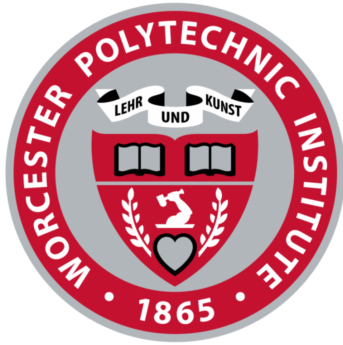


Deep Reinforcement Learning for GO AI

A Master Qualifying Project submitted to the Faculty of Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree of Bachelor of Science.

Submitted by:
Esteban Aranda
Thomas Graham

Project Advisor:
Professor Xiangnan Kong
Department of Computer Science

Project Co-Advisor:
Professor Yanhua Li
Department of Data Science

Date:
May 6, 2021

This report represents the work of one or more WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on the web without editorial or peer review.

Abstract

Go is considered to be the most challenging classical game for artificial intelligence. Following the structure of Google's AlphaGo which defeated the world's best Go player in 2016, this project aimed to develop an intelligent player that could achieve human-level of play for TicTacToe, Othello, and most importantly Go. Using Python and the machine learning framework PyTorch, the team developed two tree search algorithms, Minimax and Monte Carlo Tree Search, and two neural networks with the goal of combining them into a single intelligent agent. Testing was only performed for TicTacToe and without combining algorithms. Despite the neural networks not achieving the same performance as the tree search algorithms, their improvement over time showcase the potential of machine learning, even with a limited amount of data and computational power for training.

Executive Summary

The purpose of this report is to describe our project in which we built a mathematical model that was designed to defeat stronger opponents in each of the games Tic-Tac-Toe, Othello and Go. This was a year long process but it can be divided into different high level objectives that we followed throughout the project.

The different objectives of this project were: designing the model, developing the model, testing the model, and finally documenting our research and results. This is the basic outline of our process as there were many different smaller phases.

Designing the model was done by researching the elements of Machine Learning, Artificial Intelligence, and Deep Learning. To be more specific, the language choice was Python, and the main library that was used was PyTorch. PyTorch is an extension of the Torch library in Python for use with Machine Learning. The design was for us to start basic and use machine learning then eventually be able to use Deep Learning. Then our plan was to first use Supervised Learning then eventually getting into Reinforcement Learning. This was the end goal as this lets the model train on the environment using human interaction instead of the model requiring input to train.

Developing the model was the next objective. The code was developed over the year starting with the Game classes themselves. These being Tic-Tac-Toe, Othello and the Go game. The games needed to be edited to save data from each iteration in a Python file called Memory.py. This is because it reduced overhead by allowing the computer to load the previous file instead of rerun the iterations every time. Then when we had that working we moved on to the Player classes, they were developed in order to properly simulate each different type of player. The different types were Random Player, Monte Carlo Tree Search Player, and MiniMax Player. After getting those running properly we moved on to the neural net portion using a Value Network Player, and a Policy Network Player.

As each component of the project was developed a supporting unit test file was written as well. Testing the code was invaluable because we were able to see where the model had failed, and where the model had succeeded. This information gave us more data about how the code was working at the time.

In addition, to develop code successfully it was important to properly create a code base that was used for the project. This is a professional practice that allows code to be saved and then compared to previous uploads in case of error, or in case of system failure. Then finally the last objective was to document all of our findings by writing this report that contains our methodology, design process, results, future work and final conclusion.

Contents

Abstract 2

Executive Summary 3

Contents 4

Authorship 6

List of Figures 7

List of Formulas 8

1. Introduction 9

2. Background 9

2.1 Go

2.1.1 History

2.1.2 Rules

2.2 History of AI in Go

2.2.1 AlphaGo

3. Methodology 11

3.1 Research Methods

3.2 Testing

4. Software Development Environment 13

4.1 Programming Environment

4.2 Project Management Software

4.3 Software Requirements

4.3.1 Python

4.3.2 Pygame

4.3.3 PyTorch

4.3.4 NumPy

4.3.5 Pathlib

4.3.6 Nose Tests

5. Design **15**

5.1 Minimax

5.2 Monte Carlo Tree Search

5.3 Policy Network

5.3.1 Network Structure

5.3.2 Supervised Learning

5.3.3 Reinforcement Learning

5.4 Value Network

5.4.1 Network Structure

5.4.2 Supervised Learning

6. Experimental Results **22**

7. Future Work **26**

8. Conclusion **26**

References **28**

Appendix A: Steps of Monte Carlo Tree Search **30**

Authorship

Section	Author
Abstract	Esteban Aranda
Executive Summary	Thomas Graham
1. Introduction	Thomas Graham
2. Background	Thomas Graham
3. Methodology	Thomas Graham
4. Software Development Environment	Esteban Aranda
5. Design	Esteban Aranda
6. Experimental Results	Esteban Aranda/Thomas Graham
7. Future Work	Esteban Aranda
8. Conclusion	Esteban Aranda

List of Figures

Figure 1: AlphaGo Zero using the Monte-Carlo Tree Search **11**

Figure 2: Testing layout **12**

Figure 3: Get_valid_moves test case **13**

Figure 4: Minimax tree example **16**

Figure 5: Policy network structure **18**

Figure 6: Rectified Linear Activation Function **21**

Figure 7: Win, loss, and tie rates vs RandomPlayer for TicTacToe **23**

Figure 8: Supervised learning loss for PolicyNN **23**

Figure 9: Supervised learning loss for ValueNN **24**

Figure 10: Loss % of PolicyNN Models vs RandomPlayer **25**

Figure 11: Loss % of PolicyNN Model99 vs Models 0-98 **25**

List of Formulas

Formula 1: Upper Confidence Bound **17**

Formula 2: PyTorch's Conv2d class **18**

Formula 3: PyTorch's Linear class **19**

Formula 4: Softmax function **19**

Formula 5: Policy network's loss function **20**

Formula 6: Discounted expected reward **20**

Formula 7: Mean Squared Error **22**

Introduction

Artificial Intelligence is a rapidly growing area of Computer Science. It aims to make everyday tasks, while using computers, more automated. This is done by implementing certain tools and mathematical techniques throughout the code that help improve the algorithms accuracy and automate the datasets.

One branch of Artificial Intelligence is known as Machine Learning. According to (IBM, 2021), Machine Learning is defined as the study of computer algorithms which learn from themselves over time. All of these programs use mathematical models to be able to predict the outcomes of certain events by learning from the past experiences.

The other branch of Artificial Intelligence is Deep Learning. According to (IBM, 2021), Deep learning is defined as being made up of multiple neural networks with three or more layers each. The difference between Deep Learning and Machine Learning is Deep Learning is much more automated requiring less human interaction throughout the entire process. Supervised Learning is when the algorithm is also able to use labeled datasets which are generally much larger. Supervised Learning can be broken down into two features Regression and Classification.

In comparison, then there is Reinforced Learning which works with the environment instead of the dataset. Reinforced Learning can be broken down into exploitation or exploration. It also includes Policy Networks, and Value Networks. By combining all of these important areas of Computer Science we were able to train the model in order to make better predictions and, in turn, provide us with a better result.

Background

2.1 Go

The game of Go was invented more than 2,000 years ago in China and is viewed as the hardest classical game for artificial intelligence. There are two players Black and White, and they play each turn by placing their colored stone on one of the available intersections on the board. The game starts with an empty board of either 19x19, 13x13, or 9x9, and Black goes first. The most used board size is 19x19, which makes it an enormous search space coupled with the difficulty of accurately evaluating moves. This makes it extremely difficult for artificial intelligence to evaluate. According to (J. X. Chen, 2016) Go is increasingly difficult because the number of options at any given time is 361!, which means there are 46,655,640 possible situations.

2.1.1 History

The Go game originated in China and there it was known as weiqi, which means encirclement board game. According to (Go (game) - Wikipedia, 2021), Go was first played on a 17x17 board eventually switching to 19x19 during the Tang Dynasty from 618-907. In addition, it is theorized that the actual origin of the game comes directly from the Chinese

emperor Yao who wanted it designed for his son, Danzhu. After that it was brought to Korea between the 5th and the 7th centuries CE. The name in Korea is baduk. Finally the game reached Japan after the 7th century CE and eventually was common in the public by the 13th century.

2.1.2 Rules

The rules of Go can be quite complicated. In addition, each player has the opportunity to pass at any point in the game. Also once the player places the stone on the board it must permanently stay there unless it is captured. The official rules can be broken down into four sections, the Ko Rule, the Suicide rule, the Komi rule, and the scoring rules. The Ko Rule prevents repetition by not allowing either player to make a move that will result in the game being brought back to a previous position. The Suicide Rule is designed to prevent players from placing a stone that is not surrounded by at least one liberty around the group of stones of their color. The Komi Rule is designed to give the White player an advantage to match the fact that the Black player goes first, so Japanese play the advantage is 6.5-points, which if not in Japanese play, then the handicap is only 0.5-points. Finally, the scoring in Japanese and Koren play is by "Territory scoring" and for Chinese play "Area scoring" is used. The area scoring is the total number of stones on the board plus the intersections that are empty. The territory scoring is where the remaining stones are counted plus the number of captured "prisoners" from each player.

2.2 History of AI in Go

Before 2015 the Go AI was only able to beat players ranked Advanced Amateur, which is a good ranking but not the best players in the world, which is the rank of Professional Player, or Professional dan.

2.2.1 AlphaGo

According to (Go (game) - Wikipedia, 2021), in 2015 Google DeepMind's AI AlphaGo beat the European Go champion on a 19x19 board. This was done by implementing deep learning and by letting the AI train on itself over and over again the AI was able to learn which plays were better at each time. Then in 2016 AlphaGo beat Lee Sedoal, who was considered the best Go player in the world, four out of five times.

According to (Lapan. M, 2018) in 2017 AlphaGo beat Ke Jie, who was the world ranking Go player two years in a row, and later that year Google's DeepMind released a stronger version of the current AI called AlphaGoZero which beat the old AI 100 to 0. Lapan states the AI did this without any prior knowledge except for the game rules, and that AlphaGo Zero is all about learning from self-play and then improving its policy. He also states that the game of chess can be beat by AlphaGo Zero as well.

The way that AlphaGo Zero works is similar to our approach for this project. It uses a Monte-Carlo Tree Search algorithm to randomly sample "the most promising path." (Lapan, M, 2018). The model is training versus itself then it will go on to play the "best version" of itself

over and over again determining a “new best version” every time.(Lapan, 2018). The image below represents how the MCTS will work on the game of Tic-Tac-Toe.

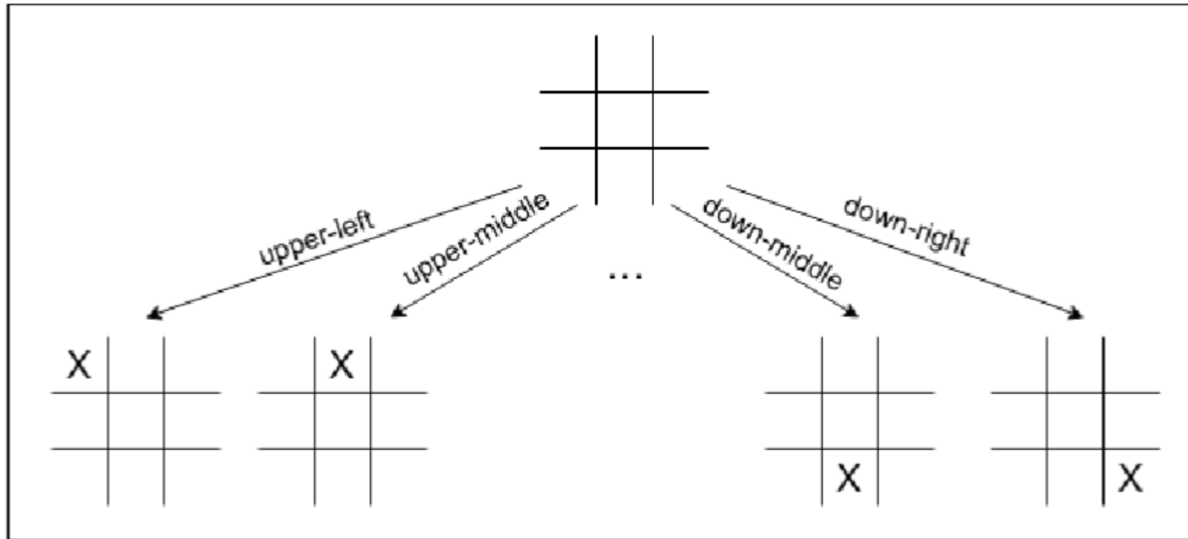


Figure 1: The Monte-Carlo Tree Search used in Tic Tac Toe from AlphaGo Zero. (Lapan, 2018)

Methodology

The problem addressed in the project was attempting to train a computer programmed model to learn from itself over iterations. The goal was to eventually be able to outperform the other AI. The code was written in Python. The main import for the project was PyTorch, which is a machine learning library based on the previous Torch library. The way the problem was addressed was by testing the model using different game types. To do this, code was written in batches starting with the easiest game to the hardest. The easiest game was Tic Tac Toe, then Othello, and then game of Go. Over the course of the year we updated the code base weekly and took notes on where we were heading next in order to properly manage our project.

The process was to begin by developing the games that would be used. The games were designed to take arguments stating which Player would be playing and the type of algorithm used. The algorithms used were MiniMax and Monte Carlo Tree Search. These are two popular algorithms in Machine Learning that helped us to perform this task. The next step was to test the games using unit testing in Python. Test cases were written for every function in the code base in order to achieve the best results.

Then the “Players” Classes were developed, these would be the “Players” that would play against the AI player. These “Players” that were developed were a Random Player, a MiniMax Player, and a Monte Carlo Player. Then it was needed to test the players in games versus each other to see the results of which algorithm was performing better in each game. The results needed to be saved somewhere so a Memory class was developed in order to do just this. This helped because each time the game could be loaded and saved instead of having to run through all the iterations each time.

Next the neural net was developed in order to increase the performance of the Players. This was crucial as the neural net is the main component of the project. The Policy Neural Net was developed which was used to predict the probabilities associated with each move and associate it with the possible rewards. Then the Value Network was developed which will predict the reward associated with a move directly. This Value Network would automatically choose the move with the highest reward connected to it.

All of these networks had their own set of unit tests that would prevent errors in the code base. In addition, all of these functions and the project in general was hosted on GitHub for source control in order to keep track of the code.

3.1 Research Methods

The goal was to produce a working AI model that can compete with another player, and learn based on the previous games results. Through combined efforts of our team and our advising we were able to constantly research the topic, and its background. This helped us to write our code, improve note taking, learn through visual aids, and collaborate each week to discuss ideas.

3.2 Testing

The development process was organized by version control, testing, deploying and maintaining the code by using GitHub. The testing step was extremely important because this allowed us to write test cases in Python and run them on the code from each class. As seen below, we created a folder to store all the test cases, and separated them by which file they were exclusively testing.

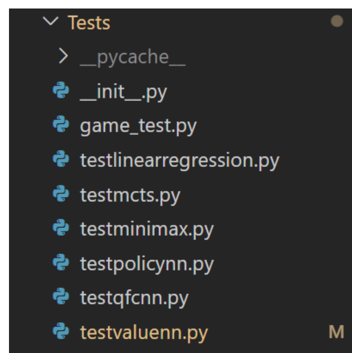


Figure 2: Testing layout

The testing phase was extensive as we wrote a test case for every function in the code base to be sure that everything was working properly at all times. There were tests written for save, load, train, choose a move, select file, and save model in most of the classes. We created the test cases to check all the conditions and parameters of the code base. This was done by creating a fake dataset and training the model on this part of the code, and then testing the equality as to what was expected versus what the actual output was.

```python
#-----------------------------------------------------------------------
def test_get_valid_moves():
    '''get_valid_moves()'''
    g = TicTacToe()  # game
    b=np.array([[  1 , 0 ,-1 ],
                [ -1 , 1 , 0 ],
                [  1 , 0 ,-1 ]])
    s= GameState(b,x=1)
    m=g.get_valid_moves(s)
    assert type(m)==list
    assert len(m)==3
    for i in m:
        assert i== (0,1) or i== (1,2) or i == (2,1)
    assert m[0]!=m[1] and m[1]!=m[2]

    s= GameState(np.zeros((3,3)),x=1)
    m=g.get_valid_moves(s)
    assert len(m)==9
    b=np.array([[  1 , 0 ,-1 ],
                [  0 , 0 , 0 ],
                [  1 , 0 ,-1 ]])
    s= GameState(b,x=1)
    m=g.get_valid_moves(s)
    assert len(m)==5
    for i in m:
        assert i== (0,1) or i== (1,0) or i== (1,1) or i== (1,2) or i== (2,1)
#-----------------------------------------------------------------------
```

Figure 3: Get_valid_moves test case

It was very important for these test cases to pass. As you can see pictured above, if that test case failed then we knew the base function Get Valid Moves was not working. This provides us with information in how to move on with debugging the issue. This made it easier for us over time to not lose track of the current issue as everything was tested modularly. This just means that we tested everything in small portions as this code base ended up with many lines of code, and later on in the development it is hard to find a software bug in previous code if you do not test this way.

Software Development Environment

4.1 Programming Environment

Visual Studio Code (VSCode) was utilized by the team as the IDE of choice. This was a matter of personal preference given the member's previous experience with the editor. In addition, it is worth noting that VSCode's flexibility and support for different languages and types of files made it a strong choice for the project given that the team had to work with .py, .csv, .p, and .pt files. Lastly, the vast number of extensions available make the editor user friendly and allow it to cater to a project's specific needs.

4.2 Project Management Software

The team used GitHub for organizing and maintaining the codebase. The GitHub repository (https://github.com/tgrahamcodes/GoAI) ensured source control and consistency among the member's code and allowed for parallel development. As well as providing tools for

version control, GitHub has a feature where users can create Issues to track a project's development history and work to be done. The team utilized this feature to keep track of tasks that needed to be completed and to track the project's progress. In addition to GitHub, the team used Google Drive to share all documents relating to the background research and to document all meetings with the advisors.

4.3 Software Requirements

4.3.1 Python

As previously mentioned, the project was developed in Python. The program requires Python 3.6 or higher to be run successfully. Once the correct Python version is set, a number of Python libraries must be downloaded and installed.

4.3.2 Pygame

Pygame is a set of modules that facilitate game development in Python, allowing its users to create multimedia games with ease. Version 1.9.6 was used in this project to build the visual interfaces for TicTacToe, Othello, and Go. These interfaces work both for humans to play the game and for the intelligent agents to play on their own. Pygame is required if a user wants to run the visual interfaces. However, it is not required for the use of any of the intelligent players.

4.3.3 PyTorch

PyTorch (version 1.7.0) is an open source Python framework that allows its users to create production ready and optimized machine learning projects. This framework was chosen due to its ease-of-use, flexibility, and comprehensive documentation and robust ecosystem. PyTorch was used by the team to build the policy and value networks, and subsequently train them. It is required to run the two agents that utilize neural networks.

4.3.4 NumPy

NumPy is an open source Python library that enables creating n-dimensional arrays and provides comprehensive numerical computing functions optimized for this type of arrays. Vectorization, indexing, and broadcasting make it fast and versatile. All of these virtues make it much more efficient to use compared to built-in Python arrays. NumPy version 1.17.1 was utilized throughout the entire project, both to build the core logic of the games and in tandem with PyTorch to facilitate the conversion between game states to Tensors and back.

4.3.5 Pathlib

This module for Python 3 offers the use of an Object Oriented programing structure to list and browse all files and directories with ease. Given the complex file structure of the project, Pathlib was essential to the project because it enabled importing elements from files in one directory to another. This library was chosen since the team was working across multiple

operating systems and with different environment variables, and Pathlib maintained all paths consistent and working across all programming environments.

4.3.6 Nose Tests

Nose Tests is a library that enables unit-testing, automatically collecting any test that follows its guidelines and allowing multiple tests to run at once. Nose was used to implement unit-testing for classes and functions to ensure that they were working as intended, following the team's test-driven development approach. The library is not required to run any core aspect of the project but is needed for testing.

Design

The development of the project can be broken down into distinctive phases. All phases followed the same test-driven development described in the Methodology section. The team first started by working on algorithms that work optimally on smaller games spaces such as TicTacToe and eventually progressing to neural networks that are more efficient for larger games such as Go.

5.1 Minimax

Minimax is an algorithm that attempts to minimize loss for a worst case scenario. For zero sum games such as the ones studied in this project, minimax is equal to the Nash equilibrium where each player's strategy is to maximize their payoff whatever the other player's strategy might be. The assumptions that minimax makes are that players take alternate moves, as opposed to simultaneous moves, and that players are going to choose the optimal decision at every decision. As shown in Figure 4, the algorithm works by expanding a game's entire search space, exploring all possible moves at every step. Once the entire tree has been expanded, all leaf nodes (possible states where the game is over) are evaluated and given a score. -1 if the opponent wins, 0 if the game ends in a tie, and 1 if the player wins. These scores are then back-propagated up the tree along their respective paths until they reach a node with branching paths. If that node represents an opponent's move then the algorithm will minimize the score (equivalent to maximizing the score from the opponent's perspective). On the other hand, if the node represents a player's move then the algorithm will maximize the score. The nodes associated with the scores that are back-propagated represent the moves to be chosen given the current path. In other words, the algorithm computes the optimal move for any possible state, even if the players do not take all optimal decisions.

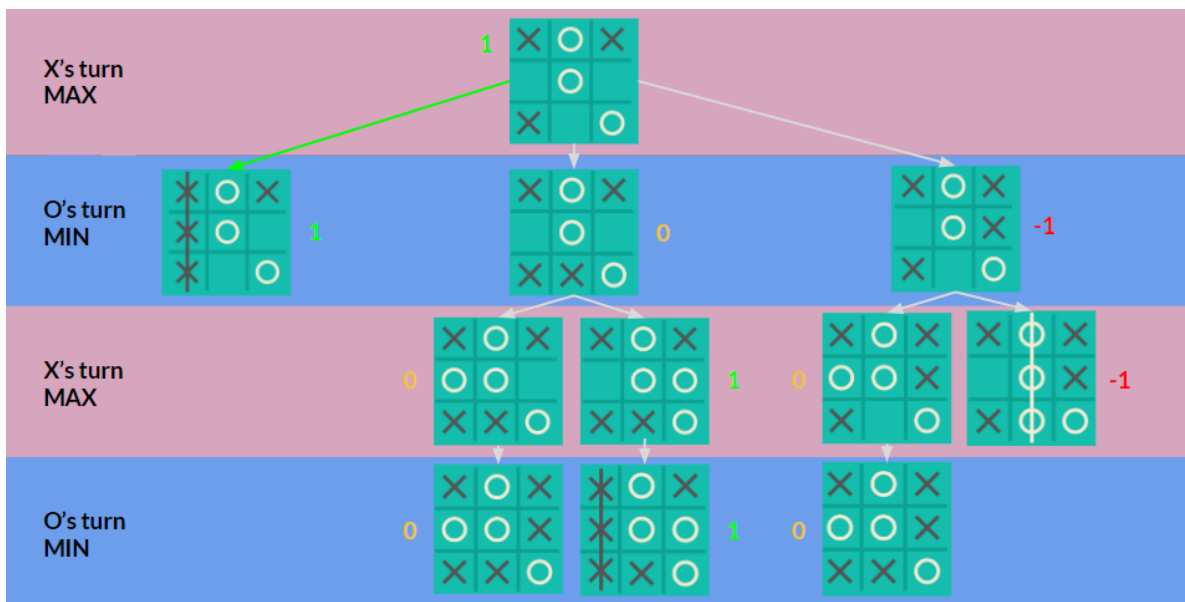


Figure 4: Minimax tree example

5.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm which utilizes randomness to solve deterministic problems. First introduced in 2006 by Rémi Coulom, MCTS is a combination of minimax with an expected-outcome model. This gives the algorithm the properties of being accurate and efficient for problems where brute-force approaches are unfeasible. The focus of the algorithm is to gradually expand the search tree based by randomly sampling moves and analyze the most promising options. MCTS can be broken down into four main steps: selection, expansion, simulation, and backpropagation (Appendix A).

- Selection: Starting from the root node (the current game state), child nodes are chosen until a leaf node L is reached.
- Expansion: Unless L is an end node, the node is expanded by creating its children nodes (valid moves from L) and choosing one C.
- Simulation: Starting from C, the game is played at random until the end of the game.
- Backpropagation: The result of the simulated playout is used to update the information in the nodes on the path from C to the root node. Each node saves the number of times it has been selected and a count of all simulated wins on that path.

This four-step process is repeated multiple times, for this project 100 simulations were performed before each move. The information stored in the nodes is then used to calculate an expected outcome, a ratio between the accumulated wins over the total number of playouts. Lastly, the move with the highest number of simulations is chosen.

The team implemented MCTS with memory by saving all simulations to a file. Therefore, for new matches the algorithm not only had 100 simulations to make a decision but had access to all previous simulations from previous matches. This results in more optimal

decision-making as due to the law of large numbers the ratio of nodes more accurately reflected their true values.

One difficulty with implementing MCTS is balancing exploration vs exploitation when having to select a child node. Though there are many formulas for balancing these two factors, the most common one and the one chosen for this implementation is called UCB (Upper Confidence Bound). The formula (Formula 1) is made up of two components, the first of which is the aforementioned win ratio, which corresponds to exploitation. The latter is for exploration, where *c* is the exploration parameter ($\sqrt{2}$ in this implementation), *N* stands for the total number of simulations, and *n* represents the number of simulations for that node. However, if for a given node there have been no simulations yet, the second term is replaced by positive infinity instead of dividing by 0. Hence, a larger UCB score translates into a child node having a better average pay-off, or that the child node needs exploring.

$$\mathrm{UCB} = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

Formula 1: Upper Confidence Bound

5.3 Policy Network

Minimax and, to a lesser extent, MCTS are only efficient for games with smaller search spaces (Pearl, 1980). However, these algorithms alone are not capable of achieving a high level of play in a reasonable amount of time under Go's incredibly large search space. That is why neural networks are key to successfully develop a Go agent. Neural networks can be described as a series of algorithms that can recognize underlying relationships in data. They are composed of artificial "neurons" that mimic the way human brains function, and through the use of linear algebra can create complex models to fit data.

5.3.1 Network Structure

The policy network's (PolicyNN) function was to output the probabilities of each available move. These probabilities were linked to the expected reward for the corresponding moves. Higher probabilities were associated with winning in the shortest amount of moves possible, and if winning was not an option, tying or losing in the longest amount of moves possible. The network's architecture was as follows: a 2-dimensional convolutional layer, a layer where features were flattened to 1-dimensional vectors, a fully-connected linear layer, and a layer to adjust logits before outputting the results.

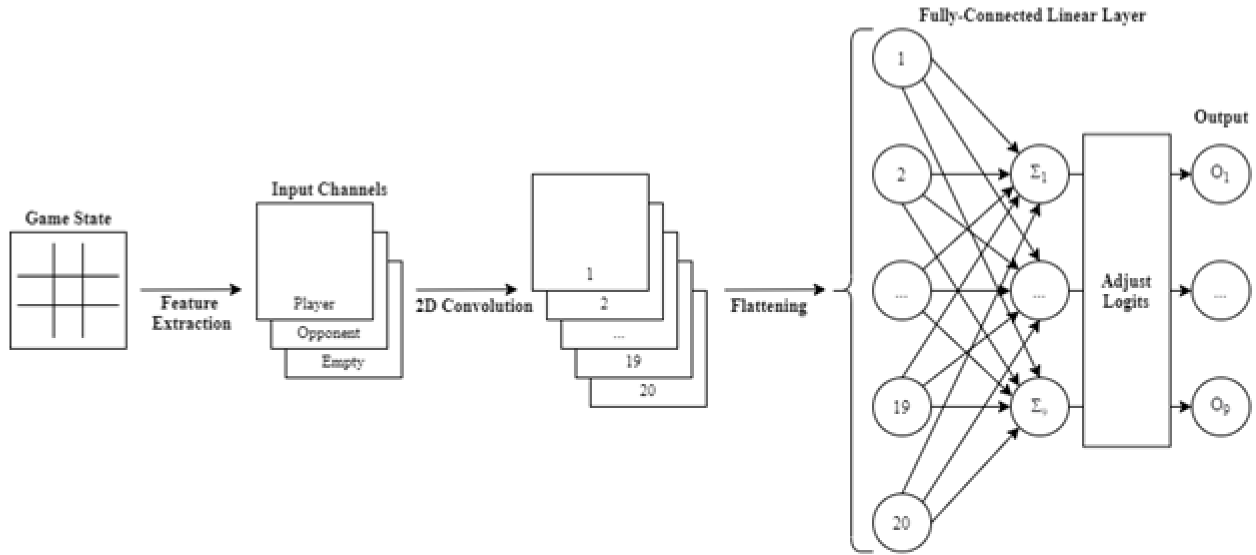


Figure 5: Policy network structure

The network took as input a single game state and would output a 1-dimensional vector of length equal to the board size. For example, for TicTacToe this vector would be of length 9. However, the input game state was decomposed into several channels. The number of input channels varied depending on the type of game the network would work on. For TicTacToe, the simplest game, game states were broken down into a player channel, opponent channel, and empty channel. The first represented all locations where the player had pieces, the opponent channel represented the location of all opponent pieces, and the latter all game spaces that were still available for use. On the other hand, for Othello and Go the number of channels increased to 4. In the case of Othello, this fourth channel defined all legal moves (given that for this game not all blank spaces are always valid moves). As for Go, the final channel contained banned move spaces, representing the spaces which the player could not use in that turn.

The convolutional layer of the network was made up of PyTorch's Conv2d. According to its documentation, Conv2d's parameters are (N, C_{in}, H, W) where N is batch size, C_{in} is the number of input channels, and H and W denote the height and width of the channels (in this case, the board size). Using the aforementioned channels as C, PyTorch's Conv2d applied convolution over the inputs. As per PyTorch's documentation, Conv2d works by applying the following formula:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

Formula 2: PyTorch's Conv2d class (PyTorch, 2019)

Conv2d's would output 20 tensors of size (H,W) which were then transformed into one-dimensional vectors. These flattened vectors were fed to the linear layer, which applied linear transformation on the incoming 20 features and would output a vector of length equal to the game's board size. PyTorch's Linear class applies the linear transformation seen in Formula 3, where *x* represents the incoming features, *A* the linear weights of dimension (output_size, input_size), and *b* the bias term which in this case was set to 0.

$$y = xA^T + b$$

Formula 3: PyTorch's Linear class (PyTorch, 2019)

In the final layer, the linear logits were adjusted to cancel out the values of moves that were no longer available or were illegal. This was achieved by giving these positions a score of -1000, significantly lower than all other scores obtained after the linear layer. In this manner, when the intelligent agent utilizing the policy network would compute softmax on the outputs of the networks, the probabilities associated with these moves would approach 0. Softmax (Formula 4) converted the linear logits to probabilities in range [0, 1] and summing up to 1. It is important to note that the moves with the highest probability were not necessarily chosen, instead after softmax had been computed, a random sample from the probability distribution was taken and used to make a decision, with higher probabilities having a higher chance of being chosen.

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Formula 4: Softmax function (PyTorch, 2019)

5.3.2 Supervised Learning

Supervised learning is the process by which a machine learns how to map inputs to specific outputs (ground truths). A machine is fed training data, consisting of pairs of input objects and desired outputs, and it develops a function and tweaks weights such that loss is minimized and the machine's output is as close as possible to the ground truths. This is achieved by having a training function that iterates over the training data hundreds of times. For each iteration, the model makes predictions and calculates the difference between predictions and labels as loss, and subsequently adjusts its parameters in response. The final model is then tested against an unseen set of data to assess its general accuracy and to ensure that overfitting has not occurred. Overfitting can be defined as overtraining a model such that it fits the training data too closely and to the detriment of accuracy on unseen data (Allamy & Khan, 2014).

For the policy network's training function, loss was computed by first obtaining the log probability of the index to be chosen, multiplying that probability by -1, and by the move's

expected reward (Formula 5). Using PyTorch's built-in Adam optimizer, the gradient of the loss was calculated and used to update the network's weights with a starting learning rate of 0.0001. Adam (adaptive moment estimation) is a form of stochastic optimization that maintains adaptive per-parameter learning rates and utilizes estimates of first and second moments of the gradients to update parameters (Kingma & Ba, 2015).

$$loss = -logp * reward$$

Formula 5: Policy network's loss function

Supervised learning was the first step in training to ensure that the training function was working as expected. Furthermore, it was used to assess the network's improvement under specific games states. Supervised learning was carried out by feeding the training function a set of game states, the rewards associated with each available move, and the index of the move to be chosen (the optimal move) in mini-batches of size 2 and trained over 500 epochs.

5.3.3 Reinforcement Learning

Reinforcement learning was the final training step for the policy network. Similar to supervised learning, reinforcement learning is another method by which an intelligent agent can be trained. However, no input/output pair is needed for reinforcement learning. Instead, the agent takes actions in an environment which are interpreted into rewards which are then fed back to the agent, creating a closed loop. By repeating this process, the agent eventually learns how to maximize reward without any ground truth or human intervention. (Lange, Gabel & Riedmiller, 2012)

The setup for reinforcement learning was as follows: two dummy players were set up to play against each other, Player1 (agent to be updated) and Player2. After playing 1000 matches, all moves made by Player1 were recorded as well as the winner of each match (-1: opponent won, 0: tie, 1: player won). For each move, a reward value was calculated utilizing Formula 6, where e represents the previously mentioned match winner value, multiplied by a discounting factor alpha (α). To encourage winning games in the shortest number of moves possible and tying/losing in the longest number of moves possible, alpha was exponentiated by a decay factor. The closer the current move was to the end of the match the higher the value of alpha (with a possible maximum value of 0.99), and the further away the smaller the value of alpha.

$$reward = e \times \alpha^{(lengthmatch - currentmove)}$$

Formula 6: Discounted expected reward

Tuples containing a game state, move, and reward were fed to the policy network's training function for 500 epochs and in mini-batches of 100 to update the current model and save its parameters to a new file. This process was repeated 100 times, for each iteration Player1

would load the previous model, play 1000 matches, train, and update its weights. In addition, every 10 iterations the dummy player would load a newer model to increase the difficulty level and prevent Player1's learning from plateauing. Reinforcement learning resulted in 100 different models indexed from 0 to 99, where Model99 was the last and would presumably achieve the highest level of play.

5.4 Value Network

5.4.1 Network Structure

The value network (ValueNN) is similar both in structure and function to the policy network. Like the policy network, the value network takes as input a single board state which is broken down into distinct channels. However, the function of the network is instead to predict the expected reward for all available moves. Regarding its structure, it is composed of a 2-dimensional convolutional layer, followed by a rectified linear activation function (ReLU), a flattening layer, and lastly a fully-connected linear layer. The three layers it has in common with the policy network fulfill the same functions. As for the ReLU layer, it applies the rectified linear unit function to each element. This linear function outputs the input, if positive, and 0 otherwise, as visualized by Figure 6. The role of this function in the network was to get the features closer to a game's possible rewards of -1, 0, or 1.

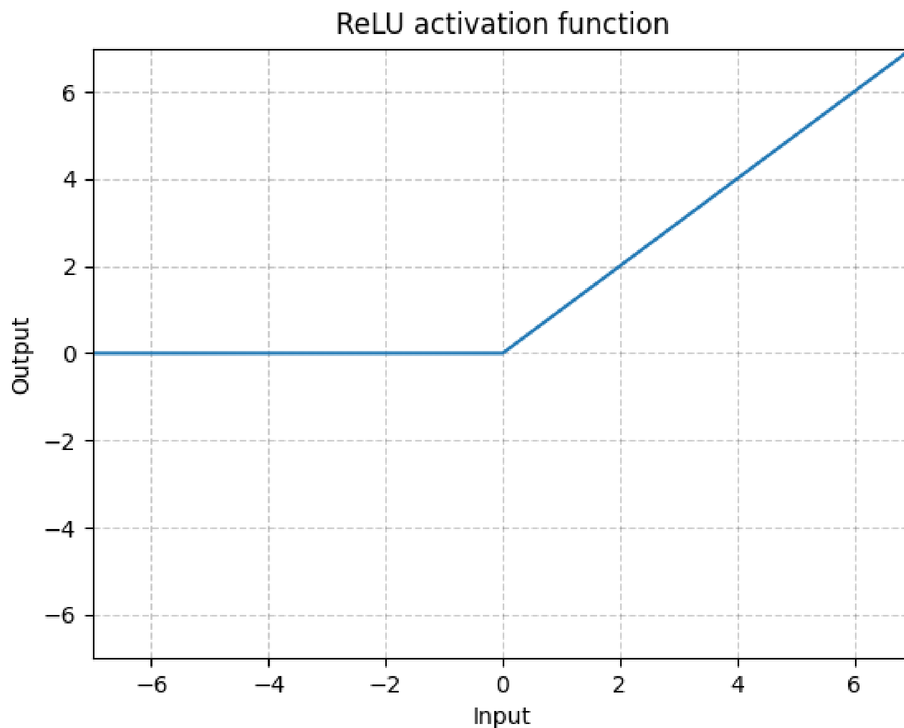


Figure 6: Rectified Linear Activation Function (PyTorch, 2019)

It is important to note that after outputting the expected rewards for available moves, and contrary to intuition, the agent utilizing the network would not always choose the move with the highest associated reward. Instead, to balance exploration vs exploitation like with MCTS, the optimal move would be chosen with a probability of 90% and the remaining 10% of the time a random move would be chosen instead.

5.4.2 Supervised Learning

The value network's training function utilizes mean squared error loss (MSE loss) and optimizes its parameters through stochastic gradient descent (SGD) with a learning rate of 0.001 and momentum of 0.9, following Sutskever et al.'s (2013) findings on the importance of momentum in deep learning. MSE (Formula 7) was calculated by averaging the square difference between the prediction and the target reward values. Utilizing PyTorch's implementation of SGD, a step of gradient descent was performed to optimize the network's parameters. SGD could be considered a simplified version of Adam, which updates weights based only on loss's first derivative and maintains a single learning rate, as opposed to per-parameter. As with the policy network, the value network was trained on a limited set of game states and the expected rewards for the respective moves for each example, for 500 epochs and in mini-batches of 2.

$$\text{MSE} = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

Formula 7: Mean Squared Error

Experimental Results

To test the level of play that the different approaches produced, five types of players were created: RandomPlayer, MinimaxPlayer, MCTSPlayer, PolicyNNPlayer, and ValueNNPlayer. As its name suggests, RandomPlayer was an agent that chose moves at random and served as a base to compare other agents to. Theoretically, two random players competing against each other would have a 50% chance of winning, plus/minus some margin to account for variance and to take into consideration the advantage of the player who goes first. Hence, if the distinct intelligent players were better than simply choosing moves at random there would be a noticeable increase in the win rate. The set up for testing was as follows: Minimax, MCTS, and PolicyNN players played 1000 matches of TicTacToe against the RandomPlayer. The results can be seen in Figure 7, with Minimax being the best at a 99.8% win rate and no losses, followed by MCTS with a 98% win rate and no losses, and finally the PolicyNNPlayer with a 64.2% win rate, 23.2% loss, and 12.6% tie. This follows expectations, as for a small and solved game such as

TicTacToe, Minimax and MCTS can search the entire game space and make optimal decisions at every move.

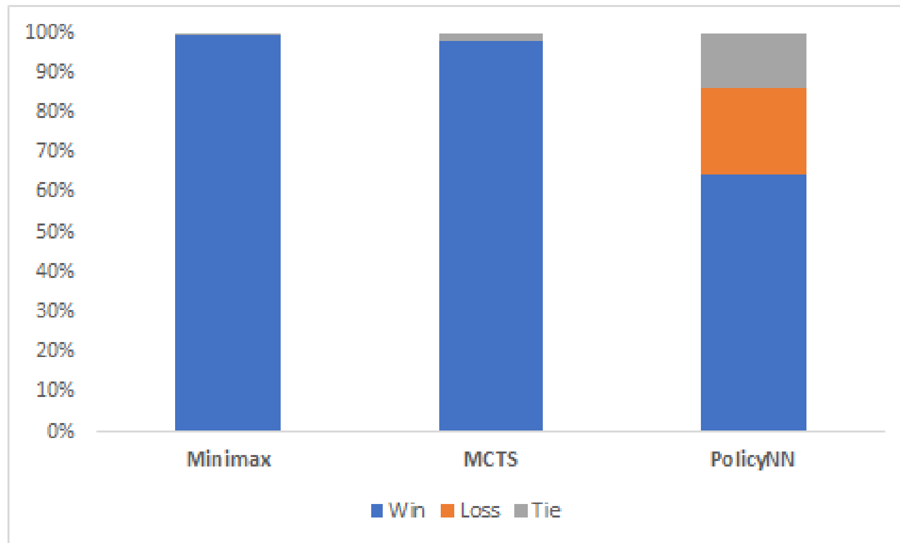


Figure 7: Win, loss, and tie rates vs RandomPlayer for TicTacToe

Even though the player utilizing the policy network did not achieve as high a level of play as the two other algorithms, it could match their win rates with more training and outperform them in Othello and Go. As can be seen in Figure 8, through supervised learning loss was reduced from 0.75 to 0.07 in 500 epochs, meaning that after training the network's outputs very closely follow the ground truths. With a more comprehensive set of data, the networks could be trained alone via supervised learning and not require reinforcement learning for smaller games.

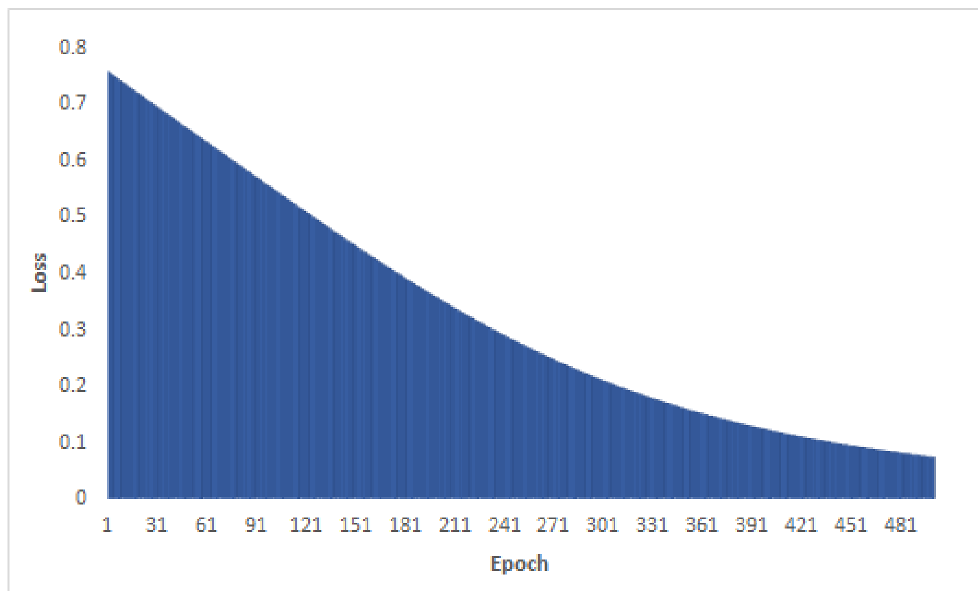


Figure 8: Supervised learning loss for PolicyNN

The value network provided results as seen in Figure 9. The test was 100 iterations in 10 sets of data. The algorithm successfully exploited close to 90% of the time as you can see in the blue bar above. If it didn't exploit then it explored nearly 10% of the time every iteration. This is good because sometimes in order to properly learn the algorithm must explore and choose a random reward 10% of the time in our case, and of course 90% we are using to choose the maximum value from the output of the neural net. These figures can be different depending on what type of data you want to collect and what your project goal is, but for ours we used 90% and 10% to be our values.

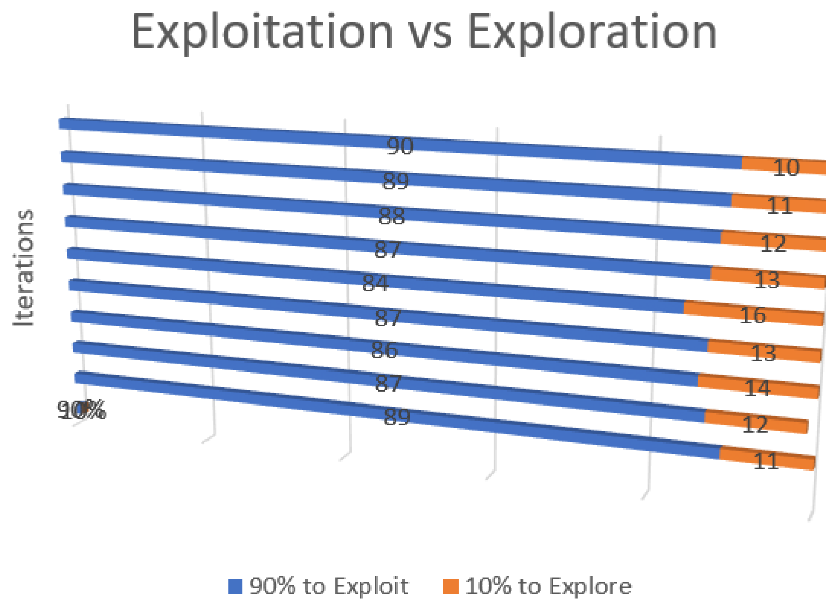


Figure 9: Supervised learning loss for ValueNN

Finally, to test if the network successfully maximized rewards and in turn learned how to play the game after reinforcement learning, two types of tests were carried out. First, each model played 1000 matches against the random player. Additionally, Model99 played 1000 matches against each other model (0 to 98). The goal of both tests was to assess the performance of the models against the base performance, and secondly to gage how much the models improved between iterations. Figure 10 displays the loss rate of all models against the random player. As can be seen there is a downward trend from around 24% to below 20%. On the other hand, Figure 11 displays the loss rate of Model99 against all other models. There is no clear trend and the loss rate remains steady, with variations, throughout. Overall, it can be said that the models improved over the 100 iterations but they could benefit from more reinforcement learning to achieve even higher levels of play.

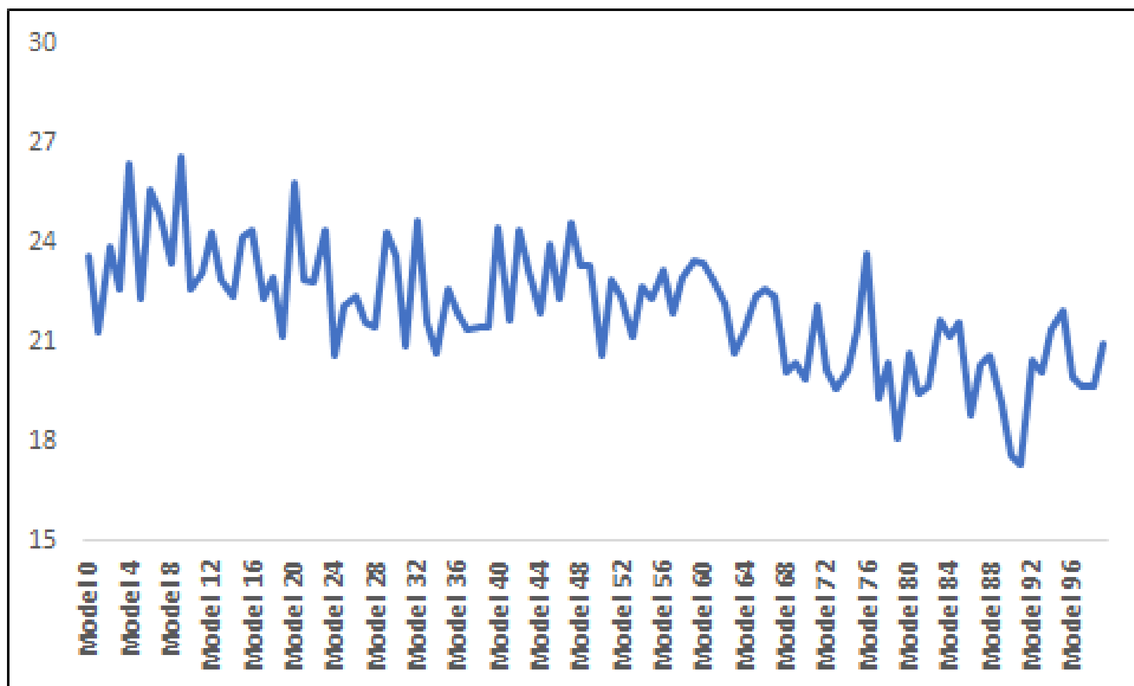


Figure 10: Loss % of PolicyNN Models vs RandomPlayer

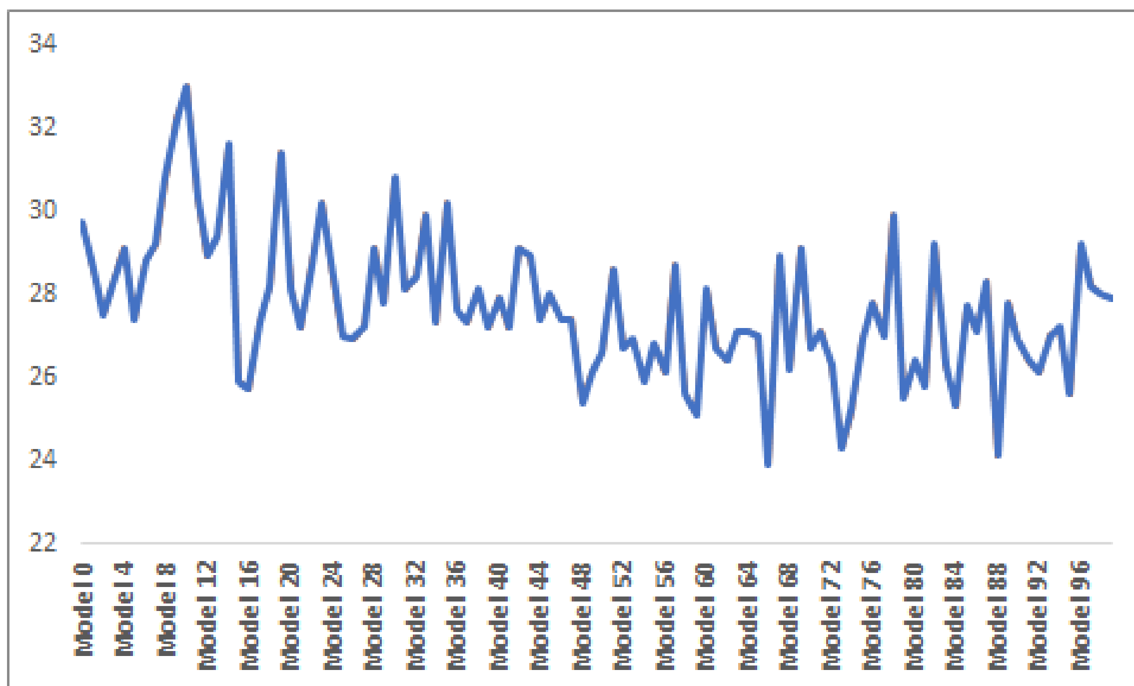


Figure 11: Loss % of PolicyNN Model99 vs Models 0-98

Future Work

Due to the challenges presented over the past year due to the COVID-19 pandemic and given the limitations of online collaboration, the team was not able to complete all of the intended tasks. Although intelligent players that employed neural networks worked for TicTacToe, time constraints did not permit testing and adjusting these agents for Othello or Go. Hence, if the project were to be picked up by another team in coming years, the natural progression from its current state would be to expand the codebase to first get reinforcement learning to train models more effectively and to adapt the AI for the two aforementioned games. Most importantly, intertwining the value and policy networks together into a single intelligent agent that could take advantage of the predictions presented by both networks to achieve a higher level of play, akin to AlphaGo, could also be implemented without much difficulty in the future.

Some additional approaches that the team would have liked to explored include: more comprehensive supervised learning for both networks by feeding them data collected and saved over thousands of MCTS simulations, and a different approach to reinforcement learning were game data is stored between models so that variations in a single match have less effect on the current model's overall effectiveness.

Finally, the team recommends utilizing cloud computing services such as Microsoft Azure, Amazon Web Services, or Google Cloud for reinforcement learning. Cloud computing services are inexpensive and offer more computational power than a local device and as such more iterations of reinforcement learning can be performed to achieve a higher level of play.

Conclusion

Despite being invented over 2,000 years ago the game of Go remains a challenge for artificial intelligence. The game presents many difficulties: a search space impossible to explore in its entirety, complex decision-making, and an optimal solution so complicated that it is almost futile to try to approximate it. These qualities prevented algorithms commonly employed to solve other classical games to achieve the same results in Go. However, thanks to rapid developments in the fields of artificial intelligence and machine learning over the past 20 years, computer Go programs have evolved from beginner-level play to surpassing the world champion. As Google's AlphaGo proves, a combination of tree search and deep neural networks, trained by supervised learning and reinforcement learning, can produce effective evaluation and move selection for a game as complex as Go.

In this project, the team developed all individual components that made AlphaGo successful, namely MCTS and the value and policy networks. Through experimental tests the team assessed the individual efficiency of different algorithms (Minimax, MCTS, and the policy network) for TicTacToe. While the performance of a single neural network for a solved game can not compare to that of tree search algorithms, the results display its ability to learn and improve its level of play over time, showcasing the potential of supervised and reinforcement learning.

Additional training of the policy network could have resulted in a higher level of play for TicTacToe and achieve a significantly different performance from a random player for Go. Unfortunately, due to constraints posed by the COVID-19 pandemic, an intelligent player emulating AlphaGo's structure could not be developed.

Nonetheless, it is evident that training and deep learning are incredibly effective tools that can surpass human level of play. With the introduction of newer algorithms such as MCTS, novel deep learning and training techniques, and more computational power available, feats that were previously thought to be far off in the future have been completed. These advances not only present new opportunities for general game-playing but for other areas as well, such as constraint satisfaction problems or planning. AlphaGo and other cutting edge intelligent players provide the hope that seemingly impossible problems in the domain of artificial intelligence could be achieved in the near future.

References

Allamy, Haider & Khan, Rafiqul Zaman. (2014). *Methods to Avoid Over-Fitting and Under-Fitting in Supervised Machine Learning (Comparative Study)*. Computer Science, Communication & Instrumentation Devices, 163-172. Retrieved from doi: 10.3850/978-981-09-5247-1_017

Chen, J. X. (2016). *The Evolution of Computing: AlphaGo*. Computing in Science & Engineering, *18(4)*, 4-7. Retrieved from doi: 10.1109/MCSE.2016.74

Coulom, R. (2006). *Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search*. In: van den Herik H.J., Ciancarini P., Donkers H.H.L.M.. (eds) Computers and Games. CG 2006. Lecture Notes in Computer Science*, 4630*. Retrieved from https://doi.org/10.1007/978-3-540-75538-8_7

IBM. (2021). *What is Artificial Intelligence?*. Retrieved from https://www.ibm.com/cloud/learn/what-is-artificial-intelligence

Lapan, Maxim. (2018). *Deep Reinforcement Learning Hands-On : Apply Modern RL Methods, with Deep Q-Networks, Value Iteration, Policy Gradients, TRPO, AlphaGo Zero and More*. Retrieved from https://ebookcentral-proquest-com.ezpxy-web-p-u01.wpi.edu/lib/wpi/detail.action?docID=5434975

Pearl, J. (1980). *Asymptotic Properties of Minimax Trees and Game-searching Procedures*. Artificial Intelligence, *14(2)*, 113-138. Retrieved from https://doi.org/10.1016/0004-3702(80)90037-5

Kingma, Diederik & Ba, Jimmy. (2015). *Adam: A Method for Stochastic Optimization*. International Conference on Learning Representations. Retrieved from https://arxiv.org/abs/1412.6980

Kohs, G. (Director). (2017, September 29). *AlphaGo* [Video file]. Retrieved from https://www.imdb.com/title/tt6700846/

Lange S., Gabel T., Riedmiller M. (2012). *Batch Reinforcement Learning*. In: Wiering M., van Otterlo M. (eds) Reinforcement Learning. Adaptation, Learning, and Optimization, 12. Retrieved from https://doi.org/10.1007/978-3-642-27645-3_2

PyTorch. (2019). *Conv2d*. Retrieved from https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

PyTorch. (2019). *Linear*. Retrieved from https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

PyTorch. (2019). *Softmax*. Retrieved from https://pytorch.org/docs/stable/generated/torch.nn.Softmax.html

PyTorch. (2019). *ReLU*. Retrieved from https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html

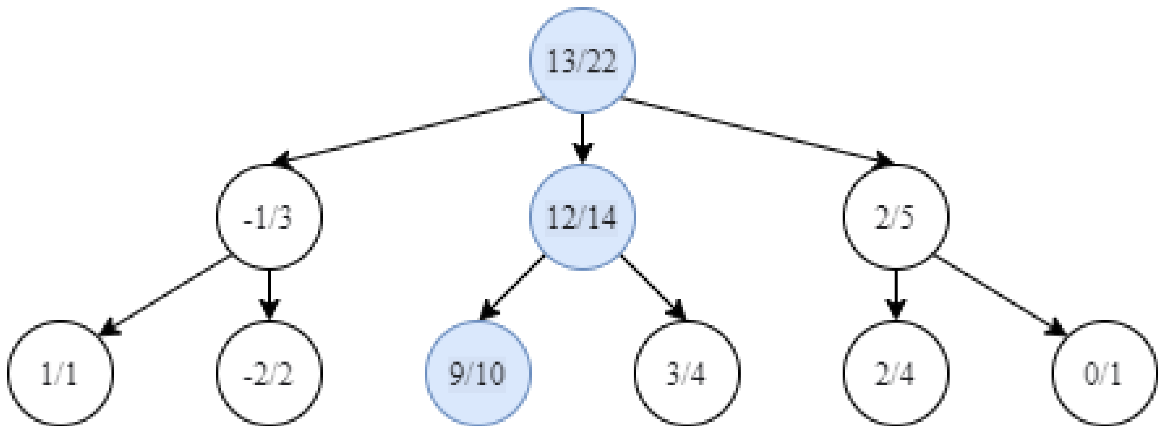
Schmidhuber, Jürgen. (2015). *Deep Learning in Neural Networks: An Overview*. Neural networks, *61,* 85–117. Retrieved from http://dx.doi.org/10.1016/j.neunet.2014.09.003

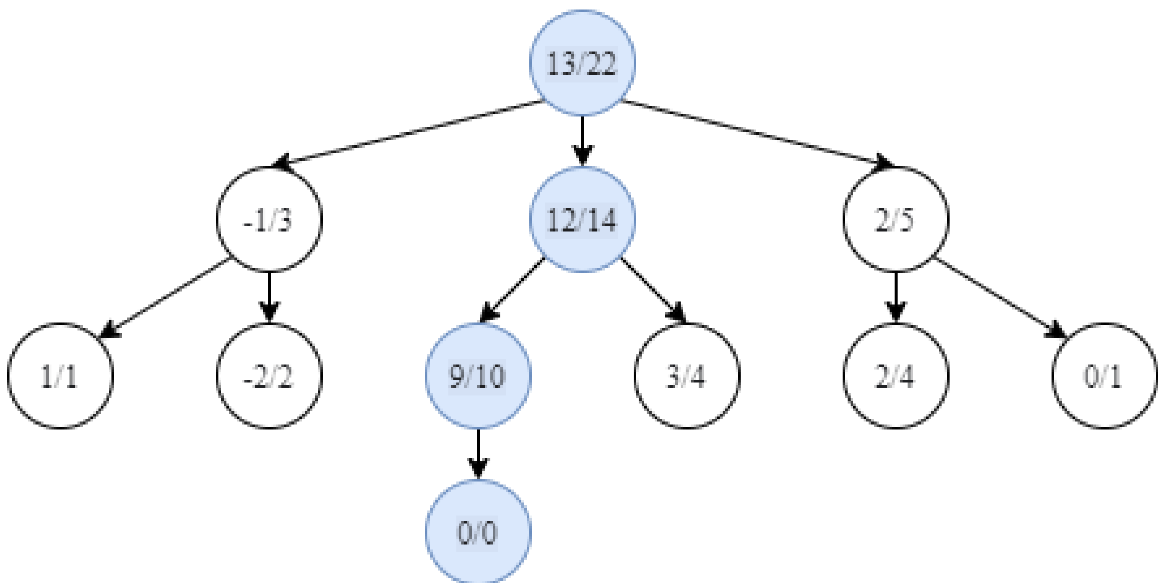
Silver, D., Huang, A., Maddison, C. *et al.* (2016). *Mastering the game of Go with deep neural networks and tree search*. Nature, *529*, 484–489. Retrieved from https://doi.org/10.1038/nature16961

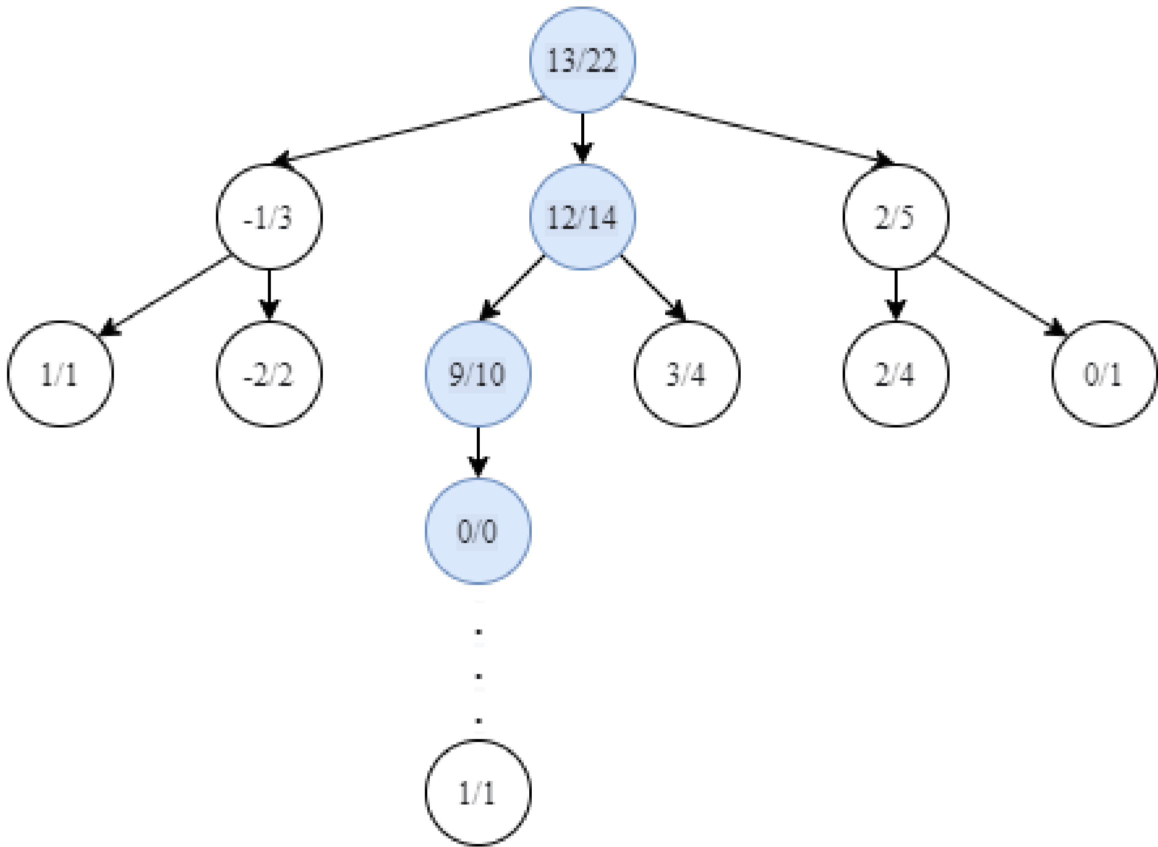
Sutskever, I., Martens, J., Dahl, G. & Hinton, G. (2013). *On the importance of initialization and momentum in deep learning*. Proceedings of the 30th International Conference on Machine Learning. PMLR, *28(3)*, 1139-1147. Retrieved from http://proceedings.mlr.press/v28/sutskever13.html

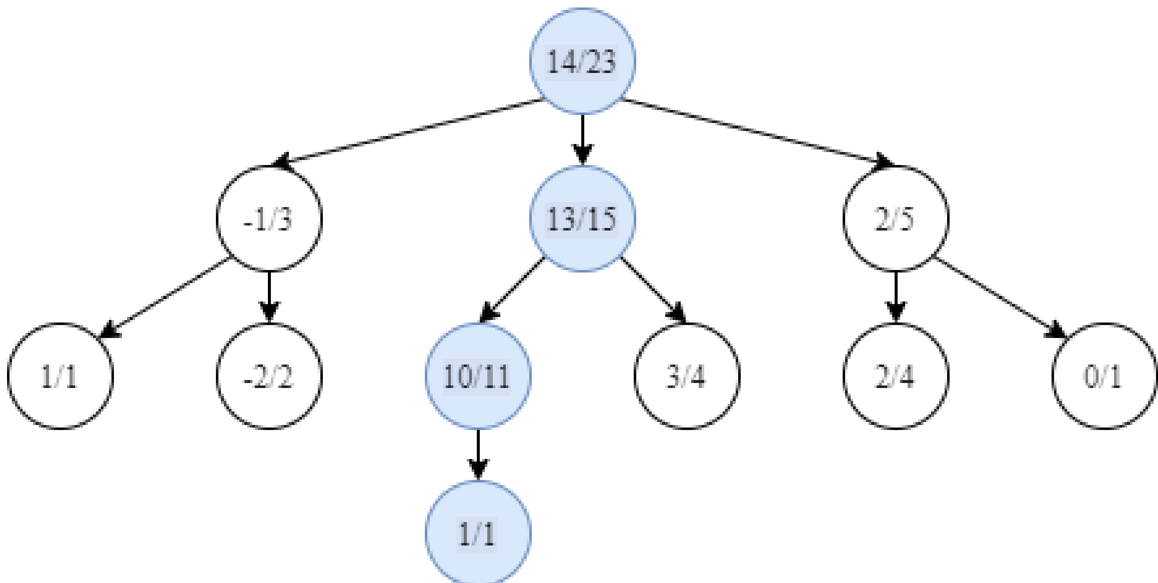
Wikipedia. (2021). *Go (game)*. Retrieved from https://en.wikipedia.org/wiki/Go_(board_game)

Appendix A: Steps of Monte Carlo Tree Search






SIMULATION
13/22
-1/3
12/14
2/5
1/1
-2/2
9/10
3/4
2/4
0/1
0/0
1/1


BACKPROPAGATION
14/23
-1/3
13/15
2/5
1/1
-2/2
10/11
3/4
2/4
0/1
1/1