Gomoku Based on AlphaZero and Game Strategy Improvement

Jiashu Chen
Electrical Engineering Department
Columbia University
UNI: jc5664

Ziyu Liu
Electrical Engineering Department
Columbia University
UNI: zl2949

Lei Lyu
Electrical Engineering Department
Columbia University
UNI: ll3433

Chaoran Wei
Electrical Engineering Department
Columbia University
UNI: cw3282

Yi Yang
Electrical Engineering Department
Columbia University
UNI: yy3089

Zikai Zhu
Electrical Engineering Department
Columbia University
UNI: zz2765

***Abstract*—AlphaGo [1], which employs deep neural networks (DNN) and Monte-Carlo Tree Search (MCTS), can learn the optimal policy to win a Go game and simulate ahead from the current state during a game to make the next move. AlphaZero [2] expands the idea of AlphaGo Zero to Chess and Shogi. In our work, we applyed AlphaZero in Gomoku game. We trained AI models to play Gomoku Game in two different deep learning frameworks: Pytorch and Keras. The trained models are utilized to play AI vs AI Gomoku Game. Additionally, We explored the influence of four factors – numbers of training iterations, model architecture, numbers of simulated games and calculation of Upper Confidence Bound(UCB) on an AlphaZero Gomoku plahyer's win percentage manage to draw some conclusions on what leads a AlphaZero player competitive in a Gomoku game.**

***Index Terms*—Reinforcement Learning, Alpha Zero, AlphaGo Zero, AlphaGo, Monte-Carlo Tree Search, Deep Neural Network, ResNet**

I. Introduction

The Go game between AlphaGo and the famous Korean Go master Lee Sedol has shown the world the great superhuman potential of reinforcement learning in large strategy games. AlphaGo [1], which employs deep neural networks (DNN) and Monte-Carlo Tree Search (MCTS), can learn the optimal policy to win a Go game and simulate ahead from the current state during a game to make the next move most likely to lead to a final win in Go game.

Basically, AlphaGo relies on a supervised learning training procedure in the first step to train a policy network, which means it still needs human knowledge to find a way on how to win a Go game. However, even the best Go player in the world can't precisely tell you what is the best choice of the next move in a Go game, thus the information given by humans, even if they are experts, to train the network can be very unreliable. As a result, AlphaGo Zero [3], a new reinforcement learning model was raised to eliminate the unreliable information given by humans. AlphaGo Zero uses a single neural network to get both the policy and value given some chess board state as the input and doesn't depends on human knowledge to train the neural network. AlphaGo Zero adopts a new reinforcement learning algorithm incorporating evaluating the policy to make the next move on the chessboard via the Monte-Carlo Tree Search method and a neural network, without using traditional simulation procedure and sampling over all data in one Go game to train neural network to be used in the next self-play game, which is trained on the aim of approaching the policy simulated by the Monte-Carlo Tree Search method and the empirical winning rate.

AlphaZero [2], which is similar to AlphaGo Zero but a fully generic algorithm, can be applied to the games other than Go such as Chess and Shogi. AlphaZaro doesn't need any additional domain knowledge except the rules of the game. AlphaZero is using a more general-purpose Monte-Carlo Tree Search algorithm which is different with AlphaGo Zero who uses a alpha-beta algorithm, which makes AlphaZero a more generic verison of AlphaGo Zero.

In our project, we managed to apply the principles of AlphaZero into a Gomoku game (also called *five-in-a-line*). We divided our group members into two teams and designed our models under two different deep-learning frameworks: Pytorch and Keras respectively, and organized an *AI vs AI* Gomoku competition. We analyzed how the choices on neural network structures, training strategies and different hyper-parameters will influence an AI player's performance on the Gomoku game in the following parts of our report. However, due to the limit of computational resources and time, we constrained all our experiments on a 8×8 chessboard, which still needs quite much time to train an AI player (about 5 hours to train 500 epochs).

II. Related Work

AlphaGo [1] is composed of several training stages. AlphaGo first uses a supervised learning network p_σ that is trained to predict the next move in a Go game based on expert human players' moves. Also, a fast policy network p_π is trained to make quick action sampling from rollouts. Then, a reinforcement learning policy network p_ρ is trained upon the result of previous training, to improve the policy

evaluated by the supervised learning policy network through self-play games with opponents randomly selected in one iteration of the training process. Finally, a value network v_θ is trained to predict winners of the self-played games with both players using the policy from p_ρ. The method of Monte-Carlo Tree Search is used to combine the value network and policy network to achieve ahead search simulation.

AlphaGo Zero [3] uses a single neural network that is trained on self-play games only. AlphaGo Zero's single network f_θ uses a temporary chess board state s and its history as input and outputs possible moves' possibility and a value (p, v). Monte-Carlo Tree Search [4] [5] [6] is used to achieve self-play games and improve the policy through searching all possible output moves with each move guided by the current neural network. AlphaGo Zero also uses sampled data in the self-play games to improve the network's policy and value estimation. The ultimate goal of the training is to minimize the difference between neural networks and the simulation results from the Monte-Carlo Tree Search process.

AlphaZero [2] is similar to AlphaGo Zero algorithm, but has differences in several respects. AlphaZero takes account of draws and other potential outcomes when estimating and optimizing the output instead of binary win/loss outcomes. Besides, the neural network is updated continually without waiting for a iteration to be completed. In addition, the hyper-parameters will be reused in all games without the game-specific tuning. Also, the optimization of the special rules of the game of Go has been eliminated too.

III. Methodology of our Project

A. Monte-Carlo Tree Search

When humans play this kind of large strategy games like Go, Chess, Chinese Chess, Gomoku, etc., they would first analyze the current situation before placing the next piece on the board. In order to beat the rival, they might need to predict how the opponent might react to their moves. The player who has a larger chance to win the game, is the player who can foresee his opponent's action as well as how the composition on the board would develop better.

Thus, the first problem comes to us is how to endow our model this ability to think ahead before making the move on the chess board. To do so, we adapted Monte-Carlo Tree Search (MCTS) [5] method to implementing the ability to make various simulations on the current situation before placing the next move on the board. With this method, not only can our model plays like a human, but also it can play with itself along with techniques introduced by AlphaGo Zero [3], so the training process can be conducted without human's involvement.

To implement a Monte-Carlo Tree Search model specifically on our Gomoku game, we designed the following key factors.

a) Selection (Tree Descent): We defined a node in the tree to present one composition in the Gomoku game. Then for its descent nodes, or named child nodes, they will be the composition after one of the players make the next move on the chess board. Therefore, all possible locations to place the chess can generate one state from the current tree node. So the number of child nodes of one node depends on how many vacant locations are in the chess board now. To select one of the child nodes from all, the simplest way is to use UCB_1, which has the formula [7]:

$$\text{UCB}_1 = Q(s,a) + C\sqrt{\frac{\log N(s)}{N(s,a)}} \tag{1}$$

Where $Q(s,a)$ here is the estimated win rate after action a, $N(s)$ is the number of times that state s has been visited and $N(s,a)$ is the number of times the action a has been chosen under state s. State s here represents a composition in the chess board and action a is which place the next step will take on the chess board. Then under the current state s, the next action will be the one with the largest value of UCB_1. Notice that if $N(s,a) = 0$, then corresponding action will have $\text{UCB}_1 = \infty$. Furthermore, we also implemented another nodes selection strategy, which is the basis of AlphaGo, AlphaGo Zero and AlphaZero [1] [3] [2] [7]:

$$\text{UCB}_{modified} = Q(s,a) + \frac{P(s,a)}{1 + N(s,a)} \tag{2}$$

Where $P(s,a)$ represents the prior knowledge we have to choose action a under the current state s and is generated to be uniform at the beginning. In AlphaZero, $Q(s,a)$ and $P(s,a)$ can be estimated by a neural network.

b) Expansion: When we reach a leaf node in the tree using the above selection procedure, the next step is to generate new leaves from it. However, before doing so, we will first check whether the game come to an end. Because if the game has ended, although their might be vacant space on the board still, this state is actually a terminal state that has no possible subsequent states. If the game hasn't come to an end, then we will expand the tree by generating all possible states as the child nodes of the current leaf node and randomly choose one of them as a new record node in the tree.

c) Simulation: After the addition of that new record node to the tree, we will implement one simulation upon that new node. In the simulation process, all actions will be made randomly until an end of the game. A winner or a tie will be decided on the result. In AlphaZero, however, the simulation process with rollouts can be avoided by using the outputs of the neural networks.

d) Back-propagation: After the simulation generates a result, all nodes in the tree that have been traversed in this run will have a new record of win, lose and tie. This record is reflected in the value of $Q(s,a)$. That is if the player make action a under the state s and win in the final simulation, then it will be encourage to choose a in the round afterwards, so the value of $Q(s,a)$ needed to be added. Otherwise, the value need to be decreased. As for adjacent nodes, they represent the play of two players, then we need to repeatedly change the sign of the change to the value of $Q(s,a)$, after every propagation upward. After reaching the parent node of the tree, a new round of game will start by selecting among children of the parent nodes based on UCB_1 or $\text{UCB}_{modified}$.

B. Neural Network and Training

In AlphaZero, a deep neural network f_θ with parameter θ [3] is employed. The whole network can be divided into two stages. The first stage is a shared feature extractor, with the raw board state s recording pieces' positions, extract a feature map. The second stage of the network is composed of two branches. The first branch outputs a vector of move probabilities $p_a = \Pr(a\,|s\,)$ which represents the probability of selecting each move. This branch is responsible for estimating policy of a Gomoku game. The second branch outputs a value v, a scalar evaluation, estimating the value function of the current player from temporary state s.

Our model samples from history data it collects during self simulation games as training data. From an optimization perspective, we need to minimize the loss function on the self-play data set:

$$l = (z - v)^2 - \pi^T \log p + c\|\theta\|^2 \tag{3}$$

s is the description of current situations, which is also the input of the Policy-Value Network (the probability distribution output of the movements). The output consists of the probability of each feasible action in the current position p and the value for the current position v. π is the policy of the game estimated by MCTS and z represents the real game winner simulated by MCTS.The loss function shows that the goal of our training is to minimize the different between predicted policy and winner between the MCTS simulation policy and winner. Also, to avoid over-fitting, we added a l_2 penalty term.

In the training process, in addition to paying attention to the changes in the loss function, we also recorded the entropy changes of the strategy from the output of the policy and value branched. At the beginning, because the strategy of the model was to randomly output the position of the drop, the entropy was relatively large. But as the training time increased, the entropy of our model would decline. This was caused by the bias of our policy. As the training process progressed, the policy network would gradually know where the probability of placing a piece in a particular chessboard state is greater, which means that the position of the drop was more definite, which caused the entropy to decline. The output of policy network would serve as prior knowledge during the search process in MCTS.

IV. Implementation

A. Network Architecture

We simplified the AlphaZero network to improve the efficiency of training and prediction. The first column of figure 1 consists of a three-layer Convolutional Neural Network, using ReLu activation function, followed by policy and value branched. At the policy side, firstly four 1×1 filters are used for dimensionality reduction, and then it is connected it to a fully connected layer. Besides, the softmax function is used to directly output the probability of each position on the chessboard. On the value side, firstly two 1×1 filters are used for dimensionality reduction, then it is connected to a fully connected layer of 64 neurons, followed by another fully connected layer. Finally, the tanh nonlinear function is used to directly output the situation between [-1,1] score.

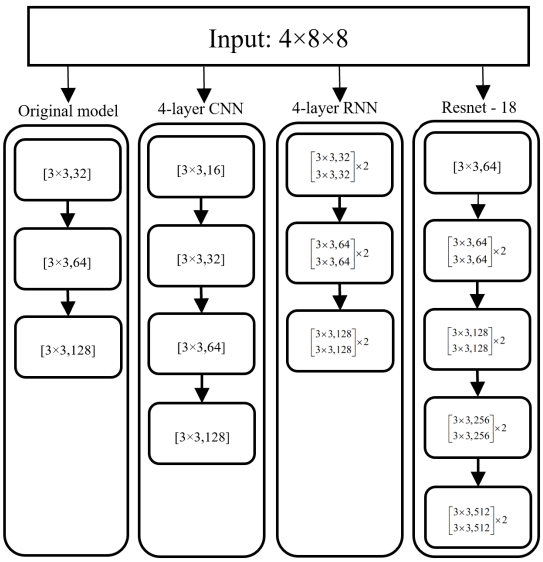


Fig. 1. Architecture of DNN Models

In order to conduct comparative experiments, we have made improvements on this basis and added two new network structures, namely 4-layer CNN model, 4-layer RNN model and ResNet-18 [8]. 4-layer CNN model adds one more Convolutional Neural Network at the beginning with 16 filters. 4-layer is adopted from the original structure with one additional layer after the second convolutional layer. 4-layer RNN model concatenates the output of the second convolutional layer with the input before going to the third convolutional layer. The new ResNet-18 is less complex than the original ResNet-18. We used padding to make the output size of every layer in ResNet-18 to be the same because our input is four matrixs with shape 8×8. The size is small so we must keep this size unchanged when using ResNet-18.

Finally, we compared the training effects between them to study the impact of different network structures on the outcome.

B. AI Game Design

In the implementation process, we simplified the environment to realize the game of Gomoku under the 8×8 board. Each board has 4 8×8 binary feature planes, which represent the positions of the pieces of the two players. The position of the opponent player's latest move, and the plane that represents the current move first. Four feature planes were used to describe the current situation information. In addition, the operations related to the board, including placing a piece, judging win or losing, and displaying, are all implemented in the Gomoku file. There are three types of players in this project, namely manual players, MCTS players and AlphaZeroPlayer. Among them, manual players can enable human players to

play against computers, MCTS players play a role in strategy testing, and AlphaZeroPlayer is the target model we use to train.

The algorithm contains two part: the training process and the test process.

Algorithm 1 training algorithm

repeat
store augmented data
enter the data into DNN, output policy π and value
enter policy and value to MCTS, output state
calulate win pertecentage of MCTS with DNN and basic MCTS
until number of iteration

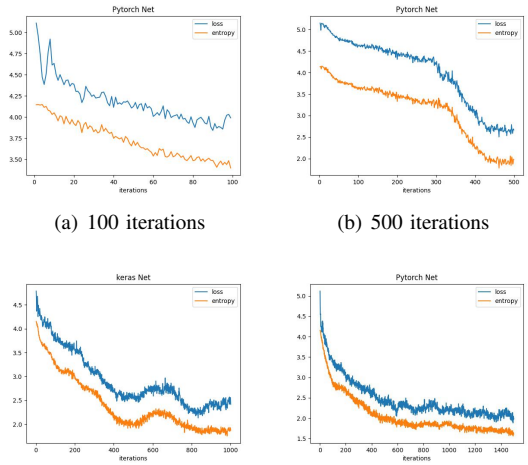
Our training process is shown in Algorithm 1. At the beginning of training, let the target model play against a MCTS player for policy improvement [9] [10] and save the data. After getting the data, we used rotation and flip method to expand the data and store all the data of 8 equivalent situations in the self-play data buffer. This kind of rotation and flipping data expansion could also improve the diversity and balance of self-play data to a certain extent. Then we updated the strategy through the policy iterations [11] [12], and finally returned the loss and entropy. When evaluating the strategy, we let the target model and the basic Monte-Carlo Tree Search model play against each other, and used the win percentage as the index of the evaluation model to judge whether it is the optimal model. During training, if the check point is reached, we would update and save the best model and the latest model at the same time. The training data is generated by the latest model. This method has a better convergence effect during training, and the speed is also faster. At the same time, in order to ensure diversity, we added a new exploration method: in the process of self-play, each step will obtain the action by sampling the probability of the number of visits of each branch at the root node of Monte-Carlo Tree Search and adding Dirichlet noise.

Additionally, the testing process contains two part: AI vs human and AI vs AI. The first function would allow human to play with AI model to test if the model is well-trained. The second function is to make a scenario that AI model plays with AI model to test the influence of training iteration, different models, the number of play of Monte-Carlo Tree Search and different methods to achieve Monte-Carlo Tree Search on the outcome. In short, our project contains the following modules to achieve the algorithm: The Gomoku module provides a basic game environment, MCTS and Monte-Carlo provide a theoretical basis for our MCTS-based players, and the Player module contains the three types of players. In addition, there is a net module that stores our network model.

V. Experiments

A. Experiment Overview

Firstly, in order to compare the impact of different training iterations on the Gomoku game performance, we changed the number of training iterations for the same 3-layer Convolutional Neural Network. Observe the game level of the Gomoku program in different situations. In training, we used PyTorch and Keras as frameworks for training. After 100 training iterations, 500 training iterations, 1000 training iterations, and 1500 training iterations on the same 3-layer Convolutional Neural Network, the results obtained are shown in Fig. 2. It shows the value changes of loss function and entropy on an 8×8 chessboard under different self-play training iterations.



(a) 100 iterations (b) 500 iterations

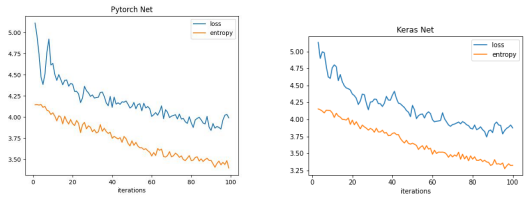
(c) 1000 iterations (d) 1500 iterations

Fig. 2. The values of loss function and entropy with different training iterations

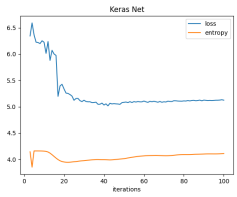
When we optimized the value of loss function during the training process, we observed that the value of loss function gradually decreases from about 5.0. In several training pieces with different iterations, the value of loss function reaches about 2.5 after 500 iterations training and then tends to decrease steadily in a long run.

In the training process, in addition to the gradual decrease of the loss function, we generally paid attention to the entropy of the Policy-Value Network. Initially, the entropy will be relatively large due to a uniform random movement probability distribution in the strategy network. After more training iterations have been implemented, the strategy network will learn which movements in different positions should have a greater probability of winning, in another word, a nonuniform probability distribution on movements. And there will be a stronger tendency for specific positions, leading to a smaller entropy. It can help Monte-Carlo Tree Search to give more simulations at those potential positions with higher winning rates, so as to achieve better performance with fewer simulations. From Fig. 2, we can see that entropy gradually decreases from about 4, and finally stabilizes at about 2.0.

After that, we employed 100 training iterations for three models and change the number of layers in the Convolutional Neural Network to establish different models. The original model has a 3-layer Convolutional Neural Network with 32, 64 and 128 filters in each layer and this model uses the ReLu activation function. Then we implemented two new network



(a) Original network with 3 layers (b) The 4-layer CNN model



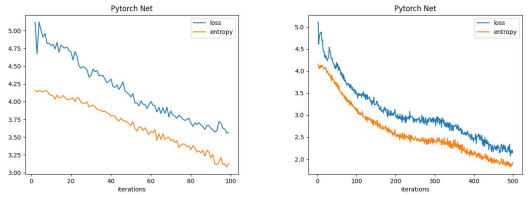
(c) ResNet-18 model

Fig. 3. The values of loss function and entropy with different model structures

structures 4-layer CNN model and ResNet-18. 4-layer CNN model adds one more Convolutional Neural Network at the beginning with 16 filters, while RestNet-18 model has a 18-layer complex network structure.

After 100 training iterations, we obtained Fig. 3 with the values of loss function and entropy for different network models. Fig. 3, we can observe that the original 3-layer network has lower values of the loss function and entropy than 4-layer CNN model after 100 training iterations, and the ResNet-18 model has the largest loss value among these three models.

Finally, we used a different tree-node selection algorithm, UCB_1, to create different models with 100 and 500 training iterations. The training results are shown in Fig. 4. From this figure, we can observe that the values of the loss function and entropy decrease at a high rate, which indicates a great performance. After 100 training iterations, the loss function value has decreased to about 3.5 and the value of entropy is about 3.0, even better than the original model performance after 100 training iterations. After 500 training iterations, the values of the loss function and entropy using UCB_1 algorithm are similar to the values of the original model.



(a) 100 training iterations (b) 500 training iterations

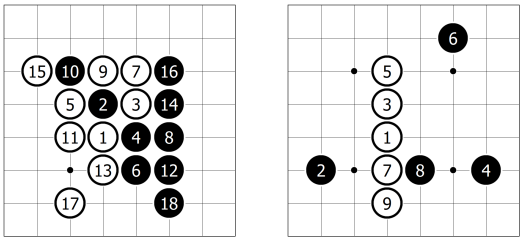
Fig. 4. The values of loss function and entropy using UCB_1

B. Comparison of Different Models

For an AlphaZero player, the skills of playing the Gomoku game and the ability to win are the most critical factors for judging the effectiveness of training. Considering the importance of the first move in the Gomoku game, we evaluated different training effects by making the models with different training iterations and different network structures play against each other. They will play with another training model twice and two AlphaZero players will make the first move in turn.

Firstly, we explored the effect of different numbers of training iterations on outcomes. Fig. 5 shows game results between models with the same network structure and different training iterations. All moves follow the principle of white first.

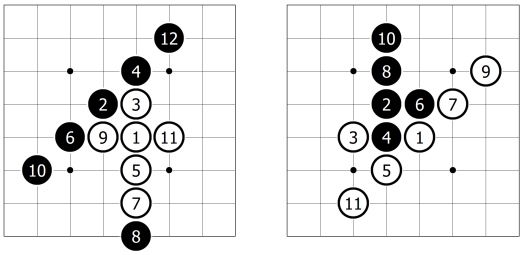
Fig. 5 shows that the the outcomes of models being trained 100 iterations and 500 iterations. The left one represents that the model with 100 training iterations move first, while the right one represents that the model with 500 training iterations moves first. We could see that when the model with 100 training iterations moves first, the model with 500 training iterations still win. However, it is a complete victory if the model with 500 iterations moves first. Therefore, the model with 500 training iterations is much better than the one with 100 training iterations. From the outcomes of these two games, we can observe that the model with 100 training iterations makes moves almost randomly, while the model with 500 training iterations has the consciousness of making moves as close as possible to win and block the opponent's potential winning lines.



(a) The model with 100 training iterations makes the first move (b) The model with 500 training iterations makes the first move

Fig. 5. 100 iterations vs 500 iterations

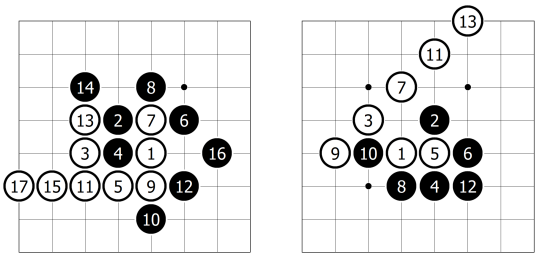
Fig. 6 shows that the the outcomes of models being trained 500 iterations and 1000 iterations. We could see that when the model with 500 training iterations moves first, the model with 1000 training iterations wins. In addition, when the model with 1000 training iterations goes first, it still wins. Therefore, the model with 1000 training iterations is much better than the model with 500 training iterations. From the outcomes of these two games, we can find out the model with 500 training iterations makes moves in purpose at first but will lose its 'mind' after the first wave of attack has been defused. On the contrary, the model with 1000 training iterations has a more comprehensive game policy of attacking and defending.



(a) The model with 500 training iterations makes the first move (b) The model with 1000 training iterations makes the first move

Fig. 6. 500 iterations vs 1000 iterations

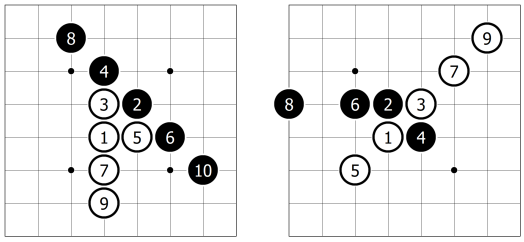
Fig. 7 shows that the the outcomes of models being trained 1000 iterations and 1500 iterations. We could see that when the model with 1000 training iterations moves first, it would win. In addition, when the model with 1500 training iterations goes first, it would beat the opponent. Therefore, at this time, the one that moves first would have more chance to win regardless of the number of training iterations. These two models both have a great game policy, and even show a tendency of aggressive moves in games.



(a) The model with 1000 training iterations makes the first move (b) The model with 1500 training iterations makes the first move

Fig. 7. 1000 iterations vs 1500 iterations

Fig. 8 shows that the the outcomes of models being trained 500 iterations and 1500 iterations, which is taken as a control group. We could see that whether the model with 500 training iterations goes first or latter would lead to a complete victory of the model with 1500 training iterations. So the model with 1500 training iterations is much better than the one with 500 training iterations. In short, as training iterations increase, models perform better than those with fewer iterations. As the number of iterations reaches to 1500, the model has been trained well. After that, if we increase the number of iterations, training iterations would not be the major factor for the model performance.

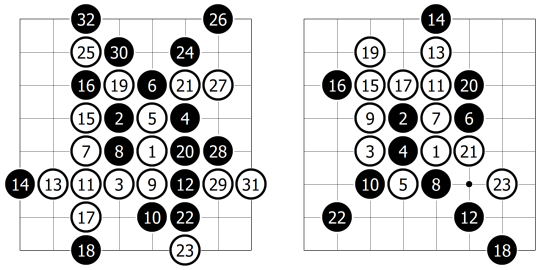


(a) The model with 500 training iterations makes the first move (b) The mode with 1500 training iterations makes the first move

Fig. 8. 500 iterations vs 1500 iterations

Secondly, we would like to explore the effect of numbers of simulated games on the models. We used the parameter 'playout' to represent the Monte-Carlo Tree Search after getting the policy and value from Deep Neural Network. Fig. 6 shows outcomes between models with the same network structure and different numbers of playout. All moves follow the principle of white first.

Fig. 9 shows that the game results between 2 same models with 1000 train iterations, while one model makes 50 playouts and another makes 1000 playouts during games. We could see that the model with 1000 playouts will always be the winner, no matter which player makes the first move. So the model with 1000 playouts has a better performance than the one with 50 playouts. It means that for models having the same train iterations, models' performance would be better as the number of playouts increases.

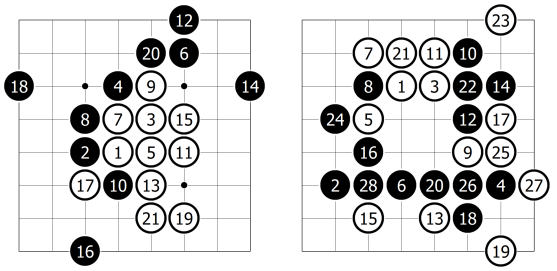


(a) 50 playouts vs 1000 playouts, 50 playouts make the first move (b) 50 playouts vs 1000 playouts, 400 playouts make the first move

Fig. 9. Game results of two original models' self play

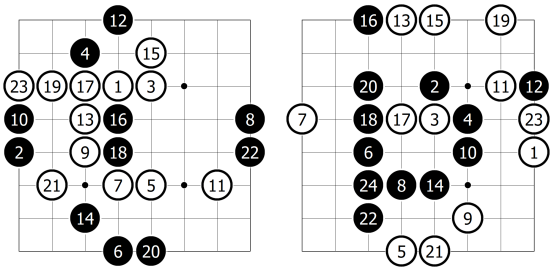
Thirdly, we compared the performance of different models. Fig. 10 shows that the results between original 3-layer model and 4-layer CNN model. We could see that when original 3-layer model moves first, it could win. In addition, if original 3-layer model goes latter, it still wins. So original 3-layer model is better than 4-layer CNN model. Fig. 11 shows that the results between 4-layer CNN model and ResNet-18 model. We could see that whether 4-layer CNN model moves first or not, it could win ResNet-18 model. In addition, ResNet-18 model's moves show great randomness on the board, which indicates its training is not enough.

In conclusion, after 100 training iterations, original 3-layer model is better than 4-layer CNN model, and 4-layer CNN model is better than ResNet-18 model.



(a) Original model takes the first move (b) 4-layer CNN model takes the first move

Fig. 10. The comparison between Original model and 4-layer CNN model



(a) 4-layer CNN model takes the first move (b) ResNet-18 model takes the first move

Fig. 11. The comparison between 4-layer CNN model and ResNet18 model

Then we compared game results between the original 3-layer model and 4-layer RNN model both with 1500 training iterations. We can observe that there is a tie result between these two models, and they all have a perfect game policy which is shown in this game. In brief, when there are enough training iterations for models, the difference of models wound not be the vital factors for game performance.

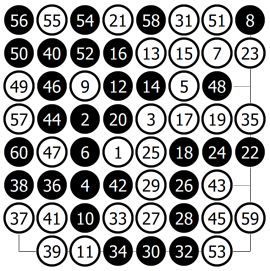
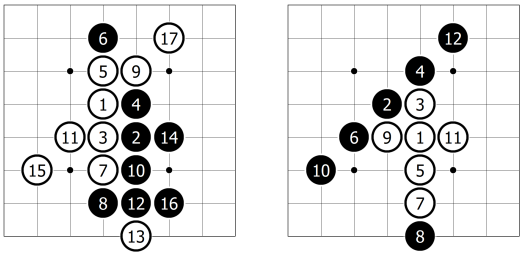


Fig. 12. The comparison between original 3-layer model and 4-layer RNN model with 1500 training iterations

Finally, we explored the influence of different UCB players to play against the AlphaZero players we trained on 3-layer CNN models with 500 training iterations. As shown in Fig. 12, whether trained 3-layer CNN model verified by UCB_1 goes first or goes last, it always wins. So UCB_1 algorithm is better than $\text{UCB}_{modified}$ algorithm here to play against the AlphaZero player to imporve its performance.



(a) UCB_1 model takes the first move (b) $\text{UCB}_{modified}$ model takes the first move

Fig. 13. The comparison between the model using UCB_1 and the model using $\text{UCB}_{modified}$

C. Discussion of Insights Gained

In conclusion, we explored four factors that may affect the model performance: training iteration, numbers of simulated games, network architecture and Upper Confidence Bound(UCB).

Firstly, the model performance becomes better as the number of iterations increase. This is easy to understand. With the increase of training iterations, the backbone network learns more features from the input chessboard states. The policy and action branches also has a better estimation of the true policy and action selection.

Secondly, the number of simulated games(playouts) also has a positive influence on the model performance. As the number of playouts increases, the model's has a higher win percentage. With the same network architecture and training iterations, more playouts mean that we utilized more Monte-Carlo Tree Search, and thus the model could get more accurate move from Monte-Carlo Tree Search. However, when the number comes to a threshold value, the model performance won't improve significantly. Different models have different threshold values. For example, the threshold for the model after 1000 iterations' training and original three-layer model is 100 playouts.

The network architecture has a very complicated influence on model's performance. In our experiment, 4-layer RNN model and ResNet-18 model seems to have a even poorer performance than that of very simple model. Complex models which containing more parameters are less likely to converge to a optimal state in a limited number of iterations.

We found that how we calculated Upper Confidence Bound also has a influence on model's performance when two models which share same architecture are trained in the same number of iterations. According to our comparison, we found that model trained under equation 1 has a better performance than equation 2.

Also, we found that our model do not always learn a better policy as training iterations accumulate. Our explanation for this problem is that the model reaches the optimal policy and will oscillate around the optimal position in further iterations. Thus, during the training process, we stored the model when

a higher win percentage is gained in every policy evaluation into the final result.

Finally, we can conclude that all these factors, such as training iteration, numbers of simulated games, network architecture and Upper Confidence Bound(UCB), are synergistic on deciding the game performance. For instance, an efficient model will show a good performance even just with a fewer training iterations, and enough training iterations will eliminate the influence of different models. These facts shows interactions and restrictions among these factors.

VI. Conclusion

Our work is based on AlphaZero and we apply it to the Gomoku game. In our work, we focus our attention on factors that affect AlphaZero player's win percentage in a Gomoku game. We build and train our deep neural network under two deep learning frameworks: PyTorch and Keras, and let them play against each other to compare different factor's influence. We conclude the influence of number of simulated games, number of training iterations, network architecture and the calculation of Upper Confidence Bound.

Source code of our project is in our GitHub repository https://github.com/yaannnik/E6885-final-project.

VII. Statement of Contribution

Jiashu Chen: Search literature and model training. Participate in writing the first four parts of the report. Revised report. Design the training script.

Ziyu Liu: Build Keras Net, train Keras models(3-layer CNN models, Resnet-18 model), output all Gomoku game results and write Experiments and Comparison parts.

Lei Lyu: Implement MCTS of UCB_1 version, train Pytorch network UCB_1 model, write Monte-Carlo Tree Search part, revise the report.

Chaoran Wei: Build Keras Net, train 4-layer RNN Keras model, train 4-layer CNN Keras model, and write comparison of different models part.

Yi Yang: Build Gomoku game environment. Design and train 3-layer CNN AlphaZero model in PyTorch. Implement class of Manual Player, AlphaZero Player and Monte-Carlo Tree Search Player.

Zikai Zhu: Design the game play script. Implementation of Network Architecture and write the corresponding part.

References

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[2] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[3] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[4] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *International conference on computers and games*. Springer, 2006, pp. 72–83.

[5] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*. Springer, 2006, pp. 282–293.

[6] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[7] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk, "Monte carlo tree search: A review of recent modifications and applications," *arXiv preprint arXiv:2103.04931*, 2021.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[9] R. A. Howard, "Dynamic programming and markov processes." 1960.

[10] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[11] D. P. Bertsekas, "Approximate policy iteration: A survey and some new methods," *Journal of Control Theory and Applications*, vol. 9, no. 3, pp. 310–335, 2011.

[12] B. Scherrer, "Approximate policy iteration schemes: a comparison," in *International Conference on Machine Learning*. PMLR, 2014, pp. 1314–1322.