Artificial Intelligence Reversi Report

SID: 11812101
Name: Cai Yufei

1. Preliminary

1.1 Problem Description

1.1.1 Goal

The problem for this project is to use artificial intelligence to play Reversi (usually this game is called Othello in the west).

1.1.2 Algorithms

The project focuses on some specific fields in order to learn some algorithms which I was not familiar with. Minimax algorithm with Alpha Beta pruning is so familiar to me since I once used them to finish some other projects. So I tried many other algorithms and get some experiences on MCTS algorithm and deep learning. It is very interesting to explore new things although these trials are not included in the final code of my project (MCTS is still preserved in final code but it does not participate in the decision process).
In this project, I realized three algorithms which are completely different. The first algorithm is based on traditional Alpha Beta Search. The second is based on pure randomly MCTS. The last version is based on the paper of Deep Mind [1], which is a combination of MCTS and deep learning (reinforcement learning).

1.1.3 Softwares

Python 3.8 - Implement the entire AI
TensorFlow 2.0 - Deep learning

1.2 Problem Applications

The AI of Reversi can help humans understand the game better, stimulate and aid Othello players to achieve higher performance. Moreover, the problem can be extended to a larger problem of how to use computers to play fighting games, which is a basic problem in AI and theoretical game theory of mathematics.

2. Methodology

2.1 Notation

In this section, we restrict our presentation to the notations and concepts used in the rest of the report. Basically, the state of one point in a chessboard has three possibilities. They are empty(notated as 0), black(notated as -1) and white(notated as 1). We define the notation $state(x, y) = z$ where $x, y \in \{0, 1, 2, 3, 4, 5, 6, 7\}, z \in \{0, -1, 1\}$ to denote the state of the chessboard point of position (x, y) is z.
To describe the state of current chessboard, we define the following function:

$$chessboard(T) = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} & a_{0,6} & a_{0,7} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} \\ a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} \\ a_{6,0} & a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} \\ a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} \end{pmatrix}$$

where $state(x, y) = a_{x,y}$ and $T \in Z \wedge T \in [0, length]$, $x, y \in \{0, 1, 2, 3, 4, 5, 6, 7\}$. The constant $length$ is the number of total steps of the game and $length \in Z \wedge length \in [0, 59]$ according to the rule of Othello. The constant T is the time point of current chessboard. Generally speaking, $chessboard(T)$ is the chessboard state matrix at the time point T.
Then here comes another important definition. We define $game_i = \{chessboard_i(0), chessboard_i(1), ..., chessboard_i(length_i)\}$ as a complete game of Othello where $length_i$ is the length of $game_i$ and $chessboard_i$ is the chessboard function of $game_i$. Explicitly, $game_i$ is a set of all chessboard states of any time point during the game.
For each chessboard state, we should have a value in order to evaluate the goodness of the state. We define $Evaluation(chessboard(T), player)$ where $player \in \{0, -1, 1\}$ as the value which indicates the degree of goodness of $chessboard(T)$ to $player$.
For every position of one specific chessboard state, there is a probability which describes the possibility to win if the player place his chessman on this position. We define:

$$prob(chessboard(T), player) = \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} & b_{0,4} & b_{0,5} & b_{0,6} & b_{0,7} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} & b_{1,5} & b_{1,6} & b_{1,7} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} & b_{2,5} & b_{2,6} & b_{2,7} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} & b_{3,5} & b_{3,6} & b_{3,7} \\ b_{4,0} & b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} & b_{4,5} & b_{4,6} & b_{4,7} \\ b_{5,0} & b_{5,1} & b_{5,2} & b_{5,3} & b_{5,4} & b_{5,5} & b_{5,6} & b_{5,7} \\ b_{6,0} & b_{6,1} & b_{6,2} & b_{6,3} & b_{6,4} & b_{6,5} & b_{6,6} & b_{6,7} \\ b_{7,0} & b_{7,1} & b_{7,2} & b_{7,3} & b_{7,4} & b_{7,5} & b_{7,6} & b_{7,7} \end{pmatrix}$$

where $b_{i,j}$ is the probability to win if $player$ place his chessman on the position (i, j) right after the time point T.
Since the game will always keep fighting, according to the characteristic of fighting games, we define

$best\_policy(chessboard(T)) = (i, j)$ where $chessboard(T)$ is a randomly given chessboard state and the vector (i, j) is the position we need to play.

2.2 Alpha Beta Algorithm

2.2.1 Data Structure

- candidate_list: A list of the positions which can be placed by color of the program.
- DIR: A constant list of eight vectors(tuples) representing eight different directions.
- placed: The number of chessmen placed on the chessboard.
- myColor: The color of the program.
- opColor: The color of the opponent of the program.
- size: The size of chessboard.
- AI: The main class of the program which stores the game information and decision function.
- vis: A list of sets which contains all the hash values of chessboard which was searched in the past. The length of the list is 65, indices of which indicates how many chessmen there are on the chessboard state. The list of sets is used to accelerate the searching rates and prune some bad states.

2.2.2 Model design

The goal for our program is to calculate a best policy $best\_policy(chessboard(T))$ for a given chessboard state $chessboard(T)$.
Because the two players have opposite goals towards the chessboard, the advantage of one player is the disadvantage of the other. So we need to define the best policy more precisely and explicitly. Here we assume that the advantage of the program player is positive, and the advantage of the opponent of the program is negative. So we have the following definition:

$$best\_policy(chessboard(T), player) = \\ \mathop{\arg\max}_{(i,j) for (T+1)} (\min(\max(...(evaluation(chessboard(T+m), player \times (-1)^m)...))))$$

This definition is typical of Minimax algorithm.
The hyperparameter m is determined by the running time limit in reality. Since the algorithm cannot end in polynomial time, we are bound not to search until the game is end. If we do that, the number of cases we search will be x^{60} where $x \in [0, 60]$. And in most cases, $x \geq 2$. As a result, this time complexity is not acceptable. Only searching for a little levels is a good idea. But then we need to judge whether a chessboard state is good without the final state of the chessboard. Here comes the evaluation function.
Evaluation function is a function which we turn to when we do not have full information about the game. But if the game is end, we can simply use the final result to judge the goodness of the chessboard. We extend the evaluation function into a broader one:

$$Evaluation(chessboard(T), player) = \begin{cases} my\_num - op\_num, & game\_end \\ my\_advantage - op\_advantage, & game\_not\_end \end{cases}$$

The basic idea of Minimax is above.
However, since the Minimax algorithm is non-polynomial and time-consuming, we are supposed to design new ways to implement them. Alpha Beta pruning is one of the best way to reduce its implement time complexity.
When we are searching and predicting the possible cases of chessboard in the future, sometimes we will find some really good or really bad points through back tracing. In such cases, we should not search forward along the bad route and must return to the shallow level.
Let alpha be the maximum value of all the returned evaluation values of maximum level. What we want to ensure is that we find the maximum value in this level. So if there are some values which are smaller than the maximum value returned from the minimum level, we should drop them away and prune the search in the minimum level. And let beta be the minimum value of all the returned evaluation values of maximum level. What we want to ensure is that we find the minimum value in this level. So if there are some values which are greater than the minimum value returned from the maximum level, we should drop them away and prune the search in the maximum level.
After grasping the principles of Alpha Beta pruning, we can prune all the useless cases in our search and make the Minimax algorithm end earlier.

2.2.3 Detail of algorithms

We are to implement the Minimax algorithm together with Alpha Beta pruning.
In the very beginning, we should have a main function which receives the chessboard state and the player as parameters and returns the best policy of the state. But before that, we are supposed to implement a function which gets the chessboard state and the player as parameters and returns a list of available legal moves.
In fact, we can use the hash value of a chessboard state to get better performances. The idea is also simple. If the chessboard state which we have visited is available move again, it means that this state is not pruned by Alpha Beta pruning and we can give this state a prior to search firstly. This can be implemented by a list of sets recording the states we have been to.
Pseudo Code:

```
vis:list(set())
def make_decision(chessboard(T),player):
    moves=get_moves(chessboard(T),player)
    alpha=-INF
    best_policy=(-1,-1)
    hash_value=get_hash_value(chessboard(T))
    depth=get_depth(chessboard(T))
    for move in moves:
        chessboard(T+1)=chessboard(T).place(move)
        evaluation=min_level(chessboard(T+1),-player,depth-1,alpha,hash_value+edit_hash_value(move))
        if evaluation>alpha:
            alpha=evaluation
            best_policy=move
    return best_policy
def min_level(chessboard(T),player,depth,alpha,hash_value):
    moves=get_moves(chessboard(T),player)
    beta=INF
    if game_end(chessboard(T)):
        evaluate(chessboard(T),True)
    elif depth==0 or is_time_out():
        evaluate(chessboard(T),False)
    vis[chessmen_num(chessboard(T))].add(hash_value)
    reorder_moves_by_hash(moves)
    for move in moves:
        chessboard(T+1)=chessboard(T).place(move)
        evaluation=max_level(chessboard(T+1),-player,depth-1,beta,hash_value+edit_hash_value(move))
        if evaluation<=alpha:
            return evaluation
        if evaluation<beta:
            beta=evaluation
    if game_pass():
        beta=max_level(chessboard(T+1),-player,depth-1,beta,hash_value)
    return beta
def max_level(chessboard(T),player,depth,beta,hash_value):
    moves=get_moves(chessboard(T),player)
    alpha=-INF
    if game_end(chessboard(T)):
        evaluate(chessboard(T),True)
    elif depth==0 or is_time_out():
        evaluate(chessboard(T),False)
    vis[chessmen_num(chessboard(T))].add(hash_value)
    reorder_moves_by_hash(moves)
    for move in moves:
        chessboard(T+1)=chessboard(T).place(move)
        evaluation=min_level(chessboard(T+1),-player,depth-1,alpha,hash_value+edit_hash_value(move))
        if evaluation>=beta:
            return evaluation
        if evaluation>alpha:
            alpha=evaluation
    if game_pass():
        alpha=min_level(chessboard(T+1),-player,depth-1,alpha,hash_value)
    return alpha
```

Now we need to consider how to implement the $Evaluation$ function.
If the game ends, it is easy enough.
If the game does not end, we should do it in another clever way with some knowledge from humans' experience.
First and foremost, what we come up with easily is the difference of number of the two players since the winner is determined by it directly. However, since the special properties of Othello, we cannot pay so much attention on it when we are at the early stage of the game.
Secondly, due to some guidelines and experience, some positions on the chessboard are much more valuable than others. For example, positions on the angles are more valuable than positions next to the angles. And positions around the centre of the board are less valuable than positions on the edge. This experience can be transmitted to a weight matrix $W_{8\times8}$. And we should calculate the following expression:

$$||W * chessboard(T)|| \times player =$$
$$||\begin{pmatrix} a_{0,0}b_{0,0} & a_{0,1}b_{0,1} & a_{0,2}b_{0,2} & a_{0,3}b_{0,3} & a_{0,4}b_{0,4} & a_{0,5}b_{0,5} & a_{0,6}b_{0,6} & a_{0,7}b_{0,7} \\ a_{1,0}b_{1,0} & a_{1,1}b_{1,1} & a_{1,2}b_{1,2} & a_{1,3}b_{1,3} & a_{1,4}b_{1,4} & a_{1,5}b_{1,5} & a_{1,6}b_{1,6} & a_{1,7}b_{1,7} \\ a_{2,0}b_{2,0} & a_{2,1}b_{2,1} & a_{2,2}b_{2,2} & a_{2,3}b_{2,3} & a_{2,4}b_{2,4} & a_{2,5}b_{2,5} & a_{2,6}b_{2,6} & a_{2,7}b_{2,7} \\ a_{3,0}b_{3,0} & a_{3,1}b_{3,1} & a_{3,2}b_{3,2} & a_{3,3}b_{3,3} & a_{3,4}b_{3,4} & a_{3,5}b_{3,5} & a_{3,6}b_{3,6} & a_{3,7}b_{3,7} \\ a_{4,0}b_{4,0} & a_{4,1}b_{4,1} & a_{4,2}b_{4,2} & a_{4,3}b_{4,3} & a_{4,4}b_{4,4} & a_{4,5}b_{4,5} & a_{4,6}b_{4,6} & a_{4,7}b_{4,7} \\ a_{5,0}b_{5,0} & a_{5,1}b_{5,1} & a_{5,2}b_{5,2} & a_{5,3}b_{5,3} & a_{5,4}b_{5,4} & a_{5,5}b_{5,5} & a_{5,6}b_{5,6} & a_{5,7}b_{5,7} \\ a_{6,0}b_{6,0} & a_{6,1}b_{6,1} & a_{6,2}b_{6,2} & a_{6,3}b_{6,3} & a_{6,4}b_{6,4} & a_{6,5}b_{6,5} & a_{6,6}b_{6,6} & a_{6,7}b_{6,7} \\ a_{7,0}b_{7,0} & a_{7,1}b_{7,1} & a_{7,2}b_{7,2} & a_{7,3}b_{7,3} & a_{7,4}b_{7,4} & a_{7,5}b_{7,5} & a_{7,6}b_{7,6} & a_{7,7}b_{7,7} \end{pmatrix}|| \times player$$

where $player$ is used to switch the perspectives.
But only these two standards are not enough to evaluate the chessboard efficiently. We should use some other criteria to evaluate the chessboard state.
The most important criteria is how many stable chessmen do the players possess. Stable chessmen are defined as chessmen which is impossible to be inverted by the opponent. The way to calculate the number of chessmen is dynamic programming.
Let $dp(i, j)$ indicates if the position (i, j) on the chessboard has a stable chessman. We define:

$$dp(i,j) = \begin{cases} 0, & (i,j)empty \\ -1, & (i,j)black\_stable \\ 1, & (i,j)white\_stable \end{cases}$$

Then we can use two-dimensional dynamic programming to calculate $dp(i, j)$. The transitional expression can be written as:

$$dp(i,j) = [dp(i, j \pm 1), dp(i \pm 1, j), dp(i \pm 1, j \pm 1)] \times Orthogonal\_vector(new\_i, new\_j)$$

To implement this process, we need to start from four angles of the chessboard and then to four edges. Conducting echelon processing operation and we get the final result.

The stable chessmen are very important throughout the game so it is necessary to include it into evaluation function.
Another important criterion is the movability. Movability is defined to be the number of positions one player can play on the chessboard. Since the available moves on an Othello chessboard is quite limited, we should pay more attention to movability especially at the beginning of the game. The algorithm to calculate movability is easy so we do not describe it.
We get the pseudo code of the evaluation function:

```
def evaluate(chessboard(T),is_end):
    if is_end:
        black_num=count_num(chessboard(T),-1)
        white_num=count_num(chessboard(T),1)
        return (black_num-white_num)*myColor
    else:
        chessmen_diff=(count_num(chessboard(T),1)-count_num(chessboard(T),-1))*myColor
        board_weight_diff=get_board_weight(chessboard(T))*myColor
        stable_chessmen_diff=(get_stable_num(chessboard(T),1))-get_stable_num(chessboard(T),-1))*myCol
        movability_diff=(get_movability(chessboard(T),1)-get_movability(T),-1)*myColor
        return chessmen_diff*chessmen_diff_coe+board_weight_diff_coe+stable_chessmen_diff*stable_chess
```

It is obviously that we have four coefficients to determine the relative weight between four different criteria. We need to divide the game into three stages to decide these coefficients.

1. Starting stage - move 1-15: This stage is quite flexible and we need to focus on the potential to win the game but not the winning condition. So in this stage we should set chessmen_diff_coe low and others relatively high. The depth of our search in this stage should be shallow because there are so many possibilities.
2. Middle stage -move 15-50: During this stage, two players are competing mainly on edges and angles and the movability becomes quite important. We need to set the movability coefficient very large and stable chessmen coefficient large. The depth of our search in this stage should be a little deeper to prevent single errors.
3. Ending stage - move 50-60: In this stage, we can search until the end of the game if we want. So what is most important is the difference of numbers of chessmen now. The depth of our search in this stage should be the deepest.

2.3 Randomly MCTS Algorithm

2.3.1 Data Structure

- Node: A basic class for MCTS search. The function of the class is to record information about every basic node on the Monte Carlo Search Tree.
- Tree: A class which is composed of a root node object in class Node, recording information like total visited times.

2.3.2 Model design

The idea of Monte Carlo tree search is to choose the best strategy or action given a game state.
We use the game tree to represent a game: each node represents a state, and moving one step from one node will reach its child node. The number of child nodes is called branch factor. The root node represents the initial state. The termination node has no child nodes.
All we want to find is the best strategy (the most promising next move). If you know your opponent's strategy, you can compete for this strategy, but in most cases you do not know the opponent's strategy, so we need to use a minimax approach, assuming that your opponent is very resourceful. He takes the best strategy every time.
Suppose the A and B are playing the game, A expect to maximize their own income. Because it is a zero-sum game, B expect to make A get the least income. The algorithm can be described as follows:

$$v_A(chessboard(T)) = \max_{a_i} v_B(chessboard(T+1))$$
$$v_B(chessboard(T)) = \min_{a_i} v_A(chessboard(T+1))$$

v_A and v_B are the benefit functions of player A and B.
That is to say, given a state $chessboard(T)$, we expect to find an action with a reward that maximizes your opponent while minimizing your reward.
Minimax is the biggest weakness of the algorithm is the need to expand the whole tree, for high-branch factor games, such as go, chess, the algorithm is difficult to deal with.
A solution to the above problem is to extend the tree structure to a certain threshold depth (expand our game tree only up to certain threshold depth d). So we need an evaluation function to evaluate non-terminating nodes. This is easy for us to judge who loses and who wins according to the current chess situation.
Monte Carlo Tree Search predicts the best strategy through multiple simulation. The core thing is search. A search is a combination traversal of the whole game tree, and a single traversal starts from the root node to an incomplete node (a node that is not full expanded). Incomplete expansion means that at least one child node is not accessed, or is called unexplored. When a node is not fully expanded, select one of its unvisited child node as the root node for a simulation (a single play out/simulation). Simulation results back propagation (propagated back) is used to update the root node of the current tree and update the statistics of the game tree node. When the whole game tree search is over, it is equivalent to getting the game tree strategy.
UCT is a core function in Monte Carlo Tree Search to select the next node for traversal:

$$UCT(v_i, v) = \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{\log(N(v))}{N(v_i)}}$$

where parameter c exploitation and exploration for balancing.
When should we end the MCTS process? When the program starts playing, "thinking time" is limited and computing power is limited (computational capacity has its boundaries too). So the safest thing to do is to do it as fully as you can with your resources.
At the end of the MCTS program, the best move is usually the node with the most access. Because in the

case of multiple visits, evaluate its goodness.
The evaluation value must be high in the end.

2.3.3 Detail of algorithms

The pseudo code of pure MCTS is very simple to implement since we do not need to put knowledge of humans into the algorithm.

```
def monte_carlo_tree_search(root):
    while resources_left(time, computational power):
        leaf=traverse(root)
        simulation_result=rollout(leaf)
        back_propagate(leaf,simulation_result)
    return best_child(root)
def traverse(node):
    while fully_expanded(node):
        node=best_uct(node)
    return pick_univisted(node.children) or node
def rollout(node):
    while non_terminal(node):
        node=rollout_policy(node)
    return result(node)
def rollout_policy(node):
    return pick_random(node.children)
def back_propagate(node, result):
   if is_root(node) return
   node.stats=update_stats(node,result)
   back_propagate(node.parent)
def best_child(node):
    pick child with highest number of visits
```

As a matter of fact, this pure randomly MCTS algorithm is used to test other two algorithms in this project. It serves as a base for the advanced algorithm - AlphaZero.

2.4 AlphaZero Algorithm

I have spent three weeks researching about the paper [1]. Before exploring the paper written by Deep Mind, I spent some time on deep learning by TensorFlow 2.0 which was unfamiliar to me. Although the result I got by this algorithm is so weak that I cannot use it to play Othello on the platform, this process of exploration is quite interesting and meaningful.

2.4.1 Data Structure

All the data structures from the pure MCTS algorithm is included in this algorithm so we omit them here.

- AlphaZeroConfig: A class which stores all the configurations of our AlphaZero algorithm.
- Node: A class of MCTS tree node which stores the information of a state.
- Game: A class which stores the whole process of one game.

- ReplayBuffer: A class which stores the sampled data from game played by networks. The data it stores is used to train the convolution neural network.
- Network: A class describing the network structure, parameters and hyperparameter.
- SharedStorage: A class which stores all the historical networks (objects of class Network) which performed well in gaming.

2.4.2 Model design

All the design was based on [1]. We just explain the entire algorithm a little bit.
What we do in this algorithm is similar to pure MCTS. However, we do not use random simulations to judge whether the situation is good or bad. Instead, we use neural networks to achieve the same goal. Just like pure MCTS, we do not need to put any knowledge of humans into the algorithm.
We should use UCB score to replace the previous UCT score used in pure MCTS.

$$UCB = UCT + value\_score$$

The $value\_score$ is given by the neural network we used to evaluate the whole chessboard state.
Let us talk more about the neural network. The input of the network is a chessboard state $chessboard(T)$, the output of it is an evaluation value and a matrix of winning probabilities $prob(chessboard(T), player)$.
Let f_θ be our neural network.

$$value\_score, prob(chessboard(T), player) = f_\theta(chessboard(T), player)$$

The cross entropy loss of the neural network is:

$$l = (z - value\_score)^2 + \pi^T \log(prob) + c\theta^2$$

where z is the final result of the game where 1 represents white winning, 0 represents game drawing and -1 represents black winning. θ is parameters of the neural network which is used as normalization.
Our training process is following:

1. Randomly generate a network and put it into shared storage;
2. Use the best network in shared storage and MCTS to generate game data (the game steps and final results);
3. Use game data we have to train the best neural network to get a new network;
4. Let the new network play with the original best network. If the new network wins, save it to shared storage to replace the best network;
5. Go back to step 2.

Because we do not need to give any experience to the algorithm, just implement the basic rules of Othello is enough. Other codes can be got directly from the pseudo code of the paper. We need to transform the psedo code into real python code.

2.4.3 Detail of algorithms

We can first write the general code and then implement some specific parts.
Pseudo Code:

```
class AlphaZeroConfig(object):
  def __init__(self):
    self.num_actors = 5000
    self.num_sampling_moves = 30
    self.max_moves = 60
    self.num_simulations = 800
    self.root_dirichlet_alpha = 0.3
    self.root_exploration_fraction = 0.25
    self.pb_c_base = 19652
    self.pb_c_init = 1.25
    self.training_steps = int(700e3)
    self.checkpoint_interval = int(1e3)
    self.window_size = int(1e6)
    self.batch_size = 4096
    self.weight_decay = 1e-4
    self.momentum = 0.9
    self.learning_rate_schedule = {
        0: 2e-1,
        100e3: 2e-2,
        300e3: 2e-3,
        500e3: 2e-4
    }
class Node(object):
  def __init__(self, prior: float):
    self.visit_count = 0
    self.to_play = -1
    self.prior = prior
    self.value_sum = 0
    self.children = {}
  def expanded(self):
    return len(self.children) > 0
  def value(self):
    if self.visit_count == 0:
      return 0
    return self.value_sum / self.visit_count
class Game(object):
  def __init__(self, history=None):
    self.history = history or []
    self.child_visits = []
    self.num_actions = 4672
  def terminal(self):
      #need to implement
    pass
  def terminal_value(self, to_play):
      #need to implement
    pass
  def legal_actions(self):
      #need to implement
    return []
  def clone(self):
    return Game(list(self.history))
  def apply(self, action):
    self.history.append(action)
  def store_search_statistics(self, root):
    sum_visits = sum(child.visit_count for child in root.children.itervalues())
    self.child_visits.append([
        root.children[a].visit_count / sum_visits if a in root.children else 0
        for a in range(self.num_actions)
    ])
  def make_image(self, state_index: int):
    return []
  def make_target(self, state_index: int):
    return (self.terminal_value(state_index % 2),
            self.child_visits[state_index])
  def to_play(self):
    return len(self.history) % 2
class ReplayBuffer(object):
  def __init__(self, config: AlphaZeroConfig):
    self.window_size = config.window_size
    self.batch_size = config.batch_size
    self.buffer = []
  def save_game(self, game):
    if len(self.buffer) > self.window_size:
      self.buffer.pop(0)
    self.buffer.append(game)
  def sample_batch(self):
    move_sum = float(sum(len(g.history) for g in self.buffer))
    games = numpy.random.choice(
        self.buffer,
        size=self.batch_size,
        p=[len(g.history) / move_sum for g in self.buffer])
    game_pos = [(g, numpy.random.randint(len(g.history))) for g in games]
    return [(g.make_image(i), g.make_target(i)) for (g, i) in game_pos]
class Network(object):
  def inference(self, image):
      #need to implement
    return (-1, {})
  def get_weights(self):
      #need to implement
    return []
class SharedStorage(object):
  def __init__(self):
    self._networks = {}
  def latest_network(self) -> Network:
    if self._networks:
      return self._networks[max(self._networks.iterkeys())]
    else:
      return make_uniform_network()
  def save_network(self, step: int, network: Network):
    self._networks[step] = network
def alphazero(config: AlphaZeroConfig):
  storage = SharedStorage()
  replay_buffer = ReplayBuffer(config)
  for i in range(config.num_actors):
    launch_job(run_selfplay, config, storage, replay_buffer)
  train_network(config, storage, replay_buffer)
  return storage.latest_network()
def run_selfplay(config: AlphaZeroConfig, storage: SharedStorage,
                 replay_buffer: ReplayBuffer):
  while True:
    network = storage.latest_network()
    game = play_game(config, network)
    replay_buffer.save_game(game)
def play_game(config: AlphaZeroConfig, network: Network):
  game = Game()
  while not game.terminal() and len(game.history) < config.max_moves:
    action, root = run_mcts(config, game, network)
    game.apply(action)
    game.store_search_statistics(root)
  return game
def run_mcts(config: AlphaZeroConfig, game: Game, network: Network):
  root = Node(0)
  evaluate(root, game, network)
  add_exploration_noise(config, root)
  for _ in range(config.num_simulations):
    node = root
    scratch_game = game.clone()
    search_path = [node]
    while node.expanded():
      action, node = select_child(config, node)
      scratch_game.apply(action)
      search_path.append(node)
    value = evaluate(node, scratch_game, network)
    backpropagate(search_path, value, scratch_game.to_play())
  return select_action(config, game, root), root
def select_action(config: AlphaZeroConfig, game: Game, root: Node):
  visit_counts = [(child.visit_count, action)
                  for action, child in root.children.iteritems()]
  if len(game.history) < config.num_sampling_moves:
    _, action = softmax_sample(visit_counts)
  else:
    _, action = max(visit_counts)
  return action
def select_child(config: AlphaZeroConfig, node: Node):
  _, action, child = max((ucb_score(config, node, child), action, child)
                         for action, child in node.children.iteritems())
  return action, child
def ucb_score(config: AlphaZeroConfig, parent: Node, child: Node):
  pb_c = math.log((parent.visit_count + config.pb_c_base + 1) /
                  config.pb_c_base) + config.pb_c_init
  pb_c *= math.sqrt(parent.visit_count) / (child.visit_count + 1)
  prior_score = pb_c * child.prior
  value_score = child.value()
  return prior_score + value_score
def evaluate(node: Node, game: Game, network: Network):
  value, policy_logits = network.inference(game.make_image(-1))
  node.to_play = game.to_play()
  policy = {a: math.exp(policy_logits[a]) for a in game.legal_actions()}
  policy_sum = sum(policy.itervalues())
  for action, p in policy.iteritems():
    node.children[action] = Node(p / policy_sum)
  return value
```

```
def backpropagate(search_path: List[Node], value: float, to_play):
  for node in search_path:
    node.value_sum += value if node.to_play == to_play else (1 - value)
    node.visit_count += 1
def add_exploration_noise(config: AlphaZeroConfig, node: Node):
  actions = node.children.keys()
  noise = numpy.random.gamma(config.root_dirichlet_alpha, 1, len(actions))
  frac = config.root_exploration_fraction
  for a, n in zip(actions, noise):
    node.children[a].prior = node.children[a].prior * (1 - frac) + n * frac
def train_network(config: AlphaZeroConfig, storage: SharedStorage,
                  replay_buffer: ReplayBuffer):
  network = Network()
  optimizer = tf.train.MomentumOptimizer(config.learning_rate_schedule,
                                         config.momentum)
  for i in range(config.training_steps):
    if i % config.checkpoint_interval == 0:
      storage.save_network(i, network)
    batch = replay_buffer.sample_batch()
    update_weights(optimizer, network, batch, config.weight_decay)
  storage.save_network(config.training_steps, network)
def update_weights(optimizer: tf.train.Optimizer, network: Network, batch,
                   weight_decay: float):
  loss = 0
  for image, (target_value, target_policy) in batch:
    value, policy_logits = network.inference(image)
    loss += (
        tf.losses.mean_squared_error(value, target_value) +
        tf.nn.softmax_cross_entropy_with_logits(
            logits=policy_logits, labels=target_policy))
  for weights in network.get_weights():
    loss += weight_decay * tf.nn.l2_loss(weights)
  optimizer.minimize(loss)
```

The remaining task is to design a useful and powerful network structure. We use 3 convolution layers and several dense layers.

```
self.input_states = tf.placeholder(
    tf.float32, shape=[None, 4, board_height, board_width])
self.input_state = tf.transpose(self.input_states, [0, 2, 3, 1])
self.conv1 = tf.layers.conv2d(inputs=self.input_state,
                              filters=32, kernel_size=[3, 3],
                              padding="same", data_format="channels_last",
                              activation=tf.nn.relu)
self.conv2 = tf.layers.conv2d(inputs=self.conv1, filters=64,
                              kernel_size=[3, 3], padding="same",
                              data_format="channels_last",
                              activation=tf.nn.relu)
self.conv3 = tf.layers.conv2d(inputs=self.conv2, filters=128,
                              kernel_size=[3, 3], padding="same",
                              data_format="channels_last",
                              activation=tf.nn.relu)
self.action_conv = tf.layers.conv2d(inputs=self.conv3, filters=4,
                                    kernel_size=[1, 1], padding="same",
                                    data_format="channels_last",
                                    activation=tf.nn.relu)
self.action_conv_flat = tf.reshape(
    self.action_conv, [-1, 4 * board_height * board_width])
self.action_fc = tf.layers.dense(inputs=self.action_conv_flat,
                                 units=board_height * board_width,
                                 activation=tf.nn.log_softmax)
self.evaluation_conv = tf.layers.conv2d(inputs=self.conv3, filters=2,
                                        kernel_size=[1, 1],
                                        padding="same",
                                        data_format="channels_last",
                                        activation=tf.nn.relu)
self.evaluation_conv_flat = tf.reshape(
    self.evaluation_conv, [-1, 2 * board_height * board_width])
self.evaluation_fc1 = tf.layers.dense(inputs=self.evaluation_conv_flat,
                                      units=64, activation=tf.nn.relu)
self.evaluation_fc2 = tf.layers.dense(inputs=self.evaluation_fc1,
                                      units=1, activation=tf.nn.tanh)
self.mcts_probs = tf.placeholder(
    tf.float32, shape=[None, board_height*board_width])
```

3. Empirical Verification

3.1 Dataset

In the usability test, it is not necessary to create some special test cases by myself from my point of view mainly because I can just turn to logics to judge whether the available moves given by my program is correct or not. The method is to check the logics of the code and write down something on paper to confirm the correctness and wholeness of the logics. Logics-oriented programming is a basic technique for us especially when we coding basic algorithms.
For the points race and the round robin, things are a little bit different. Gaming algorithms are engineering algorithms whose logics cannot be determined by single analysis, and we should use some testing means to

judge whether the algorithms run normally as expected. So what I do to test my program relies on the following three testing stages:

- Stage 1 - Third-party Othello platforms: When I finished my program, what I do firstly is to use a third-party Othello platform to check if my code is able to give legal moves. In this stage, what I do is just to test if the given moves are legal. If the moves are legal and the program will not crash, then we go to the next testing stage. The testing method is simple in this stage - just play with a program which generate legal moves and play randomly.
- Stage 2 - Creating a platform to monitor all the variables during gaming. In the stage, what I do is to ensure the functions designed by myself is useful and will not behave abnormally or out of my expectation. What I do is to write a testing platform by myself and let my program play with a random one, watching the variables when playing and judge whether they are valid.
- Stage 3 - Online third-party Othello platforms playing with other AIs: After ensuring all the functions of the program run normally, we can start to improve the level of ability to play Othello of our programs. In this stage, we should let the program to play with some well-organized programs whose level are high enough. In this stage we should edit the parameters and hyperparameter of the program to make it (seem) more clever and intelligent.

3.2 Performance Measure

In the project, how to measure the performance of a program is a significant and difficult problem. It is hard because this program solves an engineering problem and we cannot judge it by traditional methods like correctness, time complexity and space complexity. We even do not know how to define correctness.
So we define another system to evaluate the performance of the algorithm:

- Average time needed on one step: We should make the average time on one step as small as possible and try to increase search depth using the remaining time. So the average time on one step should be as large as possible but not too large.
- Time needed to evaluate in the worst case: The maximum time on one step should be restricted tightly. And if the program fails to terminate in the time limit, the decision it makes will be very stupid and silly.
- Whether the code pass the usability test: This measurement is so basic that the whole program fails if we cannot pass the test.
- The rank of the code in the points race: The rank takes up a large part of the project grades.
- The rank of the code in the round robin: The rank takes up a large part of the project grades.
- The winning percentage of the program on third-party platforms with high-level programs: The percentage is essential and is the main measurement of the real intelligence of AI. I use this measurement mainly to judge how good is a version of a algorithm.

3.3 Hyperparameter

3.3.1 Alpha Beta Algorithm

The parameters of the basic algorithm are following:

- Searching levels of the program in different stages:
  - Starting stage - move 1-15: Searching for 3 levels.
  - Pre-middle stage - move 16-35: Searching for 4 levels.
  - Post-middle stage - move 36-50: Searching for 5 levels.
  - Ending stage - move 51-60: Searching for 9 levels (till the end).
- The coefficients of different components in evaluation function in different stages:

Stage	Board Position	Chessmen Difference	Movability Difference	Stable Chessmen	Edge
1-15	1	-5	0	2000	5
16-35	1	-25	0	2000	10
36-50	1	-10	600	2000	15
51-60	0	35	50	2000	5

From the table we can find that stable chessmen is the most important whatever the stage is. And during the first three stages, what we need is good positions but not more chessmen. Situation is different in the last stage.

3.3.2 Randomly MCTS Algorithm

In randomly MCTS algorithm, the only calculation expression given by us is the function UCT:

$$UCT(v_i, v) = \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{\log(N(v))}{N(v_i)}}$$

So the only parameter is c. If we want to make the search accuracy 0.95, c should be set to 1.96.

3.3.3 AlphaZero Algorithm

Hyperparameter in this algorithm is quite much. Since the AlphaZero algorithm is based on deep learning. Here we only give some basic parameters by pseudo code:

```
# Number of self-play games playing simultaneously
self.num_actors = 5000
# Number of chessboard states when sampling once
self.num_sampling_moves = 30
# Maximum number of move in one game
self.max_moves = 60
# Number of simulations
self.num_simulations = 800
self.root_dirichlet_alpha = 0.3
self.root_exploration_fraction = 0.25
self.pb_c_base = 19652
self.pb_c_init = 1.25
self.training_steps = int(700e3)
self.checkpoint_interval = int(1e3)
self.window_size = int(1e6)
# The batch size of the neural network
self.batch_size = 4096
self.weight_decay = 1e-4
# The momentum when training the neural network
self.momentum = 0.9
self.learning_rate_schedule = {
    0: 2e-1,
    100e3: 2e-2,
    300e3: 2e-3,
    500e3: 2e-4
}
```

As a matter of fact, there are more parameters in the neural network. However, the structure of our neural network is fixed on purpose because we do not have that time to edit all the parameters. So we do not edit the parameters of neural networks. The hyperparameter of neural network can be seen in pseudo code from part 2 in the report.

3.4 Experimental results

3.4.1 Test cases the code passed in the usability test

All test cases are passed at the first time of submission.

3.4.2 Rank in the points race

In the points race, only the Alpha Beta algorithm is used and the rank is 69.

3.4.3 Rank in rank in the round robin

In the round robin, only the Alpha Beta algorithm is used and the rank is 55.

3.4.4 Gaming between the three algorithms and with other high-level AI

This part of experiment is quite essential and that is the only one done by myself only.
First and foremost, let these three algorithms play with each other. I did this by control program written by myself. The results are:

- Alpha Beta vs Randomly MCTS: Alpha Beta wins all 100 games and Randomly MCTS loses all 100 games.
- AlphaZero vs Randomly MCTS: AlphaZero wins 87 games and Randomly MCTS wins 13 games.
- AlphaZero vs Alpha Beta: AlphaZero wins 34 games and Alpha Beta wins 66 games.

The first result is very normal and the randomly MCTS is just used to test.
We need to give some explanations to the last two results:

- In theory, AlphaZero algorithm can behave much better than Alpha Beta algorithm. This algorithm is advanced and once beat human go player.
- However, there are two main problems in my implementation of this algorithm.
- Firstly, lack of powerful hardwares especially TPU/GPU. It is reported that AlphaZero Go uses 5000 TPUs to train the neural network.
- Secondly, lack of enough time to train. When I finished implementing the algorithm, the remaining time is only 3 days.
- Thirdly, the neural network is a little bit simple. The number of levels and complexity of neural network play important roles in a successful training. Since the structure of neural network in this project cannot be large due to limitation of the size of uploaded file, we can only use simple neural networks.

After testing these three algorithms by playing with each other, I finally decided to use Alpha Beta in points race and round robin.
Also, I made the Alpha Beta algorithm play with the game on 4399.com. The result is 100% winning.
On the platform botzone.org.cn, my Alpha Beta algorithm behaves well and can beat all AIs after rank 60.

3.5 Conclusion

- Advantages of algorithms: Alpha Beta algorithm has the best stability of performance. Randomly MCTS can only used as testing program. AlphaZero algorithm is the strongest in theory if the hardwares are good enough.
- Disadvantages of algorithms: Alpha Beta algorithm relies on the quality of evaluation function too much and cannot search deeper. Randomly MCTS can only perform better than purely random algorithm. AlphaZero algorithm relies on hardwares, structure of neural networks and enough time too much.
- Experience: In this project, I get familiar with many new algorithms such as MCTS and Deep Learning. And the most important experience I get is the entire understanding of AlphaZero algorithm.
- Deficiencies: Lack of hardwares and knowledge of more scientific neural networks is the main deficiency of mine.
- Possible improvement directions in the future: To learn more about convolutional neural networks and learn more about how to build more powerful artificial intelligence.

4. References

[1] Silver D, Hubert T, Schrittwieser J, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play[J]. Science, 2018, 362(6419): 1140-1144.