GO A.I. Final Report

Summer 2021 & Summer 2022

Jiageng Zheng

https://github.com/jiz322/GoCSE392

In the first part of this report (summer 2021, before page 23), we explore the algorithm, effect of parameters provided by the AlphaGo-Zero framework. We embedded the rule of the Go game to that framework and trained a Go game agent playing on a 5*5 sized game boards.

In the second part (summer 2022, after page 23), we trained a Go game agent playing on a 9*9 sized game boards. We optimized the system applying parallelism, proposed modifications to the algorithm, and found better parameters for the decision search and network training.

1. Back to the initial proposal of this study

Ideal Outcomes:

1. A programming project: For a 9x9 game board, input current game status, and then the program should output recommendations for the next move.

2. Project Report: Discuss the details of the methodologies used in this project; evaluate these methodologies; evaluate the results using real-world online Go platforms.

Hope to learn:

1. Learn various methodologies used by Go's AI. Understand the aims and results of these methodologies and why they work.

2. Gain experience in programming for decision-making AI.

3. Gain experience in AI architecture design.

4. Understand the similarities and differences between Go's AI and other AI implementations; also understand the causes of these differences.

Outcome 1:

By combining and adjusting the Alpha-Go Zero Framework (Nair, from GitHub) with Go game board (Huang, from GitHub), an understandable Go AI project is constructed and will be able to train with 9*9 game board. However, due to the time limit and the algorithm and parameters far under perfectly optimized, the 9*9 game board agent is still far away from defeating advanced human players. Instead, our agent for the 5*5 games is decent.

Outcome 2:

The report is the following content of this document. Details of methodologies will be discussed and evaluated in section 4. Since the 9*9 game board is under development, and no one will play 5*5 games, there will be no evaluation using an online platform. However, 5*5 games are still playable under a certain change of rule. We experimented with a Go game between two Humans: the agent helps only one of them.

Hope to Learn 1:

More understanding about the Monte-Carlo tree is gained by reading other people's works, debugging the program (traversing the recursion of the search function), and adjusting tree parameters, adding noises and the temp to modify the expression of the original Upper Confidence Bound algorithm. Section 4 covers these details.

Less understanding of neural network design is experienced. Nevertheless, this study still provides an experience of how to use a neural network. It also induces thoughts about how the loss function is making sense; this study also provides an opportunity to witness the decreasing value of loss function and thinking about whether this decrease is a good thing or not.

Hope to Learn 2:

The debugging process hit this point. Although an interface is provided by the original framework saying the programmer only needs to implement one "gamelogic" file, the complexity of the Go makes this improbable: you have to change other files in this framework to get the Go rule work. An instance is the rules to end the game (causes a change in the search function of MCLT.py). Another instance is the rule of 'ko' (causes a change in the getActionProb function of MCLT.py). It will be documented in the clean-up version of this code. (still cleaning it up)

Moreover, more of these experiences are gained over implementing additional features into this framework and the trail of challenging the original framework.

Hope to Learn 3:

In speaking with the AI architecture design, an investigation of the combination of Neural Network and Monte-Carlo Search provides a general understanding of A.I agent for all of these adversarial games. A thorough investigation of neural network design is lacking. Hoping to get this experience in future work.

Hope to Learn 4:

In the discussion of "Hope to Learn 2", the difference caused by game rules has already been mentioned. Compared to Othello or tic tac toe implementations, the complexity of Go also causes a difference. It might be easy and fast to train a tic tac toe agent with the original framework, but as for the Go, the Dirichlet noise has to be added, or an overfitting problem impedes the training process.

Comparing the Go to other games like poker, it is unfit for poker to use the same framework (mainly unfit with Monte-Carlo Search) because the Go is an entirely decisive game. In contrast, poker considers possibilities like the difference between the Mini-Max tree and Expectiminimax.

2. Brief Introduction of GO

The English word "GO" is retrieved from its Japanese pronunciation, whereas its intrinsic meaning, at least in Chinese, is "the Game of Surrounding," matching with the rule of capture.

The standard game board is 19*19 grid, where black and white, being opponents with each other, alternately put their stone onto the board. Each grid can be empty or occupied by black or white, so there are 3 to the power of 361 states. One may think there are a factorial of 19*19 possible sequences that lead to these states, but with the rule of capture, the total amount of possible sequences is infinite.

It is impossible, with the compute power today or near future and probably all future consider that possible sequences are infinitely many, to enumerate all possible sequences in order to find a path that guarantees a 100% win rate. However, using the Monte-Carlo method as an improvement factor, facilitated with Neural Networks, the emergence of Alpha-GO has demonstrated to us that the A.I agent can surpass the best human in the game of Go, and a heuristic search can make achievements even in the game theoretically has infinite possibilities.

This introduction will not cover all detail about Go's rules or strategies, but it intends to cover all necessary knowledge for understanding this report.

2. 1. Game Rules (copy and modified from TrompTaylorRules)

1. Go is played on a 19x19 square grid of points, by two players called Black and White.
2. Each point on the grid may be colored black, white or empty.
3. Starting with an empty grid, the players alternate turns, starting with Black.
4. A turn is either a pass; or a move that doesn't repeat an earlier grid coloring.
5. A point P, not colored C, is said to reach C, if there is a path of (vertically or horizontally) adjacent points of P's color from P to a point of color C.
6. Clearing a color is the process of emptying all points of that color that don't reach empty.
7. A move consists of coloring an empty point in one's own color; then clearing the opponent's color, and then clearing one's own color.
8. The game ends after two consecutive passes.
9. A player's score is the number of points of her color, plus the number of empty points that reach only her color.
10. The player with the higher score at the end of the game is the winner. Equal scores result in a tie.

2. 2. Balanced Game

The Go game is not balanced. The first move player will have an advantage. To balance out this advantage, for both 19*19 board or 9*9, the first-move player needs to gain six more

scores than their opponent to win. For 5*5 games, based on an advanced amateur player, the first-move players can wipe out their opponent, occupy all 25 grids, and have a 100% win rate. It is necessary to add other rules to balance these 5*5 games, like the gobang tournament, which gives second-move players a chance to exchange the black and white.

3. Project Overviews

3. 1. Introduction of the Project

There are two famous Go A.I agents existing on Github, the Leela-Zero, and the Keta-Go. We investigate the Leela-Zero in this project, and its skill of Go is highly above an amateur player. However, Leela-zero's codebase is beyond the level that the researcher of this report can shortly understand. Moreover, volunteers throughout the internet have finished the process of training. Fortunately, an understandable project, the Alpha-Go zero framework (Nair), is found. Examples of Othello, tic tac toe, and chess are provided within the framework, with game logic code and the trained neural network weights.

Another indispensable element that makes it possible to train the Go game is a GitHub work (Huang), containing understandable Go logic. Encapsulating this GitHub work into the Alpha-Go Framework, going through debugging, it is ready to train an agent for Go.

3. 2. Data Produced

1. There is one output txt file for each run of training.
    - Timed data. It includes the count of world-clock time of each self-play iteration, neural network training, and time spent in the Arena for deciding whether to accept new weight;
    - Network training data. It includes the value of loss function for both p and v before and after network training.
    - Arena data. It contains "win/lose" data of the current network weight over the previous one. The number of wins of the first move player. For two-stage go optimization, an average of turns in a game, an average of captures, and a standard division of score differences are also calculated.

2. We collected a network for each training update. Networks' weight can be loaded into the pit.py, a script provided by the original framework to create an "Arena " that each network, human, and a random player can "pit" with one another, outputting the data as the Arena data described above. Additionally, we constructed tournament.py for the experiment for cross-comparison between a group of different network weights.

4. Framework Methodologies

4.1. Overview

The AlphaGo-Zero Framework starts training with the first iteration of self-play. During the self-play, game examples are collected, including the winner of this game and all the game states. Using these data, it launches the training process of neural networks, regressing the loss function for v and p. After that, an Arena Competition is held between the network weights before and after training. If the new weight wins more games than the old, the change will commit, and the next self-play iteration starts, using network weights whoever is the winner in the previous Arena.

4.2. Input and Output of Neural Network

Taking the game board's current state as an input, the neural network returns two values. The p, policy vector, has the same length as the numbers of all possible moves. A more significant p for an action "a" means "a" is preferred. If we let the neural network play the game alone without Monte-Carlo search, the value of p, by itself, decides all moves for the agent without taking value v into consideration.

The "v," a win rate value scaled from 0 to 1, is the expected win rate provided by the neural network. This "v" is used in the Monte-Carlo search to determine the Q value.

4.3. Monte-Carlo Search

The Q value of the Monte-Carlo search is initialized with **p**; after traversing several steps forward, adding or subtracting the v value of future states will update the Q value. By combining the usage of **p** and v, the Monte-Carlo search provides a visionary action rather than depending solely on **p**.

First, we traverse through the Monte-Carlo tree for an overview. Assume we are searching on a 5*5 Go game which has 26 actions. For simplicity here, we assume an infinitely big EPS value so that the first 26 searches will create 25 nodes adjacent to the root having Q value: Q(s_0, a)=v(s'), where the " s' " is a state after the first-move player take action a.

Now, imagine we will do 1000 simulations for the first move in a game. In the first search, the algorithm detects this is an unrecorded state. It retrieves the policy vector **p** for this

state and ends the first search. In the second search, always starting at the current state, it computes the "a_t" value with vector p. Then it calls the search function again with the next state after the action "a" whichever has the largest "a_t" value. The returned value of the recursive search function is assigned to the Q value. Now we look at the expression of U. The Cpuct is constant; the numerator and denominator on the right part of this expression are both 1; Q values are initially 0; the value of U is depended solely on the P(s, a): it will select the move with the highest policy value in vector **p**.

$$a_t = \underset{a}{\operatorname{argmax}} \left(Q(s_t, a) + U(s_t, a)\right)$$

$$U(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

Then it will search on the state s' (the state s after move a), finding this is an unrecorded state, retrieve the policy vector p and return the -v to the v of the previous search. The previous search will pass the v (a value ranged from -1 to 0) as its first Q value.

Since we assume an infinite large EPS (explaining EPS: the N(s,b) is initially zero, so an Epsilon constant is added along with the sum of N(s,b) to avoid U values to be all zeros), the 3rd search through the 26th search will repeat the 2nd search with different actions until all actions are initialized with a negative Q value.

At the 28th search, the a_t expression will decide the largest "a_t." Then the algorithm will get the state s' after making this move and then call a recursive search on the state s.' Since s' is having the p' recorded, it will do a recursive search again based on the p', leading to state s". The vector p" for s" will be recorded. It then returns the win rate value v". The Q value for the first "a_t" will become (-v'+v")/2, an average of all newly discovered win rate values for each simulation.

Carefully observing this expression of action value, we find that in the beginning of simulations, even a move with a lower policy value can be searched for plenty of times. However, the most searched move is the final decision, so if the number of searches is infinite, the final decision will be the one with the highest Q value.

Notice that in the actual training, the EPS value is set to 1. A positive infinity value of EPS will require the times of simulations to increase exponentially. That will eliminate the advantage of this heuristic.

Related to knowledge from CSE327, Monte-Carlo search has the feature of a heuristic search because it searches on the moves that have the highest "a_t" value. It also has the feature of a Mini-max Tree because it computes the Q value by adding up positive and negative values of v, which values of v is what the neural network returns after taking future states as its input.

5. Observations

Observations are highly conceptual. They lack control variables to be actual experiments. They cannot make a rigorous conclusion from them, but they are still helpful because they are observed plenty of times and have a fairly decent chance to be correct.

5.1. Observation: Learning logic

At the early stage of training, the average number of total turns grows from 10 to 50, halting at 50 for a while, and then it decreases from 50 to 25 (the 10, 50, 25 are roughly speaking). The total number of turns 10 indicates both agents choose "pass" at the early game; the value of 50 indicates both agents refuse to choose "pass" even when they are making suicide (figure below) move. For two players who know when to pass, the expectation of turns is around 25. This observation indicates that the agent learned how to make timely "pass" decisions.

Picture: Both players should pass, or they will make suiside move.

5.2. Observation: overfitting problem over balanced game

When training a balanced game(see 2.2), it is observed that it takes a long while for the agent to learn to make timely "pass" decisions. If the players are new (still trying to understand the game rules), the first-move player does not have much advantage: an unbalanced game is more balanced for two beginners. Printing out the win rate value v verify this guess. The v value for black is around 5%.

The increase of difficulties in estimating the win rate v might be another cause of this problem. In a balanced game, a winning game for white might also be a losing game for black. The neural network only takes the current state. It does not know who is black and who is white.

To reinforce this guess, we print out the loss value of win rate v. Training balanced game results in v = 0.25 after 70 iterations, compared with unbalanced training -- average v=0.2 for 20 iterations. An unbalanced game has a lower loss value of v than a balanced game, even after far fewer iterations. However, without a carefully designed experiment with control variables, we cannot make rigorous conclusions.

5.3. Parameters and Tuning of Parameters

"**numEps**"

The "numEps" is the total number of games in one self-play iteration. The larger "numEps" means a self-play iteration will generate more game examples, and time spend in self-play will also linearly increase. An ideal number of the "numEps" should consider both portions of time used for self-play and network training. If a small number of Episodes can produce a decent percentage of accepting rate in the Arena, then a small "numEps" is okay. However, if Arena keeps rejecting new network weights, it is time to increase this parameter to save time from the useless network training.

"**arenaCompare", "updateThreshold**":

The "arenaCompare" is the total number of contests between previous network weights versus the current one. The "updateThreshold" is a percentage representing the lowest bound of win rate for accepting a new network weight.

After days of testing, the simulation in arenaCompare, the update threshold, and the "arenaCompare" are set to 2, 0.55, and 100, respectively. Unfortunately, we do not have enough time to experiment on this setting before July 5th, but it is still worth keeping it in this report, waiting for testing in the future.

"**Cpuct**":

It is the constant for the PUCT algorithm discussed in 4.2. A large Cpuct value results in having more moves selecting second, third, or other choices having lower p values in training data. It is hard to say if that is good for training or not. In this experiment, we trust the Alpha-Go Zero paper and set it to 1.1, but for a 5*5 game board, the best "Cpuct" value is possible to be different with the 19*19 games, making "Cpuct=1.1" a suboptimal choice.

"**numMCTSSims**":

The numMCTSSims" is the total number of Monte-Carlo searches performed for deciding each move. A larger value means a more accurate computation of Q value, and the self-play will consider more possible moves. It linearly increases the self-play time. In the experiment (section 7.1), we can conclude that this value, at least for the starting point of the train, the larger does not mean the better.

"**temp**":

The temperature constant is a threshold to increase the quality of examples created by self-play by disallowing the self-play players to choose moves other than the optimal option. We have not got a chance to try this variable after adding Dirichlet Noise before July 5th. However, before adding Dirichlet Noise (section 6.1), this parameter was once set for 15, and it causes severe overfitting problems.

6. Modification Made from the Original Framework

6. 1. Noise

In the Monte-Carlo search, 25% of Dirichlet Noise, parameter set to 0.03, is added to the root policy vector returned by the neural network. The effectiveness of the noise is measured in section7. We also add the noise mentioned in the Keta-Go paper as its "first optimization" (Wu, 2020).

6. 2. Heuristic change

The formulas below are (part of) PUCT algorithm described in the Keta Go paper (Wu, 2020) and Alpha-Go zero paper (Antonoglou, 2017).

$$\mathrm{PUCT}(c) = V(c) + c_{\mathrm{PUCT}} P(c) \frac{\sqrt{\sum_{c'} N(c')}}{1 + N(c)}$$

$$U(s,a) = c_{\mathrm{puct}} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$

The disagreement here is how to interpret the sum of N(c') and N(s, b): whether or not the c' includes c, b includes a. For convenience, we call the two interpretations" included saying" and "excluded saying." The explanation in the original paper is not found. The PUCT algorithm is different in other works because the one in AlphaGO zero paper has made modifications. For example, in The Upper Confidence Bound (UCB) Bandit Algorithm:

$$A_t = \arg\max_a \left[Q_t(a) + c \sqrt{\frac{\log t}{N_t(a)}} \right]$$

The t stands for the sum of actions at all times. This finding provides support for the included saying.

It is more important to understand the exact difference between the included saying and the excluded saying: the included saying we need more simulations than the excluded saying to make moves with a lower initial neural network policy value p. For example, move π1 has p = 0.9, and move π2 has p = 0.1. The π1 is initially preferred. Now let us assume the Q value (V(c) in keta paper's expression) for move1 remains as a constant c after nine times searches on it, and CPUCT=1 (since π1 is preferred, there is no search on π2). Now let us compute the PUCT value.

Inclusion saying
π1: PUCT = c + P(c)*0.3
π2: PUCT = c + P(c)*1
Exclusion saying
π1: PUCT = c + P(c)*0.1
π2: PUCT = c + P(c)*1

Now it is clear to see that π1 is adding with a lower value. Under included saying, the next search will still be on π1 since the 0.9*0.3 is still larger than 0.1*1, but under the excluded saying, the 0.9*0.1 is smaller than 0.1*1, which means the next search will be on π2.

Above is an over-simplified example, and even if we search infinite many times, if the Q value of π2 cannot surpass that of π1, π1 is still the preferred move in the end. The effect of whether or not to encourage searching on the move with a lower initial p is unknown. There will be experiments about it in section 7.

6. 3. Two Stages Running for Balanced Game

In section 5.2, the difficulty of training a balanced Game is discussed. By observing the pattern of how the agent learns, we develop and test a two-stage training with score reward. The two stages consist of stage1, a quick training on an unbalanced rule, and then stage2, which will switch to a balanced game rule with more simulations of each move.

Besides, a reward is added to training examples if a game is a big win. For example, if black gets all 25 territories, this game example will be added ten times. With more examples of big wins, we expect the neural network to play the game more aggressively.

The parameters setting are as follows.

Stage1:

Rule: unbalanced game rule.
“numEp”=1000;
“arenaCompares”=8; “threshold”=0.6;
“sim”=2;
“Cpuct”=1.1

“temp” =Infinite.

When in Arena avg capture <n^2, avg turns > n^2 - n, next stage

Stage2:

Rule: balanced game rule.

“numEp”=100;

“arenaCompares”=8; “threshold”=0.6;

“sim”=52;

“Cpuct”=1.1

“temp” =Infinite.

Switching from Stage1 to Stage2

The two-stages training aims to expedite the learnings of making timely "pass" decisions. The average turns, average captures, and standard deviation of score differences are computed and collected. The stage will switch after the games in Arena meet all three conditions:

1. The average turn is larger than 20, indicating the agent is unlikely to "pass" at the early game.
2. The average capture is less than 25: if blacks capture all the whites (10-20 captures for 5*5 game), this black the agent will choose "pass" in avoiding suicide moves.
3. The standard deviation is larger than 6. This standard guarantees that the agent is attempting to capture its opponents.

7. Experiment: The Tournaments

This section includes all experiments that make comparisons between multiple network weights. It uses the original version of Monte-Carlo Tree. The number of simulations, number of total competitions, and participants may vary between different tournaments. However, all CPUT (searching rate) is set to 1.1, and the temperature variable is set to infinitely large.
All training and experiments ran on Lehigh's Magic01 machine.

7.0.1 Introduction of Network Players

1. Challenger and Defender group

   Players: {cha_27, cha_52, cha_77, cha_102, def_27, def_52, def_77, def_102}

   Control Variables:

   a. CPUCT = 1.1

b. Trained with 20 iterations
c. Each iteration has 100 Episodes.
d. When comparing with previous weight training, eight games are played and new weight wins 5 to accept.
e. All codes, parameters not mentioned in the “difference” indicate they remain the same.

Difference:

a. The players having "cha" prefix are trained using the modified way to comply with the excluded saying (see Section 6.2), while "def" indicates it is trained with the original code (included saying).
b. The numbers "27, 52, 77, 102" are simulations used during training. The more simulations, the more accurate the Monte-Carlo will be about to express the network.

2. Multi-Process group

Players: {onefold, threefold, fivefold}

Control Variables:

a. All that is mentioned in the previous group.
b. The number of simulations is 150 times for each move.
c. Instead of starting a new training, the training builds on the def_27 player in the previous group.
d. Same Dirichlet Noise is added for all

Difference:

a. Onefold runs a single process, whereas "xfold" runs a multi-process (x processes as total) execution made by python's pickle library.

3. Dirichlet Noise Comparison group

Players: {onefold, no_noise} notice that onefold also in previous group

Control Variables:

a. All the same as the previous group.

Difference:

a. The no_noise play adds no Dirichlet Noise.

4. The Best Weight Group

Player: {best}

No controlled Variable. This weight, called the "best," is trained using various MTCL parameters and a mixture of modifications implemented through the experiments and explorations. At the time of a week, the win/lose from the Arena data repeatedly returns an equal value, and the win rate of the first-move player is constantly 100%, indicating that the room for improvement is not much.

5. One-stage and two-stage Group

Players: {one-stage, two-stage}

All parameters are discussed in section 6.3. As for the total iterations, the one-stage training is trained with 70 iterations, whereas the second stage of the two stages is trained with 50 iterations.

7.0.2. Types of Tournament

1. The Instinct Tournament

No Monte-Carlo. A neural network makes all decisions. This tournament is a pure measure of the network weight on its own.

2. The Training Tournament

The number of simulations is set to 2, so the first game is the same as the instinct tournament. However, by playing 100 Games, the simulation accumulates so that the Monte-Carlo Search is in place to help. It is precisely how the training handles whether to accept the new network weight.

Each player plays a game with all other players in each tournament, and then alternate first-move to second-move, play the game again. In a contest group of size 5, there will be 5*4*2 numbers of contests, and each contest may contain multiple games. For the contest in the Instinct Tournament, only two games for each contest because all the games will produce the same result: the instincts initial policy never changes without a varying number of Monte-Carlo simulations. As for the Training Tournament, the number of games is set to 100.

7. 1. Simulation Tournament

It is a training Tournament (see 7.0.2.2) held in between network weights having only "numMCTSSims" as its variables. Each player plays 50 games with each of the other players, and there are 600 games for each player to play. We recorded the number of wins.

Contest group1:	{cha_27,	cha_52,	cha_77,	cha_102}
Wins:	{201,	344,	325,	330 }
Contest group2:	{def_27,	def_52,	def_77,	def_102}
Wins:	{201,	259,	365,	375 }

Analysis of Result:

A larger value of simulation indicates a more accurate result a Monte-Carlo Search provides. The increase of Wins from 27 simulations to 53 simulations is the most obvious. In the defender group, the difference between 77 simulations and 102 simulations is trivial. In the challenger group, the 52 simulations network weight even outperformed the 77 and 102. These results indicate that at the starting point of the training, doubling the times of simulations doubles the time it takes but does not guarantee a better result if the original times of simulations are good enough.

7. 2. Challenger-Defender Real Training Tournament

It is a training Tournament (see 7.0.2.2) held in between network weights having only the modification of algorithm as its variables (included saying v.s.excluded saying); (see 7.0.1.1). There are 200 games for each player to play. We record the number of wins.

Contest group1:	{cha_27,	def_27}
Wins:	{118,	82}
Contest group1:	{cha_52,	def_52}
Wins:	{105,	95}
Contest group1:	{cha_77,	def_77}

Wins:	{81,	119}
Contest group1:	{cha_102,	def_102}
Wins:	{109,	91}

Analysis of Result:

In no way can we say whether it is better to use challengers' approach to training or use defenders'. The impact of whether to "include" or "exclude" is still obscure.

7. 3. Parallel Real Training Tournament

This tournament is a training Tournament (see 7.0.2.2) held in between network weights having only numbers of processes as variables (introduced in 7.0.1.2). There are 400 games for each player. We recorded the number of wins.

Contest group1:	{onefold,	threefold,	fivefold}
Wins:	{184,	191,	225 }

Analysis of Result:

The five-process training has the most Wins, and threefold is slightly better than one fold. Beyond this result, it is worth noting that even though all three players are trained with 20 iterations, onefold ends up using its 20th network weight as the best one, for threefold is 18th, and fivefold is 15th. The fivefold reach to the current skill level even earlier than threefold and onefold, demonstrating the multi-process is efficacious.

7. 4. World-scale Training Tournament

A training Tournament and a first instinct tournament (see 7.0.2) are held between all network weights of unbalanced games. There are 2200 games for each player in the Training Tournament and 44 games for the First Instinct Tournament. We recorded the number of wins.

Training Tournament Result:

{cha_27, cha_52, cha_77, cha_102, def_27, def_52, def_77, def_102, one, three, five, best}
[610 , 861 , 975 , 964 , 579 , 700 , 939 , 1098, 1513, 1574, 1692, 1695]

First Instinct Tournament Result:

{cha_27, cha_52, cha_77, cha_102, def_27, def_52, def_77, def_102, one, three, five, best}
{ 8, 20, 20 , 26, 8, 14, 20, 18, 34, 24, 34, 38}

Analysis of Result:

This experiment reinforces the result of 7.1 and 7.2: The "cha" and "def"s pattern remains the same, and 52 seemed to be a reasonable time for "cha"s simulation, and 77 for "def." The best is still the best, but the fivefold is very close to it.

7. 5. Balanced World-scale Tournament

This training Tournament (see 7.0.2.2) was held between selected network weights, balanced and unbalanced games. They will play a balanced game. It seems unfair for network weights trained with unbalanced games because they tend to win the game only by one. When the number of simulations is small, the decision made by the original policy vector p is impossible to win as a first move player. There are 2200 games for each player.

The number of wins x/2200 is recorded:

{cha_27,cha_52,cha_77,cha_102,def_27,def_52,def_77,def_102, five, best, onestage, twostage}
{1011 , 878 ,830 ,1317 ,915 ,956 ,1041 , 878 , 1745, 1196, 1067 ,1366 }

This result is interesting.

The expectation was that the "onestage" and "twostage" would have the best result since they train with balanced game rules while others only play unbalanced games. Moreover, the number of self-play iterations for "onestage" and "twostage" are respectively 70 and 50, well for "cha_x" and "def_x", the number of iterations is 20, for "five" is 40. However, the result is beyond expectations. The "five" wins the championship.

The two-stage training achieves a better result than the one-stage even with 20 fewer iterations. This finding implies that setting the sim=2 at the starting point of training is worth considering.

Moreover, the "five" earn the championship indicates another way to train a balanced game agent: train with unbalanced rules, then switch the rule to the balanced one. In the future, it will be interesting to see the performance of "five"s network weight trained with the balanced game rule.

7. 6. Noise Tournament

The Noise Tournament (see 7.0.2.2) is held between selected network weights, both balanced and unbalanced games. There are 200 games for each player to play. The tournament runs three times.

Result:

{onefold,	no-noised}
{110,	90}
{107,	93}
{105,	95}

Result Analysis:

The network weight that is trained with Dirichlet noise is slightly better than the one without noise. It is worth mentioning that adding-noise versions accept the new weight ten times, whereas the "no-noise" accepts 17 times; the loss value for the win rate value for the one without noise is 0.2, whereas the "onefold" with noise is 0.118. The noise is likely helpful for the regression of loss function for the win rate value. However, a lower win rate loss value does not imply that adding noise is preferred because it is the noise in the training examples making wrong moves that cause the win rate of such an example easier to estimate. But if we set a temperature value, the noise will be necessary.

8. Real-World Experiment

The 5*5 gameboard is an unfair game, regardless of whether it is balanced or not. The best network game wins an advanced amateur 100% in real-world testing when it moves first, but this advanced amateur can win over A.I when he moves first. It is hard to tell who is better. However, the following real-world experiment attempts to demonstrate that the AI agent is better than this amateur human player.

Change of rule: in addition to the basic rule of Go, we add one more policy. We give the second move player a chance to switch from white to black at the second round of his/her move. Instead of playing the best moves, the first-moved black player has to confuse the white to make the wrong decision about whether to switch the role.

In this experiment, the “best” is used (see 7.0.1.4). The number of searches is set to 150 times.

8. 1. First Round: Human v.s. Human

Player1 and player2 and both humans play five games at their best. Player1 moves first in all of five games. Player 1 has tried this new rule with the best agent, whereas player2 only prepares by himself.

The result is player1 wins three times, player2 wins two times.

8. 2. Second Round: Human vs. Human Assisted with Best Agent

Player1 and player2 and both humans play five games. Player1 moves first in all of five games. Player1 plays on his own while player2 uses the best 5*5 agent as help. At the second round of player2, player2 checks the Q value (-1, +1) of its next best move. If this value is larger than 0, he will not switch, and vice versa.

The result is player1 wins one time, player2 wins four times.

8. 3. Real-World Experiment Analysis

Although five times experiment cannot make a decisive conclusion, it provides a better demonstration of the A.I agent's skill is likely to be better than amateur players who need time to adjust their strategy with the additional game rule. On the other hand, the player without A.I's

opposite to the A.I agent win one game, meaning that the current 5*5 is not making the perfect decision: at least it may make a mistake to anticipate the game's final result at the fourth turn (second round for white to move).

The data (the game records) of this experiment will be pushed to the GitHub page.

9. Future Works

The experiment omits the temperature parameter by setting it to infinity. Before adding Dirichlet noise, any temperature parameter value less than the total turns can cause severe overfitting problems. Now since the noise is added to the original framework, it is worthwhile to see the effectiveness of the temperature parameter.

A larger game board needs to be trained to make more accurate and thorough assessments of the algorithm and parameters. The 9*9 game boards will be trained on Lehigh's Magic 01 machine during the summer, planning to use the two-stage training, Dirichlet noise, and the noise discussed in the Keta-GO paper.

Moreover, I will regret it if I never have a chance to scrutinize the neural network design, so this is also a crucial part of future work.

Finally, if there is a chance to continue this independent study, I would like to slow it down, spend more time expressing the discoveries and make my work more clean and concise. I wish I could present my work in a way more formal and understandable. There is room for improvement.

Tips for training 9*9 in the future:

1. Start with the low value of Monte-Carlo simulations, e.g. sim=2
2. Start with a low value of num Episode, e.g. 50
3. If it keeps rejecting new changes, increase the value of Episode
4. Then slowly increase the value of the simulations
5. Can have a multi-process set with different combinations of simulations and episodes.
6. Try the temp variable. Unlike Othello, the Go game should make the first several moves constant instead of the last several (Alpha-Go Zero paper, first 30 moves for 19*19 board).
7. Is there a way to add more state information to increase the performance of a balanced game? (It might cause a higher win rate for white in a balanced game) E.g. beside the board status, add an input to the neural network to clarify if it is the turn of black or white. The current Go A.I in the market predicts a higher win rate for white. Could this be the reason?

Biography

Antonoglou, Huang, Guez, Hubert, Baker, Lai, Bolton, Chen, Lillicrap, Hui, Sifre, Driessche, Graepel, Hassabis

Mastering the game of Go without human knowledge

DOI:10.1038/nature24270

October 2017

Wu, Jane Street Group

Accelerating Self-Play Learning in Go

arXiv:1902.10565

November 2020

Surag Nair

alpha-zero-general (github project)

https://github.com/suragnair/alpha-zero-general

Ted Huang

go (github project)

https://github.com/teddy57320/go

Tristan Cazenave, Jean Méhat, Abdallah Saffidine.

UCD: Upper confidence bound for rooted directed acyclic graphs. Knowledge-Based Systems, Elsevier, 2012, 34, pp.26-33.10.1016/j.knosys.2011.11.014 . hal-01499672

Johannes Czech1, Patrick Korus1, Kristian Kersting

Monte-Carlo Graph Search for AlphaZero. Department of Computer Science, Technical University of Darmstadt Centre for Cognitive Science, Technical University of Darmstad hessian.ai — The Hessian Center for Artificial Intelligence Darmstadt, Germany 64289, December 22, 2020

Shantanu Thakoor, Surag Nair, Megha Jhunjhunwala

Learning to Play Othello Without Human Knowledge

Bruno Bouzy,Bernard Helmstetter

Monte-Carlo Go Developments. October 2003 DOI: 10.1007/978-0-387-35706-5_11

Leela-Zero (the GitHub project)

https://github.com/leela-zero/leela-zero

Christopher D. Rosin

Multi-armed Bandits with Episode Context. Parity Computing, Inc. 6160 Lusk Blvd, Suite C205, San Diego, CA 92121

Aditya Prasad

Lessons from AlphaZero (part 3): Parameter Tweaking

https://medium.com/oracledevs/lessons-from-alphazero-part-3-parameter-tweaking-4dceb78ed1e5

Acknowledgements

Thanks to Hanchi, Sun helped in Real-World testing as player2. Thanks to Yuming Tian, who provided one GPU at the starting point of the training.

Part 2: from 5*5 to 9*9

Previously, we trained a go agent playing on a 5*5 sized board. In part 2, we increased the board size to 9*9. Training a more complex game, we fixed bugs, adjusted the learning system, and proposed modifications. By monitoring the self-play process, we use human knowledge to help the game agent make better decisions and generate better game examples.

Bugs from Previous Code

Bug Example 1: stack-overflow
The learning system threw a stack-overflow exception in the recursive search function.

It is the action sequence [(6,8) by ' w', (8,8) by 'b' , (7,8) by 'w', pass by 'b'] that causes this problem. This action sequence is allowed, and the ending state after these actions is the same as the beginning. The agent finds the cached value for this state, and it will repeat the same sequence, as a result it fails to add a leaf node to the tree.

We fixed this issue by adding the turn number to the state-string-representation so that states are different even though the board appears to be the same. This way, the starting of the second sequence will add a leaf node rather than going back to the cached state.

Bug Example 2: Overwriting when Multiprocessing
In the code we used in section 7.3, the example generated by self-play overwrites each other, so during training, the amount of examples used by 'five fold' is no more than that of 'one fold.' For the new version, we designed a more structured pipelining method which will be discussed in the Optimization section.

The previous parallelism of self-play has no effect on example generation, but the ‘five fold’ (part1, section 7.3) has the best performance. It is interesting to explore what has brought this improvement. We brought the concept of Genetic Algorithm to explain this. In the testing phases, we compare the neural network weight with the previous one. The new neural network weight will be accepted if it is better. In Genetic Algorithm, generating a new neural network is a procedure of reproduction and testing is a process of filtering and making selection. The “5 fold” is the best (part1, section 7.3) because it generates the largest randomness and updates neural networks of the highest frequency.

Bug Take Away
There were bugs in the code, parameters that are not tuned, but, using ideas from genetic algorithms, they are still able to train a decent agent for simple games like gobang, tic-tac-toe, etc. It cannot train the 9*9 Go game because the 9*9 Go games are a lot more complex. To fix potential issues, we refined the computation of training losses, monitored the self-play process, and tried to optimize each part of the system.

Search Tree Modification

The parameter “numMCTSims” controls the amount of computation for the agent to decide one move. We want to know what is the optimal value for the “numMCTSims.” . From the AlphaGoZero paper, this parameter was set for 1600, meaning that for each turn of self-play, 1600 more nodes were added to the search tree. Since we are training a smaller game (9*9 rather than 19*19 which AlphaGoZero did) and we have less computing power (4 GPUs having 11GB memory each, less than the 64 GPUs used in AlphaGoZero paper), we have to use a smaller amount of searches. Our modifications belows intend to improve efficiency of searches.

New parameters are proposed: max-leaves, max-depth, depth-weight, leaves-increase, and Qdecision. We set up experiments to evaluate efficiency improved by the Qdecision, but for other parameters we cannot quantify their effect due to the randomness of this learning system.

Modification 1 - Max-leaves
We observed that when one player has a huge advantage, it only searches for the move that has the largest policy value (the **pi** vector returned from the neural model). When one player has a huge disadvantage, it searches on all the moves. This phenomenon can be explained by the PUCT Algorithm used by the search tree.
To determine which move to search (AlphaGo Zero paper):

$$a_t = \underset{a}{\operatorname{argmax}} \left(Q(s_t, a) + U(s_t, a) \right)$$

To activate the U (AlphaGo Zero paper):

$$U(s,a) = c_{\text{puct}} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$

When the agent determines the action to search, it compares the value of a_t or U for all actions. When no action has been searched, none of them have a_t. The move having the largest U value will be searched. And since the Cpuct is constant, N is 0, the only determinator is P, the policy returned by the neural network. When a player has a huge advantage, the Q value will be large (a positive number nearly 0.5), you can see the a_t will always be larger than U of other moves. Therefore, when comparing the only one a_t value with other U values, the a_t will always be larger as long as the Q value is positive. As a result, for many moves that have the potential to have the largest Q value, their Q values are not even activated (initiated).

In the real game, we observed that the player having advantage always misses the correct move, and we observed that the correct move it missed always has the second or third largest p value. On the other hand, when a player has a huge disadvantage, Q value is nearly a negative one, and the agent will search all valid moves once, including moves that are not worth any searches.

To improve efficiency of the searches, we propose a new parameter: “max-leaves.” For each state, there will always be “max-leaves” numbers of action to be initiated. It is similar to the beam search algorithm for Chatbot or machine translation. From this modification, by constraining the shape of the search tree, we mitigate the problem, and as a result, all nodes in the tree have an equivalent number of children.

Modification 2 - Max-depth

We need to determine the number for “numMCTSims”, the number of searches for each move. To explain in detail, we first clarify that each search produces one winning rate v for a new state. The Q value is the average of all winning rates (explained in part 1, section 4.3). Intuitively, the more v values we have, the more accurate the Q value would be, and thus the activation (the a_t) can be computed more thoughtfully. From the experiments, it is proved that more searches give better performance for most cases. However, we cannot allow the agent to search infinite times since searches take time. We want to figure out a cutoff point which makes searches more efficient. We proposed a new parameter “max-depth” to replace the “numMCTSims”, which reflects more on decision quality.

Now, we call the “search” method to add new nodes until the depth of the tree reaches to the “max-depth,” because depth of the tree can represent the quality of the move better than the total

amount of searches.

From our observation, we found many desirable moments as it makes slow decisions when two moves are similarly good. However, in fact, reaching the maximum depth does not guarantee the same quality of all moves. Some quick decisions may have low quality if the agent tries to "adjourn the game" a lot (moves that you can anticipate your opponent's next move since there is one easy correct answer.).

Modification 3 - Depth-weight

In the AlphaGo Zero paper, the value of Q is computed in the backup process as below.

$$W(s_t, a_t) = W(s_t, a_t) + v, Q(s_t, a_t) = \frac{W(s_t, a_t)}{N(s_t, a_t)}.$$

It is just the average of the winning rates over all states in the subtree.

We proposed a new parameter called "depth-weight." When we play a board game, it is apparent that we can anticipate our winning rate better if the state is closer to the end. Therefore, instead of taking average as what they did before, we give a larger weight if the exploration is deeper. Experiments will be done to test the effect of this modification.

Modification 4 - Leaves-increase

We have defined the Max-leaves, but we still want to know what is the optimal value for Max-leaves (number of beams)? Regardless of the value, we do not explore all the valid moves, so there will be a chance that the optimal move is not included. The deeper the tree, the higher the chance we missed the optimal move on some depth level. We want to know whether increasing the Max-leaves of the searches (e.g. increase from 5 to 12) can be more efficient than setting Max-leaves to a very rigid large number (e.g. always 12) and perform better than setting to a rigid small number (e.g. always 5).

Modification 5 - QDecision

In the modification 2, we claimed that the "max-depth" can reflect the quality of decision better than "numMCTSims," the total amount of searches. For this modification, we further improve the effectiveness of searches by using the Q value (the average of winning rate v after searches). When doing an infinite number of searches, the activation value will be approximately equal to the Q value (see formula in **Modification 1 - Max-leaves**). This means the agent will spend more of its following searches on the actions that have a larger Q. However, it is not the way an agent makes decisions. The agent makes decisions by selecting the actions that searches passed through it the most times, and this action may not have the largest Q value. This design makes sense, because if it decides purely on the Q value, it mitigates the effect of the neural network's policy vector.

For this modification, we made use of the Q value without harming the effect of the policy vector. Now, the agent will stop searching only when the current best action is having the largest Q value. In real practice, we ignore the minor difference (less than 0.02) to decide the best move faster.

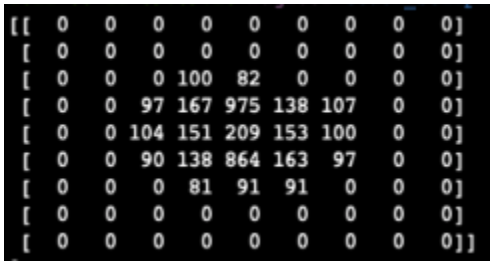
Improve Self-Play Example Quality

Modification 1 - Noise

The original version we used from the AlphaGo Zero framework does not have any noise except for the "temp" parameter. This is bad since we have to set "temp" (part1, section 5.3) to a high value. In the new version, by using and modifying the idea from the Kata-Go paper, we set the "max-depth" to be a random value in a range. The game agent will alway choose the best move it sees, but the best move can be different if it searches for different "max-depth" value.

Then, the dirichlet noise mentioned in the AlphaGo Zero paper is added. By adding the dirichlet noise to the policy vector, now the actions that originally have small policy value also have a chance to be simulated.

Moreover, we found the first several moves still have a high chance to overlap. In our previous neural network models, the first move is alway action (4,4), the center. We forced the first move to be randomly decided over the first 20 best policies. It gives better randomness without downplaying the quality of the game, since advanced players cannot tell why the best move is better than the 20th best one. After days of training, the number of searches for the first move is the figure below. It agrees with the Go knowledge of an advanced human player.



Modification 2 - Winner Expectation

We use the self-play data to train the neural network model. In the self-play data, each state and move are labeled by the final result. That is, if black wins, all the moves made by the black are labeled the same. This may cause problems. For instance, if the black wins by making bad moves at the start but good moves at the end, during training, the bad moves at the start will disturb the learning since they are also labeled as winning moves.

To explore the impact of the problem, we design a new way for self-play. Before one game, we randomly assign black or white to be the expected winner. When making decisions, searches only happen in the turn of the expected winner. As a result, the expected winner will always search 1 level deeper than its opponent. An analogy can be when one plays Go with oneself, this person only spends time in the black turn and white always plays according to his/her assumption.

During experiments, we found that the expected winner does not always win the game. To explore the impact of winner expectation, we divided game examples into 2 groups based on the correctness of expectation. We set up experiments to test which group can train a better neural network model.

Modification 3 - Ending Game Logic

Unlike most other board games, a Go game ends after both players agree it is finished. In this project, we add a pass option to AlphaGo-Zero framework. If both players choose to pass, then the game has ended. It takes some human interventions to help the agent to learn how to terminate the game properly.

At the very beginning of the training, the Go agent takes iteration to learn not to pass at the early stage, and it seems to never learn how to avoid suiside moves at the end. We used to observe this agent taking thousands of turns per game (there are only 81 territories on the board), indicating that suiside moves were frequently made. We tried to solve this issue by disabling the pass option before turn 70 and terminate the games at turn 130. After training, the self-play generates better examples, but by monitoring these examples we can still find wrong passes and suiside moves in around 20% of game examples until we refine its ending game logic.

First, we disable the pass option for the player who has less scores. Because if it chooses to pass, it is equivalent to surrender since its opponent will win the game by also passing the move. It will only choose to pass if pass is the only valid move (and then it loses correctly). This modification solves the problem of wrong pass decisions.

Second, after turn 80 (almost all 9*9 Go game ends before turn 80), for players who have more scores, we force the pass option to have the largest activation value by the chance of one third. Since the passes can never be the suiside moves, it decreases suiside chance of the winning player. Whether or not to suiside is still determined by the neural network returns, but the neural network can learn this better after we gave human interventions as above.

We made this modification on August, 7th, 2022. By the time this report is written, we can hardly find any cases of ending game issues.

Modification 4 - Fairness of the Game

The Black has the first-move advantage, so all versions of the Go rule give White some advantage when comparing final territories. For a nine by nine board (having 81 territories), Black needs 44 to win but White only needs 38. The game becomes more fair to both players.

At the starting phase of training, the player who gets 41 of 81 will be the winner. Under this setting, Black (first move player) has a higher winning rate. This is not the standard way of playing the 9*9 Go games, but we can train the neural network model easier.

To theoretically explain this phenomenon, first we clarify that when making decisions, both Black and White interpret the game board's current state using 0 as the current player, 1 as opponent. The standard game setting punished the Black, but it confused the model since the model cannot tell black from white. This can be verified in the experiments from the previous study (part1, section 7.5).

Later, we trained our agent to play standard games (black wins by 44). We found that the winning rate estimated by the neural network is reasonable when one player has a score more than 44, but if both players are scored in the range from 38 to 44, the winning rate estimation will be around 50%, meaning that the neural network cannot tell which player is in advantage. This is expected since in our training examples, a score from 38 to 44 can be a win of white or a loss of black. The vagueness of winning rate estimation may impede our agent from making the best decision. We tried to fix this issue by training two different neural network weights for Black and for White separately. In the training examples for Black, we labeled all states as a loss if the game ended with a score in the range 38 to 44. For white, we labeled states in that range as wins. We expected this strategy to solve the problem of winning rate estimation.

Optimizations

Modification 1 - Pipelining

In section 7.3, we experimented with parallelism. It was running multiple processes of iterations containing self-play, training, and testing iterations. However, it has problems. First, it does not

sufficiently use the GPU memory. The training takes more memory and GPU Utility than self-play and testing. If multiple processes start training at the same time on the same GPU, both processes are slowing down. Second, we did not accurately control the execution order. The training process updates the neural network, so training processes' updates will be overwritten with each other. Moreover, we cannot quantify how many game examples and how much time one iteration takes.

For the new version, we use a pipelining execution strategy to reduce overhead and gain more control over the entire system. Instead of having all processes running self-play, training and testing iterations, we divided them into separate processes. At the first phase, all the processes are running self-play (example generation). When it has enough game examples (we typically generate 3000 games per iteration), we terminate all the self-play processes. We train the model, and after we have a new neural network weights we test its performance using multiple processes.

We also implemented an optimistic execution order. At the early stage of the entire reinforcement learning system, the new neural network weight is unlikely to be worse than the previous one. Therefore, we skip the testing process. We run self-play processes, terminate them and train the model, and then start self-play again without testing. Training only takes one process, so we have computing power to make up the previous test so if our assumption is wrong we are able to notice that. (when any new neural network weight is worse than the previous one)

Modification 2 - “Call-by-need” Optimization Taking Advantage of NN Predictions
In the previous version of search tree, we called the function ‘getValidMoves’ to mask the policy vector returned by the neural network. The valid moves for each state are cached. In the new version, since the model is trained, we trust the neural network’s policy vector **p** to give valid moves larger possibilities, so we removed the ‘getValidMoves’ function during searches. If an invalid move is searched, our Go Logics will catch this error. The search will be rolled back, and the move having the next activation value will be searched. This modification brought a speedup of about 1.5 times.

Experiments

By the time this report is written, we trained the neural network using 44 game example sets. Each iteration generates neural network weights for experiment. We have generated 160 neural

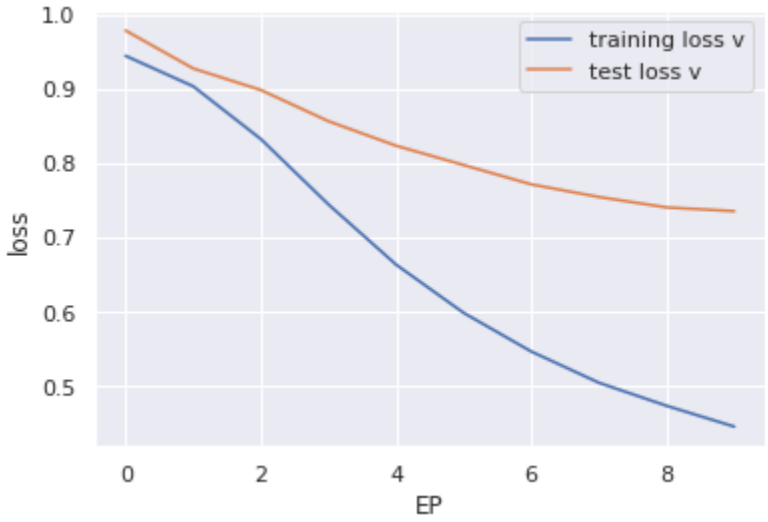
network weight for various experiments. A selective group of experiments are presented in this section.

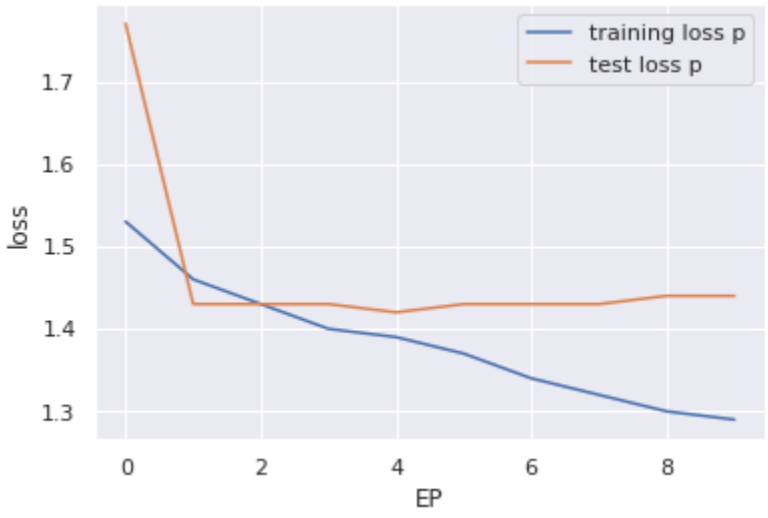
1 - Number of Epoch

The original AlphaGo-Zero framework trained neural networks for 10 epochs. In this experiment, we wanted to know whether 10 epochs is the best choice.

1.1 - Number of Epoch & Losses

The AlphaGo-Zero framework did not measure test losses. We split the example into 80% training set and 20% for the test. The average testing losses for each epoch is as follows (we used examples in Iteration 21).





Analysis 1.1
Figure shows that the test loss of winning rate keeps decreasing, indicating that the overfitting problem does not seem to exist. The overfitting issue exists for the policy vector after the second epoch. One may naively expect epoch 10 to be better since its v loss is the lowest and its p does not seem to overfit too much. However, the following experiment falsifies this expectation.

1.2 - Arena Test
In this experiment, we tested the ability of the Go agent using the neural network weight produced after epoch 1 and 10 (named ep1 and ep10) with the previous best (best of iteration 20). The result is as follows.

Weight Version: Iter 21 (l5d3)	Result
ep1:prev	211:209
ep10:prev	193:277

(the l5d3 means we set the max-leaves to 5 and max-depth to 3)

We also compare the ability of ep1 and ep10 using different iterations. The result is as below.

(l5d3)	Result
ep1:ep10 (Iteration 2)	582:468
ep1:ep10 (Iteration 32)	102:96

Analysis 1.2
The results of both tables indicate that ep 1 is better than ep10. Based on this experiment result, we found it is efficient to train the model using only one epoch.

Associated with the figures from experiment 1.1, the result suggests a lower loss for policy vectors is more important than that of the winning rate. This makes sense intuitively. Since when test loss of policy vector reduces, the new neural network weight can activate moves better than previous version (or at least counter moves of the previous). When the test loss of winning rate value reduces, it can better predict the winning rate of the previous version, but not necessarily the current one. In this sense, neural network's winning rate prediction is always one iteration behind that of the policy.

2 - Impact of Winner Expectation

We introduced winner expectations in the section “Improve Self-play Quality” modification 2. In this experiment, we start a game with an expected winner, and it will always search one level deeper than its opponent. Since searches are cached, as a result, we only search at the expected winner’s turn. To generate more cases that white wins the game, there is a 70% chance the white will be the expected winner. At the end of this iteration, it generates 8GB data (about 2500 games), consisting of 4GB examples that the expected winner won and 4GB examples that it lost.

We fine-tuned previous best neural network weights with each of these examples. The loss value of policy pi and winning rate v is as below.

Training with expected wrong examples: Loss_pi=9.59e-01, Loss_v=2.40e-01
Training with expected correct examples: Loss_pi=9.93e-01, Loss_v=4.25e-01
Training with all examples: Loss_pi=9.71e-01, Loss_v=4.31e-01

Then, we compare the new neural network weight with the previous best. The testing result is as below.

(Iter13) (l5d5)	
Expected_correct: prev	109:101
Expected_wrong: prev	83:127
all:prev	137:73

Analysis 2

Training with a subset of examples may give smaller loss values depending on how this subset is selected, but for investigating whether this subset successfully filtered the noise, it is better to measure the Go agent's Go ability. Here, training with all examples gives the best neural network weight, so this winner expectation strategy did not improve the quality of self-play examples even though we get smaller loss values.

3 - Impact of New Parameter

3.1 - Depth-Weight

Using the same neural network weight, we compare the Go ability when setting Depth-Weight to different values. The result of testing is as below.

(Iter9)	
Depth-Weight 1 : 1.5 (l5d7)	10:10
Depth-Weight 1 : 2 (l5d7)	44:40

Analysis 3.1
We never observed Go ability improvement by changing the Depth-Weight. In other experiments we observed a decrease of Go ability if we set Depth-Weight to a large value (i.e. 3). We decided to set this value to one, the original value.

3.2 - Leaves Increase
Using the same neural network weight, we compare the Go ability when setting leaves-Increase to different values. The result of testing is as below.

(Iter9)	
Leaves Increase 0 : 1 (l5d5)	36:48
Leaves Increase 0 : 1 (l5d7)	26:58

Analysis 3.2
It is clear that a larger value for Leave-Increase increases the Go ability of the agent. However, we did not measure the total amount of searches used by different players. We are unsure about whether this modification also increases the efficiency of searching.

3.3 - QDecision
Using the same neural network weight, we compare the Go ability of the agents having QDecision enabled or disabled. We also measure the average searches per move to measure the efficiency of searching. The result of testing is as below.

QDecision True:False (Iter32)	
True:False (both d5l7)	90:76 (avg sim per move 362:275)
True(d5l7):False(d7l7)	85:83 (avg sim per move 382:576)

Analysis 3.3
From the first row of the chart, the use of QDecision increases the ability of the Go agent but the average amount of search is also increased. From the second row, the max-depth 5 using

QDecision is having similar testing results with max-depth 7 no QDecision, but it requires less average searches for the first setting than the second.

Since the sample size we used is small (168 games), the result might not be perfectly precise, but we observed progress that is made right after we enable QDecision during training. It is very likely that this modification will increase the efficiency of generating qualitative game examples.

4 - Two Networks
After we shifted to the fair game (black wins by 44, white wins by 38), we set up experiments to train two different neural network weights for black and for white, aiming to eliminate the vagueness in winning rate prediction.

If the final score of one play is in range (38,44), all moves are labeled as a loss for training the black network, and all moves are labeled as a win for training the white network. We fine-tuned previous best neural network weights with each of these examples. The loss value of policy pi and winning rate v is as below.

Training black: Loss_pi=1.70e+00, Loss_v=5.03e-01
Training white: Loss_pi=1.70e+00, Loss_v=5.08e-01
Training (original): Loss_pi=1.69e+00, Loss_v=7.01e-01

We tested the ability of two network weights versus single network weights (original). The single network weight wins by (19:5).

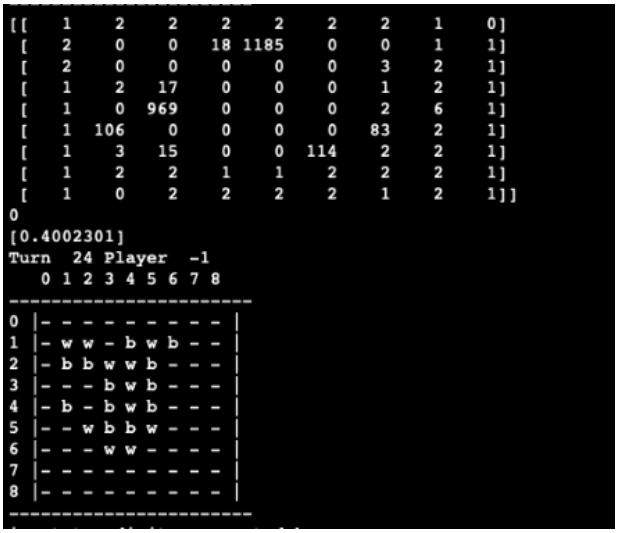
Analysis 3.4
A lower loss value for training black and white is expected since the vagueness is eliminated by setting up a rigid standard or winning the game (44 for black's network and 38 for white's network). However, we did not expect the two networks to perform poorly when comparing with the original version. By the time this report is written, we still need time to figure out the reason.

Evaluation

Human Evaluation
We invited an advanced player to play with our game agent. The human player won this game.

Our agent made a mistake in the 23th turn. Below is a screenshot of the game. The black is AI, white is human. For the turn 23, black should take (4,2) to win. However the AI chose (1,4), and as a result, the black stones on the left side are all captured.

We can see that the incorrect move (1,4) is searched for 1185 times while the correct one (4,2) is searched 969 times, the second largest. Our neural network can give the policy and winning rate close to an advanced player's knowledge, but the current agent is not as accurate.



Future Work

Seven days prior to the time we wrote this report, we refined the ending game logic, and we observed the agent to make progress faster than before. We will keep training this agent, see how well it can be without changing the current structures and parameters.

Meanwhile, it is also worthwhile to think how this type of learning system can be applied to other fields outside of the board games.

Biography (New for Part 2)

David J. Murray-Smith

Experimental modeling: system identification, parameter estimation and model optimisation techniques

Doshi, Ketan

Foundations of NLP Explained Visually: Beam Search, How It Works https://towardsdatascience.com/foundations-of-nlp-explained-visually-beam-search-how-it-works-1586b9849a24

Acknowledgements

Thanks to Hanchi, Sun helped in Human Evaluation.