MSDQNBot: Multi-Stage Learning Gomoku AI Agent Based on DDQN

Group12 - Jiaxin Li(521030910096), Junzhe Shen(521030910108), Benhao Huang(521030910073)†

***Abstract*—The game of Go is the longest-studied domain in the history of artificial intelligence. A number of AI agent based on a combination of sophisticated search techniques, domain-specific adaptations, and handcrafted evaluation functions or the Reinforcement Learning are emerging over decades. One of them , AlphaZero , has achieved superhuman performance in many challenging games. However, we made a series of experiments on AlphaZero in Gomoku and found that it encounters certain issues including *Too Long training time*, *Edge Blindness*,*Merciful Strategy* in our experiment setting within the constraints of limited training time and computational resources.**

To address the above issues, we conducted a series of meaningful exploration, such as *parallel acceleration* which implement the acceleration of training speed on Windows PCs, *Gumnel Muzero* which aim to implement the policy improment using fewer simulation, *Posiitional Bias Design* which aim to guided our model to pay more attention to the edge and so on. Altough most of them are not able to attained the performance levels we anticipated, this experiment trials all guided us in designing a ultimately superior model: MSDQNBot .

The training pipeline of MSDQNBot,multi-stage Learning Gomoku Ai Agent Based on DDQN, is consist of 3 stage, *Imitation Learning* , *Self-Playing* and *Competing with Master*. In the whole process, we also established a dataset of one hundred thousand game boards and implement our *Rule-Based GomokuBot* , which has excellent performance in our experiment to train our MSDQNBot in this pipeline.

Finally, after 3-stage training pipeline, Our MSDQNBot achieves a success rate of 95% in games against Pure Monte Carlo Tree Search (MCST) with a simulation count of 2000. By the way , our Rule-Based GomokuBot even achieves a success rate of 97% and a tie rate of 2.33% . Therefore, we think our MSDQNBot still has significant potential, and there is room for further fine-tuning in training. However, due to computational constraints, we do not allow our model to reach its optimum within the limited time. Exploring ways to further enhance the performance of our model is a direction worthy of further investigation in the future.

***Index Terms*—Gomoku , DDQN , Multi-Stage Learning**

I. Introduction

Gomoku, a two-player game on a 15×15 grid, involves placing black or white stones to form a row of five horizontally, vertically, or diagonally. Success demands a strategic balance between offense and defense. High-level play includes intricate board analysis, position evaluation, and anticipating the opponent's moves, creating a vast decision space. Developing a formidable Gomoku Agent to defeat humans is challenging. In our project, we integrated DDQN into the AlphaZero framework, employing Multi-Stage Learning for effective training. We also explored parallel acceleration, experimented with GumbelMuZero, and delved into other intriguing aspects.

Our project code is avaliable on Github[1], and a live demo is avaliable on HuggingFace[2].

II. Related Works

A. Searching Algorithms for Zero-Sum games

Gomoku is considered a zero-sum game. In zero-sum games, the total gain of all players in the game adds up to zero. This means that any advantage gained by one player is exactly balanced by the losses of the other player(s).

The Minmax algorithm [1] operates on the principle that players alternate turns, trying to maximize their own benefit while minimizing that of their opponent. It explores all possible moves in the game tree up to a certain depth. The major limitation of Minmax is its computational intensity, as it requires exploring a vast number of game states, especially in games with high branching factors. Alpha-Beta Pruning [2] is an enhancement of the Minmax algorithm. It significantly reduces the number of nodes evaluated in the search tree, thereby increasing efficiency.

Monte Carlo Tree Search (MCTS) has been originally proposed in [3] and [4], as an algorithm for making computer players in Go. In particular, pure Monte Carlo tree search does not need an explicit evaluation function. Simply implementing the game's mechanics is sufficient to explore the search space. However, a disadvantage of MCTS is that it might overlook subtle losing lines in certain positions, where moves appear strong but lead to defeat, especially against expert players, due to its selective node expansion policy.

B. Advanced Self-Learning AI Algorithms

The development of Go AI agents has been a remarkable journey in artificial intelligence, showcasing significant advancements over the years. Developed by DeepMind, AlphaGo [5] was the first AI to defeat a professional human Go player. It combined advanced machine learning techniques, including deep neural networks and Monte Carlo Tree Search (MCTS), to evaluate and select moves. However, its heavy reliance on large, often inaccessible datasets of human-played games could significantly constrain its strategic variety. An advancement over AlphaGo, AlphaGo Zero [6] learned to play Go entirely through self-play, without using any human game data. AlphaZero [7] generalized the approach of AlphaGo Zero

† All three authors are senior students from Shanghai Jiao Tong University and are of equal contribution.

[1]Github: https://github.com/Lijiaxin0111/AI_3603_BIGHOME

[2]HuggingFace: https://huggingface.co/spaces/Gomoku-Zero/Demo

beyond Go to other games like Chess and Shogi. MuZero [8] could handle environments where the rules were not explicitly known, learning both the game dynamics and strategy through self-play. This made MuZero more applicable to real-world problems where rules and dynamics are not always clearly defined.

III. Initial Explorations

Before officially figuring out effective methods, we first explore some possibilities to locate our research directions. We record these meaningful explorations here to demonstrate our thoughts and ideas.

A. Observations

In this section, we'll highlight some intriguing findings from our preliminary investigations. Recognizing that a pure Monte Carlo Tree Search (MCTS) approach may not fully uncover all the potential shortcomings of the model, our observations are drawn from scenarios where a human player competes against the AI. This setup provided a more comprehensive perspective on the AI's performance and limitations.

1) Long Training Time: A complete training procedure can take more than a day. During training, the primary performance bottlenecks are the collection of self-play data and the validation process against the pure MCTS (Monte Carlo Tree Search) method. Although decreasing the frequency of validation can effectively shorten the training duration, the collection of self-play data continues to be a significant bottleneck.

2) Edge Blindness: During our testing of AlphaZero's performance in the game of Gomoku, we observed that while the model is quite competent in situations where Go st are concentrated in the central area of the board, it tends to overlook moves made by its opponent at the board's periphery. As displayed in Fig. 2, this oversight often resulted in AlphaZero's defeat when facing real human players, indicating a significant area for improvement in its strategic approach.

3) Merciful Strategy: Beyond Edge Blindness, the model exhibits another notable behavior we've termed 'Merciful Strategy'. Even when presented with a decisive, game-winning move, the model often opts to prolong the game instead of executing the winning play immediately. This tendency not only extends the game unnecessarily but also inadvertently increases the chances for its opponent to recover and possibly win. Such an approach is contrary to our desired outcome of a strategically efficient and ruthlessly effective AI player. A gif display for this case could be found here.

B. Meaningful Trials

To address the issues identified in the observations section, we conducted several experimental trials. Although these trials did not yield concrete results, we consider them as significant steps that contribute to demonstrating our efforts and workload. These explorations not only helped in refining our understanding of the challenges but also guided future directions for more effective solutions.

1) Parallel Acceleration: In order to accelerate the process of collecting self play data, we implemented parallelization of self play data collection through multi-processing. This involved running each game in a subprocess and then aggregating the game data in the main process. However, this approach led to a noticeable drop in performance on Ubuntu servers, in contrast to the performance enhancement observed on Windows PCs.

We profiled the program to see what was going on. The profiling results on Ubuntu server and Windows PC are shown in Fig. 3 and Fig. 4 respectively. From left to right represents time, with longer bars indicating longer durations for function calls, from top to bottom represents the hierarchy of function calls, where functions at the top call those below them. The profiling tool is provided by [9].

As illustrated in Fig. 3, there is an occasional, significant increase in the time required for a playout on Ubuntu, whereas it remains relatively constant on Windows PCs. This sudden increase in time consumption on Ubuntu is caused by a slowdown in torch operations. The underlying cause, however, remains unclear. In fact, even when conducting self play data collection in a single-process manner, there is still a slowdown in torch operations from time to time, except that it's not as significant as in multi-processsing.

2) Gumbel MuZero: To accelerate training time by achieving policy improvement with fewer simulation iterations, we aim to implement the Gumbel Muzero described in [10] on Gomoku.

Gumbel Muzero utilizes the Gumbel Top-k trick to sample actions without repetition, enabling policy improvement. This approach differs from the heuristic action selection method used in AlphaZero.In this experiments, we utilizes the Gumbel Top-k trick using sequential halving algorithm, as pseudocode 1.

Algorithm 1 sequential halving algorithm

Require: k: number of actions
Require: m $\leq$ k : number of actions sampled without replacement.
Require: n: number of simulations
Require: logits $\in R^k$: predictor logits from a policy network π

1: Sample k Gumnel variables:

$$(g \in R^k)\ Gumbel(0)$$

2: Find m actions with the highest $g(a) + logtis(a)$:

$$A_{topm} = argtop(g + logtis, m)$$

Use Sequential Halving with n simulations to identify the best action from the A_{topm} actions, bt comparing $g(a) + logits(a) + \sigma(\hat{q}(a))$

In order to make full use of the information from searching, Gumbel Muzero Construct the completed Q-values and the improved policy for the root node as follows:

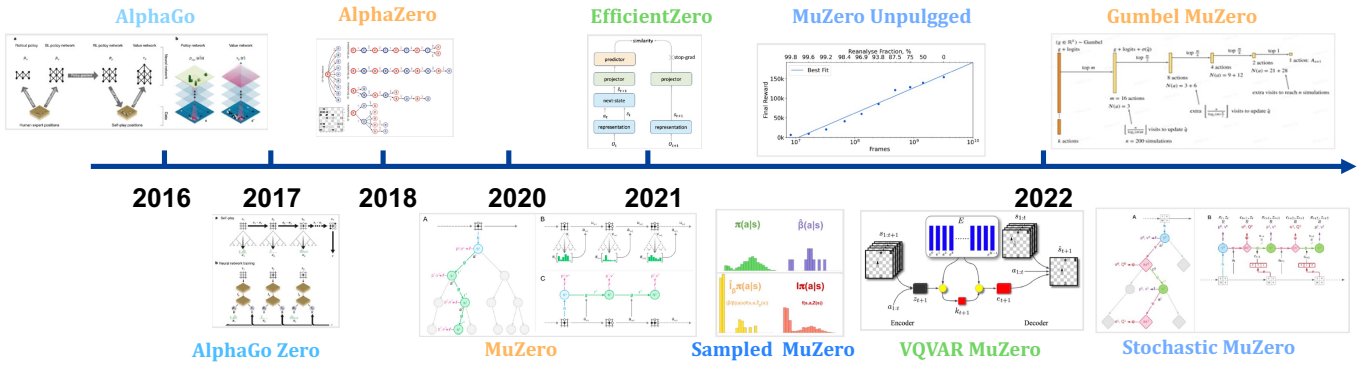

Fig. 1. A Timeline of AlphaGo and AlphaZero's Evolution: From the initial breakthrough of AlphaGo in 2016, which first defeated a human Go professional, to the 2017 advancements of AlphaGo Zero and 2018's AlphaZero, expanding prowess beyond Go to Chess and Shogi. The journey continued with MuZero in 2020, advancing AI's understanding of games without pre-set rules. Gumbel MuZero represents a recent and development in the field, capturing our interest with its innovative approach.

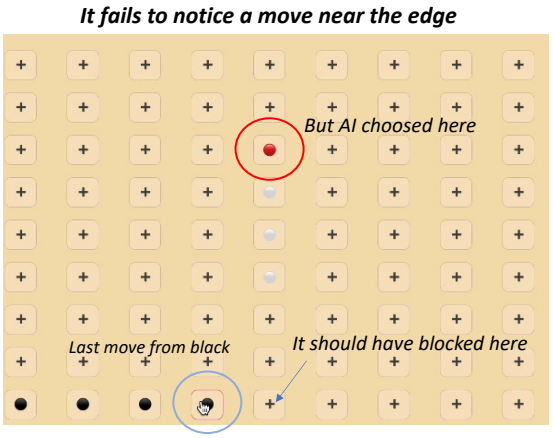

Fig. 2. Edge Blindness: In the illustrated scenario, the human player controls the black pieces, whereas AlphaZero controls the white. This example highlights a crucial error by AlphaZero, where it fatally overlooks an opponent's move on the board's edge, leading to a significant strategic disadvantage.

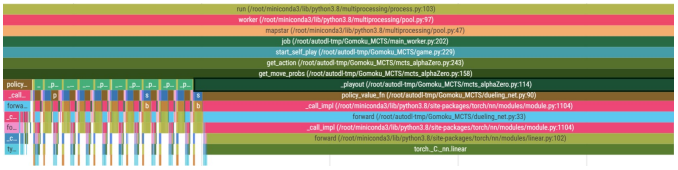

Fig. 3. Profiling result of a subprocess on Ubuntu

Fig. 4. Profiling result of a subprocess on Windows PC

$$\text{completedQ}(a) = \begin{cases} q(a) & \text{if } N(a) > 0 \\ v_\pi & \text{otherwise} \end{cases}$$

$$\pi' = softmax(logits + \sigma(completedQ))$$

where, $\sigma(\hat{q}(a)) = (c_{visit} + max_b N(b))c_{scale}\hat{q}(a)$

For the non-root node , Gumbel Muzero proposes a deterministic method for action selection, to minimize the distance between the improved policy π' and the normalized visit counts after selecting a:

$$argmax_a(\pi'(a) - \frac{N(a)}{1 + \sum_b N(b)})$$

In order to extracted the improved policy into the policy network, the loss function to update the policy network as follow:

$$L_{completed(\pi)=KL(\pi',\pi)}$$

Although we made every effort to faithfully reproduce the methods and formulas mentioned in the paper as above, we were unable to achieve the expected results in the final task. Even in our experiment sets, Despite training for multiple epochs and trying various parameter settings, we were unable to achieve any significant performance improvement.

Finally, we analyze and discuss with TA about the possible reason :

- The Sequential Halving algorithm, to some extent, distributes the search depth evenly to a series of possible action compared to the progressive widening search, which hinders effective deep search.
- Model-based policy optimization Muzero is less accurate than AlphaZero in most cases. One of possible reasion is that as the number of inference steps in the environment model increases, the accumulated compound error in the model quickly grows, resulting in unreliable outcomes from the model.

Based on the above reason which make the model sensitive to the experiment setting, it is reasonable that reproduced results did not meet the initial expectations satisfactorily. It is one of important reason for us to shift our research towards developing a more deterministic Gomoku agent, aiming to achieve better performance more easily.

3) Positional Bias Design: To address the issue of edge blindness, we considered an alternative approach to the one-hot matrix input for the Policy Network. Our strategy involved making the model more aware of moves at the board's edges. To achieve this, we experimented with adding positional bias to the input, aiming to enhance the model's sensitivity to edge positions.

The original usage of the Policy Network is described as follows:

$$P_i, value_i = \mathcal{V}_{policy}(BS_i)$$

Here $\mathcal{V}_{policy}$ denotes policy network, P_i represents the probabilities of executing specific actions on the board at that state, and $value_i$ signifies the evaluated value for the Board State(BS_i). The subscript i indicates that the current state corresponds to the i^{th} move.

We added a positional bias to the input BS_i. This bias is depicted as shown in the figure. To increase attention towards the board's edges, we set the center of the board as $(0, 0)$. In this context, a 2D-Gaussian distribution becomes an ideal choice for generating such a bias. The positional bias is mathematically expressed as:

$$bias = \mathcal{G}(x, y)$$

$$\mathcal{G}(x, y) = \frac{1}{2\pi\sigma_x\sigma_y} \exp\left(-\frac{1}{2}\left[\frac{(x-\mu_x)^2}{\sigma_x^2} + \frac{(y-\mu_y)^2}{\sigma_y^2}\right]\right)$$

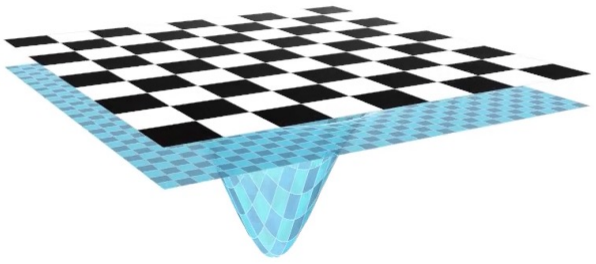

Fig. 5. Positional Bias with Gaussian Distribution.

where $\mathcal{G}(x, y)$ represents the Gaussian function at coordinates (x, y). Here, μ_x and μ_y are the means, and σ_x and σ_y are the standard deviations of the distribution along the x and y axes, respectively. In our trials, we set $\sigma_x = \sigma_y = 1, \mu_x = \mu_y = 0$. Therefore, the input become:

$$P_i, value_i = \mathcal{V}_{policy}(BS_i - bias)$$

Despite our efforts, the experiments indicated that this strategy was not effective in resolving the problem. While intuitively appealing, it appears that a more refined and complex approach is necessary for further exploration.

IV. Methods

In this section, we introduced methods used in achieving our final results.

A. Training Pipeline

Although AlphaZero's ability to master complex rules through self-play is impressive, we have observed that this method is not particularly efficient. Firstly, it requires a substantial amount of training time. Secondly, it might take even longer to overcome two significant drawbacks: edge blindness and a merciful strategy. Given our limited time and computing resources, we are inclined to pursue an alternative approach, which we refer to as "multi-stage learning." This strategy aims to achieve more efficient learning outcomes by leveraging diverse learning stages and methods.

1) Stage1 - Imitation Learning: Consider the analogy of a child learning to walk. Typically, a child cannot walk independently without guidance and assistance from adults. In this scenario, the child represents our model, and the adults are akin to experts. To facilitate rapid and efficient learning in our model, a straightforward strategy is to have the model emulate how these experts make decisions and take actions. This process is referred to as Imitation Learning, where the model observes and replicates the actions of experts to acquire new skills or behaviors more effectively. This approach is particularly useful in machine learning contexts where direct learning from scratch is either impractical or too time-consuming.

Imitation learning [11] is a framework in reinforcement learning where the model learns to imitate a certain behavior based on demonstrations. Typically, these demonstrations are presented in the form of state-action trajectories, which guide the learning process. To achieve this, we need to prepare lots of high quality data. The process of data preparation will be introduced detaily in IV-D.

In this stage, we train our model using the Gomoku chess manuals gotten in IV-D to implement policy improvement for Strategy Networks and enhancing predictive performance for Value Networks as follow Loss function:

$$Loss_v = MSE(v, v_{expert})$$

$$Loss_\pi = -\sum_{i=1}^{n} \pi(x_i) log(\pi_{expert})$$

2) Stage2 - Self-Playing: After completing the initial phase of training through imitation learning, we transition to a strategy similar to that employed by AlphaZero [7], where the model engages in self-play to further its learning. Obviously, this is more effective than starting with self-play from scratch, since after the first stage of learning from experts, the model

has already acquired substantial knowledge. When the model competes against itself, it's essentially challenging an opponent with expert-level skills. Conversely, if the model were to begin with self-play from scratch, it will probably result in suboptimal strategies. This can lead to tricky issues such as edge blindness and a merciful strategy, which will be reflected in its relatively low winning percentage in our experiments.

3) Stage3 - Competing with Masters: After completing the initial two stages of training, we then move on to a third stage designed to further refine and test the model. In this stage, we introduce a challenging scenario by having our model compete with a more advanced and capable Gomoku Agent, which we refer to as the 'master model'. This setup is meant to assess how well the model has learned from the earlier stages and to push its capabilities even further. By facing off against this master model, our AI is exposed to higher levels of strategy and skill, encouraging it to improve and adapt to more complex challenges. This approach not only tests the model's current abilities but also promotes continuous learning and development. However, we should design and implement a powerful Gomoku Agent first.

B. Training Algorithm

We adopt the training procedure of AlphaZero under minor changes, see Algorithm 2 and Algorithm 3.

AlphaZero adopts MCTS algorithm as its backbone. Each round of MCTS AlphaZero consists of four steps:

- *Selection*: Start from root R and select successive child nodes until a leaf node L is reached. The root is the current game state and a leaf is any node that has a potential child from which no simulation (playout) has yet been initiated.
- *Expansion*: Unless L ends the game decisively (e.g. win/loss/draw) for either player, create one (or more) child nodes and choose node C from one of them. Child nodes are any valid moves from the game position defined by L. Each child node is assigned with a moving probability p predicted by the policy and value network.
- *Simulation*: Complete a playout from node C. A playout is choosing moves with largest score defined as $Q + c \cdot p \cdot \frac{\sqrt{N}}{n}$ until the game is decided (game is won, lost, or drawn), where Q stands for Q-value, c is a temperature hyper-parameter, N is the visit counts of parent node, n is the visit counts of the current node.
- *Backpropagation*: Use the result of the playout to update information in the nodes on the path from C to R. The update of each node includes increasing the visit counts by 1, and updating $Q = \frac{leaf\_value - Q}{n}$. $leaf\_value$ is set to 1 if the current player wins, -1 if the current player loses, 0 if the game if drawn.

C. Model architecture

AlphaZero utilizes Convolutional Neural Network(CNN) as policy and value network, whose architecture is shown in Fig. 6.

Algorithm 2 Training Algorithm

Require: $game\_batch\_num$: Number of training iterations
Require: $data\_collect$: Method to obtain training data
$data\_buffer \leftarrow$ empty deque: An empty Python collections.deque object to store training data
for $i = 1$ **to** $game\_batch\_num$ **do**
  if $data\_collect =$ SELF_PLAY **then**
    Collect self play data
    extend $data\_buffer$ with augmented self play data
  else if $data\_collect =$ LOAD_DATA **then**
    Load expert data from a sampled batch of generated high quality games
    extend $data\_buffer$ with augmented expert data
  end if
  if $data\_buffer$ has sufficient amount of data **then**
    Policy Update
  end if
end for

Algorithm 3 Policy Update

Require: num_epoch: Number of epochs
Require: $policy\_value\_net$: A policy and value network
Require: $batch\_size$
Require: $data\_buffer$
for $i = 1$ **to** num_epoch **do**
  $batch\_data \leftarrow$ Sample from $data\_buffer$
  $state\_batch \leftarrow$ Extract state from $batch\_data$
  Forward call $policy\_value\_net(state\_batch)$, store the results to $log\_act\_probs, value$
  Calculate $Loss = Loss_v + Loss_\pi$ between $log\_act\_probs, value$ and ground truth in $batch\_data$
end for

In this work, we adopt dueling network architecture proposed in [12](see Fig. 7), a variation of CNNs that improves upon traditional CNNs used in reinforcement learning. The key idea in dueling network architecture is to separate the representation of state value and the advantage of each action. In traditional CNNs used for Q-learning, a single stream of computation outputs Q-values for each action. In contrast, dueling networks have two streams:

- One stream estimates the state value function $V(s)$, which is the value of being in a given state.
- The other stream estimates the advantage function $A(s, a)$, which represents the additional value of taking a particular action compared to others.

After processing through these separate streams, the outputs are combined to estimate the Q-value function: The Q-value for a state-action pair $Q(s, a)$ is computed by combining the value $V(s)$ and the advantage $A(s, a)$ in a way that maintains the original Q-learning objective. Typically, this is done using the formula: $Q(s, a) = V(s) + A(s, a) - \frac{1}{|A|} \sum_{a'} A(s, a')$.

By separating the estimation of state values and action advantages, the network can learn to evaluate the value of

each state without having to learn the effects of each action. This is particularly beneficial in environments where the value of the state does not vary much across actions.

Dueling networks often lead to improved stability in training and better policy evaluation, especially in scenarios where the action choice does not have a significant impact on the outcome.

Furthermore, as the board size increases and deeper networks are required for more accurate predictions of policy and value, the convolutional layers can be extended into Residual blocks (see Fig. 8).

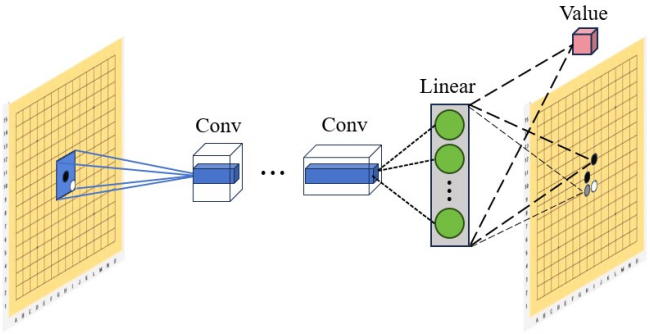

Fig. 6. AlphaZero's Policy and Value Network

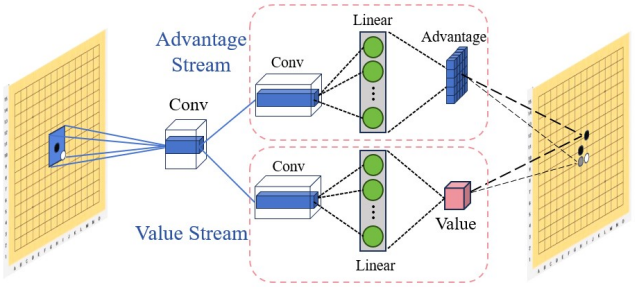

Fig. 7. Dueling Network

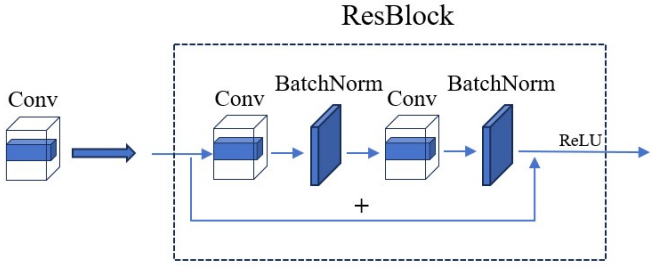

Fig. 8. Residual Block

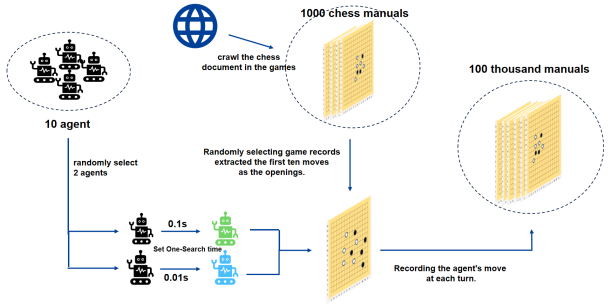

Fig. 9. The Pipeline of Data Collecting

D. Data Collecting procedures

Considering the numerous classic opening strategies in the game of Gomoku, such as the "Cold Star Opening" and the "Stream Moon Opening," it is meaningful for our Gomoku Agent to learn these openings as they offer a certain level of determinism, which can greatly enhance its performance.

However, in actual Gomoku gameplay, the board state can change rapidly, and the deterministic nature of opening strategies often cannot fully achieve the intended effect. Therefore, human players tend to adjust these openings based on the actual circumstances they encounter during the game.

Based on the aforementioned observation, we aim for our Gomoku Agent to learn classic opening moves from high-level human players' actual gameplay in tournaments, including learning the initial moves of these opening strategies as well as the adjustments made by human players based on the game's specific board state.

During our process of method survey, we discovered that Gomoku agents have a long history of development. Whether based on deterministic algorithms or neural network-based approaches, these agents have demonstrated performance in real-world matches that surpasses that of average players and even rivals that of professional players.

Moreover, Gomoku agents have the capability to engage in continuously playing game, make it possible for us to gather a substantial amount of data in a relatively short timeframe.

Therefore, we design the two-edge data collecting pipeline as Fig. 9.

1) Collecting Gomoku chess manuals from the Human Game: Our opening is from the Website . There are many Gomoku match in this website, including the 5th World Team Gomoku Championship and so on. We used python library `selenium` to request the JavaSript webpage , which includes the Gomoku chess manuals .[3] Then, we use `BeautifulSoup` to parser the webage and get the first ten moves. Considering the fact that the size of board is 15 * 15, we discard the opening whose the first ten moves is out of boundary 9*9. In this process, we crawled one thousand openings in totally.

2) Generating More Gomoku chess manuals by robots' gameplaying: We download ten excellent Gomocup Bots

[3]The source of our data is shown in the appendix A.

on the leaderboard of Gomocup [4], an annual international Gomoku AI programming competition,which erves as a platform for researchers and developers to showcase their Gomoku-playing agents and compete against each other.

Base on the Gomocup Manager `Piskvos`, we build our one-hundred thousand Gomoku dataset as process 4

Algorithm 4 The process of generating data

Require: B: the bots pool downloaded from the Gomocup
Require: O: the opening pool crawled from the web
1: Randomly choose two bots from B : $Player_1 \leftarrow b_i$, $Player_2 \leftarrow b_j$
2: Set the time limitation for one search : $t_{Player_1} \leftarrow 0.1$; $t_{Player_2} \leftarrow 0.01$
3: Randomly choose one of opening o_i from O
4: Let the Bots $Player_1$, $Player_2$ make the moves according to the opening o_i
5: Let the Bots $Player_1$, $Player_2$ make the moves according their algorithm and record every their moves

3) Data Processing and Parsering : In the IV-D2, we only get the moves of two bots in every iteration. In order to make the training more effective in our Gomoku agent, we split the dataset into 3 parts according to the winner in every games. In our following training using this datasets, we randomly select an equal number of Gomoku chess manuals from these splits every time.

Based on the bots we have selected all exhibit excellent performance in the Gomocup. Therefore, we believe that regardless of whether they are the winning or losing side, their moves are of high quality. To train the policy network, we represent the policy function need to be learned is a one-hot vactor corresponding to the position of the move.

For evaluating the value of the game board, we assign a value based on the final outcome. If the winning side is evaluating the board, it is assigned a value of 1. If the losing side is evaluating the board, it is assigned a value of -1. In the case of a draw, the value assigned to the board is 0 for both sides.

E. Implementations of Rule-Based-GomokuBot

For the third stage of training, we developed a master Gomoku agent, which we call the "Rule-Based-GomokuBot". This agent serves as a tutor and is designed using traditional game strategies, instead of complex neural networks like AlphaZero.

The Rule-Based-GomokuBot uses established Gomoku rules to analyze and score different board positions. It then applies the Alpha-Beta Pruning algorithm to efficiently determine the best moves. This approach is simpler yet effective, demonstrating that advanced AI performance can be achieved with rule-based strategies and smart decision-making processes. Fig. 10 demonstrates the frameworks of Rule-Based GomokuBot. Some key technologies used are described in subsections below.

[4]The Bots we used for data generating is shown in the Appendix B.

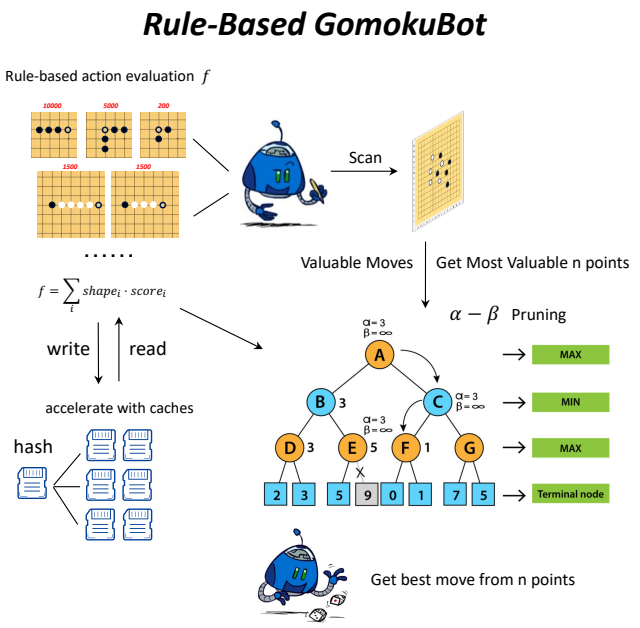

Fig. 10. The Framework of Rule-Based GomokuBot.

1) Rule-Based Evaluation: Unlike learning a neural network to evaluate state's value, we directly evaluate the state value using pattern matching strategies. It involves adding up the values of various line formations on the board. For examples:

- A line of five in a row scores 100,000 points.
- An open four (a line of four pieces with open ends) scores 10,000 points.
- An open three (three in a row with open ends) scores 1,000 points.

According to these pattern rules, we subtract the score of one player from the other, based on their role in the game. This subtraction gives us a final assessment of the board's state, helping to determine the best possible move for the player.

$$f = \sum_i pattern_i \cdot score_i$$

Here $score_i$ is determined by a pattern-score dict $\mathcal{D}$, which takes patterns as keys.

$$Score_{(black)} = f(Black) - f(White)$$

2) Alpha-Beta pruning: Then we choose Alpha-Beta pruning algorithm to decide which move to take based on current board states. As an enhancement of the Minmax algorithm, Alpha-Beta Pruning [2] significantly reduces the number of nodes evaluated in the search tree, thereby increasing efficiency.

To enhance the Alpha-Beta pruning process, we start by scanning the whole board to spot the most strategic positions for making moves. We select a fixed number of these top positions, based on their calculated importance. This number,

n, is adjustable in the code and affects both the winning chances and the time it takes to make a move. Using our scoring rules, we estimate the value of each selected position if it were the next move. This helps us figure out which positions are worth considering. Finally, we use the Alpha-Beta algorithm to pick the best moves from these selected positions. This strategy helps us narrow down our search, making the algorithm faster and more efficient.

3) Z-Cache: As we known, if two situations on the board are identical, they are essentially the same game state, despite the different order of moves to get to it.

When we come across such identical scenarios during our search, we can score one and save this score. If the same scenario comes up again, we don't need to rescore it; we can just use the saved score. This method of reusing scores from previous identical situations makes our process quicker and more efficient. We can also store results from previous searches to use in future heuristic searches, further enhancing the effectiveness of our strategy.

Zobrist hashing [13] is a fast array hash algorithm that suits our needs perfectly. The efficiency of Zobrist hashing is remarkably high. With each move in the game, it only requires a single XOR operation. Compared to the computational effort of scoring each move, the performance cost of this XOR operation is negligible.

4) Calculate to kill: To address the "merciful strategy" issue mentioned in III-A3, we incorporated an additional module called the "killer" module. This module is designed to identify any immediate winning moves by searching for any potential line of five pieces. If such a winning opportunity is detected, the module directs the AI to execute this move immediately. Otherwise, the AI then reverts to using its default strategic algorithms. This enhancement ensures that the AI does not miss clear opportunities to win, thereby solving the problem of being unnecessarily "merciful" in its gameplay.

V. Experiments and Results

In this section, we would display the results of our experiments with our methods.

A. Edge Blindness and Merciful Strategy Problems Solved

After applying the imitation learning strategy to train DuelNet(simulation time = 200) equipped with killer module in IV-E4, we have already solved the Edge Blindness and Merciful Strategy problems. A live demo is available here.

B. First Stage: Imitation Learning

TABLE I
PLAYER VS PUREMCTS (SIMULATION TIME =2000) IN 50 MATCHES.

Agent	Win	Lose	Tie	Average Time (s)
DuelNetPlus	48	1	1	≈ 0.0015/0.3
GomokuBot	50	0	0	≈ – / 0.005
AlphaZero	35	15	0	≈ 0.004/0.8
PureMCTS	31	19	0	≈ 0.0027/5.5

1) Much Better Performance: To prove the effectiveness of our approach, we set up a series of 50 games where the fixed challenger was Pure MCTS, configured to perform 2000 simulations per move. The participants in this benchmark were four players: DuelNetPlus, GomokuBot, AlphaZero, and Pure MCTS itself.

It's important to note that DuelNetPlus and AlphaZero were subjected to identical training conditions, ensuring a fair comparison. This included same learning rates, batch sizes, and the number of training epochs (1000) among other settings. Additionally, their simulation times are also the same as 200.

In Table I, except for GomokuBot, which is not MCTS-Based, thus only has the time taken to make a move, we've also record the time it takes for the other players to run a single simulation. These times are rough estimates — they aren't exact due to our limited verification but give us a sense of each player's time efficiency.

It's obvious from the table that the GomokuBot performs best, with highest wining ratio and fast reaction time. And merely after the first stage training, DuelNet has achieved a much better performance than AlphaZero both in win ratio and reaction time.

2) Higher Time Efficiency: In addition to a better performance, it's also amazing that though with a 0.1 million scale dataset, it still has better time efficiency compared to self-play training strategies. As demonstrated in the upper plot of Fig. 11, where we train Duel Net separately on two strategies, it could be seen from the figure that imitation learning could have model achieve its best performance earlier and keep doing well than the self-play training one. Note that we **increase the simulation time of the opponent Pure MCTS used for testing during training with epochs**, thus it's normal to see a decrease in winning ratio during training process.

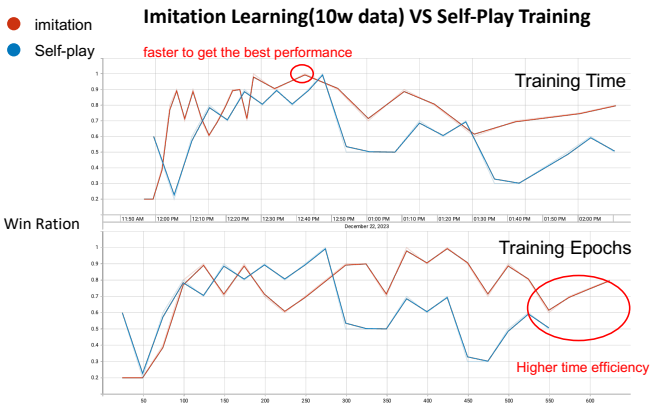

Fig. 11. The Training Procedure Show-casting Time Efficiency.

The data also reveals that imitation learning can complete more epochs than self-training within the same time, highlighting its better time efficiency. This suggests that imitation learning is not only faster but also potentially more effective in

certain scenarios, allowing for more extensive training within the same time frame.

C. Second Stage: Self-Play Training

In this stage, we use the best model derived from imitation learning and let it practice by playing games against itself for another 1000 rounds. To see how much it improved, we set up 200 test games against PureMCTS with 2000 simulation time. The results are shown in the Table II.

TABLE II
GAME OUTCOMES BY STAGE AND STRATEGY

Stage	Use Kill	Win	Lose	Tie
First Stage	No	163	33	4
First Stage	Yes	183	16	1
Second Stage	No	185	15	0
Second Stage	Yes	193	7	0

As we can see, a second stage self-play training keeps improving model's performance a lot. The model from second stage without killer module even outperform the one from first stage with killer module! Besides, these results also show the effectiveness of the killer module, the models indeed have carried out merciful strategies which lead to lose, as mentioned in section III-A3.

D. Third Stage: Competing with Master Model

In this section, we would display the outcomes of the third stage in our pipeline. Here, we not only designed and implemented GomokuBot but also refined our DuelNet by engaging it in competitions with GomokuBot, allowing it to learn and evolve from these experiences.

TABLE III
300 GAME OUTCOMES OF DIFFERENT MODELS

Model	Win	Lose	Tie
GomokuBot	291	2	7
Second Stage	284	16	0
Third Stage	285	15	0

The results in Table III are derived from 300 games playing against PureMCTS with simulation time of 2000. And the Second Stage model is exactly the same one as in Table II. The Third Stage Model is trained by the data collected from playing against GomokuBot.

As evident from the results, GomokuBot outperforms other models with impressive efficiency. The model in the third stage shows a slight improvement over the second stage, though not a massive leap. We suspect that pushing a Neural Network-based model, already excelling, towards a fully rule-based agent becomes tougher, resulting in limited performance gains. However, this outcome might be due to insufficient training or improper settings, aspects we need to delve into further. Considering our tight schedule and computing resources, we find these results to be quite satisfactory.

E. Game Visualizations

For better visualizations, we build a website using `streamlit` to display our Gomoku Agent lively. You could visit it at https://huggingface.co/spaces/Gomoku-Zero/Demo. We recommend **cloning the repository locally for a smoother gaming experience**, as the Gomoku Agent's reaction time on HuggingFace's space tends to be much more slower, as we have tested.

ACKNOWLEDGMENT

We gratefully thank Prof. Gao and the TAs for their generous help throughout the course.

REFERENCES

[1] E. Zermelo, "Über eine anwendung der mengenlehre auf die theorie des schachspiels," in *Proceedings of the fifth international congress of mathematicians*, vol. 2. Cambridge University Press Cambridge, 1913, pp. 501–504.

[2] Wikipedia contributors, "Alpha–beta pruning — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=1188156145, 2023, [Online; accessed 5-January-2024].

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

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

[5] 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.

[6] 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.

[7] 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.

[8] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. P. Lillicrap, and D. Silver, "Mastering atari, go, chess and shogi by planning with a learned model," *CoRR*, vol. abs/1911.08265, 2019. [Online]. Available: http://arxiv.org/abs/1911.08265

[9] T. Gao, "Viztracer," https://github.com/gaogaotiantian/viztracer.

[10] C.-Y. Kao, H. Guei, T.-R. Wu, and I.-C. Wu, "Gumbel muzero for the game of 2048," in *2022 International Conference on Technologies and Applications of Artificial Intelligence (TAAI)*. IEEE, 2022, pp. 42–47.

[11] A. Nair, B. McGrew, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Overcoming exploration in reinforcement learning with demonstrations," in *2018 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2018, pp. 6292–6299.

[12] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1995–2003.

[13] Wikipedia contributors, "Zobrist hashing — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Zobrist_hashing&oldid=1170414646, 2023, [Online; accessed 5-January-2024].

VI. APPENDIX

A. Appendix A: The souce of openings

- The 3rd Anji World Gomoku Open Tournament Game Record
- 2020 National Gomoku Championship Game Record
- 2019 National Gomoku Open Tournament Game Record

- The 12th National Gomoku Championship Game Record
- The 5th World Team Championship Game Record
- 2012 Pjuga Sbory Game Record
- The 17th All Japan Renju Championship (Pearl King Battle) Game Record
- The 41st St. Petersburg Renju Championship Game Record
- The 4th National Intellectual Sports Games Women's Individual Gomoku Tournament Game Record
- "Kangbo Cup" Gomoku International Invitational Tournament Game Record

B. Appendix B: The Bots for generating data

- Embryo18
- Embryo21
- Rapfi
- Rapfi19
- Rapfi21
- pbrain-SlowRenju
- pbrain-whose2019
- pbrain-pela