

Final report

ZeroGo

ML Based Go Player

Prepared by:

Jing Ouyang

Sunny Patel

Wesam Alwehaibi



Table of Contents

| | |
|--|-----------|
| Table of Contents | 2 |
| Abstract | 4 |
| Background: Go rules | 5 |
| Liberty | 5 |
| Ko rule | 6 |
| Scoring and end of game | 7 |
| Background: Ranking players | 8 |
| Background: Game play strategies by experts | 9 |
| By Bruno Curfs, 1-dan | 9 |
| General strategies | 9 |
| If you play Black | 9 |
| If you play white | 10 |
| Why Go as a project? | 11 |
| Goals | 12 |
| Prior Work | 13 |
| AlphaGo | 13 |
| AlphaGo Zero | 14 |
| Methodology | 16 |
| Tools Utilized | 16 |
| Agents built | 16 |
| Random Agent | 16 |
| Greedy Agent | 17 |
| Minimax agent | 17 |
| Depth pruned agent | 17 |
| Alpha beta agent | 17 |
| MCTS agent | 17 |
| Neural Network Agent | 18 |
| Actor Critic | 18 |
| ZeroGo Model | 20 |
| Results and Analysis | 26 |

| | |
|---|-----------|
| Traditional Algorithm Agents | 26 |
| Greedy Agent (25-kyu) | 26 |
| Minimax Agent (Ranking Not Applicable) | 26 |
| Depth Pruned Agent (Ranking Not Applicable) | 26 |
| Alpha Beta Agent (Ranking Not Applicable) | 27 |
| MCTS Agent (15-kyu to 25-kyu depending on parameters) | 27 |
| Deep learning Agents | 27 |
| Neural Network Agent (5-kyu to 15-kyu depending parameters) | 27 |
| Actor Critic Agent (Ranking Not Applicable) | 29 |
| ZeroGo Agent (1-dan to 10-kyu depending on parameters) | 29 |
| Limitations, Conclusion, and Future Directions | 31 |
| References | 32 |

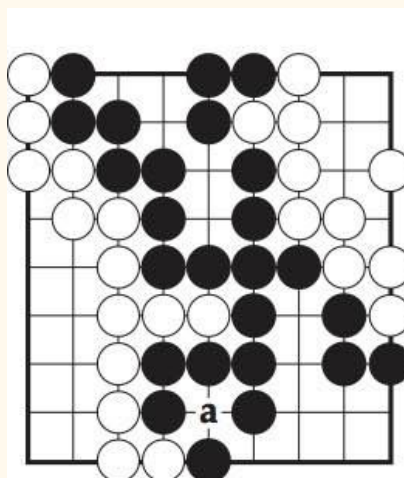
Abstract

ZeroGo is an attempt to use machine learning techniques to develop agents for the game of Go. Go is an abstract strategy board game for two players, in which the aim is to surround more territory than the opponent. Traditional search-based Go agents could hardly reach the level of human players due to the overwhelmingly large search space of Go. Recent researchers focus on machine learning algorithms such as Neural Network and Reinforcement Learning. While strong Go programs such as AlphaGo from DeepMind could reach super-human level with the help of strong hardwares, we combined the traditional techniques with the new emerging machine learning algorithms to reach amateur human player level.

This paper discussed the process of our team building ZeroGo program. We first introduce some background knowledge about Go such as rules and basic strategies. Then we discuss the incentives and goals of our project. Some prior works in the field of Go agents are then introduced. Next, we introduced various methods applicable for Go and the agents we have built using those methods. We analyzed our results and finally discussed the limitations and future directions.

Background: Go rules

The game begins with an empty board. It is a 2-player game with one player being White and the other one being Black. They get unlimited supply of what is called stones. The stones are the objects that they play on the board. As mentioned earlier, the goal of the game is to cover each of as many areas of the board as much as possible. A player can place his stone on any empty position. If an opponent's stone is covered by the player's stone from all sides then the opponent's stone is captured and the position becomes free for a future move. Pretty unusual but the game is played on the intersection between lines and not on the grid cells.

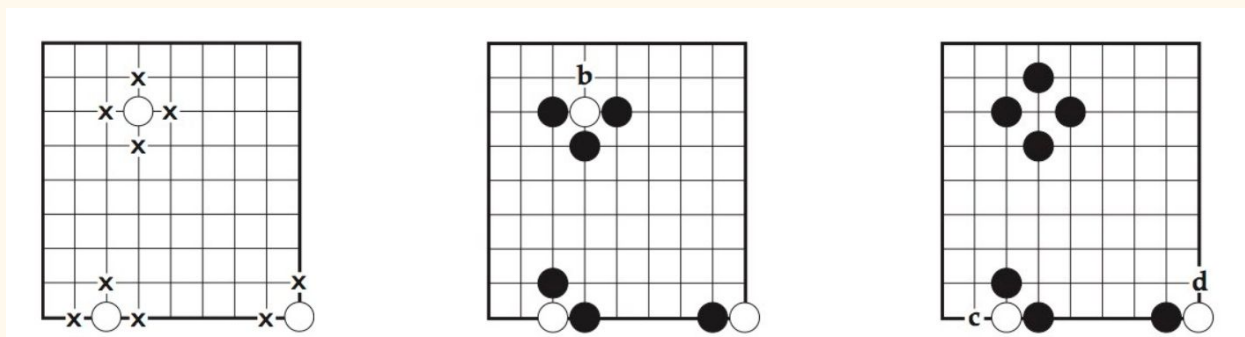


The stone on the position “a” will be captured by the surrounding black stones.

At the end of the game the winner is decided by counting the number of stones of each kind.

Liberty

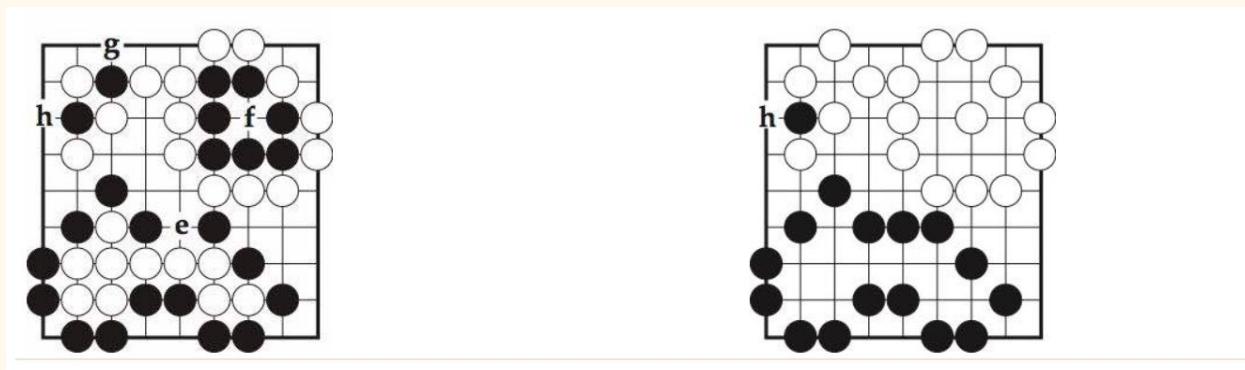
A liberty is defined as an empty position, either horizontally or vertically, around a stone.



The leftmost diagram shows 3 stones and their liberties marked as crosses.

The middle diagram shows b as the only liberty left to the white stone and can be captured in the next black turn.

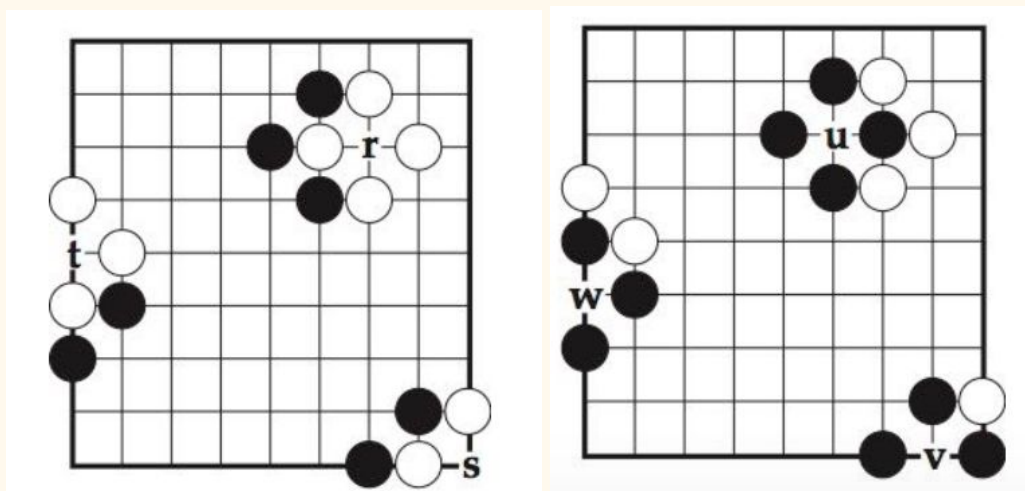
The right diagram shows that b was captured and in the next turn c or d could be captured too. Strings are stones of the same color connected to each other by being adjacent. Strings can be captured in the same way as individual stones. The entire string/collection is captured when there are 0 liberties left for each of the stones in the string.



In the left diagram above, playing a white in “f” will capture all surrounding black stones as shown on the right.

Also, playing a black in “e” will lead to all surrounding whites being captured. A player can not play a self-capturing move, i.e. playing a move that makes any of their own stones captured (reduce the number of liberties to 0)

Ko rule



If from the state shown in the left black player plays a move on position “r” the white stones will be captured as shown in the diagram right. At this point, white can play a move at position “u” and that will lead to the same state as the earlier one. This leads to black having the chance to play “r” again and this can result in an infinite game. This is prevented by the ko rule.

The ko rule removes this possibility of indefinite repetition by forbidding the recapture of the ko, in this case a play at u, until White has made at least one play elsewhere. Black may then fill the ko, but if Black chooses not to do so, instead answering White's intervening turn elsewhere, White is then permitted to retake the ko. Similar remarks apply to the other two positions in these diagrams; the corresponding plays at w and v in must also be delayed by one turn.

Scoring and end of game

Both players have the ability to pass at any point in the game. When both players pass consecutively the game is declared as ended.

Before the scoring phase, the stones that have no chance of making two eyes or connecting up to friendly stones are identified. These are called dead stones. Dead stones are treated exactly the same as captures when scoring the game. If a disagreement occurs, the players can resolve it by resuming play. The goal of the game is to control a larger section of the board than your opponent.

There are two common ways to calculate the score and decide the winner of a game.

- Territory scoring: The player gets one point for each point of the board that is completely surrounded by that player's stones and one point for each of the captured stones. The player with more points is the winner.
- Area scoring: the player gets one point for every point of the territory plus one point for each stone left on the board. The player with more points is the winner.

Most of the time, these two methods lead to the same winner. The only time this can lead to a different winner is when neither of the players pass early. In such cases, the difference in the stones on the board equals the captures. For most of our implementation we are going to assume area scoring as our method.

In addition, the white player gets extra points as compensation for going second. This compensation is called komi. Komi is usually 6.5 points under territory scoring or 7.5 points under area scoring—the extra half point ensures there are no ties. We assume 7.5 points komi.

Background: Ranking players

Traditionally, the level of players has been defined using “kyu” and “dan” rank. Kyu ranks are considered student ranks. Dan ranks are considered master ranks. Beginners who have just learned the rules of the game are usually around 30th kyu. As they progress, they advance numerically downwards through the kyu grades. The best kyu grade attainable is therefore 1st kyu. If players progress beyond 1st kyu, they will receive the rank of 1st dan, and from thereon will move numerically upwards through the dan ranks. In martial arts, 1st dan is the equivalent of a black belt. The very best players may achieve a professional dan rank.

Here is a table that describes the ranking system in brief:

| Rank Type | Range | Stage |
|---|--------|----------------------|
| Double-digit <i>kyu</i> (級,급) (<i>geup</i> in Korean) | 30–20k | Beginner |
| Double-digit <i>kyu</i> (abbreviated: DDK) | 19–10k | Casual player |
| Single-digit <i>kyu</i> (abbreviated: SDK) | 9–1k | Intermediate amateur |
| Amateur <i>dan</i> (段,단) | 1–7d | Advanced amateur |
| Professional <i>dan</i> (段,단) | 1–9p | Professional Player |

With the invention of calculators and computers, it has become easy to calculate a rating for players based on the results of their games. Commonly used rating systems include the Elo and Glicko rating systems. Rating systems generally predict the probability that one player will defeat another player and use this prediction to rank a players strength.

For our report, we will be using the same ranking system. When we create a new player, we can rank it against multiple players. One obvious approach is to rank each of our players with other players of our own. There are ways like making our agent test against other players online or even make humans play against it. These are described along with individual players. More information on this can be found at [9].

Background: Game play strategies by experts

There are many strategies which people use to play. Experts with years of experience have been communicating, sharing and deriving new strategies regularly. Here are some of the strategies devised by the players in the world.

By Bruno Curfs, 1-dan

General strategies

- The same area of territory requires less stones to surround in the corner than at the border, and less stones at the border than in the center of the board. It makes sense to play in the corner first, then at the border and last in the center.
- Many stones in a small area spell inefficiency.
- There is a fine balance between influence and territory. More territory is good, more influence is good. Influence is the beginning stage of territory. But influence can only turn into territory by using skill.
- Try to be creative, play your own moves and review/replay your game to see why it worked or why it didn't.
- Record and replay your game (apps abound); ask an experienced Go player to comment on your games.

If you play Black

- Keep your stones connected.
- Keep an eye on your (prospective) territory and defend it.
- Keep the game as simple as possible. White is stronger and will easily thwart what you consider attacks. In most cases “attacks” on white will only weaken you.
- Keep your groups alive.
- Learn to look one or two moves ahead. Simply imagine where white could play and what you would do in that situation. Start simple to prevent second-row captures. Increase your skill to read out more complex situations and with more moves.

- Never give up on dead stones, but try to find a way to make them a bother to white. If you can force white to capture already dead stones without making your losses bigger, it will cost him moves.
- Build impenetrable walls.
- Consider taking territory over defending an invasion.
- Look for big moves.

If you play white

- Play light and flexible and keep good shape. Be oil through porous rocks.
- Learn to play and win a ko.
- Complicate the game by cutting. Keep your cutting stones alive for maximal damage.
- Complicate the game by invading. Minimize black's territory by making points dame.
- Create influence, plan how to use it to gain territory.
- Kill weak groups.
- Invade the corners.
- Play tengen (middle) and connect up with it or use it as a ladder breaker to make your invasions work.
- Surround lonely stones until they are too weak to live.
- Cut black in pieces and keep the pieces separated.
- If black invades, confine him and if he keeps adding stones, find a way to capture at a large scale.

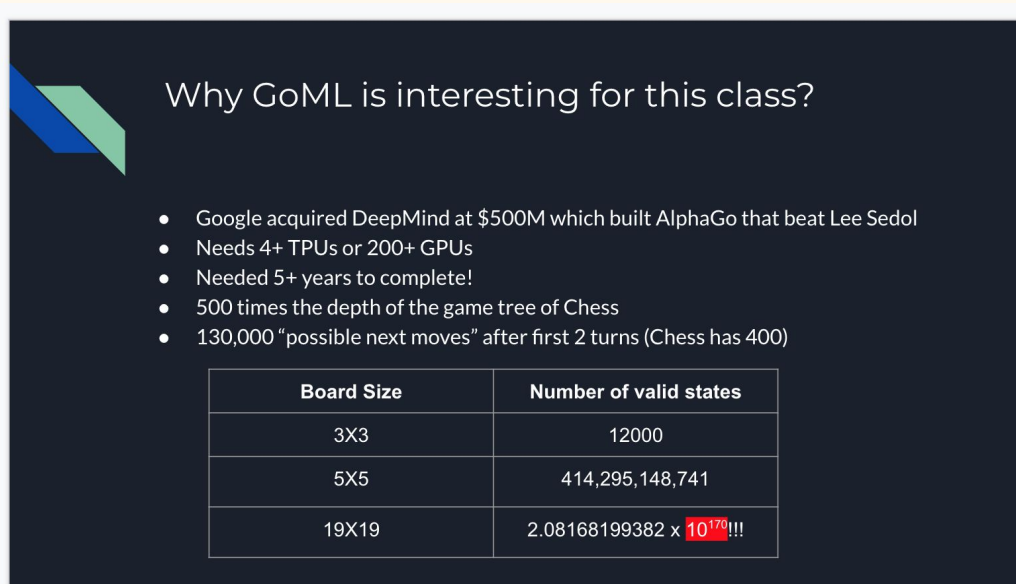
Why Go as a project?

Despite decades of work, the strongest Go computer programs could only play at the level of human amateurs. Standard AI methods, which test all possible moves and positions using a search tree, can't handle the sheer number of possible Go moves or evaluate the strength of each possible board position.

Go is often stated to be the most challenging game for an AI system to train for and win. Go has similar ideas to chess. Chess was a difficult problem for AI too. It was unsolved for years before IBM Deep Blue achieved success. Go is definitely way more complex than Chess. For example, at the opening move in Chess there are 20 possible moves. In Go the first player has 361 possible moves, and this scope of choices stays wide throughout the game. After the first 2 moves, Chess has 400 valid moves whereas go has 130,000. A full 19x19 go board has 10^{170} valid states! For any traditional algorithm, it can safely be stated to be impossible to tackle this scale.

Google recently acquired a company called DeepMind. They created AlphaGo. One of the few AI systems that could master the game of Go. Their algorithm and the ideas are described in the next section. Alpha Go obviously is a beautiful combination of a good algorithm/software and a very heavy hardware. To get a full version of Alpha Go running, it can take millions of dollars.

Here is a slide from our proposal that shows why Go is an interesting project for this class.



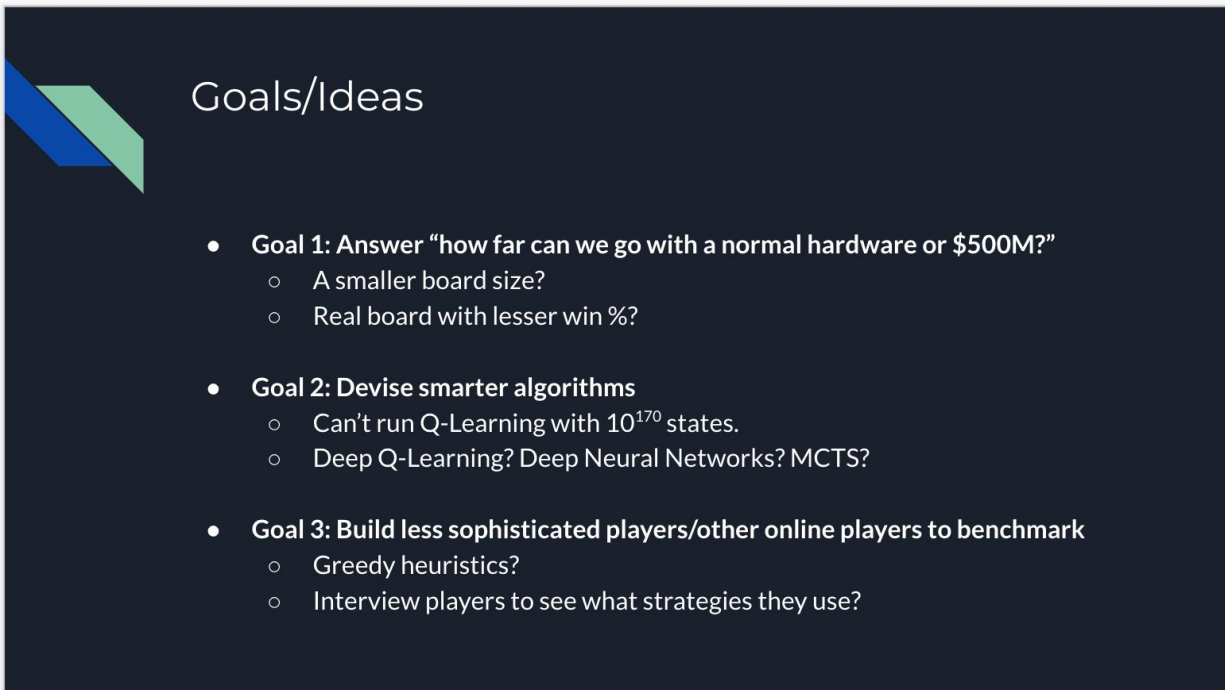
Why GoML is interesting for this class?

- Google acquired DeepMind at \$500M which built AlphaGo that beat Lee Sedol
- Needs 4+ TPUs or 200+ GPUs
- Needed 5+ years to complete!
- 500 times the depth of the game tree of Chess
- 130,000 "possible next moves" after first 2 turns (Chess has 400)

| Board Size | Number of valid states |
|------------|-------------------------------------|
| 3X3 | 12000 |
| 5X5 | 414,295,148,741 |
| 19X19 | $2.08168199382 \times 10^{170}$!!! |

Goals

This is a slide from our original proposal in week 2.



Goals/Ideas

- **Goal 1: Answer “how far can we go with a normal hardware or \$500M?”**
 - A smaller board size?
 - Real board with lesser win %?
- **Goal 2: Devise smarter algorithms**
 - Can't run Q-Learning with 10^{170} states.
 - Deep Q-Learning? Deep Neural Networks? MCTS?
- **Goal 3: Build less sophisticated players/other online players to benchmark**
 - Greedy heuristics?
 - Interview players to see what strategies they use?

Hence, we decided to take up this project with these 3 goals in our minds.

In simple terms, we want to see what the hardware limitation on the game is. How far we can go with the usual hardware but novel algorithms. We use macbooks and google colaboratory throughout the process.

Prior Work

There have been several attempts to crack the game of Go. There are multiple attempts with many different kinds of algorithms, hardware and software techniques. One of the most notable attempts is Alpha Go.

AlphaGo

AlphaGo, a computer program that combines advanced search trees with deep neural networks. These neural networks take a description of the Go board as an input and process it through a number of different network layers containing millions of neuron-like connections. One neural network, the “policy network”, selects the next move to play. The other neural network, the “value network”, predicts the winner of the game. AlphaGo was introduced to numerous amateur games to help it develop an understanding of reasonable human play. Then it was made to play against different versions of itself thousands of times, each time learning from its mistakes. Over time, AlphaGo improved and became increasingly stronger and better at learning and decision-making. This process is known as reinforcement learning. AlphaGo went on to defeat Go world champions in different global arenas and arguably became the greatest Go player of all time.

In October 2015, AlphaGo played its first match against the reigning three-time European Champion, Mr Fan Hui. AlphaGo won the first ever game against a Go professional with a score of 5-0. AlphaGo then competed against legendary Go player Mr Lee Sedol, the winner of 18 world titles, who is widely considered the greatest player of the past decade. AlphaGo's 4-1 victory in Seoul, South Korea, in March 2016 was watched by over 200 million people worldwide. This landmark achievement was a decade ahead of its time.

This is what they mean by “enormous search space.” Moreover, in Go, it’s not so easy to judge how advantageous or disadvantageous a particular board position is at any specific point in the game — you have to play the whole game for a while before you can determine who is winning. But let’s say you magically had a way to do both of these. And that’s where deep learning comes in! DeepMind used neural networks to do both of these tasks. They trained a “policy neural network” to decide which are the most sensible moves in a particular board position. And they trained a “value neural network” to estimate how advantageous a particular board arrangement is for the player. They trained these neural networks first with human game examples. After this the AI was able to mimic human playing to a certain degree, so it acted like a weak human

player. And then to train the networks even further, they made the AI play against itself millions of times (this is the “reinforcement learning” part). With this, the AI got better because it had more practice.

With these two networks alone, DeepMind’s AI was able to play well against state-of-the-art Go playing programs that other researchers had built before. These other programs had used an already popular pre-existing game playing algorithm, called the “Monte Carlo Tree Search” (MCTS). DeepMind’s AI isn’t just about the policy and value networks. It doesn’t use these two networks as a replacement for the Monte Carlo Tree Search. Instead, it uses the neural networks to make the MCTS algorithm work better... and it got so much better that it reached superhuman levels. This improved variation of MCTS is “AlphaGo”, the AI that beat Lee Sedol and went down in AI history as one of the greatest breakthroughs ever. So essentially, AlphaGo is simply an improved implementation of a very ordinary computer science algorithm.

AlphaGo Zero

AlphaGo Zero uses a novel form of reinforcement learning which is using its own brain to teach itself. The system starts off with a neural network that knows nothing about the game of Go. It then plays games against itself, by combining this neural network with a powerful search algorithm. As it plays, the neural network is tuned and updated to predict moves, as well as the eventual winner of the games. This updated neural network is then recombined with the search algorithm to create a new, stronger version of AlphaGo Zero, and the process begins again. In each iteration, the performance of the system improves by a small amount, and the quality of the self-play games increases, leading to more and more accurate neural networks and ever stronger versions of AlphaGo Zero.

This technique is more powerful than previous versions of AlphaGo because it is no longer constrained by the limits of human knowledge. Instead, it is able to learn tabula rasa from the strongest player in the world: AlphaGo itself.

It also differs from previous versions in other notable ways.

- AlphaGo Zero only uses the black and white stones from the Go board as its input, whereas previous versions of AlphaGo included a small number of hand-engineered features.
- It uses one neural network rather than two. Earlier versions of AlphaGo used a “policy network” to select the next move to play and a “value network” to predict the winner of the game from each position.

- These are combined in AlphaGo Zero, allowing it to be trained and evaluated more efficiently. AlphaGo Zero does not use “rollouts” - fast, random games used by other Go programs to predict which player will win from the current board position. Instead, it relies on its high quality neural networks to evaluate positions.

There are many papers, including the original AlphaGo paper published on the internet. Reading them makes it clear that the solutions are not straightforward or easy to implement. Most of them also require us to have hardware that is more than just a usual computer. Here's one attempt to summarise the mountain of information.

ALPHAGO ZERO CHEAT SHEET

The training pipeline for AlphaGo Zero consists of three stages, executed in parallel

MONTE CARLO ZERO

SELF PLAY

Create a "training set"

The best current player plays 25,000 games against itself.

- Saves MCTS results to understand how AlphaGo Zero selects which move.

At each move, the following information is stored:

The game slate
(see Effect at a Glance State action)

π

The search probabilities
(from the MCTS)

The winner
+1 if we played well -1 if the other player did +0.5 once the game has finished!

RETRAIN NETWORKS

Optimise the network weights

A TRAINING LOOP:
Sample a new batch of 2048 positions from the last 500,000 games.
Retrain the current neural networks on these positions.
(The game slates are the raw test data Deep Neural Networks Architecture)
Loss function:
Compares predictions from the neural network with the search probabilities and actual scores.

PREDICTIONS

\times Cross entropy
 $-$ Mean squared error
 $+/-$ Regularisation

ACTUAL

After every 1,000 training loops, evaluate the network

EVALUATE NETWORK

Test to see if the new network is stronger

Play 100 Games between the latest neural network and the current best neural network

Both players use MCTS to select their moves, with their respective neural networks to evaluate leaf nodes.

Latest player must win 55% of games to be declared the new best player

THE DEEP NEURAL NETWORK ARCHITECTURE

How AlphaGo Zero assesses new positions

The network learns "tabula rasa" (from a blank slate).
At no point is the network trained using human knowledge or expert moves.

The value head

game value for current player (+1, 0, -1)

batch normalisation

fully connected layer

ReLU after non-linearity

hidden layers size 256

Batch normalization

Input

The policy head

W x H = 19 x 19 (to avoid more high probabilities)

Batch normalization

fully connected layer

ReLU after non-linearity

Batch normalization

2 convolved filters (x3)

Input

A residual layer

Batch normalization

ReLU after non-linearity

Skip connection

Batch normalization

256 convolved filters (x3)

Batch normalization

ReLU after non-linearity

Batch normalization

256 convolved filters (x3)

Input

The network

160 residual layers

Value head

Policy head

Input: The game slate (raw input)

MONTE CARLO TREE SEARCH (MCTS)

How AlphaGo Zero chooses its next move

First, run the following simulation 1,600 times...

- Choose the action that maximises...

$Q + U$

Q : A function of P and N , that increases if an action hasn't been explored much, relative to the other actions, or if the prior probability of the action is high.

U : The mean value of the root state.

Early in the simulation, U dominates (more exploration), but later Q is more important (less exploration).
- Continue until a leaf node is reached.
- Backup previous edges.

Then select a move

After 1,600 simulations, the move can either be chosen:

- Deterministically (for competitive play) Choose the action from the current state with greatest N
- Stochastically (for exploratory play) Choose the action from the current state from the distribution $\pi \sim N^{-\frac{1}{T}}$ where T is a temperature parameter controlling exploration

Other points

- The sub-tree from the chosen move is retained for calculating subsequent moves
- The rest of the tree is discarded

Methodology

Tools Utilized

This project aims at solving the game of Go using Machine Learning. While this might sound like repeating what AlphaGo has already done, it is not. We aimed to do a few things differently. As mentioned in the goals section above, our intention is to leverage algorithmic power rather than hardware. For this reason, We purposely used a regular google colab with google drive to test the limitations and abilities of the algorithms chosen as our methodology. We used the GPU runtime that comes with 12GB RAM.

Furthermore, we are utilizing python 3 programming language. And for our deep learning agents, we used Keras 2.3.1 abstract API with tensorflow 1.15.2 backend.

Also, as the framework for our project, we used the library dlgo by the authors of the book “Deep learning and the game of go”. The library offers basic data types needed, game encoders and decoders, hashing and scoring techniques, and other utilities. We have utilized some of the offered functionalities and modified and used our own techniques whenever needed throughout the process.

Agents built

To fully explore the problem, We have created multiple agents, some using traditional algorithms and some using deep learning algorithms. Given the limitations we have set for the resources, we believe that some of our approaches might completely fall apart, however, we will still have takeaways that can be used to answer the questions that we have. We have attempted the following algorithms to tackle our problem statement:

Random Agent

The goal of writing this agent is just to generate moves at random. This agent helped us establish an interface that can be extended for multiple different types of players/agent. The accuracy of this agent is obviously low and can not be used in a real setting. But this player can also be made to play against other players for them to learn and explore new states. This player can work for any scale and size since there is no computation needed.

Greedy Agent

A greedy agent is the next logical (and easiest) agent to make. The greedy agent, as the name suggests plays greedily by making moves that the agent thinks will make him win the game. At each state the agent makes a greedy decision. The greed here is to “capture the highest possible number of opponent coins”. We use this agent as part of our future training where we want our neural network to learn how to play against greedy strategies.

Minimax agent

A minimax agent is the one where the game is modelled as an adversarial search problem. One player assumes the role of maximising the score while the opponent tries to minimise it. They make choices that help their own goal and also prevent the opponent from meeting their goal. A minimax agent ideally looks at the whole game tree to the end and checks which move is the best.

Depth pruned agent

This is a variation to a minimax agent that can explore the tree at a certain depth. That is, instead of letting minimax run the whole algorithm, it allows the agent to go only a certain depth further from the current point. This enables the tree to be pruned and enables the selection of a move in an acceptable amount of time.

Alpha beta agent

This is another variation to minimax, where parts of the search tree are pruned to increase the efficiency of exploration. The algorithm cuts parts of the tree that are deemed to produce outcomes worse than previously visited nodes. The algorithm does that by calculating an alpha value and a beta value for each node visited. For this to work, there needs a good utility function to evaluate the score of a given node. Utility functions include area scoring, as well as territory calculation.

MCTS agent

MCTS is an algorithm that is made specifically for games like Go which have a huge state tree and where evaluation of a given non-terminal state is not easy. MCTS relies on random sampling to expand the search tree. In this algorithm, the agent plays full games to the very end by choosing random moves, then the result of the game is used to evaluate each move selected during the game. This evaluation is used for future plays to select the most promising moves.

Neural Network Agent

The mechanism of Neural Network Algorithm in Go is to predict the move according to a given board state. This is called a Go move-prediction model. The board is first encoded to several 2D feature planes, then fed into hundreds of convolutional filters, and a single final dense layer in the board size with softmax activation.

It is most common to use human-played games to train a Neural Network model for Go. In this way, the agent learns the way that humans play. However, most Go professional players only play on the 19x19 board, so it's only possible to train the Neural Network agent on the 19x19 board.

According to major Go move-prediction Neural Network models, we construct a model with 4 convolutional layers, each with a “same” padding which keeps the original board size 19x19. Then, all the filters are flattened and fed into a 19x19 dense layer with a softmax activation to predict the move for a given board state. We choose only 4 convolutional layers due to our limited computing resources and also to meet our goal of how far we could get with limited hardwares.

We also tried tweaking various parameters of the neural network. That includes trying different

- Number of layers
- Number of neurons
- Net structures (with or without pooling, padding etc.)
- Board state encoders
- Activation functions
- Amount of data
- Regularization
- Dropout
- Optimizers

Actor Critic

An actor critic network is a network that combines policy function (an actor) and value function (critic) into a single network. In our implementation, We decided to specifically use the advantage actor-critic algorithm. The reason is that in many implementations, all moves are treated equally, however, normally, not all moves the agent has chosen are good moves. Hence, there is a need for a function that gives higher credit to the good moves and overlooks the bad moves. The use of the advantage concept in actor critic network fulfills this requirement.

Our training process constitutes of the following:

Step 1: Generating self-play games

Initially, we created two randomly initialized agents (using the untrained actor critic network), and had them play against each other and collect all the game states, moves, advantages and all relevant data and store them.

Step 2: Training the agent on the self-play games

In this step we use the pre-generated self-play data from the previous step, and train the actor critic network (agent 2).

Step 3: Evaluating the agent for improvements

Finally, we have the previous agent (agent 1) play against the agent trained from step 2 (agent 2), if it appears that agent 2 has learned anything significant, then agent 1 will be updated to become the same as agent 2.

Step 4: Repeat.

Follows is a screenshot of the training process:

```

-----
Starting from agent: /content/drive/My Drive/G0/workdir/agent.hdf5
Total games so far 0
Won 237 out of 480 games (0.494)
Total games so far 1000
Won 241 out of 480 games (0.502)
Total games so far 2000
Won 232 out of 480 games (0.483)
Total games so far 3000
Won 243 out of 480 games (0.506)
Total games so far 4000
Won 242 out of 480 games (0.504)
Total games so far 5000
Won 249 out of 480 games (0.519)
Total games so far 6000
Won 263 out of 480 games (0.548)
New agent: /content/drive/My Drive/G0/workdir/agent_00007000.hdf5
Total games so far 7000
Won 230 out of 480 games (0.479)
Total games so far 8000
Won 214 out of 480 games (0.446)
Total games so far 9000
Won 222 out of 480 games (0.463)
Total games so far 10000
Won 231 out of 480 games (0.481)
Total games so far 11000
Won 269 out of 480 games (0.560)
New agent: /content/drive/My Drive/G0/workdir/agent_00012000.hdf5
Total games so far 12000
Won 247 out of 480 games (0.515)
Total games so far 13000
Won 252 out of 480 games (0.525)
Total games so far 14000
Won 300 out of 480 games (0.625)
New agent: /content/drive/My Drive/G0/workdir/agent_00015000.hdf5

```

ZeroGo Model

ZeroGo Model is an imitation to the AlphaGo model but can be run on light hardware. As AlphaGo, it has three components: a policy agent, a value agent and a Monte-Carlo Search Tree wrap-up. We'll introduce those components one by one.

Policy Agent

In general, the policy agent is based on the Neural Network agent, as it also accepts a board state and predicts the best move of the given board. But it could perform better through a reinforcement learning process including self-play games and learning from its own mistakes. Following is the full process of the play-train cycle:

1. Generate self-play games and store in experience buffers

The idea of reinforcement learning is to learn from its own experiences. We let the agent play against itself for multiple games. At the end of each game, we reward the agent with 1 if the agent wins or -1 if the agent loses. We then store each board state

with its corresponding stone placement and reward in the experiences buffers in the format of hdf5 file.

2. Train the agent with the experiences buffers

We then use the experiences buffers to train the agents. For each sample in the buffer, we have a board state, an action, and a reward of 1 or -1. We use the board state as the input just as we did in the part of neural network agent, the difference is in how we handle the label. When one-hot encoding the action, we place an 1 to the index of that action if the reward for this game is 1, and place a -1 to the index of that action if the reward is -1. In this way, the agent reinforces its intention to place the stones that led to its winning, and tries to avoid the actions that led to its losing.

3. Evaluate the agent to decide if we save the agent

After multiple rounds of training, we need to evaluate the agent to see if it has made some improvements. We do this by letting the learned agent to play with its previous version for multiple games, typically 100 or 500 games, and then use Binomial test on its winning rates to test if it has significant improvement that overperforms the previous agent.

There are two ways to use the policy agent in our ZeroGo model. First, it gives intuition when it sees the board, so that it provides guidance on which directions to explore when we do Monte-Carlo tree search. We call this part “u-value”. Second, we use it to simulate a game when we do rollouts at the leaf nodes of the Monte-Carlo tree. The final result of the rollouts will be a part of the “q-value”.

Value Agent

The value agent is used to judge a board position, which means to judge how much in advance or behind compared to its opponent. The value agent has almost the same structure with the previous network, except that it uses a single perceptron with a tanh activation to predict a value ranging from 0 to 1, indicating its advantage.

The way to train the value agent is similar to the policy agent that we train the agent using the self-play experiences. But it is not a good idea to use a binary reward 0 and 1 for the value agent. There are two reasons. First, it’s hard to tell how much one is in advance or behind from the opponent in the early game -- both players start with equal strength, so either giving the agent a reward of 0 or 1 confuses the agent. Second, even in a winning game, there are some

moves that are not that wise. We need to distinguish the good moves and the bad moves in each game to reward the agent accordingly.

Here is how we calculate the reward for each state and action. We first define the discounted reward to be the following:

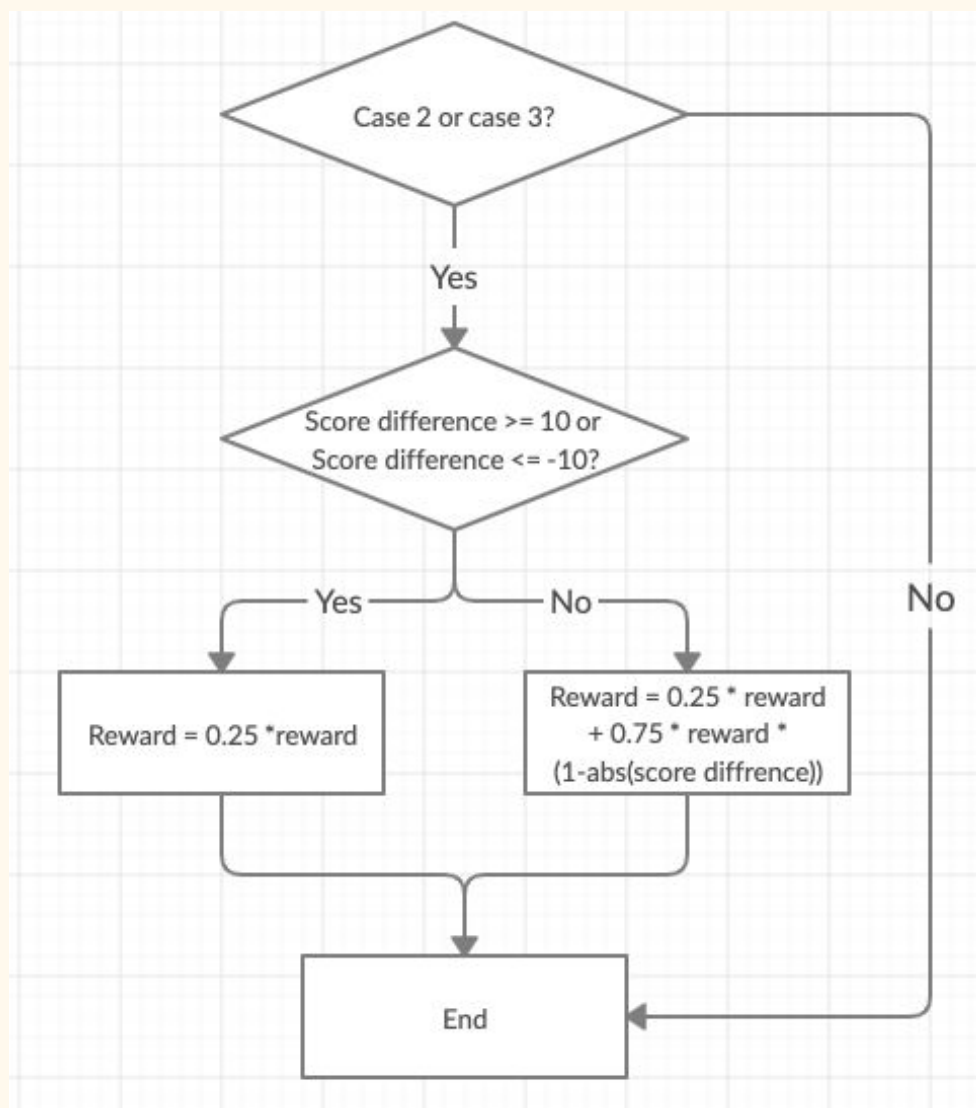
$$r_d = \begin{cases} 0.5 & \text{if } step \leq 30 \\ r * (step - 30)^2 * 240^2 & \text{if } 30 < step \leq 270 \\ r & \text{if } step > 270 \end{cases}$$

Where r is the original reward, 0 for losing and 1 for winning. By doing this, we penalize the reward for early games and emphasize the reward for the late games.

Second, we want to distinguish the good moves and the bad moves, and reward them accordingly. We do this by scoring the mid-game for each board state, calculating the score difference of the player and its opponent, and use the score difference to rectify the reward. There could be four cases for each game state:

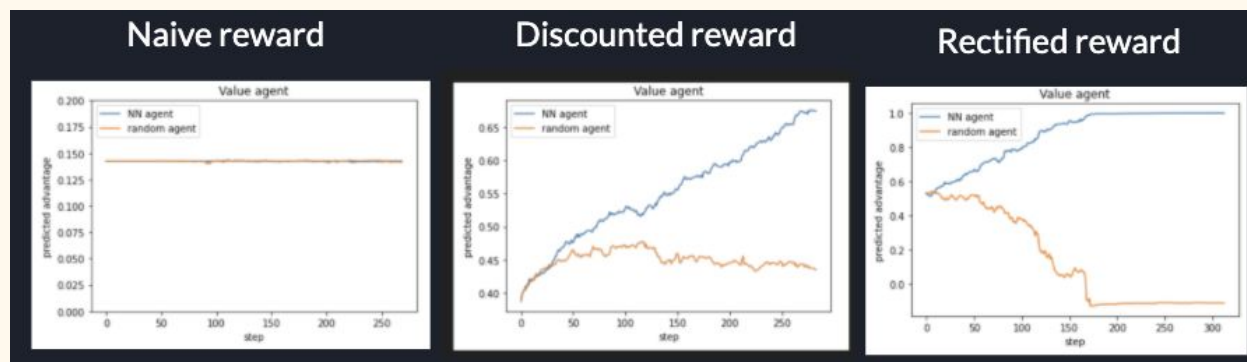
1. Final game result is winning, and the current score difference is positive.
2. Final game result is winning, but the current score difference is negative.
3. Final game result is losing, but the current score difference is positive.
4. Final game result is losing, and the current score difference is negative.

Among the four cases, we want to keep the reward for case 1 and 4, reduce the reward for case 2 and augment the reward for case 3. We achieve this through the following steps:



After discounting the early rewards and rectifying the rewards by the score difference, the reward gets closer to the real advantage value, thus making the training process more efficient.

The graph below shows the predicted value for a match between a powerful bot (Neural network agent) and a naive bot (random agent) from different versions of value agent. The left graph shows the agent trained by the naive reward of 0 and 1. It is unable to distinguish between the good player and bad player through the whole game. The middle graph shows the agent trained by the discounted reward but without rectifying by the score difference. It is able to distinguish the good and bad player to some extent. But the gap is very small even at the end of the game. The right graph shows the agent trained by the reward we discussed above, and it is able to accurately judge the board position starting from the early game until the end game.



In the ZeroGo model, we use the value agent to judge the board position for each board state. This is another part of “q-value”. Adding up the rollout result and the predicted value, we get “q-value” for each MCTS node.

MCTS

When humans play Go, people generally rely on two abilities to select a move -- instincts and reading ahead. In the ZeroGo model, we simulate those two with “u-value” and “q-value” respectively, and we use Monte-Carlo Tree Search to manage the process. Here is what ZeroGo does when it selects a move:

1. **Create a root.** Given a board state, the MCTS creates a root node for that board state.
2. **Expand.** Given the board state, we let the policy agent and the value agent to predict each move with a probability and a value, which is the initial “u-value” and “q-value” each move. Then the MCTS expands child nodes for each move and stores the u-value and q-value. It does not expand all possible childs due to efficiency issues. Typically it expands 10 nodes with the top 10 q-value.
3. **Select.** The MCTS selects a node according to the sum of u-value and q-value. It selects the node with the highest sum of values.
4. **Repeat step 2 and 3.** We repeat expanding and selecting nodes until the maximum MCTS depth is reached. Until then we reach a leaf node.
5. **Rollout.** We simulate a rollout from the board state at the leaf node. The policy agent takes turns to play both Black and White until the maximum rollout depth is reached or the game is finished. The rollout result comes as another part of the q-value.
6. **Update.** The MCTS back-propagates the rollout result to the parent nodes recursively, and updates the u-value and q-value.
7. **Repeat step 2 through 6.** The MCTS returns to the root to explore other possibilities. We repeat the process for multiple times which we call the “rollout times”.

8. **Select final move.** After fully exploring the search tree, we select a final move with the most visit count.

Here are the details to calculate leaf node value, u-value and q-value:

1. Leaf node value

$$V(l) = \lambda * value(l) + (1 - \lambda) * rollout(l)$$

In this equation, $value(l)$ is the result of your value network for l , $rollout(l)$ denotes the game result of a fast policy rollout from l , and λ is a value between 0 and 1, which you will set to 0.5 by default.

2. q-value

$$q(n) = \sum_{i=1}^n \frac{V(l_i)}{N(n)}$$

In this equation, $V(l)$ is the leaf node value as we just discussed, $N(n)$ denotes the number of visits of node n . The q-value basically takes the average value for each rollout.

3. u-value

$$u(n) = c_u \sqrt{N_p(n)} * \frac{P(n)}{1+N(n)}$$

In this equation, c_u is a fixed constant for all nodes to scale the utility, $N(n)$ denotes the number of visits of node n , while $N_p(n)$ denotes the number of visits of the parent of node n . $P(n)$ is the original probability to select the move that leads to node n , indicated by the policy agent.

The process for the ZeroGo model to select a move is actually quite similar to the human players. It starts to provide candidate moves with instincts, and tries to explore possibilities to figure out which candidate is the best.

Results and Analysis

In this section, we provide results and analysis of each agent developed. To provide more insight, we also provide the ranking of each agent whenever applicable. In general, due to the nature of the game and the huge branching factor, traditional agents fail in producing acceptable performance. Deep learning agents perform better than traditional agents as they learn from professional game plays. However, as explained below, the combination of both traditional and deep learning agents provides the best of both worlds.

Traditional Algorithm Agents

Greedy Agent (25-kyu)

Greedy algorithm fails for multiple reasons, but mainly for its nature where it is never willing to sacrifice any stone, regardless of its importance in the game. This makes key stones in less advantageous positions as the agent is never willing to sacrifice “disposable” stones in order to save key stones. Greedy gameplay also fails to look further into the possible consequences of its actions in the future. Though, the greedy agent can have successful game plays if it is against weaker opponents, this includes the random agent previously built as well as beginner human players, this is because the agent can make certain captures before the opponent can save itself. Our Greedy agent is ranked 25-kyu.

Minimax Agent (Ranking Not Applicable)

The minimax agent requires looking at the whole game tree, till the leaf nodes to determine the best next move, however, it is not computationally feasible to do that as the branching factor is huge and the number of states increases exponentially as the game size increases. The agent was experimented with a 5x5 board and even with such a small-sized board it fails miserably as it cannot play a single move using our regular hardware limitations.

Depth Pruned Agent (Ranking Not Applicable)

To handle weaknesses of minimax agent, we implemented the depth pruned agent where the depth of the tree is limited to certain depth. This has improved the speed of the agent selecting the next best move, however, since the agent only checks a shallow depth of the tree, the agent ends up neglecting some vital part of the tree that can be advantageous or it might end up exploring parts which seems “promising” but are actually a guaranteed loss. We can describe

the depth pruned agent as a short-sighted agent. The agent is still slow, for the board of size 5X5 we could only explore a depth of 2.

Alpha Beta Agent (Ranking Not Applicable)

To make the pruning more intelligent than simply cutting the tree to certain length, we implemented the alpha beta agent. However, to be able to make the alpha beta pruning effective and with minimum loss of important parts of the tree, a good utility function is vital in the process. However, in the game of go, finding a good utility function to implement with alpha beta algorithm is challenging. It is possible to use the agent territory or the number of stones as possible utility functions, However, they still cannot foresee possible outcomes of the next states. The outcome of a state is only determined after the game is complete. For these reasons, this agent also does not perform well.

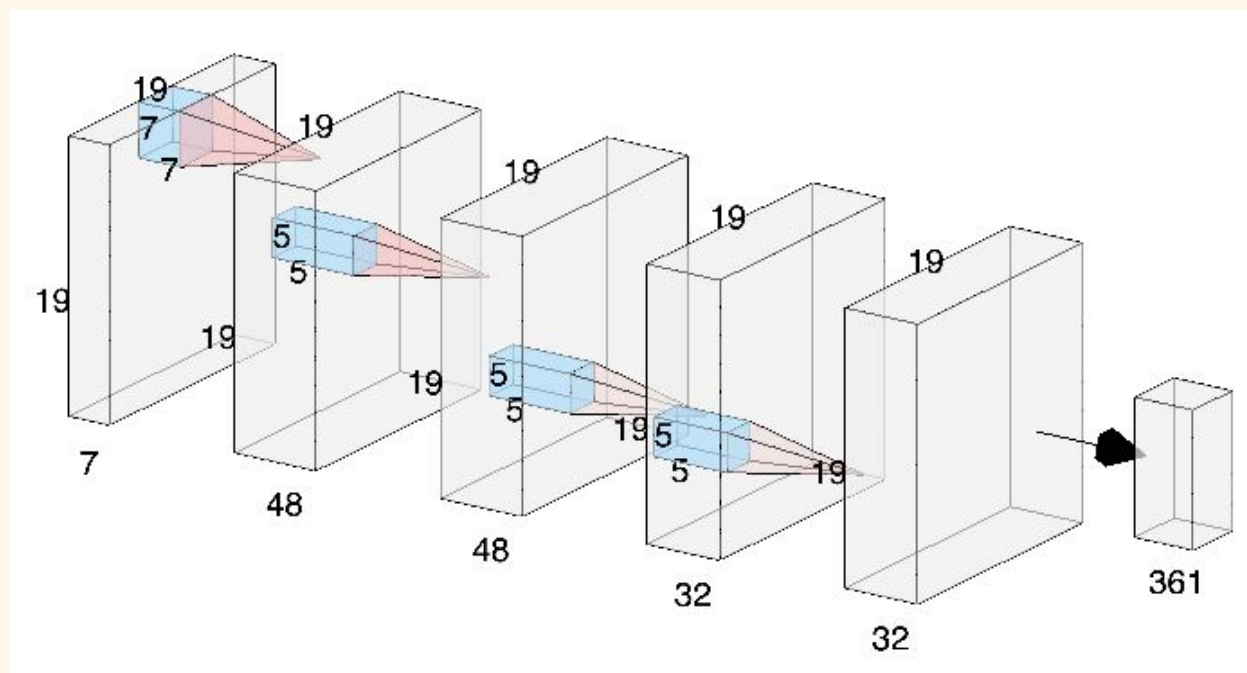
MCTS Agent (15-kyu to 25-kyu depending on parameters)

MCTS agent was then implemented to explore random parts of the game tree. The MCTS agent performs much better than all previous agents developed, and probably much better than all other tree-search algorithms. However, MCTS agent's performance is highly dependent on exploring as many vital parts of the tree as possible. This means that the more powerful hardware available, the better the expected performance will be. Our current agent is not close to being perfect or even usable for a full size game. The agent implemented is ranked 15-kyu to 25-kyu, depending on the algorithm parameters and our available hardware capabilities.

Deep learning Agents

Neural Network Agent (5-kyu to 15-kyu depending parameters)

After multiple attempts and hours of training we were able to find a NN which performed better than others. Following is the NN architecture that we were able to implement that achieved roughly 27% accuracy with 10 epochs of training with 5000 human-played games. This was done with a single Google Colab notebook with only 2 GPUs.



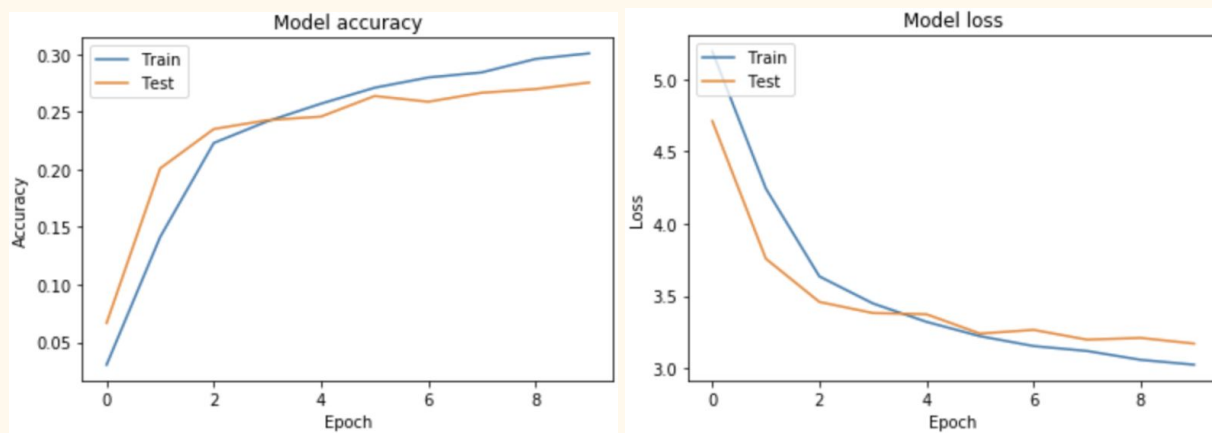
Structure description

1. Encoded seven-plane board
2. 48 filters of same-padding 7*7 convolutional layer
3. 48 filters of same-padding 5*5 convolutional layer
4. 32 filters of same-padding 5*5 convolutional layer
5. 32 filters of same-padding 5*5 convolutional layer
6. Dense layer with 361(19*19) number of classes

We also summarized some rules of thumb during the process of parameter tuning:

1. It is better to retain the original board size for the hidden layers.
2. Adding pooling layers might not be a good idea because the agent needs to make decisions on each position, but pooling summarizes the feature for an area of positions.
3. Either feeding more training data or adding dropout will fix the issue of overfitting.

Below is the learning curve of this neural network model.



The trained agent can reach 5-kyu level, which is close to beginning professional level. It shows decent strength to both attack the opponent's territory and defend its own territory. It is also able to select moves in a very short time (less than 0.1 second).

But the agent still has weaknesses. As it concentrates on only the current board state to select a move, it cannot forecast the future states. As a result, it focuses on immediate reward and neglects long-term development. While testing with the agent, we found that the agent is quite powerful in a local fight. But the human players easily win over the agent if they sacrifice some local stones but get a better global strategic layout.

Actor Critic Agent (Ranking Not Applicable)

The Actor critic network implemented was trained on a 5x5 board, with more than 15,000 self play games. Upon observing the way it plays after the training process is completed, we notice that the agent needs to train for a much larger number of self play games as the improvements it is experiencing is very small, especially considering the problem size (5x5) is much smaller than the target problem size (19x19). The agent ended up playing just a little bit better than a random agent after all these self-play games. This is expected since it is a pure reinforcement learning implementation, it is basically learning everything from scratch. For this very reason, we concluded that a pure actor critic agent is not very suitable for the problem in hand, and hence we decided to explore other solutions. However, we also believe the agent has the potential to improve given unlimited time and resources. It is not feasible to rank the agent because the agent was trained on a 5x5 board.

ZeroGo Agent (1-dan to 10-kyu depending on parameters)

The ZeroGo agent is an organic combination of all the methods we've tried -- a traditional MCTS wrap-up of two deep reinforcement learning agents. It could reach up to 1-dan level

ranking given enough time to explore the search tree. It performs much better than just a neural network policy agent in the following ways.

1. It can trade off between its instincts and future rewards by adding up the u-value and the q-value. In the actual matches with humans, it shows decent strength in both local combat and global strategic layout. Below is a screenshot showing its thinking to trade off between u-value and q-value:

```
=====Candidate moves=====
D12
num of visits: 77, q value: 0.3202, u value: 0.0635, sum value: 0.3838
A18
num of visits: 75, q value: 0.2681, u value: 0.1148, sum value: 0.3829
O3
num of visits: 48, q value: 0.2601, u value: 0.1237, sum value: 0.3838
G14
num of visits: 0, q value: 0.2700, u value: 0.1109, sum value: 0.3809
E13
num of visits: 0, q value: 0.2697, u value: 0.0923, sum value: 0.3620
```

2. It is able to judge local “living” or “dead” stones by reading ahead. This is an extremely important skill for all-level players. It can prevent the agent from placing suicidal stones in a dead area, and also save some work placing unnecessary stones to the living areas.

The ZeroGo model is not only the best on its strength, but also in some other nice features as follows:

1. It is able to set a threshold of value to make a decision to resign. It is extremely important to know when to resign in the game of Go, as Go has a very strict rule to finish a game (only when both players pass).
2. It is able to visualize its process of thinking. It is easy to read which child moves it explores, and the corresponding immediate rewards (u-value) and long-term rewards (p-value). It is helpful for humans to understand the board state.
3. It is flexible to show different levels of strength by tuning its parameters, such as rollout times, rollout depth, tree depth etc., so that different levels of human players can find their proper levels to play against.

It also has several weaknesses:

1. The thinking time is a bit long to reach a high level of strength. To reach the level of 1-dan, it takes about 200 seconds to select a single move, given the limited computational power we have.
2. It is not possible to reach the level higher than 1-dan, given the shallow network and limited games of training. We are not able to reach super-human power such as AlphaGo or AlphaGo Zero.

Limitations, Conclusion, and Future Directions

During the implementation and analysis process, we acknowledge that one of the limitations of this project is the computational abilities. It is a limitation set by us to explore what algorithms can achieve considering constrained hardware capabilities. Another limitation is the agreed-upon goal of this project which emphasized exploring many algorithms, this could have, in some way or another, compromised the time needed to fully explore the potential of each algorithm. We prioritized more exploration in the expense of deep exploitation.

Regardless of these limitations, we believe we successfully accomplished the goals of the project in which we investigate the possibilities given constrained hardware capabilities, the abilities and potential of various algorithms in tackling the problem, and lastly using less sophisticated agents to benchmark other agents.

For future extensions of this project, we believe that using game specific techniques (rule-based playing) and game characteristics can boost the abilities and performance of our ZeroGo agent. Also with enhanced hardware, the agent can provide stronger, better, and faster performance.

References

1. [https://en.wikipedia.org/wiki/Go_\(game\)](https://en.wikipedia.org/wiki/Go_(game))
2. https://en.wikipedia.org/wiki/Go_and_mathematics
3. <https://www.britgo.org/intro/intro2.html>
4. <https://www.manning.com/books/deep-learning-and-the-game-of-go>
5. https://en.wikipedia.org/wiki/Go_strategy_and_tactics
6. <https://www.freecodecamp.org/news/explained-simply-how-an-ai-program-mastered-the-ancient-game-of-go-62b8940a9080/>
7. <https://medium.com/applied-data-science/alphago-zero-explained-in-one-diagram-365f5abf67e0>
8. <https://deepmind.com/blog/article/alphago-zero-starting-scratch>
9. https://en.wikipedia.org/wiki/Go_ranks_and_ratings#Kyu_and_dan_ranks
10. Barratt, Jeffrey & Pan, Chuanbo. (2019). Playing Go without Game Tree Search Using Convolutional Neural Networks.
11. Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016).
<https://doi.org/10.1038/nature16961>