

Engineering Design Document

ZeroGo

ML Based Go Player

Prepared by:

Jing Ouyang

Sunny Patel

Wesam Alwehaibi



Table of Contents

Table of Contents	2
Background	3
The Go game	3
Go rules	4
Liberty	4
Ko rule	5
Scoring and end of game	6
Game play strategies by experts	7
By Bruno Curfs, 1-dan	7
General strategies	7
If you play Black	7
If you play white	8
Why Go as a project?	9
Goals	10
Prior research	11
AlphaGo	11
AlphaGo Zero	12
Proposed methodology and Ideas	14
Random Agent (implemented)	14
Greedy Agent (implemented)	14
Minimax agent (implemented)	14
Depth pruned agent (implemented)	15
Alpha beta agent (implemented)	15
MCTS agent (implemented)	15
Neural Network (In progress)	15
Actor Critic (In progress)	17
ZeroGo Model (Work in Progress)	19
References	20

Background

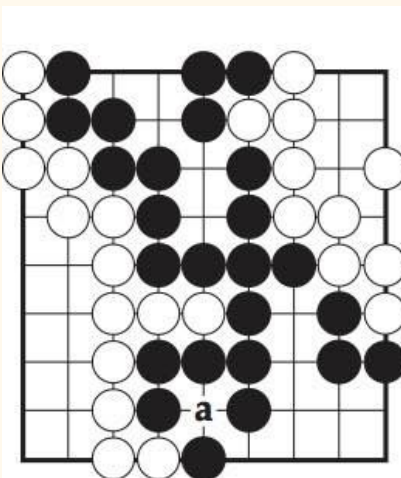
ZeroGo is an attempt to solve the game of Go!

The Go game

Go is an abstract strategy board game for two players, in which the aim is to surround more territory than the opponent. The game was invented in China more than 2,500 years ago and is believed to be the oldest board game continuously played to the present day. A 2016 survey by the International Go Federation's 75 member nations found that there are over 46 million people worldwide who know how to play Go and over 20 million current players, the majority of whom live in East Asia.

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 much as area 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 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.

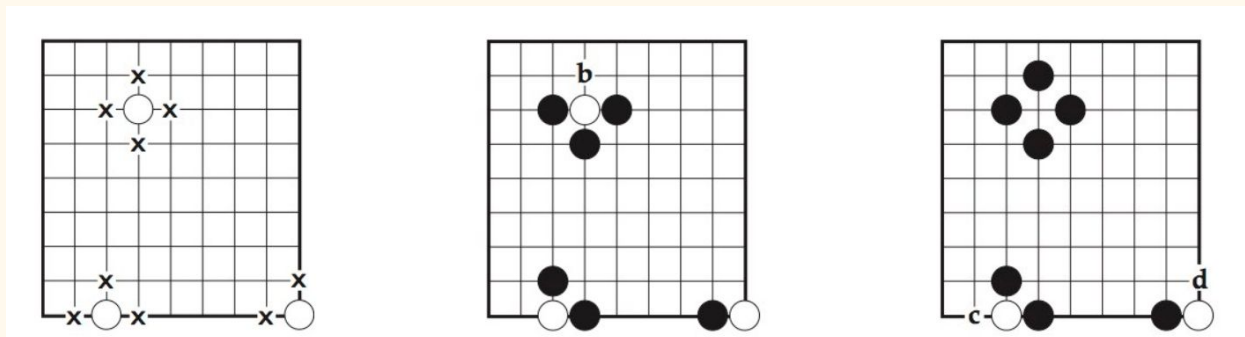


The stone on the position “a” will be captured by the surrounding black stones.

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

Liberty

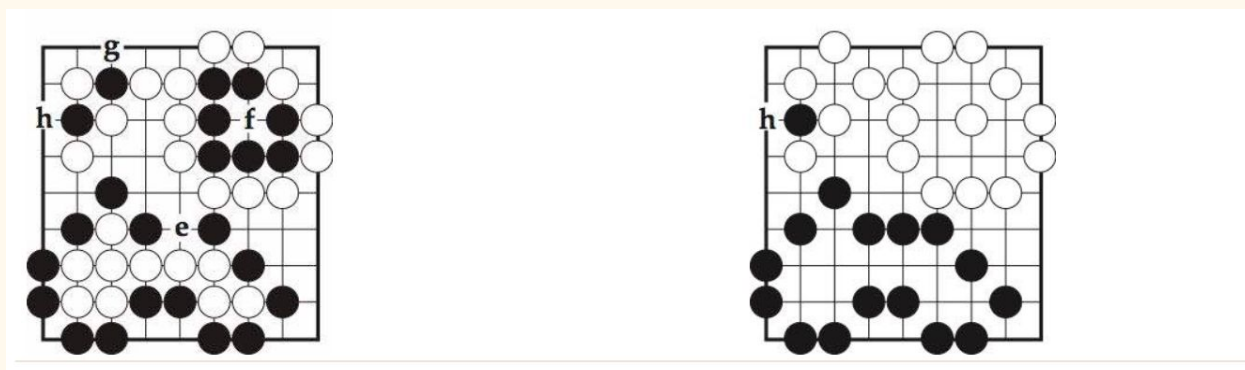
A liberty is defined as an empty position, either horizontally or vertically, around a stone.



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.

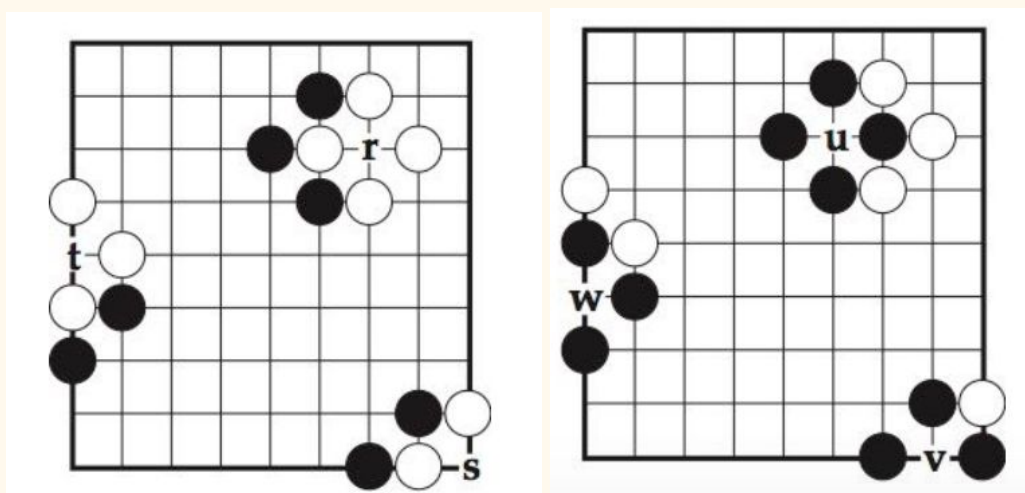
The right diagram shows that b was captured and in the next turn c or d could be captured too. Strings are stones of the same color connected to each other by being adjacent. Strings can be captured in the same way as individual stones. The entire string/collection is captured when there are 0 liberties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

Before 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.

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 Alpha Go. 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.



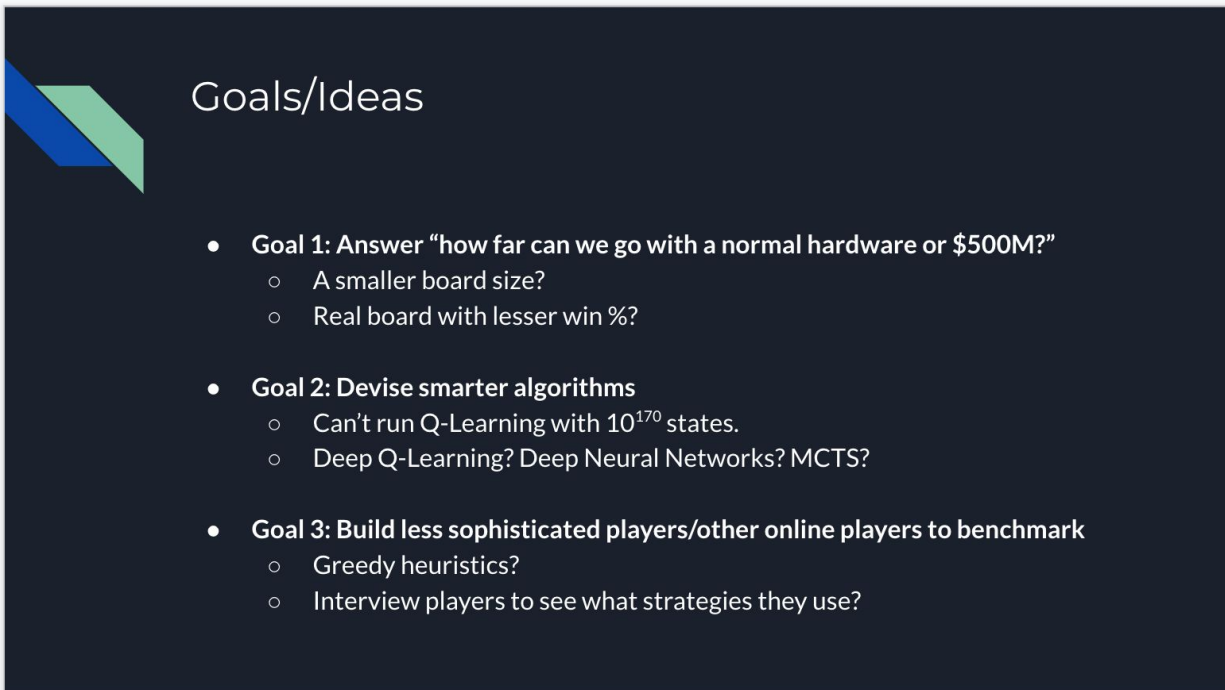
Why GoML is interesting for this class?

- 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 plan to use macbooks, google colab and Google cloud Platform if need be.

Prior research

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.

Proposed methodology and Ideas

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 aim to do a few things differently. As mentioned in the goals section above, we aim to leverage algorithmic power rather than hardware. While this approach may completely fall apart, we will still have takeaways that can be used to answer the questions we have. We have (or are going to) attempt the following methodologies and algorithms to tackle this problem statement.

These are certain algorithms that we are trying/will try:

Random Agent (implemented)

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 (implemented)

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”. The greedy agent performs slightly better than random agent because it can make certain captures before the opponent can save itself. This agent can also play slightly better than beginner human players. We will 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 (implemented)

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. This approach fails miserably because of the branching factor mentioned above. It is not possible to use such an agent for even a board size of 5X5.

Depth pruned agent (implemented)

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. For the board of size 5X5 we could only explore a depth of 2.

Alpha beta agent (implemented)

This is another variation to minimax, where alpha beta values are assigned to each state and the pruning is done intelligently. The player still performs badly. The reason is that a very challenging aspect of game of go is to evaluate the utility of a given state. I.e. given a random incomplete game/state there is no easy way to predict who will win from there. Hence this agent also does not perform well.

MCTS agent (implemented)

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. We implemented MCTS agent to see how it can perform. It did much better than Alpha-Beta and depth pruned, and probably much better than all other tree-search algorithms. But this is not even close to being perfect or even usable for a full size game.

Neural Network (In progress)

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 commonly 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 in 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

Below is our final Neural Network structure implemented using Keras.

```
ZeroPadding2D(padding=3, input_shape=input_shape, data_format='channels_first'),
Conv2D(48, (7, 7), data_format='channels_first'),
LeakyReLU(alpha=0.1),

ZeroPadding2D(padding=2, data_format='channels_first'),
Conv2D(48, (5, 5), data_format='channels_first'),
LeakyReLU(alpha=0.1),

ZeroPadding2D(padding=2, data_format='channels_first'),
Conv2D(32, (5, 5), data_format='channels_first'),
LeakyReLU(alpha=0.1),

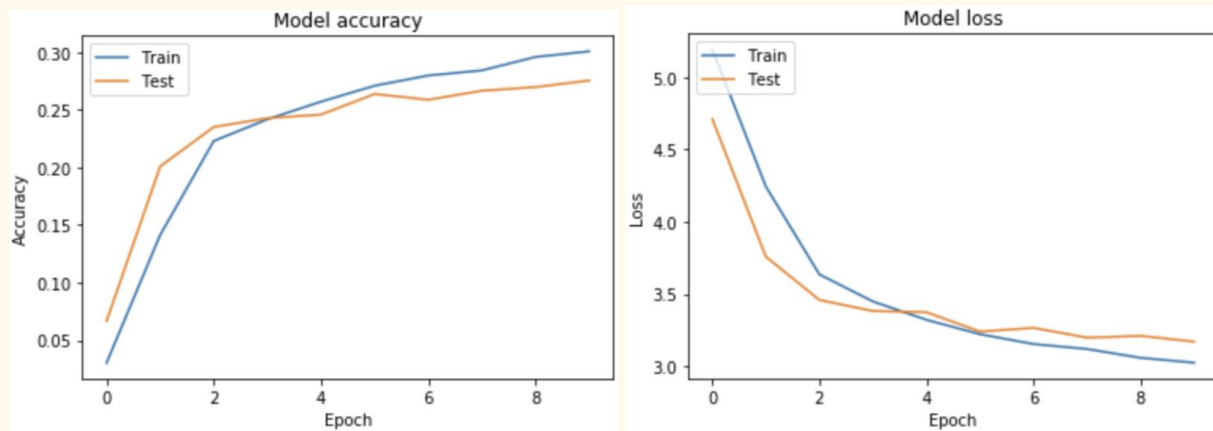
ZeroPadding2D(padding=2, data_format='channels_first'),
Conv2D(32, (5, 5), data_format='channels_first'),
LeakyReLU(alpha=0.1),

Flatten(),
Dropout(0.25),

Dense(num_classes, activation='softmax')
```


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. This was done with a single Google Colab notebook with only 2 GPUs.

Below is the performance evaluation of this Neural network.



Actor Critic (In progress)

We also started experimenting with an actor critic network. 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.

Currently, our training process constitutes of the following:

Step 1: Generating self-play games

Initially, we create two randomly initialized agents (using the untrained actor critic network), and have 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 same as agent 2.

Step 4: Repeat.

The current Actor critic network implemented was trained on a 5x5 board, with 15,000 self play games. Upon observing the way it plays after the training process 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. This is expected since it is a pure reinforcement learning implementation.

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 (Work in Progress)

Next, we plan to use a combination of reinforcement learning, NN, search based techniques, as well as utilizing some game heuristics. The general idea is to have a neural network agent that is already trained on historical games to use reinforcement learning and search based algorithm to wisely select its moves. We also plan to use game heuristics (like the symmetry, specific patterns of the game) to improve the agent and have it make better decisions. We are hoping to achieve a higher accuracy than Neural networks.

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>