Reinforcement Learning: Theory and Applications

by
DONG,Hanwen

A report
submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
The Chinese University of Hong Kong

November in 2022
Supervisor: Prof. John C.S.Lui

Abstract

The AlphaZero algorithm is an improvement on the AlphaGo Zero algorithm, which has demonstrated a great performance in the game of Go. In this report, we attempt to implement an algorithm based on AlphaGo Zero and AlphaZero to play the game of Gobang on an 8×8-sized board, which is called Alpha Gobang. We use Monte Carlo tree search as search algorithm and design a convolutional neural network containing 5-6 layers. The tree search is executed to perform self-play, guided by the neural network. The generated self-play data are then used to train the neural network, which further improves the strength of the tree search. We combine both AlphaGo Zero and AlphaZero together, utilizing data augmentation, as applied in AlphaGo Zero, and maintaining only one single neural network to generate self-play data, as done in AlphaZero. Based on some properties of Gobang, we simplify the input features to the neural network. Besides, we try to take some initiatives to address the problem of data imbalance during the training in a more effective way. Finally, our model demonstrates a good performance when playing against with human. In most cases, on the 8×8-sized board, the game ends with a draw, sometimes the model can even defeat human.

Contents

1 Introduction

The AlphaGo Zero algorithm [1] has achieved superior performance in the game of Go, and the AlphaZero algorithm [2] [3] is a general-purpose version of AlphaGo Zero. Compared with the AlphaGo [4] series of algorithms AlphaGo Fan (beating Fan Hui), AlphaGo Lee (beating Lee Sedol) and AlphaGo Master (beating Ke Jie) that initially defeated human players, the AlphaZero algorithm learns from scratch entirely based on self-play reinforcement learning. Instead of utilizing human expert data for supervised learning, it directly starts exploration from random action selection. AlphaZero has two key parts: (1) use Monte Carlo tree search in self-play to collect data; (2) use the data to train a deep neural network and use the network to estimate action probabilities and state value during tree search. The algorithm is not only applicable to Go, but also beat the world champion program in chess and shogi, proving the generality of the algorithm. In this report, we implement a algorithm based on AlphaGo Zero and AlphaZero to play the game of Gobang, named as Alpha Gobang. We first introduce the concept of combinatorial games (including go, chess, Gobang, etc.), and gives the basic rules of Gobang; then we introduce the details of implementation of Alpha Gobang, including the specific process of Monte Carlo tree search and the design of deep neural network, using Gobang as the game environment. Finally, we demonstrate our experimental results, including the results of training process, the results of self-play and the results of human playing with Alpha Gobang.

2 Background

2.1 Combinatorial Game

Combinatorial Game Theory(CGT) [5] is a branch of mathematics and theoretical computer science that usually studies serialized games given perfect information. Such games usually have the following characteristics:

1. Games usually involve two players (e.g. Go, Chess). Sometimes a game involving only one player (such as Sudoku, poker) can also be regarded as a combinatorial game between the game designer and the player. Games involving more than two players are not considered combinatorial game problems because more complex game problems such as cooperation occur in games [6].
2. The game does not contain any chance factors that affect the game result, such as dice and so on.
3. The game provides players with perfect information [7], meaning that each player is fully aware of all the events that occurred before.
4. Players perform actions in a turn-based manner, and both action space and state space are finite.
5. The game ends in a finite number of moves, and the result is usually a win or loss, with some games having draws.

Many combinatorial game problems, including single-player games such as Sudoku and Solitaire, and two-

player games such as Hex, Go, and Chess, are classical problems that computer scientists aims to solve. Since IBM's Deep Blue system [8] defeated the chess master Gary Kasparov, Go has become the next bridgehead of artificial intelligence. In addition, there are many other games, such as Othello, Amazons, Shogi, Chinese Checkers, Connect Four, Gobang, etc., have attracted a large number of people to use computers to find solutions.

Gobang, also known as Gomoku or Five in a Row, is a typical combinatorial game. Usually there are two players, black and white. The black player plays first. Starting with an empty board, when there are five pieces of the same color connected in a line (horizontal, vertical or diagonal), it means a player wins, otherwise it is a draw. Gobang has various rules, the most common rule is Freestyle rule. For the freestyle rule, one of the player only needs to have at least five pieces of the same color connected in a line to win the game. Freestyle Gobang has no restrictions on either player. Under this rule, the first player could take a great advantage. It has been proved by a Japanese player in 1899 that if the first player plays perfectly at each move, it must win. Nowadays, Gobang has developed various rules to balance the advantage of the first player. Here, we simply use the most common rule, the freestyle rule.

2.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [6] is a method of sampling actions and building a search tree based on the sampled actions to find the optimal decision in a given space. This approach has revolutionized combinatorial games, and has pushed the performance of AI algorithms that solve combinatorial games such as the game of Go to unprecedented levels.

MCTS mainly includes two parts: tree structure and searching algorithm. A tree is a data structure that consists of nodes connected by edges. Usually, in addition to the states and actions, each node in the search tree also stores the the number of visits and the evaluation of rewards. In the AlphaZero algorithm, a node also contains the action probability distribution of the state.

As shown in Figure 1, in the search tree of AlphaZero, each node stores the following information.

1. A: The action to be performed to reach this node (to index its parent node)
2. N: The number of visits of the node. The initial value is 0, indicating that the node has not been visited.
3. W: The total action-value of the node. The initial value is set to 0.
4. Q: The average action value of the node, calculated by $\frac{W}{N}$, represents the value estimate of the node. The initial value is set to 0.
5. P: The prior probability of choosing action A. This value is obtained by inputting the state of its parent node to the neural network, and stored in the child node

Since there are two players in the game, when building a search tree, there are two player perspectives in one

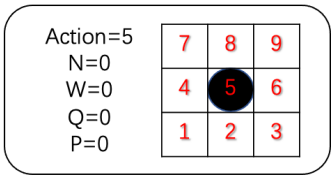


Figure 1: A example of node information. The action can be represented as $(b, 5)$, which means that the black player (b) performs action 5 and reaches this state. $N = 0$ means the number of visits to the current node is 0, W means the total action value, Q means the mean action value, and P means the prior probability of choosing action A = 5. Since the current node has not been visited yet, all initial values are set to 0

tree. The information stored in the node is updated either from the perspective of the black player or from the perspective of the white player. For example, in Figure 1, the board state represented by this node has only one black piece, so next it should be the turn of the white player to take the next move. However, it should be noted that the information in this node is stored from the perspective of its parent node (the black player). Since this node is newly generated by the parent node when expanding its child nodes, A, N, W, Q, and P stored in this node are all initialized by the black player and used for subsequent updates from the black perspective. Therefore, only the black player chooses action $A = 5$ to reach this node, and at the same time initialize the current information as $N = 0, W = 0, Q = 0, P = 0$.

After the search tree is established, the Monte Carlo Tree Search explores action space through a heuristic method to estimate the action value function of the root node $Q_\pi(s, a)$. The whole process can be described as starting from the root node and exploring until it reaches the leaf nodes, and repeating the process multiple times so that the estimation of action value is gradually accurate, thus finding the optimal action in the search tree. The action value function without discount factor can be expressed as [9]:

$$Q_\pi(s, a) = \mathbf{E}\left[\sum_{h=0}^{T-1} P(S_{h+1}|S_h, A_h)R(S_{h+1}|S_h, A_h)|S_0 = s, A_0 = a, A_h = \pi(S_h)\right] \tag{1}$$

where $Q_\pi(s, a)$ is the action value function.

The tree search method has four steps: select, expand, simulate, backup. All these steps are performed in the search tree, and there are no pieces on the real game board.

1. **Select**: Based on a certain policy, the action is selected from the root node until it reaches some leaf node.
2. **Expand**: Add child node to the current leaf node.
3. **Simulate**: Starting from the current node, the game is simulated by some policy (e.g. random policy) until the end of the game, and the result is obtained: a win, a loss, or a draw. Depending on the result, a reward is usually given: +1 for a win, -1 for a loss, and 0 for a draw.
4. **Backup**: The results of simulation are updated in a backward way by visiting the nodes passed in the current round of tree search and updating the information for each node.

The most commonly used tree search algorithm is the UCT (Upper Confidence Bound in Tree) algorithm [10],

which is a good solution to the balance between exploration versus exploitation in the tree search process. The UCT algorithm is an extension of the UCB (Upper Confidence Bound) algorithm [11] to the tree structure. The UCB algorithm is a classical algorithm for solving the Multi-Armed Bandit problem. The UCB algorithm chooses an action at time t according to the following strategy:

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{2}$$

where $Q_t(a)$ is action value, which increases the possibility of dominant action being selected. In the second term, $N_t(a)$ represents the number of times that action a was selected during the first t time steps. This term increases the possibility of exploration. The fewer times the action was selected, the more likely the action is to be selected. c is a positive real number used to balance the level between Exploration and Exploitation.

The UCT algorithm is an implementation of the UCB algorithm in a tree structure, which selects the action that maximize the UCT value in the search tree. The UCT value is defined as follows:

$$UCT = \bar{X}_j + C_p\sqrt{\frac{2\ln n}{n_j}} \tag{3}$$

where n is the number of visits of the current node and n_j is the number of visits of its child node j. C_p is a constant controlling the level of exploration. The average reward term $\bar{X}_j$ encourages to exploit actions with higher values and the square root term $\sqrt{\frac{2\ln n}{n_j}}$ encourages to explore actions with fewer number of visits.

The UCT algorithm solves the problem of balancing the exploration and exploitation of the actions in tree search, and subverts many large-scale reinforcement learning problems, such as Hex, Go, and Atari, etc. Levente Kocsis and Csaba Szepesvari [10] proved that: consider a Finite-Horizon Markov Decision Process (Finite-Horizon MDP) where the reward is between [0, 1], the number of states is D, and the number of actions for each state is K. Consider the UCT algorithm such that the square root term of the UCT is multiplied by D. The estimated deviation of the expected reward $\bar{X}_j$ is of the same order as $O(\frac{\log n}{n})$. Furthermore, the probability of estimation error at the root node converges to zero at a polynomial rate as the number of searches increases. This indicates that the UCT algorithm ensures that the tree search converges to the optimal solution as more searches are executed.

AlphaZero discards the simulation step and directly predicts the results using deep neural networks. Thus, the tree search algorithm in AlphaZero contains three essential steps, which is shown in Figure 2.

1. **Select**: Based on a certain policy, the action is selected from the root node until it reaches some leaf node.
2. **Expand and evaluate**: Add child node to the current leaf node. The prior probability of selecting each action and the action value of each state-action pair is evaluated directly by policy network and value network.
3. **Backup**: After the expansion and evaluation are completed, The results are updated in a backward way by visiting the nodes passed in the current round of tree search and updating the information for each node. If the leaf node is not the end of the game, then the game cannot return the result of winning or losing, which

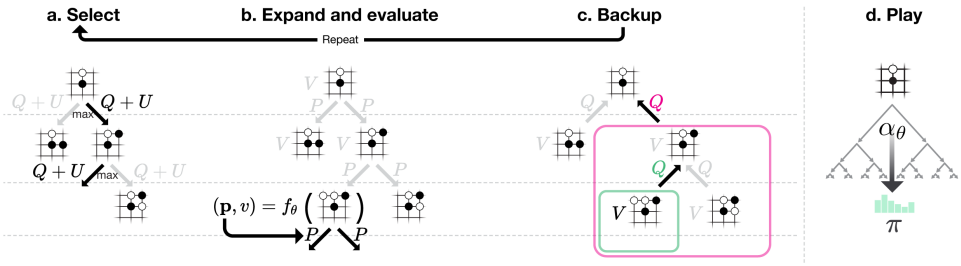


Figure 2: Monte Carlo tree search in AlphaZero [1]. The tree search process is repeated many times before each actual move. It first selects an action from the root node until it reaches a leaf node, then expands the leaf node and evaluate the value, and finally performs a backup step to update the node information

is instead predicted by the neural network. If the leaf node has reached the terminal state of the game, then the result is given by the game.

In the select stage, the action is given by $a = \arg\max_a(Q(s,a) + U(s,a))$, where $Q(s,a) = \frac{W}{N}$ encourages to exploit actions with high action value, $U(s,a) = c_{puct}P(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$ encourages to explore actions with higher prior probability and lower visit count, and c_{puct} determines the level of exploration.

In the expand and evaluate stage, policy network outputs the probability $P(s,a)$ of each action to be selected under current state. Value network outputs the value v of current state s. $P(s,a)$ is used to calculate $U(s,a)$ described in select stage. The value v is used to calculate W in backup stage, where $W(s,a) = W(s,a) + v$.

In the backup stage, all the information in each node is updated, where $N(s,a) = N(s,a) + 1$, $W(s,a) = W(s,a) + v$, $Q(s,a) = \frac{W(s,a)}{N(s,a)}$.

3 Alpha Gobang based on AlphaGo Zero and AlphaZero

The AlphaZero is applicable to various combinatorial games such as Go, Chess, Shogi, etc. Here, we take the freestyle Gobang mentioned in Section 2.1 as an example to introduce the details of the AlphaZero. We take a 3×3 game board as an example, where three pieces connected in a line means win, to illustrate how AlphaZero plays the game. Besides, AlphaZero is the upgrade of AlphaGo Zero. They are very similar but distinguish in several ways [2].

1. AlphaGo Zero is only applicable to the game of Go, while AlphaZero is a general algorithm for board games. In AlphaGo Zero, since the rules of Go are the same under rotation and reflection, data augmentation has been used to generate more data by rotating or flipping the game boards. However, AlphaZero does not augment data because the rules of games like chess and shogi are position-dependent.
2. AlphaGo Zero calculates and optimizes the winning probability based on win/lose outcomes. Instead, AlphaZero calculates and optimizes the expected outcomes, taking draws into consideration.
3. In AplhaGo Zero, self-play data are generated by the best player from the historical training. After each iteration, if the new player wins the best player by around 55%, it will replace the best player and the following

self-play data will be generated by this new player. In contrast, in AlphaZero, self-play data are generated by a single neural network. It only maintains one network and generate data by using the latest parameters for this network.

Our implementation references both of these two algorithms. We utilize data augmentation, like in AlphaGo Zero, since the rules of Gobang are the unchanged under rotation and reflection. We take draws into account and only maintain one single neural network to generate self-play data, like in AlphaZero.

In this section, we will introduce the details of AlphaZero. The whole algorithm can be divided into two parts: (1) using Monte Carlo tree search to collect self-play data; (2) using deep neural network to fit the data and using it in Monte Carlo tree search. The process is illustrated in Figure 3.

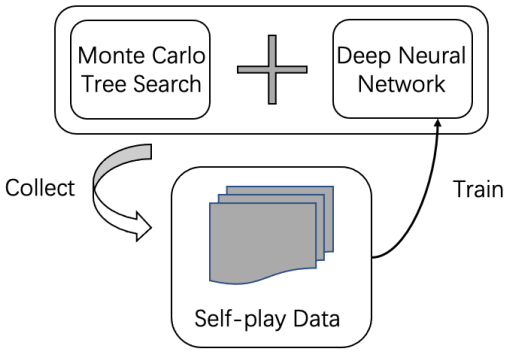


Figure 3: The algorithm process. In the AlphaZero, Monte Carlo tree search, self-play data, and neural network form a loop. Monte Carlo tree search combined with neural network is used to generate data, and the generated data is used to improve the prediction accuracy of the network. The more accurate the network prediction, the higher quality of data generated by Monte Carlo tree search; the higher the quality of the data, the more accurate the trained network prediction; the whole process forms a virtuous circle

3.1 Monte Carlo Tree Search in AlphaZero

First, we demonstrate how to use Monte Carlo tree search to collect data. We assume that the game starts in the state shown in Figure 4 (usually games start with an empty board). At this point, it is the white player's turn to perform the action.

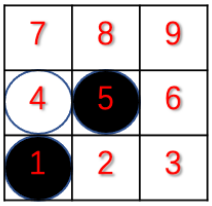


Figure 4: The starting game board state.

We start to build the tree structure from this state, sequentially performing the aforementioned three tree search steps: selection, expansion and evaluation, and backtracking. At this point there is only one node in the tree, and

since it is at the top of the tree, it is the root node. Because it has no children, it is also a leaf node. This means that we have reached a leaf node, which means we have completed the selection step. Therefore, the second step is expansion and evaluation. Figure 5 shows the process of node expansion. All child nodes of the node are expanded. The policy network takes the node state as input and calculate the probability of each action being selected.

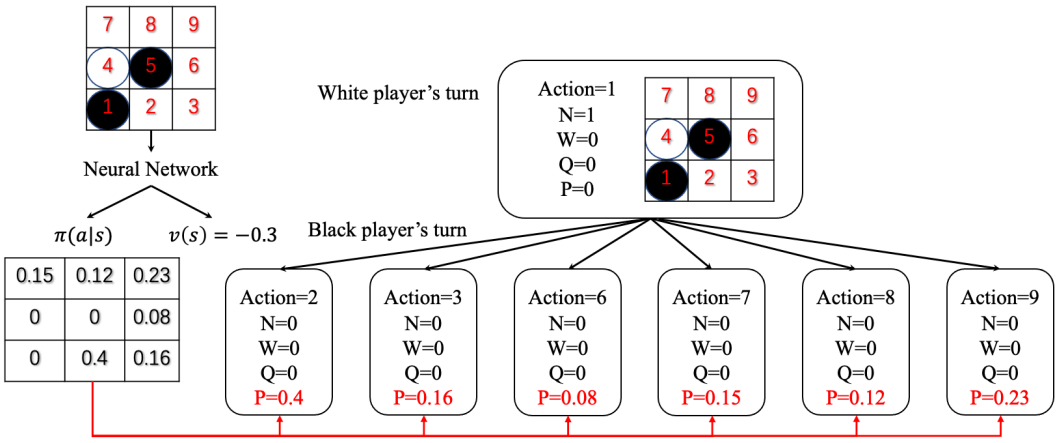


Figure 5: Expansion and evaluation at the root node. All the feasible action nodes are expanded. The probabilities are given by neural network.

The last step is backup. Since the current node is root node, we only need to update visit count N, rather than W and Q. After updating $N = 0$ to $N = 1$, the search process has completed.

Every time we perform the tree search, we will start from the root node. As shown in Figure 6, the second tree search process will also start from the root node. This time, there is a child node under the root node, which means the current node is not the leaf node. The action is selected by the formula $a = \arg\max_a(Q(s,a)+U(s,a))$, where $Q(s,a) = \frac{W}{N}$ and $U(s,a) = c_{puct}P(s,a)\frac{\sqrt{\sum_b N(s,a)}}{1+N(s,a)}$. Here, white player selects action $A = 2\ (w,2)$ and reaches a new node. This node is a leaf node, and it is black player's turn to select an action.

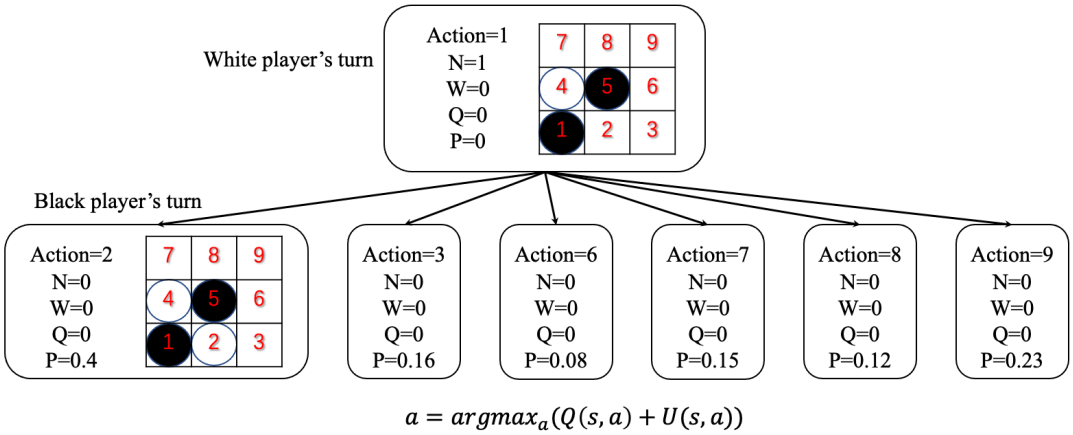


Figure 6: Selection at the root node. White player selects action $A = 2\ (w,2)$ and reaches the leaf node. Now it is black player's turn to select a move.

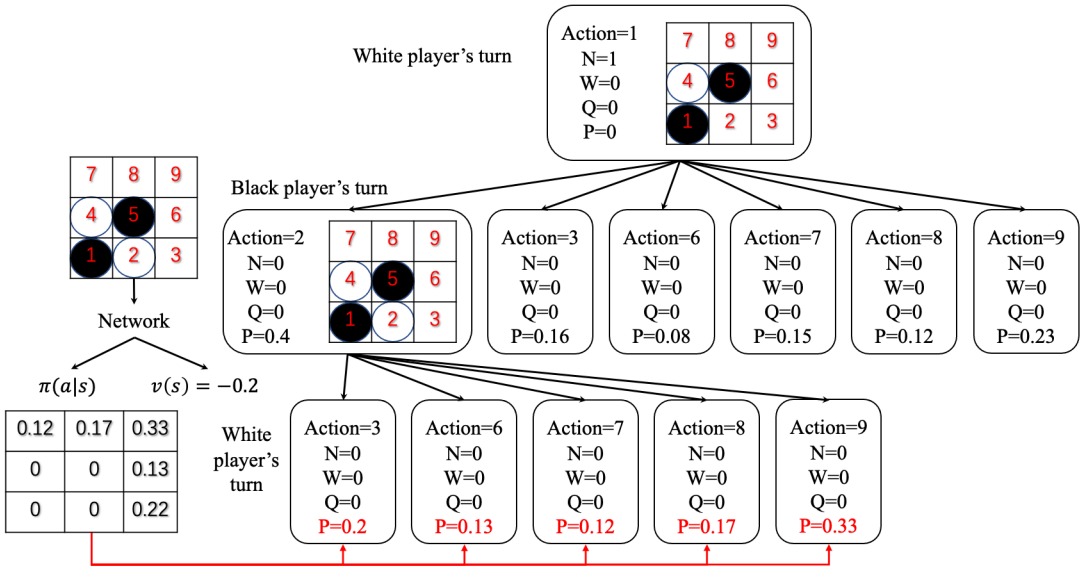


Figure 7: Expansion and evaluation at the new node. All the feasible action nodes are expanded. The probabilities are given by neural network.

Next We expand and evaluate this leaf node, as shown in Figure 7. Same as the first time: all feasible actions are expanded and the probability of each action is given by the policy network.

Now it's time to perform backup step. Since there are two nodes in the tree, we first update the current node, and then update the previous node. The updates of these two nodes follow the same way: $N(s,a) = N(s,a) + 1$, $W(s,a) = W(s,a) + 1$, $Q(s,a) = \frac{W(s,a)}{N(s,a)}$. Note that there are two views in the tree: the view of black player and the view of white player. We need to notice the updated view and always update the value from the view of current player. In Figure 8, the value $v(s)$ given by neural network is -0.2, which is calculated from the view of black player. However, this is the value of action 2 selected by white player. Therefore, it is white player that should know this value instead of black player. When updating the information of white player, we should use the opposite number of the value, which is $v(s) = 0.2$. Thus, we have $N = 1$, $W = 0.2$, $Q = 0.2$.

After that, we return to its parent node. As before, since the node in the current state is the root node, we do not need to perform backup to update W and Q, only need to update the number of visits N. After setting N = 2, the second tree search process is complete, as shown in Figure 9.

The third tree search process also starts from the root node. According to the formula a=arg $\max_a(Q(s,a) + U(s,a))$ and the information in the current tree, the white player selects action 2 $(w, 2)$, and the black player chooses action 9 $(b, 9)$. As shown in Figure 10, after the selection, the game reaches the terminal state. In the expansion and evaluation stage, the node will not be expanded, while the value v can be obtained directly from the game. Therefore, the value network does not estimate state value, and the policy network does not output action probabilities.

Next is the backup. There are three nodes need to be updated. Nodes are updated recursively from leaf nodes

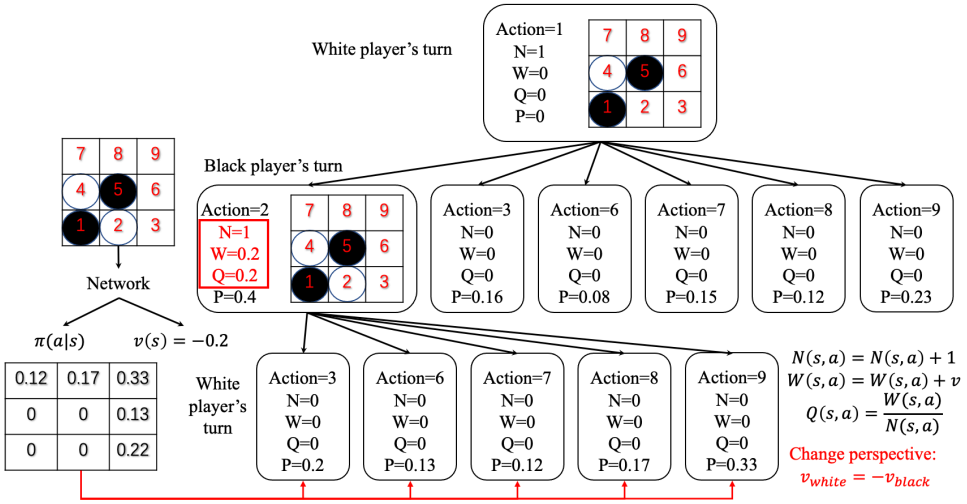


Figure 8: Backup at the new node. Information stored in the current node is updated from the view of white player: $N = 1$, $W = 0.2$, $Q = 0.2$

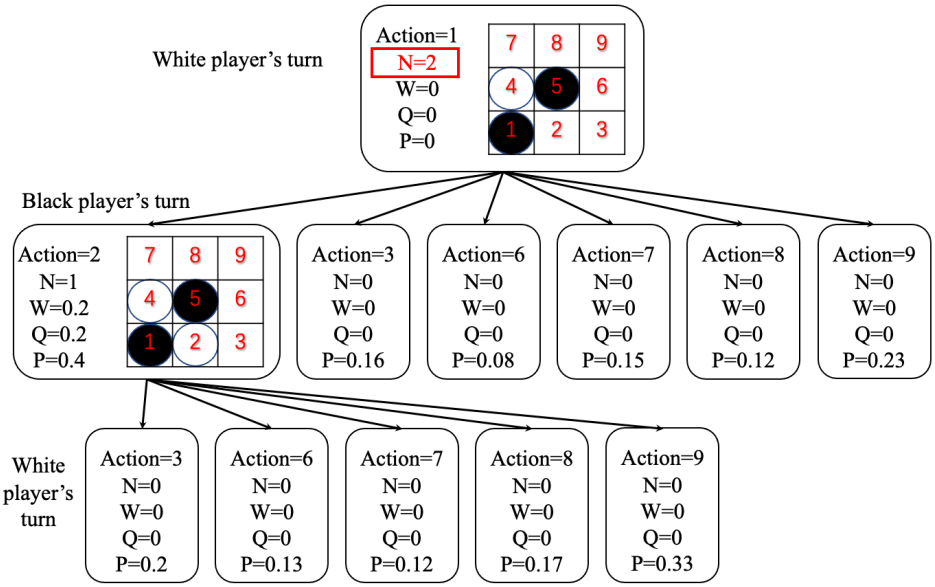


Figure 9: Backup at the root node. N is updated to 2, W and Q remain still.

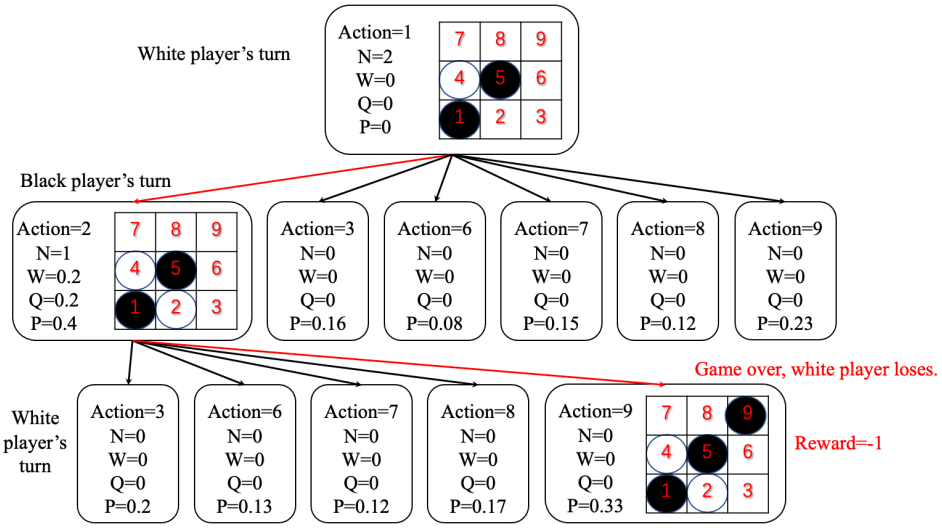


Figure 10: Expansion and evaluation at the final node. Since the game ends at this node, there will not be any expansion. The reward is obtained from the game. Therefore, policy network and value network are not used here.

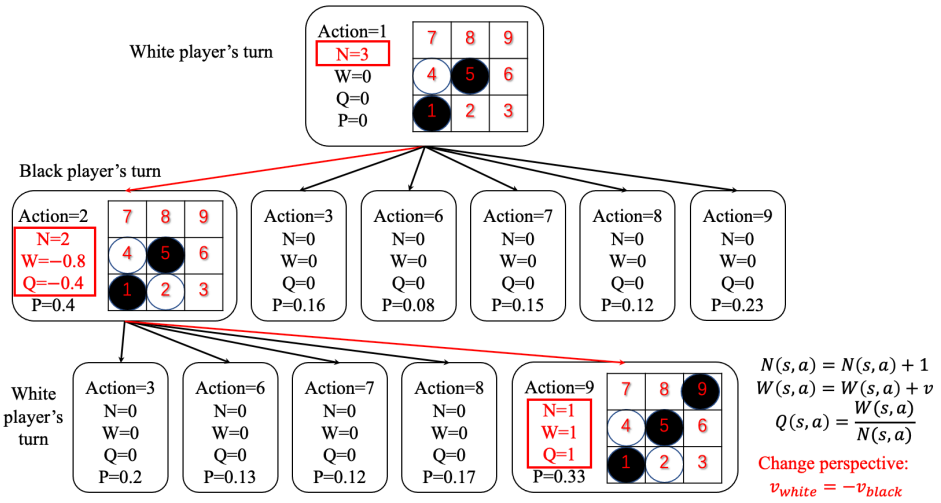


Figure 11: Tree structure after backup. During the third tree search, the backup step recursively updates the information of the three visited nodes. Since both players are in the same tree structure, and $v_{white} = -v_{black}$, we need to update the information from the correct view

to root nodes, where $N(s,a) = N(s,a) + 1$, $W(s,a) = W(s,a) + 1$, $Q(s,a) = \frac{W(s,a)}{N(s,a)}$. In addition, when updating the nodes in different levels, we should change the view as well, meaning that $v_{white} = -v_{black}$. In this game, black player selects action 9 $(b, 9)$ and reaches a new node. Then it is white player's turn to play. However, the game is over at this time and white player loses the game. White player receives a $reward = -1$ calculated from the view of white player, which is $v_{white} = -1$. The new node is a leaf node. The information stored in this node is used by black player to select action 9. Therefore, the information should be represented from the view of black player. Thus, we have $v_{black} = -v_{white} = 1$. The information in this node is updated to $N = 1$, $W = 1$, $Q = 1$. The information of the remaining two nodes is also updated in the same way. After finishing backup, tree structure is shown in Figure 11.

So far, we demonstrate three iterations of Monte Carlo tree search. After 400 iterations, as shown in Figure 12,

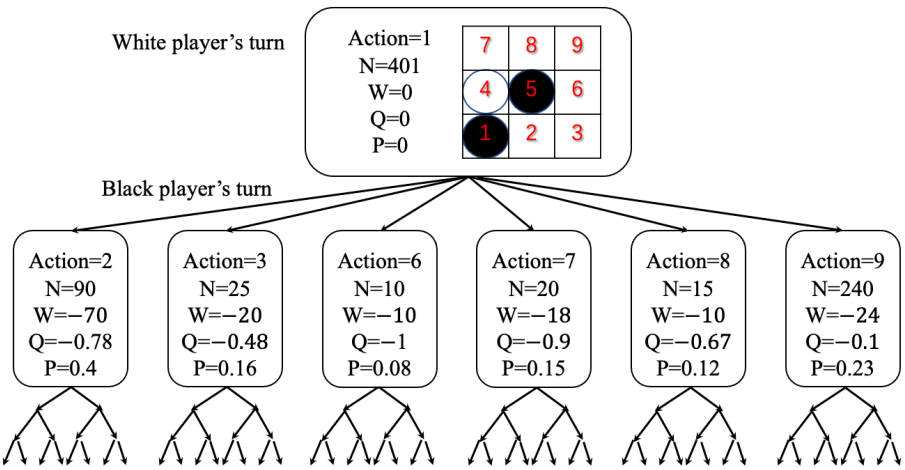


Figure 12: Tree structure after backup. During the third tree search, the backup step recursively updates the information of the three visited nodes. Since both players are in the same tree structure, and $v_{white} = -v_{black}$, we need to update the information from the correct view

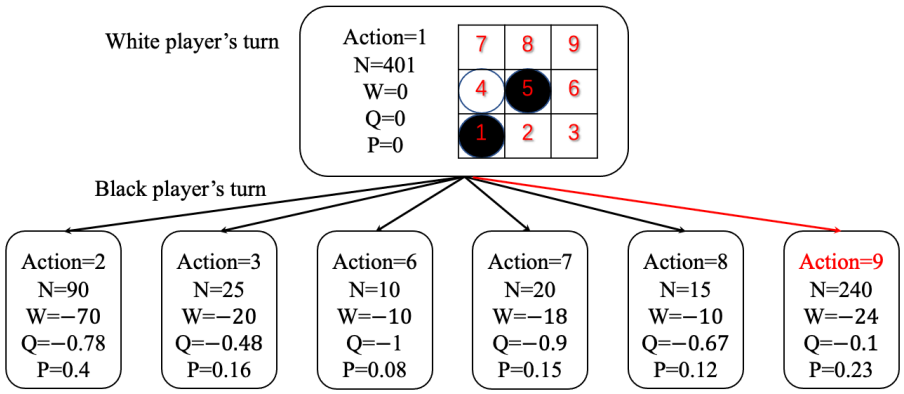
the estimated values are more accurate.

After finishing tree search process, AlphaZero is able to select an move to play. Actions are selected by calculating the number of visits for each action and normalizing them to probabilities, instead of outputting action probabilities through the policy network, $\pi(s,a) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,a)^{1/\tau}}$, where τ is a temperature parameter, $b \in A$ represents the feasible action under state s. Here, the selected action is 9 $(w, 9)$, as shown in Figure 13.

The temperature parameter τ controls the level of exploration. If $\tau = 1$, the probability of selecting an action is proportional to the number of visits, which has a high level of exploration and ensures the diversity of data. If $\tau \to 0$, action with the highest visit count is preferred. In AlphaZero and AlphaGo Zero, when performing self-play to collect data, for the first 30 moves of each game, the temperature parameter is set to $\tau = 1$. For the rest of the game, the temperature is set to $\tau \to 0$. When playing with human, the temperature is always set to $\tau \to 0$, which means the optimal move is always selected.

At this point, a white piece has been placed at position 9, so the information of the root node in the tree will be modified. As shown in Figure 14, the Monte Carlo tree search will continue to search from the new root node. Other sibling nodes and their parent nodes will be pruned and discarded to save memory.

The whole searching process is repeated until the end of the game, as shown in Figure 15. The data for each time step is stored as tuple (s_t, π_t, z_t) [1]. s_t is the current state of the current player at time step t. Next section we will introduce what it consists of. π_t is the action probabilities for each feasible action at time step t. $z_t = \pm r_t$ is the game winner from the view of current player at time step t, where $r_t = \{+1, -1, 0\}$ is the final reward. Each game contains many tuples (s_t, π_t, z_t) and all this tuples are used to train the neural network.



a	2	3	6	7	8	9
$\pi(a\|s)$	$\frac{90}{400}$	$\frac{25}{400}$	$\frac{10}{400}$	$\frac{20}{400}$	$\frac{15}{400}$	$\frac{240}{400}$

Figure 13: AlphaZero selects an action. After 400 iterations of searching, it selects an action based on $\pi(s,a) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,a)^{1/\tau}}$. Here, $\tau = 1$. The selected action is 9.

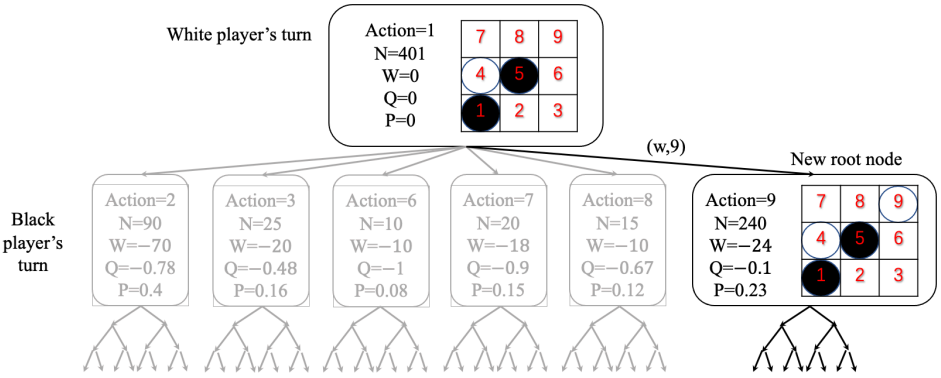


Figure 14: AlphaGobang selects an action. After 400 iterations of searching, it selects an action based on $\pi(s,a) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,a)^{1/\tau}}$. Here, $\tau = 1$. The selected action is 9.

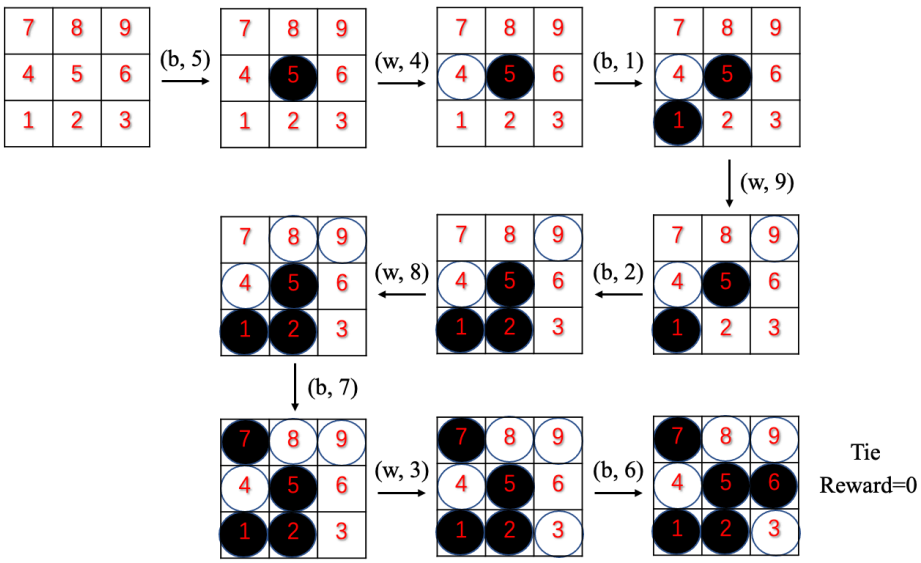


Figure 15: The self-play data. All the board states are saved together with action probabilities and values.

3.2 Training a Neural Network

With the help of Monte Carlo tree search, we are able to get the self-play data (s_t, π_t, z_t). These data are then utilized to train a deep neural network f_θ with parameters θ. A set of board representations s is fed into the neural network, which then outputs action probabilities and a value $((p, v) = f_\theta(s))$ [1]. The action probabilities p vector represents the probability of choosing each possible action. The value is a scalar that represents the probability that player s will win the game from the perspective of the current player.

The loss function l consists of the cross-entropy loss of the action distributions between π and p, the mean square error of values between z and v, and the $L2$ regularization of the parameters. The loss function is given by:

$$l = (z - v)^2 - \pi^{\mathrm{T}} \log p + c \parallel \theta \parallel^2 \tag{4}$$

where parameter c controls the regularization weight to prevent over-fitting [1]. The neural network is updated to reduce the error between the predicted value v and the self-play winner z, as well as to maximize the similarity between the action probabilities p constructed by neural network and search probabilities π.

In fact, the goal of neural network training is to allow the network to learn from MCTS. The MCTS-generated policy π is usually superior to the raw policy p generated by the neural network f_θ. We may consider the process of learning better action probabilities via MCTS to be an policy improvement. The process of using MCTS to select action and using game winner z as sampled value can be considered as a policy evaluation. The entire procedure is a policy iteration [1]: we update the parameters of the neural network such that action probabilities p and value v generated by neural network are as close to the better search probabilities π and game winner z generated by MCTS as possible. The newly updated parameters are then utilized in the next search to generate a stronger search policy.

As mentioned in Section 3, we only maintain one single neural network to generate self-play data, like in AlphaZero. However, to evaluate the performance of neural network, we design another random MCTS player without using neural network. It plays randomly against with the network-based MCTS. We not only preserve the latest model, but also save the model with best performance against random MCTS.

Next we introduce more details about the design of neural network.

3.2.1 Input Features

In AlphaGo Zero, the input is a 19×19×17 image stack consisting of 17 feature planes of the game board [1]. Positions on the game board are encoded from 1 to 361. Each 19×19 feature plane X_t consist of binary values representing the presence of the players' moves. $X_t^i = 1$ if player selects action i and place a stone on position i at time step t; $X_t^i = 0$ if opponent selects an action or position i is empty. There are totally 17 feature planes.

8 feature planes X_t represent 8 past moves of current player. Another 8 feature planes Y_t represent 8 past moves of opponent. The last feature plane C represents the color and is always either 1 when black is playing or 0 when white is playing. These 17 planes are concatenated together to form the input features as well as the states $s_t = [X_t, Y_t, X_{t-1}, Y_{t-1}, ..., X_{t-7}, Y_{t-7}, C]$. Historical game boards are necessary, because on one hand, in the game of Go, the move to be selected depends not only on the current situation, but also on the past moves; more past games may also be helpful for training. On the other hand, the judgment of winning or losing in Go also differs for the first player and the second player, so it is also necessary to tell the network whether the current player is the first player or the second player.

For the freestyle Gobang, the input features can be simplified. We aim to train a model using based on a board with size 8×8. The move to be selected only depends on the current situation of the game board and has nothing to do with past moves. Therefore, in Alpha Gobang, it is not necessary to input the historical game boards. We only take two feature planes X_t, Y_t as input features, where X_t represents the moves from the view of current player at time step t, and Y_t represents the moves from the view of opponent at time step t. Apart from these, we also add one last move of opponent L_t to the input features. $L_t^i = 1$ if opponent made his last move on position i, 0 for the rest of the places. We add L_t to the input features because the position of the next move from current player is often near the position of opponent's previous move. The the third plane makes it easier for the network to decide which position has a higher probability. The fourth feature plane C is the color to play. In the freestyle Gobang, the first player could take a great advantage. Whether to go first or not has a significant impact on the evaluation of current situation.

As a result, our input has a shape of 8×8×4 consisting of 4 feature planes, each of which is 8×8. These 4 feature planes are combined together to form the input features $s_t = [X_t, Y_t, L_t, C]$.

3.2.2 Network Structure

In AlphaGo Zero, the input features s_t first pass through a single convolutional blocks followed by 19 or 39 convolution-based residual blocks, and then connects to a 4-layer or 7-layer network to obtain policy and value repectively. The entire network has more than 40 or 80 layers, and it is very slow when training and predicting.

In our implementation, we try to simplify the structure of the network. The network starts with 3 convolutional layers, with 32, 64, 128 filters respectively and kernel size 3×3, using ReLU activation function. It is then connected to two separate 'heads' to output the policy and the value respectively. For the policy head, we first use 4 filters with kernel size 1×1 for dimensionality reduction, then connect a fully connected layer, and use the softmax function to output the probabilities of each position on the game board. For the value head, we use 2 filters with kernel size 1×1 for dimensionality reduction, then connect a fully connected layer with 64 neurons, and finally connect a fully connected layer, and use the tanh function to output value between $[-1, 1]$. The entire neural network only contains

5-6 layers, so both training and prediction are relatively faster. The network structure is shown in Figure 16.

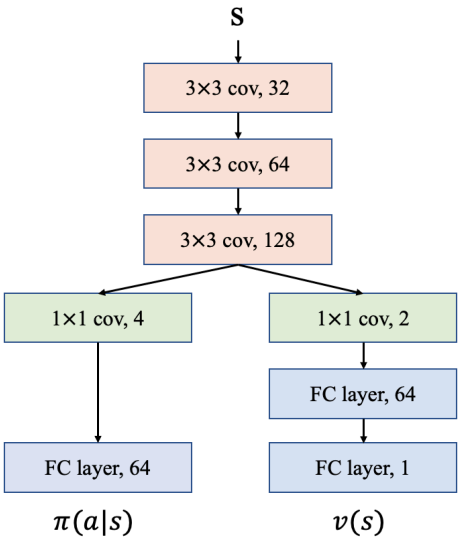


Figure 16: Neural network structure.

3.2.3 Further Optimization for the Training

So far, we are able to construct a deep neural network and train the model. However, there is still one problem that needs to be solved.

Problem of Data Imbalance As mentioned in Section 2.1, for the freestyle Gobang, if the first player plays perfectly at each move, it must win in the end. The consequence of this is that during the training, the amount of winning games for the first player will be far more than that for the second player. The data is thus unbalanced and this will lead to an unbalanced preference, causing the network to favour the first player to win even more. This preference further allows the self-play training to produce more winning games for the first player. In the end, the model may collapse, that is, the value of prediction for the first player is close to 1, and the policy is more accurate; the value of prediction for the second player is close to -1. As a result, there will be a large number of games with a length of only 9 or 11, which means that the first player quickly forms five pieces in a row, but the second player forms only four pieces or stones are arranged in a messy manner.

In fact, the average length of games should gradually increase with further training, as the model becomes more powerful and generates more games with longer length. One thing to note is that, under the freestyle rule, we found that the advantage of the first player can not be completely eliminated during the training. If the noise is large enough and the training is long enough, this problem can be alleviated, but it is rather time-consuming. Our objective is to alleviate this problem more effectively and try to keep the data balanced on both sides as much as possible.

The first approach is to keep track of how many games the first and second players have won in the data buffer. If the number of games won by a certain side is insufficient, whenever a winning game for this side is collected, this game is added to data buffer again. For instance, let N_1 and N_2 be the numbers of winning games for the first player and the second player respectively. When a game won by the second player is collected, we add this game to the data buffer again with a probability of $(\frac{N_1}{N_2 \times 0.8} - 1)$. When a game won by the first player is collected, we add this game to the data buffer again with a probability of $(\frac{N_2}{N_1} - 1)$. This solution aims to alleviate the problem of data imbalance between the two players.

The second approach is to discard the game with short length with a certain probability. The game with a length of 9 will be discarded with a probability of 0.9; the game with a length of 20 will not be discarded. As the length of a game rises from 9 to 20, the probability of discarding it decreases linearly. This solution is designed to maintain the longer games and use them to train the neural network.

The third approach is to apply training weights to each state. During the self-play training, the later state in one game appears less frequently, while the earlier state appears more frequently. Apart from this, later moves may be more associated with winning or losing. Therefore, a weight growth factor is created such that the training weight obtained by the later state is higher than the training weight obtained by the earlier state. When sampling from data buffer, states with higher weights are more likely to be sampled, which allows neural network to learn more features from complex board states. The weights calculation is given by:

$$w_i = \frac{\gamma^{L-i}}{\sum_{i=1}^{L} \gamma^{L-i}} \times L \tag{5}$$

where w_i is the weight for the i^{th} board state in one game, L is the length of a game, and γ is a growth factor, which is set to 0.95. The last board state in one game gives the factor of 1, and the rest of states give decaying factors. The weights are relevant to the length of game to avoid a large number of repeating weights. The summation of all weights is equal to the length of a game.

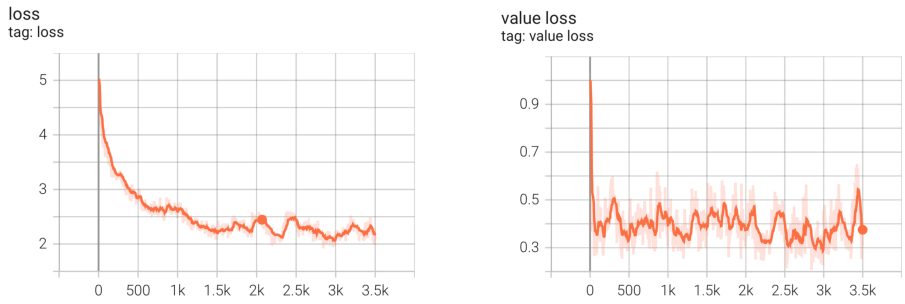
Algorithm 1 Algorithm A with a Reservoir (A-Res) [12]

Input: A population V of n weighted items
Output: A reservoir R with a WRS of size m
The first m items of V are inserted into R
for each item $v_i \in R$ **do**
  u_i = random(0,1)
  Calculate a key $k_i = u_i^{1/w_i}$
end for
for $i = m+1, m+2, .., n$ **do**
  The smallest key in R is the current threshold T
  For item v_i: Calculate a key $k_i = u_i^{1/w_i}$, where u_i = random(0,1)
  if the key k_i is larger than T **then**
    The item with the minimum key in R is replaced by item v_i
  end if
end for

We also apply a another algorithm called Algorithm A With a Reservoir (A-Res) [12] to help us to solve the problem of weighted random sampling (WRS) without replacement. It inputs a population V of n weighted items and outputs a reservoir R with a WRS of size m. The process is illustrated in Algorithm 1. We can apply this algorithm to sample the weighted states without replacement.

4 Experimental Results

We implemented Alpha Gobang using Pytorch based on Python3.7. We trained the model for 3500 iterations on the board with size 8×8. The total loss function and the loss of value are shown in Figure 17. We can see both of these decrease as training continues. The total loss function decreases from 5 to around 2 and 2.5. The value loss decreases as well and finally fluctuate between 0.3 and 0.5.



(a) Total loss (b) Value loss

Figure 17: Total loss function and loss of value

During the training, in addition to observing that the loss function is slowly decreasing, we also pay attention to the changes in the entropy of the policy (action probability distribution) output by the policy network, which is demonstrated in Figure 18. Normally, at the very beginning, policy network output a uniform random probability, so the entropy will be relatively large. As the training continues, the policy network will gradually learn which position should have a greater probability to place in different situations. The distribution of action probability is no longer uniform, and there will be a stronger preference, so entropy becomes smaller. It is because of the preference of the output probability that it can help MCTS to perform more simulations in more potential positions during the search process, so as to achieve better performance with fewer simulations.

There is another important feature, the length of each game or episode. Episode length represents the total number of moves in one game during the self-play stage. Figure 19 shows the average length of episodes during the training. It can be seen from the figure that in the early stage of training, due to the completely random moves, the number of moves in the game is very long. As the training continues, the number of moves in one game drops rapidly, indicating that the model has gradually mastered the initial rules of the game. According to the rules, one needs to place five pieces in a line to win, so the model just quickly put them in a line. At this time, it only knows how to attack, but do not know how to defend. Therefore, from 500 iterations to 1500 iterations, the average length

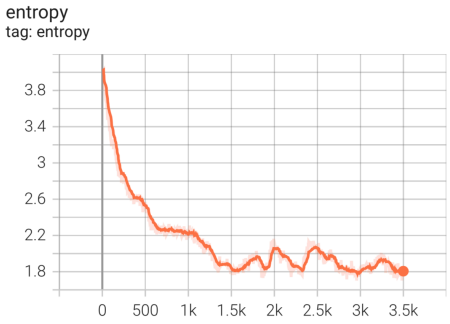


Figure 18: Cross entropy

is only about 14 to 16. As the training goes on, the model gradually learns how to defend, and the game gradually becomes longer. From Figure 19, although there are some fluctuations after 1500 iterations, the average length is still growing slowly.

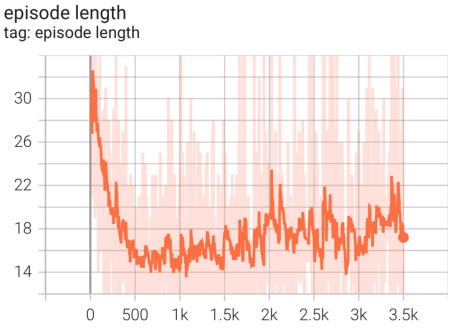


Figure 19: Episode length for 3500-iterations training

We further train another 1700 iterations based on current model and obtain the figure of average episode length, which is shown in Figure 20. As the training continues, the average length stays at about 20, above or below.

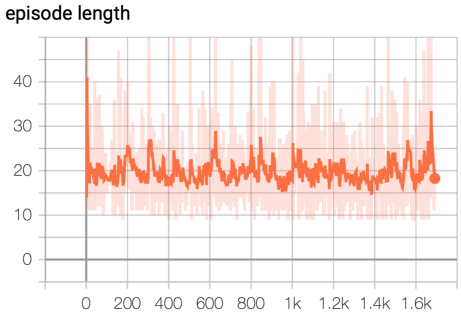


Figure 20: Episode length after retraining 1700 iterations

Figure 21 shows one self-play game generated by the model trained for 5500 iterations. We can see the black player keeps attacking and forming multiple four-pieces lines. The white player has to defend, but it sometimes organizes several counterattacks. However, it appears that the white player is indeed at a disadvantage, and in the end the black wins. From this figure, we can see that Alpha Gobang has already learned how to attack and defend.

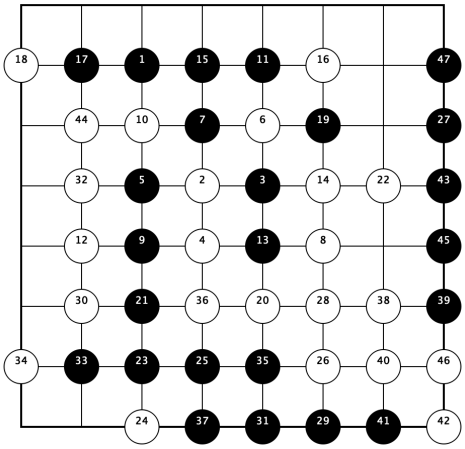


Figure 21: Detailed information of one self-play game

We also play against with the model trained for 5500 iterations. To play with it, we only need to enter the coordinate of the position. Figure 22 shows the interface of the game board. For example, if we want to put a stone at position (4,3), we only need to enter '4,3', then the stone will be placed at position (4,3).

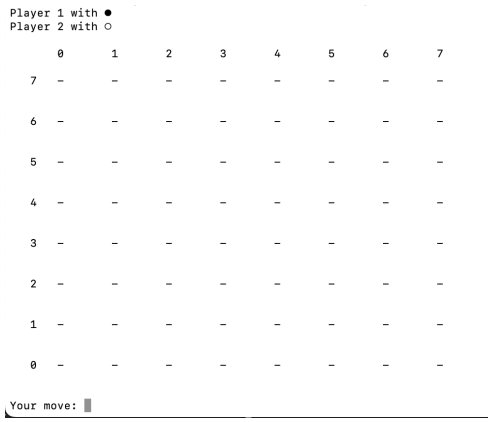
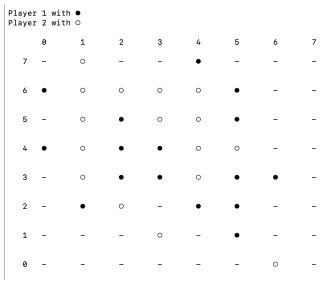


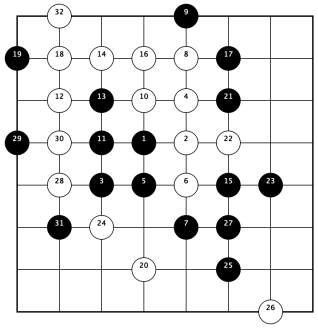
Figure 22: Game board interface

In fact, most of time, when we play against with the model, the game ends with a draw, because of the small board size (the size of 8×8 is much smaller than the formal size of 15×15). However, there exists some exceptions. Figures 23 show the two games of human playing against with Alpha Gobang. In both games, human is the black player and plays first, but human loses in the end. The lengths of both games are higher than 30, which indicates Alpha Gobang performs quite well even in the later stage. From the detailed information, at the very beginning, the stone positions of both players are very close to each other. They both want to prevent each other from connecting three pieces in a line. Alpha Gobang quickly realizes the attacks and defends effectively. Despite being on the defensive in the early stage, it develops some tricks to win the game in the end.

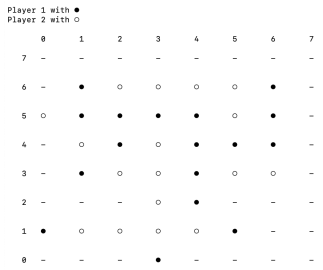
Figure 24 demonstrates one game won by human. In this game, human is the black player and plays first. Instead of connecting four pieces in a row, human tries to form a large amount of three-pieces lines to restrict Alpha Gobang and make it unable to take care of itself. It has to defend passively and loses in the end. It is



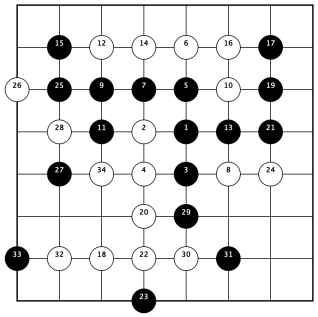
(a) Interface of the first game playing with the model



(b) Detailed information of the first game



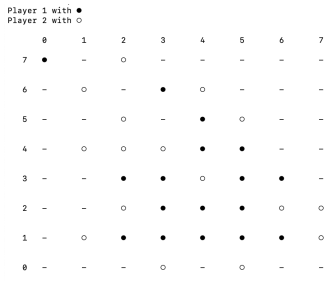
(c) Interface of the second game playing with the model



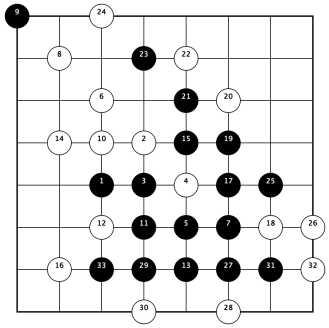
(d) Detailed information of the second game

Figure 23: Two games playing against with Alpha Gobang. In both games, human is the black player and plays first, but loses in the end.

undeniable that the black player takes a great advantage, but it is obvious that Alpha Gobang does not develop more effective defensive policies when facing more complex and unusual attacks.



(a) Interface of the game playing with the model



(b) Detailed information of the game

Figure 24: One game playing against with Alpha Gobang won by human. Human is the black player and plays first.

To summarize, Alpha Gobang has already learned the basic rule of the freestyle Gobang. It knows how to attack and defend. Sometimes it can even play some tricks. For example, it may deliberately stagger one stone on the board and place it to another position, in order not to be found that it is connecting three or four pieces in a line. However, because of the limit of the board size, Alpha Gobang can not develop more sophisticated methods of attack and defense , which is one of aspects to be optimized in the future.

5 Conclusion

This report introduces the details of implementation of Alpha Gobang based on AlphaGo Zero and AlphaZero, including the Monte Carlo tree search process, the design of neural network and the experimental results. It turns out that the implemented algorithm performs quite well, both in the training process and the game between human and model. However, there are still several aspects that need to be improved. First, our model only runs on the board with size 8×8, which is much smaller than 15×15, the formal board size of Gobang. Due to the small board size, the model cannot derive more policies to attack and defend during training. In the future, we need to train a model that meets the size of 15×15. Second, in order to meet a board with larger size, we need to further optimize the structure of neural network, since the larger board usually has action space and the state space. We need more layers to extract the features from larger feature planes. Last but not least, we can train different models to match different rules of Gobang. So far, we only train the model under the freestyle rule. This is the original rule and not applicable to the formal competition. Under this rule, the advantage of the first player can not be completely eliminated, which causes a series problems during training, such as data imbalance. By applying other more complex rules, it is possible to resolve the problem of data imbalance.

References

[1] 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.

[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] ——, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[4] 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.

[5] M. H. Albert, R. J. Nowakowski, and D. Wolfe, *Lessons in play: an introduction to combinatorial game theory*. CRC Press, 2019.

[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. J. Osborne and A. Rubinstein, *A course in game theory*. MIT press, 1994.

[8] F.-h. Hsu, "Ibm's deep blue chess grandmaster chips," *IEEE micro*, vol. 19, no. 2, pp. 70–81, 1999.

[9] A. Couëtoux, M. Milone, M. Brendel, H. Doghmen, M. Sebag, and O. Teytaud, "Continuous rapid action value estimates," in *Asian conference on machine learning*. PMLR, 2011, pp. 19–31.

[10] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006: 17th European Conference on Machine Learning Berlin, Germany, September 18-22, 2006 Proceedings 17*. Springer, 2006, pp. 282–293.

[11] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, pp. 235–256, 2002.

[12] P. S. Efraimidis and P. G. Spirakis, "Weighted random sampling with a reservoir," *Information processing letters*, vol. 97, no. 5, pp. 181–185, 2006.