search-based feature to the input of the neural network.

Of the four, they found the first approach to be the most promising.

The first approach had Pachi make moves based on both Pachi's default move selection and the result of the neural network. In order to do this, they took the move selection probability matrix of the neural network as well as the probability matrix output by Pachi's prior heuristics and combined them with different weightings. The neural network results were given a weight between 0 and 1, with 0 meaning that Pachi chose the move independently, 0.5 meaning each had an equal say, and 1 meaning that the neural network's move choice was the only one considered. Pachi with the weighted neural network influence was played against another Go AI, Fuego [7].

While the results from the first approach appeared quite encouraging, there were some anomalies in the data. As can be seen in Figure 2.9, Pachi appeared to consistently play better when playing as white, except in the case of the neural network receiving a 0.5 weighting, where Pachi won a majority of games as black. Additionally, at any weighting higher than 0.5, Pachi lost every game as black and won every game as white. We began by trying to investigate the cause of these anomalies.



**Figure 2.9 Results of Integrating Neural Network with Different Weightings**

## 3. Methodology

As mentioned earlier, we began by investigating the anomalous data from last year's MQP. We pursued several different causes of the discrepancies, ultimately determining that it was due to a misunderstanding of the way in which GoGui, the tool used for playing the games themselves, reports its data. We then moved on to modifying Pachi to adaptively weight the neural network in an effort to optimize playing ability. We did this by taking two parameters, game turn and Pachi's move confidence, and using them to decide the influence given to the neural network on any given turn. The parameters were weighted using values determined through simultaneous perturbation stochastic approximation.

### 3.1 - Investigating Anomalous Data

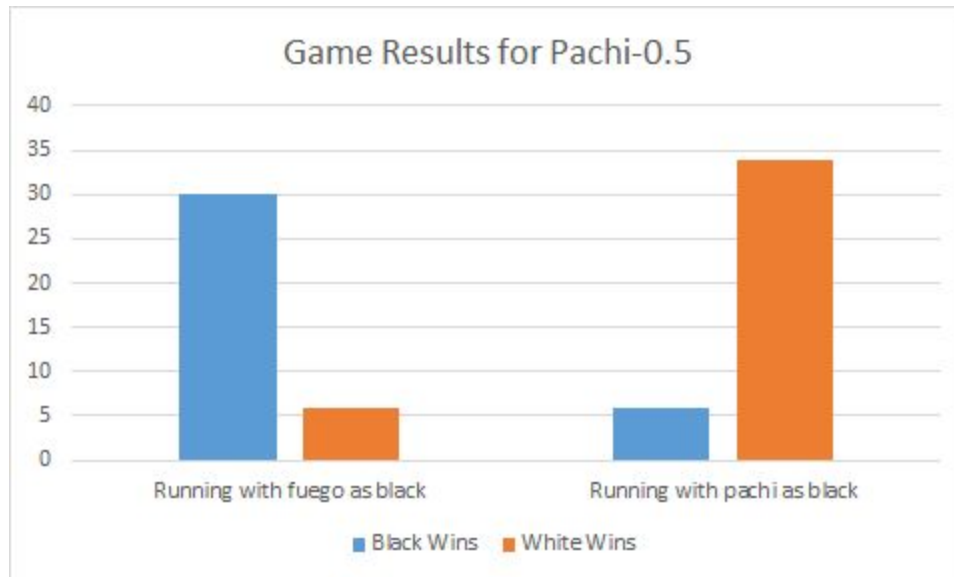
We began our work by attempting to recreate the data seen in last year's MQP, shown in Figure 2.9. We first investigated the testing environment used to produce the data. As described earlier, last year's MQP team played two Go AIs against one another, Pachi and Fuego, another open source Go AI. Pachi was modified to incorporate the results of the neural network with different levels of importance, Fuego was left untouched as a baseline. In order to play the two Go AIs against one another, they used a tool called GoGui. For each weighting of the neural network, 100 games in total were played.

We set about downloading and setting up the tools used by last year's team so as to match their specifications. We encountered our first hurdle when setting up Fuego. In last year's

tests, Fuego was run with a configuration file that was absent from the existing codebase. We solved this by exploring the different parameters available and creating a configuration file that would produce the same output as last year's. Next, we reworked some of the modified code for Pachi, so as to allow it to compile and run properly on our machines. Lastly, we examined all of the code uploaded by last year's MQP, found the section in which they applied the neural network weightings, and compiled versions of Pachi using each neural network weighting.

Once we began running tests, we quickly realized that each game took between 30 and 50 minutes to play out. This meant that we could not rapidly run the same tests as last year's group. Instead, we ran our tests in batches, trying to get a picture of the artificial players' performance over the different neural network weightings. As we ran more tests, we started to see the same pattern. We looked at the sections of Pachi's code that last year's group modified, but the modifications were sound. The neural network weightings were being applied properly and there was nothing in the code to explain the behavior.

We had a breakthrough when we experimented with different ways of running the tests. One of these ways involved passing the arguments to GoGui, the game coordinator, in a different order. GoGui runs games in an alternating color order, so we expected to see the same number of black and white wins as before. Surprisingly, the win rates flipped when the starting colors were flipped. This can be seen in Figure 3.1. This implied that the order in which the engines were fed to GoGui changed the results. As that should not have been the case, we dug deeper.



**Figure 3.1 Results of Flipped Colors Experiment**

We examined the interactions between GoGui and the AIs themselves by modifying Pachi to print out what color it was being run as. In doing this, we found something very strange. When GoGui was given Pachi as initially playing black, it continued to refer to it as black, even in games where it played as white due to the alternating. This can be seen in Figures 3.2(a) and 3.2(b).

```
B<< Black
B<< Black
B<< =
B<<
B>> genmove b
IN: genmove b
```

**3.2(a) Pachi Playing Black**

```
B<< White
B<< White
B<< =
B<<
B>> genmove w
IN: genmove w
```

**3.2(b) Pachi Playing White (Still labeled B)**

Based off of this observation, we concluded that GoGui may label the results not by what color the AIs were playing, but by which color they were assigned when the batch of games began. We used this to remodel the data from last year, achieving the graph seen in Figure 3.3. This was a much more reasonable graph, and showed that previous interpretations of GoGui's

results, including our own, were incorrect<sup>1</sup>. The reinterpretation showed that the higher the neural network influence, the worse Pachi performed. Given this, the question became, how can we best use the neural network if it seems to negatively affect Pachi's performance.

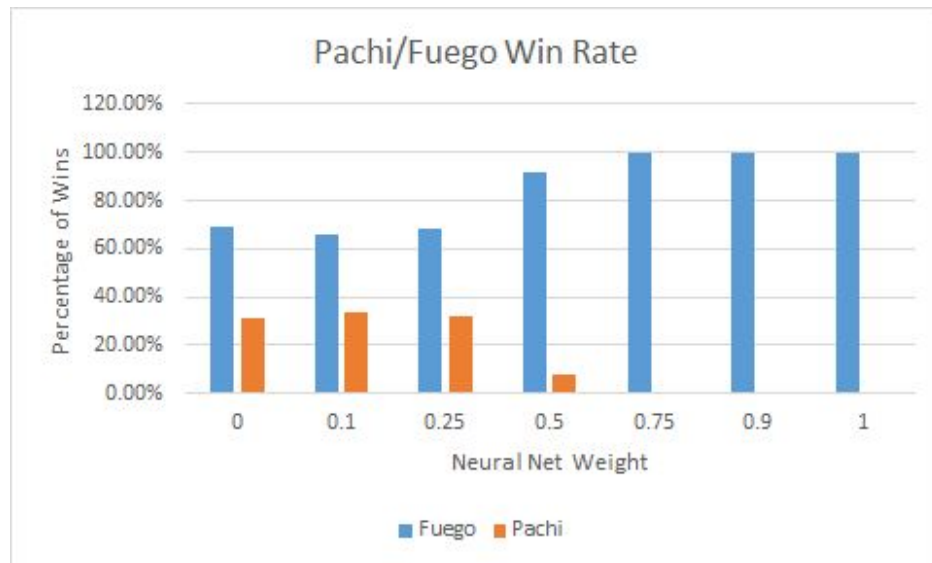


Figure 3.3 Remodeled GoGui Results

## 3.2 - SPSA Analysis of Neural Network Performance in Pachi

We investigated methods of adapting the neural network weighting during a game. We believed using a dynamic weighting, as opposed to the static weighing used by last year's MQP, would increase Pachi's playing ability. We ultimately decided to pursue two parameters: current game turn and Pachi's move preference. We suspected game turn to have a strong, negative impact in the late game when the game board becomes more complicated and divergent from the neural network's training set. Changing the neural network based on Pachi's move preference

<sup>1</sup> In remodeling the data, we corrected for the spike at 0.5 by assuming the tests were run with the two engine colors flipped with regards to the other tests. This closely matched our own results when testing both configurations. We say assume as we did not have access to the data at 0.5.

was expected to support very good moves chosen by Pachi and to use the neural network more when Pachi was less confident of winning. We started with the following function:

$$a * turn + b * pachi\ value + c \quad (3.1)$$

The goal remained to determine the optimal values for  $a$ ,  $b$ , and,  $c$ . We did this through simultaneous perturbation stochastic approximation (SPSA).

SPSA is a technique designed by Spall as a technique to estimate the gradient of a cost function in noisy data problems [1]. It acts as a fast alternative to finite difference stochastic approximation methods. As Monte Carlo Tree Search inherently has some randomness, it has some amount of noise. This noise adds considerable difficulty in determining an optimal function. As SPSA is designed for noisy data function generation, it was a natural choice for determining the constants in equation (3.1).

Suppose there is a loss function  $f$  that has known parameters, but unknown weightings (for said parameters)  $\bar{\theta}$ . That is  $f$  is of the form,  $\theta_1 * x_1 + \theta_2 * x_2 + \dots$ , where  $x_1, x_2, \dots$  are known parameters but  $\theta_1, \theta_2, \dots$  are unknown constants. SPSA works by perturbing  $\bar{\theta}$  by a small, random amount,  $\bar{\Delta}$ , over the course of the algorithm. SPSA then compares function results from said perturbations and uses said comparison to update  $\bar{\theta}$ . The general equation is as follows,

$$\bar{\theta}_{k+1} = \bar{\theta}_k - a_k * \frac{f(\bar{\theta}_k + c_k \bar{\Delta}_k) - f(\bar{\theta}_k - c_k \bar{\Delta}_k)}{2c_k} * [\bar{\Delta}_k^{-1}] \quad (3.2)$$

where  $a_k, c_k$  are gain sequences subject to restraints set forth in [18] and [19]. In this way, every element of  $\bar{\theta}$  is perturbed in each iteration, requiring only two loss function calls regardless of the dimensions of  $\bar{\theta}$ . While the conditions of convergence to an optimal  $\bar{\theta}^*$  are very specific as laid forth in [19], they are easy to achieve in practice.



For the purposes of determining the optimal neural network weighting at any given point, we were required to use a somewhat complex loss function  $f$ . We had Pachi, with perturbed neural network weightings, generate its optimal move at a given board state and then had Fuego estimate the value of each move. In this way, our loss function was Fuego's evaluation of Pachi's move. We aimed to train a low iteration (and thus low intelligence) Pachi to match the play of a high iteration (and thus high intelligence) Fuego by using the neural network optimally. We ran SPSA for over 13,000 iterations using games from U-Go [18], a repository of high dan games from the K Go Server. A random board state for each game was loaded, Pachi's neural network was perturbed and the optimal move output. Fuego then determined the values of each move, in this case estimated win percentage, and equation (3.2) was used to update  $\bar{\theta}$  for the next board state. This required significant changes to Pachi's source code as well as parallelization of the SPSA algorithm to compensate for Pachi's slowness generating moves, even at low iterations. We later ran SPSA again, with a change to how Pachi interpreted the perturbed  $\bar{\theta}$ , as the results of the first function were not impressive.

### 3.3 - Testing

After determining the two potential neural network optimization functions, we needed to evaluate just how effective they were. This required running optimized Pachi against standard Pachi, Pachi with static neural network weighting, and Fuego.

Our initial tests involved getting a baseline for how the optimized Pachi compares to default Pachi (no neural network) and Pachi with a static neural network weight of 0.5, that is the case in which Pachi's default move evaluation and neural networks move evaluation are

weighted equally. To run these we had each of the three versions play 50 games against each other with alternating colors. Each version of Pachi was given 10,000 iterations per move due to time and computing resource constraints. One test was run where each version of Pachi was given 25,000 iterations to see if results persisted at a higher iteration count. We also tested how the optimized neural network at 10,000 iterations compared to default Pachi at 25,000 iterations. This would illustrate if the addition of the neural network allowed Pachi to perform equally well at lower iterations.

We also tested the optimized Pachi against Fuego. This required determining the number of iterations at which the programs played at an equal level. We ultimately concluded that default Pachi at 10,000 iterations and Fuego at 12,500 iterations played about evenly. We also found that when Fuego was raised to 50,000 iterations it had a distinct advantage over default Pachi. We then tested how optimized Pachi at 10,000 iterations compared to Fuego at 12,500 iterations and 50,000 iterations. In each situation, Pachi and Fuego played 50 games with alternating colors. This gave us a clear idea as to how the addition of the optimized neural improved Pachi's playing potential against another Go program. These tests ultimately let us determine how effective adding the neural network to Pachi was.

The final set of tests we ran was on how the first function  $\bar{\theta}$  values converged to the optimal solution over time. To do this, we had optimized Pachi play 30 games, alternating colors, against Pachi using intermediate  $\bar{\theta}$  values. These were the  $\bar{\theta}$  values every 2,000 iterations during the SPSA function training. These helped us determine if the values converged to optimality or if more iterations were required.

## 4. Results & Evaluation

### 4.1 Clarifying Last Year's Results

In addition to the results we generated this year, we also looked back at Keller and Perez's work from last year. This was necessary due to the misunderstanding of GoGui's reporting system. While some of the testing data was unavailable to us, there was still a substantial portion to review. The first two methods last year's MQP team attempted, adding a neural network to Pachi's move selection process and helping Pachi choose which nodes to explore with two different neural networks, were the only ones that are relevant for reinterpretation. We began with their first problem, determining how much influence the neural network should have compared to Pachi's default heuristics. Figure 4.1 has been repeated below for convenience.

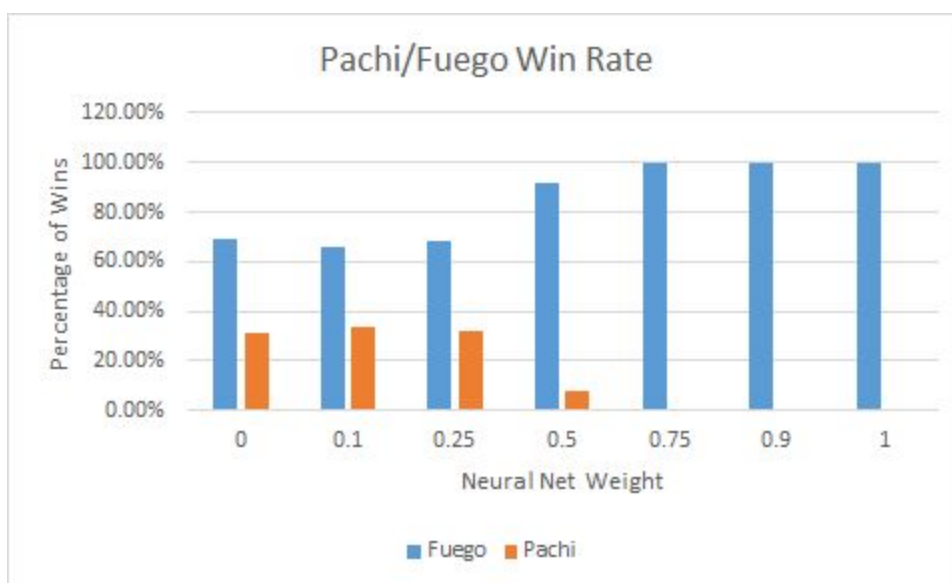


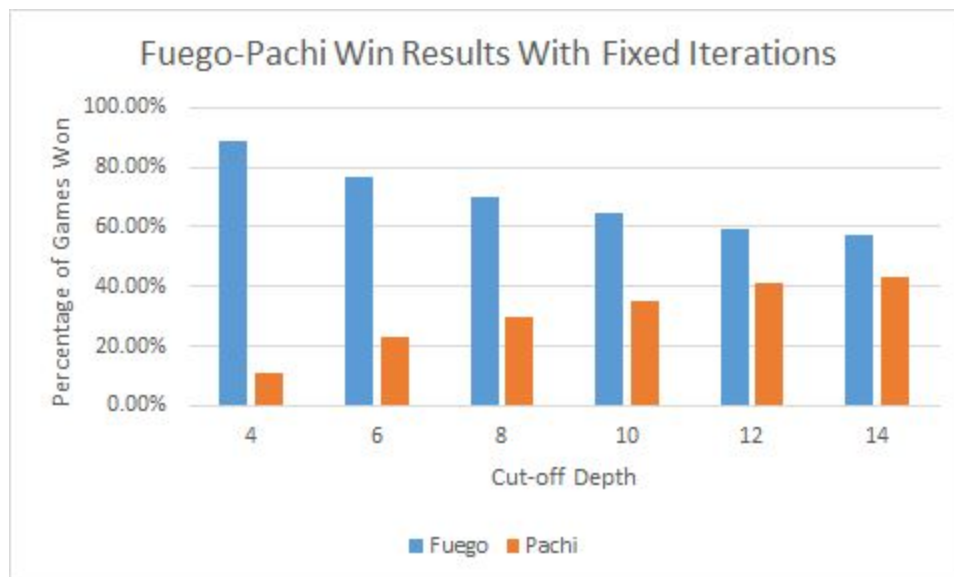
Figure 4.1 Remodeled GoGui Results

Figure 4.1 shows Pachi and Fuego's win rates as the neural network weight for Pachi increased. At each weighting, Pachi and Fuego played 100 games against each other, alternating colors every game. Pachi was run with 27,000 iterations while Fuego was run with 180,000. At the time, this was thought to be an even matchup, but as can be seen, it is not. At 0.0 weighting (default Pachi), Pachi won 31 games while Fuego won 69. At 0.1 weighting, Pachi won 34 games while Fuego won 66. At 0.25 weighting, Pachi won 32 games while Fuego won 68. At 0.5 weighting, Pachi won 8 games while Fuego won 92. At 0.75, 0.9, and 1.0 (only neural network considered) weightings, Pachi won 0 games while Fuego won all 100. In addition, as the neural network influence increased, average game length decreased with Pachi resigning, on average, earlier and earlier each weighting.

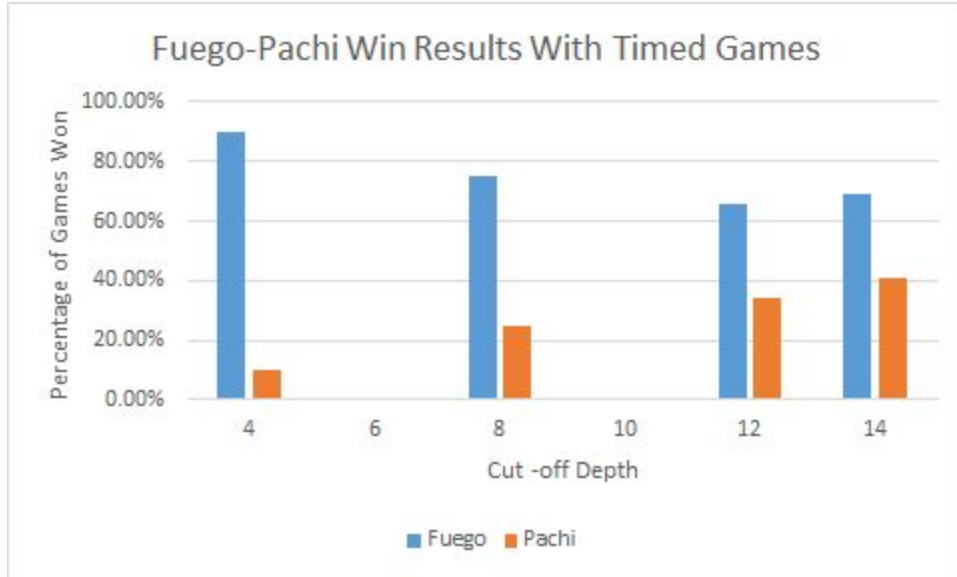
The second problem last year's MQP team tackled was investigating at what search tree depth a faster, less accurate neural network should be used to expand nodes rather than a slower, more accurate neural network. They set cutoff tree depths,  $n$ , for when the slower neural network should stop being used and the faster neural network should start being used instead. For both of these neural networks, they used a weighting of 0.5. They also tested if limiting Pachi's time to make a move versus controlling its iterations made a difference. This was to prevent the slower neural network from taking too long to run and causing Pachi to play too slow. For the fixed iterations test, Pachi was given 27,000 iterations and Fuego was given 180,000 iterations. For timed games, both Pachi and Fuego were given 30 minutes total time to play. For each test, Pachi and Fuego played 100 games against each other, alternating colors

For  $n = 4$ , Pachi won 11 games while Fuego won 89 games with fixed iterations and Pachi won 10 games while Fuego won 90 while timed. For  $n = 6$ , Pachi won 23 games while

Fuego won 77 games with fixed iterations. There was no data for timed games. For  $n = 8$ , Pachi won 30 games while Fuego won 70 games with fixed iterations and Pachi won 25 games while Fuego won 75 while timed. For  $n = 10$ , Pachi won 35 games while Fuego won 65 games with fixed iterations. There was no data for timed games. For  $n = 12$ , Pachi won 41 games while Fuego won 59 games with fixed iterations and Pachi won 34 games while Fuego won 66 while timed. For  $n = 14$ , Pachi won 43 games while Fuego won 57 games with fixed iterations and Pachi won 41 games while Fuego won 69 while timed. Figures 4.2 and 4.3 below are graphical representations of the fixed and timed win rates as depth increases.



**Figure 4.2 Remodeled Depth Results with Fixed Iterations**



**Figure 4.3 Remodeled Depth Results with Timed Games**

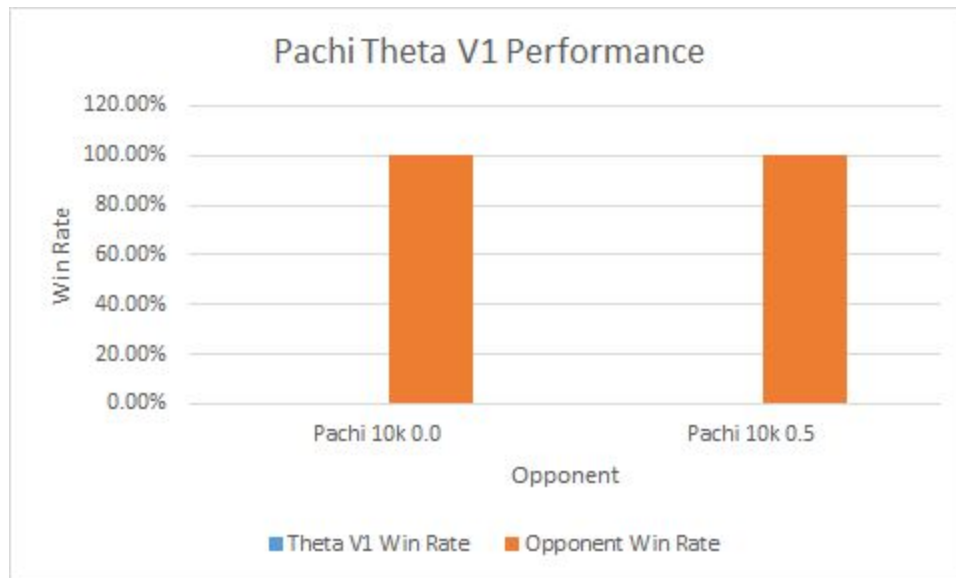
As can be seen, in both cases increasing the cutoff depth significantly improves Pachi’s win rate. As this occurs for both the fixed number of iterations and the fixed time games, it appears that continuing to use the slower, more accurate neural network does not cut far enough into Pachi’s play time to negatively impact it. This implies that it might be worth using the slower neural network all the time rather than just at low depths. It also suggests that the faster neural network might offer no, or minimal, advantage over the slower neural network.

## 4.2 Results from Optimized Neural Network Weight

### 4.2.1 Performance of V1

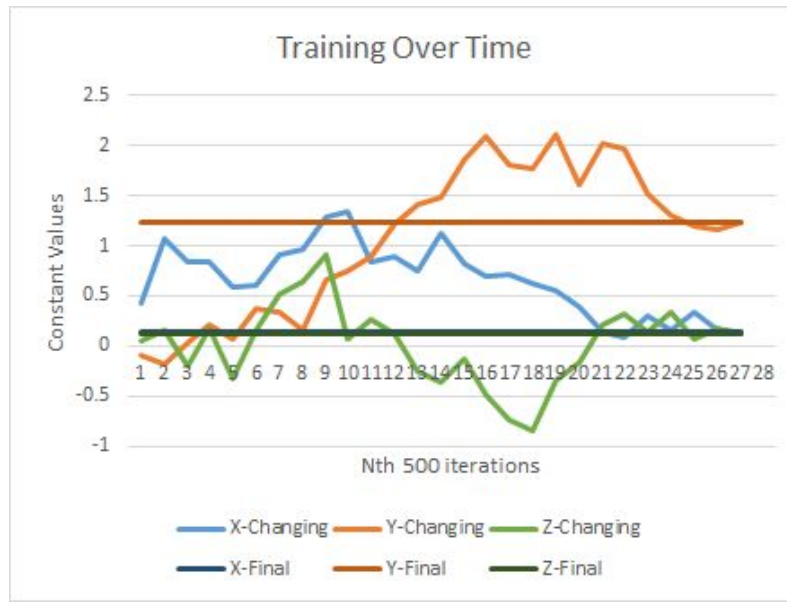
During our first attempt at using SPSA to “train” a neural network weighting function, we ran the testing for over 13,000 iterations. For the sake of clarity, we will call this version of the function (with the final  $\bar{\theta}$  from the SPSA training) **V1**. When V1 played default Pachi (Pachi 0.0) 50 times, each program having 10,000 iterations, V1 lost every game. The same result

occurred when V1 played Pachi with a static neural network weighting of 0.5 with both programs having 10,000 iterations. These results can be seen in Figure 4.4 below.



**Figure 4.4 Performance of Pachi Theta V1**

These results indicated the function was even worse than static weighting. To verify that the intermittent  $\bar{\theta}$  converged to the final  $\bar{\theta}$  (an indication that the SPSA training worked as intended), we plotted the values as can be seen below in Figure 4.5. The eventual convergence to final values indicated that the problem was in our initial function rather than the SPSA training. We created and trained a new function, **V2**, that capped the neural network's influence at 0.25.

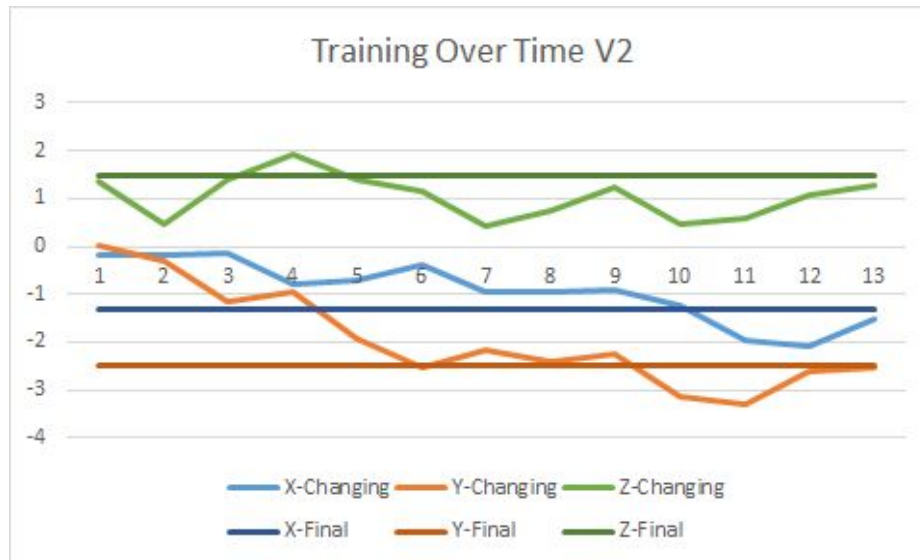


**Figure 4.5 Convergence of Theta V1 values**

#### 4.2.2 Performance of V2

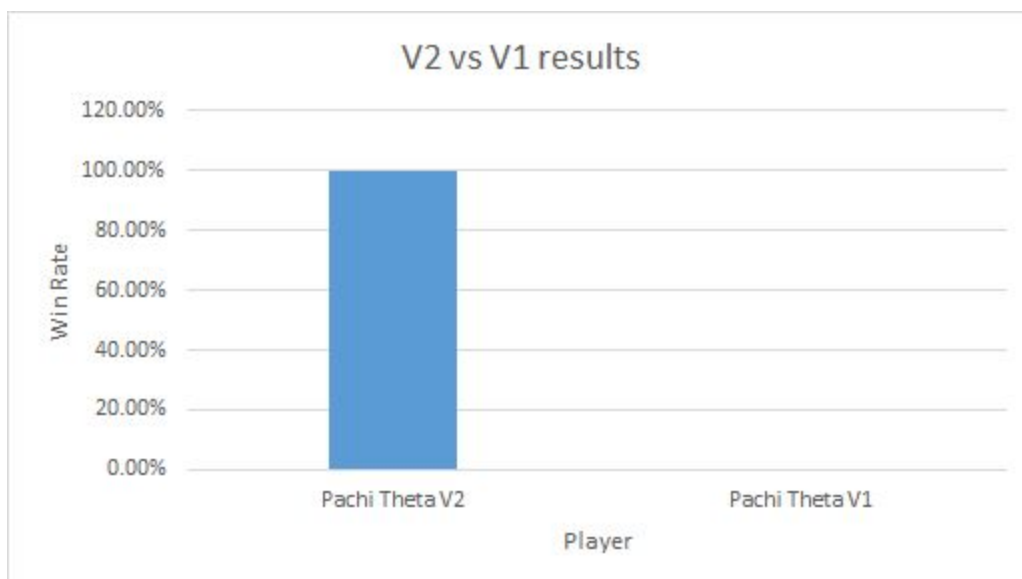
Due to time constraints, we were only able to train the new function over 7,000 iterations. Because of this, before testing the performance of the new values, we wanted to confirm that they had indeed converged. As can be seen in Figure 4.6, the initial values were closer to the final values than in the training of the first set. Consequently, the values in the second set converged more quickly.





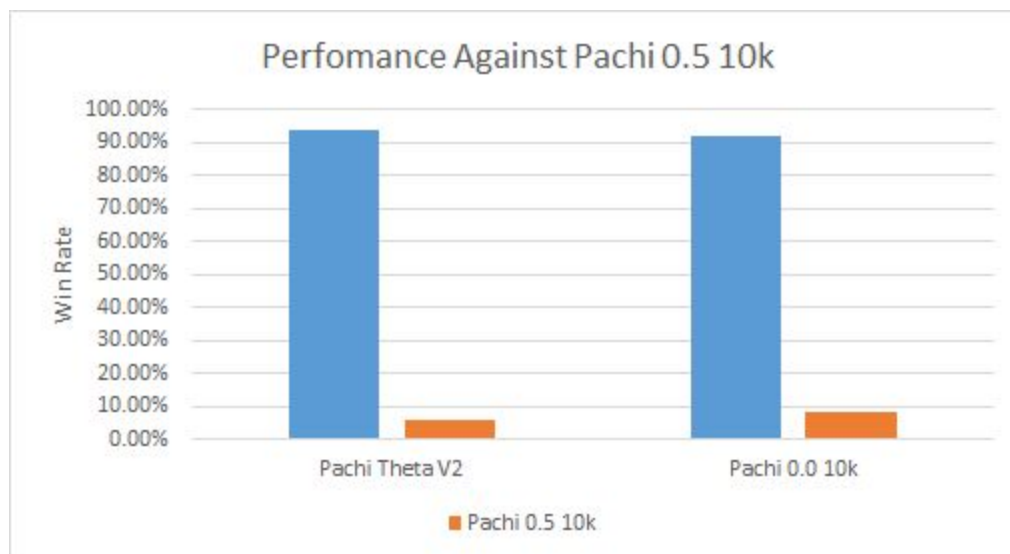
**Figure 4.6 Convergence of Theta V2 values**

After we verified that the  $\bar{\theta}$  converged to the final values, we had V1 and V2 play 50 games against each other, each using 10,000 iterations. V2 won every game, as can be seen in Figure 4.7 below. This was a clear indication that even with limited training, V2 was significantly better than V1 and gave us strong grounds to move forward with testing.



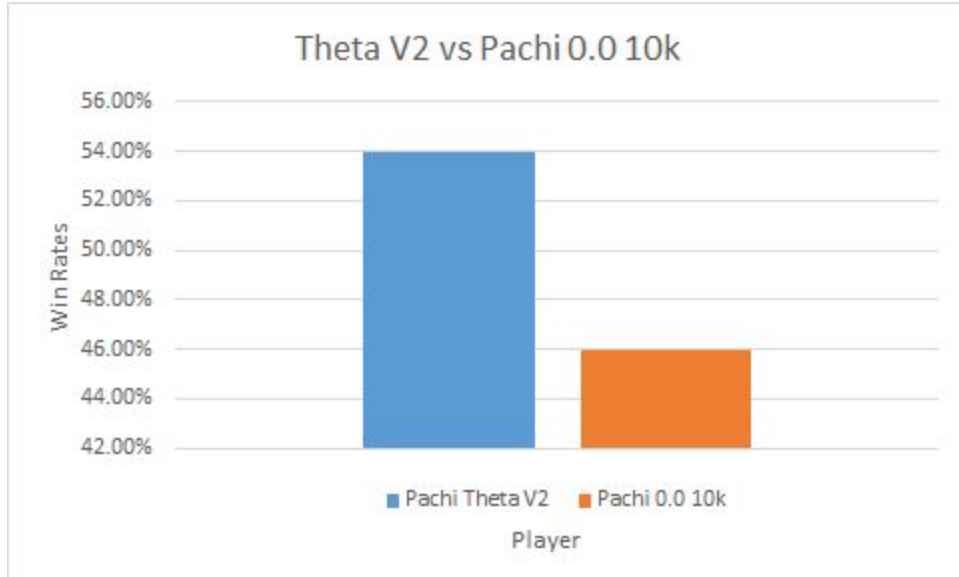
**Figure 4.7 Pachi Theta V2 performance against Pachi Theta V1**

Having confirmed that V2 was a stronger player than V1, we ran the same tests on it that we had performed on V1. We began by testing V2 with 10,000 iterations against Pachi with a neural network weighting of 0.5 and 10,000 iterations. At the same time, we tested default Pachi with 10,000 iterations against Pachi with a neural network weighting of 0.5. Our results, shown in Figure 4.8, demonstrated that V2 significantly outperformed Pachi 0.5, as did default Pachi. An interesting feature of this data was that V2 seemed to do even better against Pachi 0.5 than default Pachi did. This implied that V2 might be comparable in performance to default Pachi.



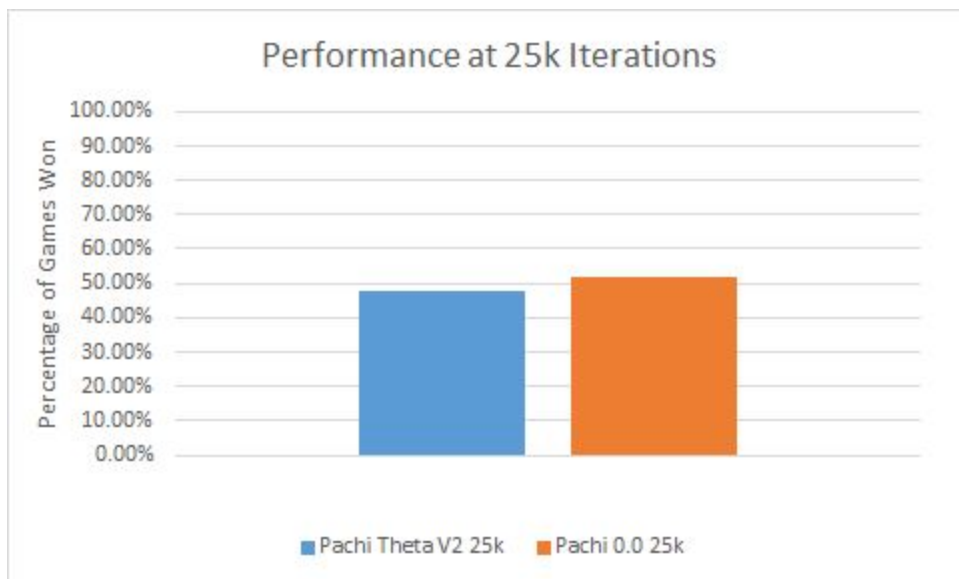
**Figure 4.8 Comparative Testing Against Pachi 0.5 with 10k Iterations**

Following up on this observation, we had V2 and default Pachi play 50 games against each other, again with 10,000 iterations. V2 won 27 games while default Pachi won 23. As such, we concluded that the two were about even in performance. These results can be seen below in Figure 4.9.



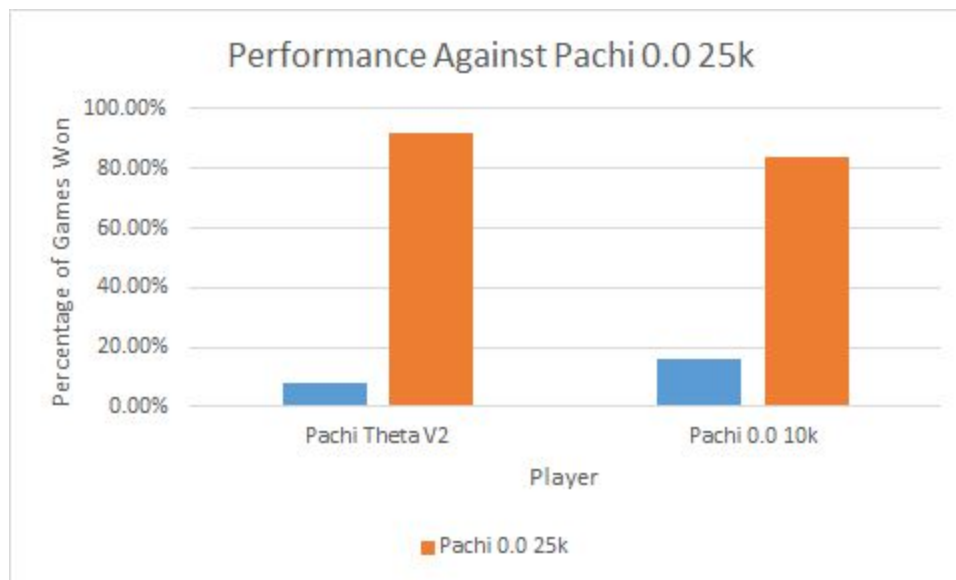
**Figure 4.9 Pachi Theta V2 Performance Against Pachi 0.0 with 10k Iterations**

We wanted to verify that these results persisted when running with a higher number of iterations. In order to do so, we had V2 play 50 games against default Pachi, each with 25,000 iterations. V2 won 24 games while default Pachi won 26. As with 10,000 iterations, the two programs at 25,000 iterations are comparable. This is indicative that the neural network does not detract from Pachi’s performance, even when Pachi is “smarter” (run with more iterations).



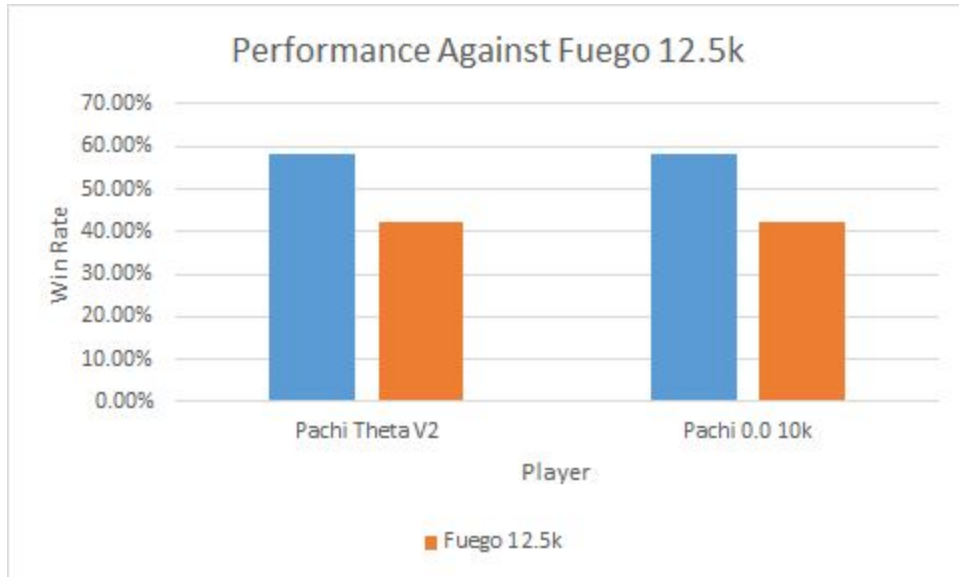
**Figure 4.10 Results of V2 Versus Default Pachi At 25k Iterations**

We then wanted to see how V2 and default Pachi performed against stronger opponents. We had V2 with 10,000 iterations play 50 games against default Pachi with 25,000 iterations. V2 only won 4 games while the stronger Pachi won 46. We also had default Pachi with 10,000 iterations play 50 games against another instance of default Pachi with 25,000 iterations. The 10,000 iteration default Pachi won 8 games while the 25,000 iteration default Pachi won 42. The results can be seen below in Figure 4.11. As can be seen, default Pachi at 10,000 iterations performed twice as well as V2. While this appears to indicate default Pachi is stronger than V2, the difference in games is too small to be conclusive.



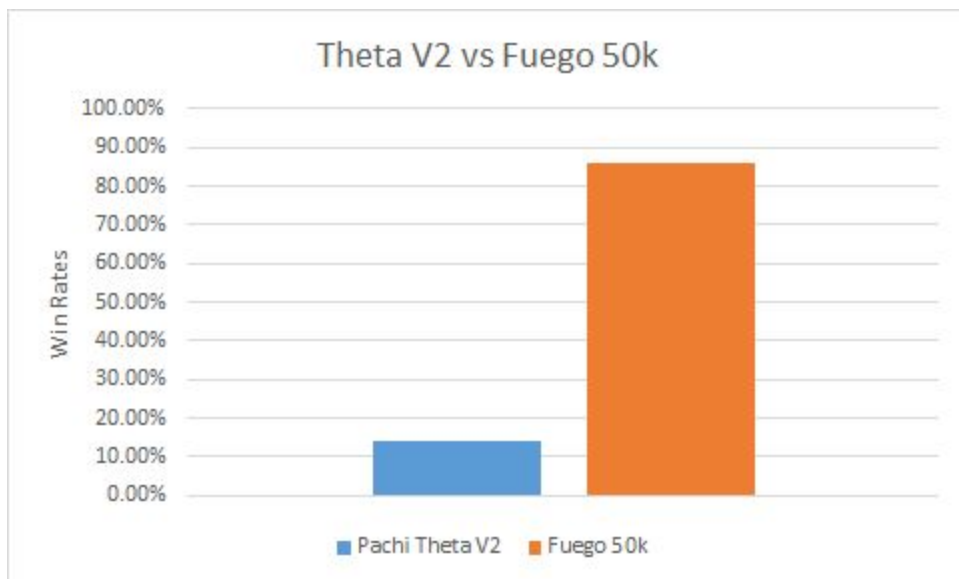
**Figure 4.11 Performance Against Pachi 0.0 with 25k iterations**

After testing V2's performance against Pachi, we wanted to see how it would perform against Fuego at different numbers of iterations. Playing against Fuego with 12,500 iterations, we found that V2 won exactly as many games as default Pachi (29, while Fuego won 21). This can be seen in Figure 4.12.



**Figure 4.12 Comparison of Performance Against Fuego with 12.5k Iterations**

Lastly, continuing our testing of V2's performance against a much stronger opponent, we played it against Fuego with 50,000 iterations. The results are shown in Figure 4.13. We found that V2 won 8 out of 50 games. This was surprising in that Fuego was running at four times as many iterations as in the previous test. This indicated that V2 could, at least some of the time, hold its own against a stronger opponent.



**Figure 4.13 Theta V2 Performance Against Fuego with 50k Iterations**

## 5. Conclusions & Future Work

### 5.1 Conclusions

After reinterpreting last year's work, we found that the neural network seemed to perform best at low levels of influence. Though it functioned well as a move predictor on its own, it seemed to harm performance when integrated with a larger degree of influence.

Reinterpreting last year's MQP results also showed that the slower, more accurate neural network they used in their depth cutoff experiment is fast enough to be used by Pachi. With this slower neural network and a static weighting of 0.5, Pachi and Fuego did comparably well, even when the number of iterations was vastly skewed in Fuego's favor. As such, this slower neural network appears to be very powerful and strongly augments Pachi.

Even with an adaptive weighting function trained via SPSA, the strongest performance we were able to achieve with neural network integration (V2) was comparable to that of default Pachi. The difference between V1 and V2 was that V2 restricted the neural network to a low level of influence. Though this significantly improved V2's performance compared to that of V1, it was still unable to do better than default Pachi. This showed that this neural network was unable to positively affect Pachi's game playing in a significant way.

## 5.2 Future Work

The neural network used in last year's depth adaptive search seemed to significantly outperform that used in our experiments. As such, we believe that a possible means of improving Pachi's performance would be to implement adaptive weighting with the second neural network. Training an adaptive weighting function via SPSA allowed us to significantly improve the performance of Pachi using an inferior neural network to the point that it matched that of default Pachi. Were the same to be done with a more effective neural network, it might be possible to exceed default Pachi.

Additionally, the integrated version of Pachi could likely be improved by adding more parameters to  $\bar{\theta}$ . This would allow more flexibility in the weighting function and thus let it more intelligently change the influence of the neural network based on game state.

Finally, running SPSA training for more iterations or for higher level games could lead to a better function. V2s training was limited to 7,000 iterations on strong amateur games. More training iterations on more professional games could lead to a stronger performance.

# References

- [1] Abramson, B, *Expected-Outcome: A General Model of Static Evaluation* (IEEE Transactions on Pattern Analysis and Machine Intelligence 1990).
- [2] “Artificial Intelligence - Neural Networks”, accessed March 24, 2017, [https://www.tutorialspoint.com/artificial\\_intelligence/artificial\\_intelligence\\_neural\\_networks.htm](https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_neural_networks.htm)
- [3] “Artificial Neural Networks/Print Version”, last reviewed March 14, 2013, [https://en.wikibooks.org/wiki/Artificial\\_Neural\\_Networks/Print\\_Version](https://en.wikibooks.org/wiki/Artificial_Neural_Networks/Print_Version)
- [4] Deshpande, Adit, “A Beginner’s Guide to Understanding Convolutional Neural Networks”, accessed March 26, 2017, <https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/>
- [5] Engelking, Carl, “Artificial Intelligence Just Mastered Go, But One Game Still Gives AI Trouble”, last modified January 27, 2016, <http://blogs.discovermagazine.com/crux/2016/01/27/artificial-intelligence-go-game/#.WN4GbW-GPIV>
- [6] Enzenberger, Markus, *The Integration of A Priori Knowledge into a Go Playing Neural Network*, September 1996, accessed April 6, 2017, <http://www.cgl.ucsf.edu/go/Programs/neurogo-html/neurogo.html>
- [7] Fuego, last accessed April 14th, 2017, <http://fuego.sourceforge.net/>
- [8] Gelly, Sylvain, *The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions* (EPC TAO, INRIA Saclay & LRI Bât), Accessed March 30, 2017, [http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Applications\\_files/grand-challenge.pdf](http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Applications_files/grand-challenge.pdf)
- [9] Keller, Joshua; Perez, Oscar, *Improving MCTS and Neural Network Communication in Computer Go*, accessed March 1, 2017, [http://web.wpi.edu/Pubs/E-project/Available/E-project-042616-172920/unrestricted/MQP\\_Budapest2016\\_Report.pdf](http://web.wpi.edu/Pubs/E-project/Available/E-project-042616-172920/unrestricted/MQP_Budapest2016_Report.pdf)
- [10] Kocsis, Levente; Szepescari, Csaba, *Bandit Based Monte Carlo Planning* (Computer and Automation Research Institute of the Hungarian Academy of Sciences), Accessed March 31, 2017, <http://ggp.stanford.edu/readings/uct.pdf>



- [11] Lawler, Richard, *Google DeepMind AI wins final Go match for 4-1 series win*, last modified March 14, 2016, <https://www.engadget.com/2016/03/14/the-final-lee-sedol-vs-alphago-match-is-about-to-start/>
- [12] Maddison, Chris J et al., *Move Evaluation in Go Using Deep Convolutional Neural Networks* (ICLR 2015), Accessed March 29, 2017, <https://arxiv.org/pdf/1412.6564.pdf>
- [13] Metz, Cade, “In Two Moves, AlphaGo and Lee Sedol Redefined the Future”, last modified March 16, 2016, <https://www.wired.com/2016/03/two-moves-alphago-lee-sedol-redefined-future/>
- [14] Pachi, last accessed April 16th, 2017, <https://github.com/pasky/pachi>
- [15] “Scoring this 9x9 board”, last modified september 2015, <https://forums.online-go.com/t/scoring-this-9x9-board/5165>
- [16] Schraudolph, Nicol, et al., *Temporal Difference Learning of Position Evaluation in the Game of Go*, last accessed April 5, 2017, <http://www.gatsby.ucl.ac.uk/~dayan/papers/sds94.pdf>
- [17] Silver, David; Huang, Aja et al. *Mastering the Game of Go with Deep Neural Networks and Tree Search* (Google Deepmind, 2016), last accessed March 28, 2017, <https://gogameguru.com/i/2016/03/deepmind-mastering-go.pdf>
- [18] Spall, James C., *Implementation of the Simultaneous Perturbation Algorithm for Stochastic Optimization*, last accessed April 8, 2017, [http://www.jhuapl.edu/SPSA/PDF-SPSA/Spall\\_Implementation\\_of\\_the\\_Simultaneous.PDF](http://www.jhuapl.edu/SPSA/PDF-SPSA/Spall_Implementation_of_the_Simultaneous.PDF)
- [19] Spall, James C., *An Overview of the Simultaneous Perturbation Method for Efficient Optimization*, last accessed April 4, 2017, <https://pdfs.semanticscholar.org/bf67/0fb6b1bd319938c6a879570fa744cf36b240.pdf>
- [20] U-Go, *Game Records*, last accessed April 8th, 2017, <https://www.u-go.net/gamerecords/>
- [21] US Go, *A Brief History of Go*, last accessed April 2, 2017, <http://www.usgo.org/brief-history-go>