The Chess AI

Jiacheng Bao, Yicheng Fan, Yunfei Wang, Shunkang Jia, Yunxiang He
Shanghaitech University
Shanghai, China
{baojch, fanych1, wangyf9, jiashk, heyx1}@shanghaitech.edu.cn

***Abstract*—Chess is one of the most classic problems in artificial intelligence. The overall strategy of chess can be summarized as: to gain material advantage and to occupy a favorable position by eating pieces, with a pattern of thinking that consists of constantly sorting and discussing, calculating possible positions and judging the advantages and disadvantages of the later ones. In this project, we implemented several AI agents to play Chess using Minimax method, Expectimax method, Neural Network method and Monte Carlo tree search method. Based on other open-source chess applications, we borrowed and adapted them to develop a chess application of our own, combined with with random and mouse/keyboard agents, we compare the performance of these four AI agents, including their superiority over random agents and their relative performance.**

***Index Terms*—Minimax, Expectimax, Neural Network, Monte Carlo tree search, Chess, AI**

I. Introduction

A. Statespace

Chess is a typical Combinatorial Game, which possesses properties of zero-sum, perfect information, deterministic, discrete and sequential. The property of perfect information means all possible moves are known to both players. This feature greatly reduces the size of state space compared to other games that do not have perfect information, such as mahjong. In addition, due to the simplicity and limitation of the rules of chess, its state space is even further reduced. For example, the size of the chess board is 8x8, so there are 8x8 ways to pick a piece in the first move, and the second move is more complicated. Consider the queen in chess, which can move in 8 directions, up to 7 squares, so that's $7 * 8 = 56$ scenarios. Similarly, there are 8 ways for a knight to move. Add to this the 9 scenarios where pawns are promoted to knights, bishops, and castles (other promotions are subsumed in the queen's move), for a total of $8*8*(56+8+9) = 4672$ scenarios. Therefore, compared to GO, the state space of chess is quite small.

B. Motivation

As described in the abstract, we used several different AI agents to play Chess. All of them are widely used in the field of artificial intelligence. Minimax and Expectimax are classical algorithms for solving zero-sum games. Monte-Carlo Tree Search is a heuristic search algorithm that combines Monte-Carlo simulation and tree search. Neural Network is also a way of machine learning to train a model as the agent. For example, AlphaZero is a very famous AI agent trained by neural networks. We would like to compare the performance of these algorithms in chess as well as incorporate some new improvements and optimizations, and try to explore a more efficient approach to state evaluation.

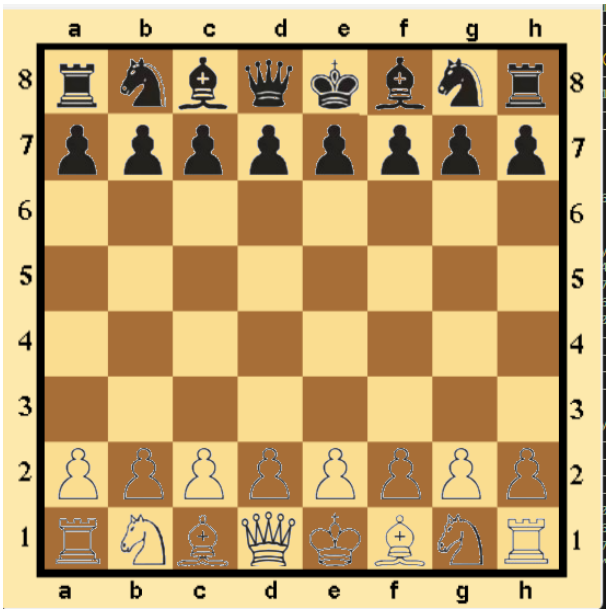

Fig. 1. 2D Board graph

II. Agents

In the section below, we will introduce four different agents selected to serve as AI: the Minimax Search agent, the Expectimax Search agent, Monte-Carlo Tree Search agent, and a neural network agent. To assess the feasibility of these agents, we pit them against our random agent in gameplay, considering the winner as deemed feasible.

A. Baseline–Random agent

Our random agent strategy consists of using "random.choice" to randomly select an action from all possible actions in all our pieces. This makes the agent's behavior a bit silly because when generalized, the agent does not move the king or move other surrounding pieces to protect the king.To make it less naive, when generalized, we instruct the agent to move the king's position or to achieve the effect that it can either move the king or protect the king by moving one of the pieces around the king. This ensures that the agent is less likely to be captured in subsequent playtests.

B. Minimax agent

Minimax search is a decision-making algorithm commonly used in two-player games, it also works well in chess, where the outcome is determined by the interaction of both players.The goal of the algorithm is to minimize the possible loss for a worst-case scenario while maximizing the potential gain.

In our implementation, there are three major functions:

- **Evaluation Function**: We assign a numerical value to a chess state based on several factors. The value is then used to compare and select the best move among all the potential actions. Here gives our factors:
  1) The values of pieces: Considering the various strength of the pieces, we assign various values to each piece. The more important role it plays in the game, the higher value it gets. Generally, King $>$ Queen $>$ Rook $>$ Bishop $>$ Knight $>$ Pawn. Because king's survival determines the game, his value is set breakneck high.
  2) The positions of pieces: For each chess piece, its value is influenced by its specific position on the board; higher values are associated with more favorable positions. Referring to data from the existing chess AI, Sunfish, the value matrix exhibits slight asymmetry, mainly attributed to the asymmetrical positions of the King and Queen on the board.
  3) The threats by opponent's pieces: It is crucial for each piece to survive after a move, so we contemplate deducting a fraction of the score to account for the potential risk to the player's safety, discouraging actions that may jeopardize the well-being of the piece.
  4) The capture of opponent's pieces: Each chess piece has the ability to capture the opponent's pieces, and for every captured piece, the capturing player earns the corresponding points as the values assigned to each piece. In conjunction with the threats posed by the opponent's pieces, a chess piece may opt for self-sacrifice, exchanging itself—having a lower value—with an opponent's piece of higher value, aiming to gain a strategic advantage and maximize overall value.
  5) The protection from friendly chess: This implies that if there are friendly pieces capable of capturing the next position based on the chosen move, the score should be augmented.
- **MinValue**: In this part, we consider the opponent's turn. For each potential response by the opponent, our algorithm considers the player's countermove. We judge whether the input state leads to the finish of the match. If so, it evaluates the resulting positions and selects the move that leads to the lowest score for the opponent. If not, it continues evaluating by calling function **MaxValue** for each potential action.
- **MaxValue**: In this part, we consider the player's turn. Similar to **MinValue**, it considers all possible moves that the player can make, stimulates the opponent's response and evaluates the resulting positions, resulting in the highest score the player can make.

The depth of the minimax search is initialized to 2, it means that the algorithm considers only two levels of moves – the current player's move and the opponent's response. In other words, it looks at one player's move, then the opponent's countermove, and evaluates the resulting positions. Besides that, every time the total number of pieces decreases by eight, we increment the exploration depth by 1, allowing us to obtain more profound strategic insights. The advantage of this approach is that our agent will make better choices when we consider a deeper depth. However, the drawback of this approach is that with the increase in depth, the search space undergoes exponential growth, resulting in longer computation times.

Additionally, we have added alpha-beta pruning based on the minimax agent we have implemented. This helps to eliminate branches of the game tree that are known to be suboptimal and significantly reduces the number of nodes explored during the search, making the search more efficient, leading to a shallower effective search depth.

C. Expectimax agent

Minimax and alpha-beta are great, but they both assume that your opponent will make optimal decisions. Anyone who has won a game of tic-tac-toe will tell you that this is not always the case. The Expectimax algorithm is an algorithm for dealing with random events in a game tree. It is mainly used in games or problems that contain probabilistic decisions, where a player's opponent may make random decisions with uncertainty. Expectimax's ability to effectively deal with uncertainty in a game or problem makes it excel in situations involving randomness, whereas in real-world applications, many decision-making problems involve uncertainty and randomness, so Expectimax is more compatible with real-world decision-making situations

After implementing the **Minimax** agent, we enhance the **MinValue** function by incorporating the mean score of each potential action. Given the inherent uncertainty in chess games, **Expectimax** proves to be a fitting approach as well.

D. Monte-Carlo Tree agent

The Monte-Carlo Tree Search (MCTS) agent is a decision-making algorithm commonly used in games and simulations. It combines elements of random sampling and tree exploration to efficiently navigate large decision spaces. So that, we figure that MCTS can be effective when applied to chess, as it is a game with a high branching factor and complex decision trees.

Our implementation is classical Upper Confidence bounds for Trees(UCT) algorithm, which includes four steps, namely Selection, Expansion, Simulation and Backpropagation.

- **Selection**: To find an optimal node worth exploring in the tree, the strategy involves first selecting an unexplored child node. If all child nodes have been explored, the

Algorithm 1 The UCT algorithm

function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TreePolicy(v_0)
    $\Delta \leftarrow$ DefaultPolicy($s(v_l)$)
    BackUp(v_l, Δ)
  end while
  return a(BestChild($v_0, 0$))
end function
function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return Expand(v)
    else
      $v \leftarrow$ BestChild(v, Cp)
    end if
  end while
  return v
end function
function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
  where $s(v') = f(s(v), a)$ and $a(v') = a$
  return v'
end function
function BESTCHILD(v,c)
  return $\underset{v' \in \text{children of } v}{\operatorname{argmax}} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$.
end function
function DEFAULTPOLICY(s)
  while s is nonterminal **do**
    $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  end while
  return reward for state s
end function
function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v
  end while
end function

choice is then based on the child node with the highest UCB value.

- **Expansion**: Creating a new child node by taking one step in the previously selected child node. The strategy is that randomly selecting an action that is not a repetition of the action taken in the preceding child node.
- **Simulation**: Starting from the newly expanded node, simulate the game until reaching a terminal state. This allows us to determine the score obtained for this expanded node.
- **Backpropagation**: Feedback the score obtained from the previously expanded node to all its parent nodes, updating their quality values and visit counts. This facilitates the calculation of the UCB values for subsequent steps.

Here gives the pseudocode of our algorithm 1: In the function **BestChild**, we use the Upper Confidence Bound(UCB) algorithm, Where v' represents the current tree node, v is the parent node, Q denotes the cumulative quality value for the tree node, N is the number of visits to the tree node, and C is a constant parameter equaling $\frac{1}{\sqrt{2}}$ during the exploration.

E. Neural Network agent

The neural network we implemented takes the game state as input and outputs the probability of choosing each class of pieces. However, to train neural network agents, a large amount of training data is required to generate effective models.

To address this problem, we implemented a self-trainer that generates training data by pitting random agents against Monte Carlo Tree Search (MCTS) agents. The generated samples include the state of the game board at each step, the current player, and the corresponding score. We preprocess this data to create input samples, which consist of 16 dimensions of features such as the state of each type of piece on the board and information about the starting player. All input samples are represented as 16x64 tensors. Subsequently, we train our neural network moedel using five fully connected layers to enhance its performance. Finally, we will create a training dataset to train the model and use this to implement our neural network agent. See the code for specific hyperparameter settings.

But when we use a large amount training data to train the model, it will lead to a really long training time. We only use less training data and the simplest fully connected layer neural network for inference because of equipment resources limitation. Therefore, We believe that when it is difficult to obtain enough high-quality data, this type of data-driven approach is likely to perform much less well than traditional policy-based approaches, and even with a considerable amount of data, how to address the generalizability of the model is a very worthwhile question. Therefore, this method is not a focus in our experiments and appears only as a control group.

III. PERFORMANCE RESULTS

A. Random Agent v.s. Our Agents

Since several of the AI agents we have implemented have better performance than the random agent, we let these agents fight against the random agent on the white side and the black side respectively 50 times to count the winning rate and the required steps to win.

B. Other Agents v.s. MCTS

Because the test time is really long, so we just test 20 times in this section.

Play against the Random agent, almost 100% winning rate. However, the neural network works not well due to the inadequate training data and the limitation of the equipment.

White	Black	Winning rate	Steps
random	Minimax(Naive)	100%(Black)	32.8
random	Minimax(alpha-Beta pruning)	100%(Black)	30.6
random	Expectimax	100%(Black)	35.2
random	Neural Network	60%(Black)	195.2
random	MCTS	100% (Black)	36.3

TABLE I
PERFORMANCE WHEN V.S. RANDOM AGENT

White	Black	White Winning rate	Steps
Minimax(Naive)	MCTS	80%	52.4
Minimax(alpha-Beta pruning)	MCTS	75%	54.1
Expectimax	MCTS	75%	57.6
Neural network	MCTS	0%	40.4

TABLE II
PERFORMANCE WHEN V.S. MCTS

Play against the MCTS agent, Minimax and Expectimax play better than MCTS with about 80% wining rate. But neural network plays worse than MCTS with about 0% wining rate.

For Minimax agent, one possible improvement is to find a better evaluation function, thus to find a better acton policy. Also, alpha-beta pruning and setting the appropriate depth of search are also helpful.

For MCTS agent, maybe early determination can make sense.

For Neural network agent, optimization involves identifying more significant input features, configuring appropriate hyper-parameters, generating additional input samples, and refining the output structure to facilitate better and more intelligent learning by the model.

Therefore from the results and analysis above, we can know some basis knowledge about chess AI and furthermore, there are also many places we can improve to get a better performance and result.

IV. EXTERNAL LIBRARIES

- tkinter, which is a Python library, also helping us to generate the Python project GUI.
- torch, which is a deep learning framework developed in the Python language. It is simple to use and has features characteristics such as dynamic computation graphs and so on.
- numpy, which is an extension library for the Python programming language, supporting extensive operations on multidimensional arrays and matrices, and provides a rich mathematical function library specifically designed for array operations.
- random, which is a module used to generate random numbers.

V. REFERENCES

- Sunfish Chess AI, https://github.com/thomasahle/sunfish