CS311 Project1 Report: An Implementation of Reversed Reversi Agent through Monte Carlo Method

Zhengdong Huang *12212230*

***Abstract*—This article is a project report of CS311 Project 1. In this report, I proposed a Deep-learning-based MCTS algorithm and a training process based on reinforcement learning to implement a Reverse Reversi Agent. Also, I analyzed the model's complexity and proposed optimizing strategies. At the same time, I conducted experiments to verify the effectiveness of the proposed model and optimizing strategies, with analysis and summaries for the experimental results.**

***Index Terms*—Reversed Reversi, Monte Carlo Tree Search, Reinforcement Learning**

I. Introduction

IN the realm of artificial intelligence and game theory, the development of effective strategies for board games has long been a challenging yet intriguing pursuit. Among these games, Reversi, also known as Othello, stands out for its dynamic gameplay and strategic depth. Originating in the late 19th century, Reversi has garnered attention from researchers and enthusiasts alike due to its simple rules yet complex strategic possibilities.

In the classic game of Reversi, players strive to convert as many adversary pieces to their own color, effectively asserting dominance over the board. However, Reversed Reversi presents a twist, challenging players to minimize the presence of their own colored pieces on the board by game's end. This inversion of traditional strategy demands a fresh perspective, prompting players to reevaluate their tactics and innovate new approaches in pursuit of success.

In 2017, David Silver and his team introduced a groundbreaking Monte Carlo search tree-based deep reinforcement learning algorithm [14], [15]. Their innovative approach delivered exceptional results across various board games, highlighting a potential avenue to develop an advanced agent for the game Reversed Reversi. Through probabilistic simulations and iterative refinement, Monte Carlo Tree Search provides a robust framework for decision-making in uncertain and dynamic environments, bypassing the need for manually predefined gaming strategies. Additionally, the integration of Reinforcement Learning enhances game agents through a self-training process, offering a hopeful solution for implementing Reversed Reversi Gaming Agents.

Moreover, the insights garnered from this study extend beyond the realm of Reversed Reversi, holding potential for addressing real-world challenges characterized by complex, adversarial dynamics. By modeling strategic decision-making in competitive and dynamic environments, the methodologies developed here may find applications in diverse domains such as cybersecurity, resource allocation, and negotiation strategies.

This paper will primarily demonstrate the fundamental principles of the Monte Carlo Search Tree (MCTS) algorithm and the Reinforcement Learning (RL) method. It will delve into the application of MCTS and RL algorithms in developing a Reverse Reversi Agent, and endeavor to compare its performance with traditional search algorithms. We seek to elucidate the effectiveness of our methodology and its implications for problem-solving domains.

II. Preliminary

A. Game Description

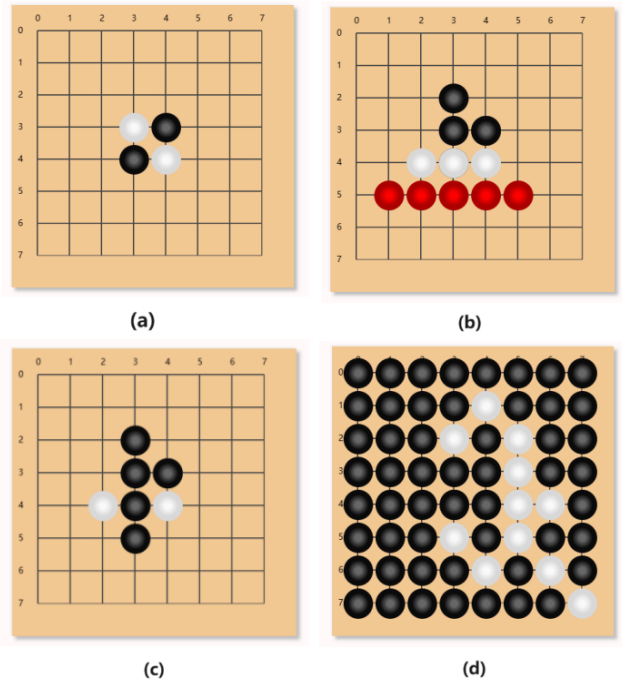

Fig. 1: **Different State in Reverse Reversi**. (a) The initial board state at the onset of the game. (b) Red points represent the valid moves for next step in black turns (c) Showing the board state after placing black chess at (6,3) from board state in b (d)

Othello is a strategic, turn-based game where players, represented by black and white disks, vie for control over an 8x8

grid. Throughout the game, players take turns placing legal discs on the board and flipping their opponent's discs until the game ends. Each turn, players must place their own disks in a way that sandwiches the opponent's disks horizontally, vertically, or diagonally between their own. If a player cannot make a valid move and their opponent can, the turn passes to the opponent. The game continues until the board is full or neither player can make a valid move. Traditionally, victory is achieved by having the most disks on the board. However, in our project, Reversed Reversi, the player with the fewest disks emerges as the victor. Sample game state is shown in Fig. 1

B. Problem Formulation

Table 1 presents the important notations utilized in the problem formulation. The formulation of the problem is defined as follows: Given a chessboard s, determine the optimal move a_{best} that maximizes the probability of winning the game. In the method we used that employs the Monte Carlo Tree Search (MCTS) and Reinforcement Learning (RL) algorithms, we have also defined several parameters that aid in making the decision for the move.

Table 1: Notations

Notation	Name	Explanation
s	chessboard state	Current state of the chessboard
a	action	Position for a $board$ to drop a disc
a_{best}	Best Action	Best position for a $board$ to drop a disc
$U(s,a)$	Node Evaluation	Evaluation of (s,a) values
$Q(s,a)$	Node Experience	Average reward of (s,a) by simulation
c_{puct}	Hyperparameter C	PUCT hyperparameter for selection
$P(s,a)$	Prior Probability	PUCT prior probability for (s,a)
$N(s,a)$	Node Visits	Number of node (s,a) visits
$R(s,a)$	Node Reward	Cumulative reward on node (s,a)
f_θ	Neural Network	Neural network parameterized by θ
v_θ	NN Evaluation	Neural network Evaluation for given s
$\vec{p_\theta}$	NN Policy	Neural Net work policy for given s
v_s	State Evaluation	Evaluation of win rate for state s
r	Game Result	Result of the game indicating winner
$\vec{p_s}$	Action Probability	The prior probability vectors for state s
s_t	State Train	State in train example
$\vec{\pi_t}$	Possibility Train	Enhanced possibility in train example
r_t	Result Train	Game Result in train example
N_e	MCTS Iteration	Iteration time of single MCTS
b	Avg Branching Factor	Average branching factor of search tree
d	Tree Depth	Search Tree Depth of MCTS
s_t	Simulation Time	Time consumed per simulation
n	Node	Node in Search Trees

III. Methodology

The agent's design is centered around the implementation of a Deep-Learning based Monte Carlo Tree Search algorithm and training neural networks through reinforcement learning. This approach is complemented by various optimization strategies to boost the agent's performance. The workflow of this method will be presented in subsection A, followed by a detailed explanation of the agent algorithm in subsection B. The reinforcement-based training approach will be introduced in subsection C, with the performance analysis and optimizing strategies covered in subsections D and E.

A. General Workflow

The general workflow of the agent algorithm can be described as Fig.2:

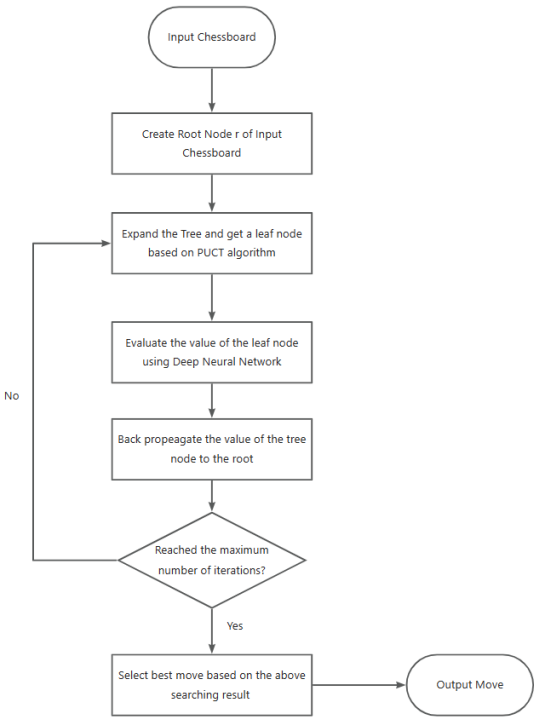

Fig. 2: Agent Workflow

The algorithm described above primarily utilizes the Monte Carlo Tree Search (MCTS) algorithm in conjunction with a Deep Neural Network (DNN) for node and action evaluation. We employ a reinforcement learning algorithm to create a dataset and train the neural network through self-play. The overall training workflow is shown in Fig.3.

B. Agent Algorithm

1) Monte Carlo Tree Search: The agent algorithm primarily utilizes the Monte Carlo search tree method, a decision tree search algorithm that relies on statistical simulation [4]. The fundamental steps of this algorithm are outlined below:

Selection: Commencing from the root node, a node is selected for expansion based on a specific strategy.

Expansion: The chosen node is expanded to produce its children.

Simulation: The expanded child node is simulated to derive the node's evaluation.

Back-propagation: Following the simulation results, the statistics of each node on the path from the current node to the root node are updated.

The pseudo-code for the overall process is shown in Algorithm 1.

In our implementation, we have utilized the Prediction Upper Confidence Trees (PUCT) algorithm for node selection, a refined version of the Upper Confidence Trees (UCT) algorithm [3], [8]. The UCT algorithm employs the Upper

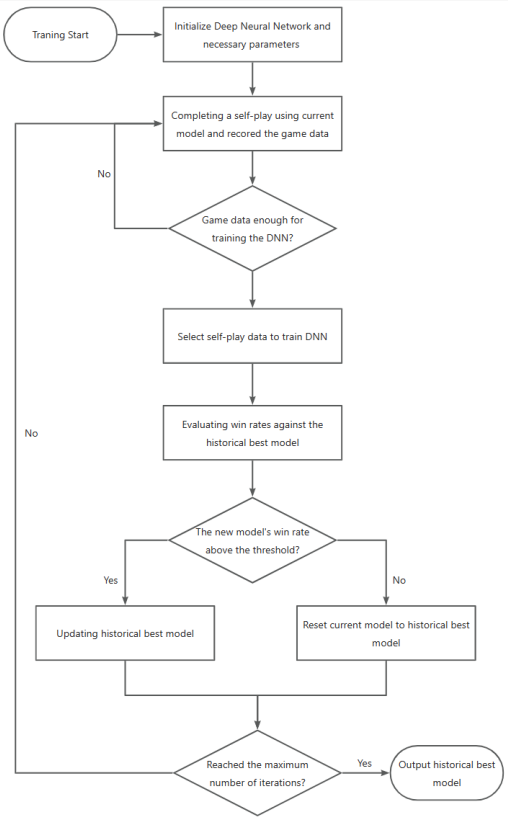

Fig. 3: Training workflow

Algorithm 1 MCTS

Input: Current Board State s_0
Output: Predicted optimal action a^*

$n_0 \leftarrow$ create_node(s_0)
while Haven't reach the max number of iterations **do**
    $n_{leaf} \leftarrow$ Selection(n_0)
    $r_{leaf} \leftarrow$ Simulation(n_{leaf})
    BackPropagation(n_{leaf}, r_{leaf})
end while
$a^* \leftarrow$ PUCB(n_0)

Confidence Bounds (UCB) theory to gauge the level of exploration of nodes in the search tree. Higher UCB values of a node indicate greater uncertainty and signify the need for more exploration and simulation. The PUCT algorithm enhances this by introducing the concept of action a prior probability $P(s, a)$. This probability, derived from the neural network and integrated into the original UCT algorithm, aids in the improved selection of nodes requiring expansion at the current stage. The comprehensive formulation of the PUCT algorithm is outlined as follows:

$$U(s,a) = Q(s,a) + c_{puct} \cdot P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$$

Here $U(s, a)$ is the final evaluation of the action a of a state s, combining expected reward $Q(s, a)$ and the prediction upper confidence bound. $N(s, a)$ represents the number of visited time of action a of a state s. The formula for $Q(s, a)$ is as follows, where $R(s, a)$ represents the cumulative reward:

$$Q(s,a) = \frac{R(s,a)}{N(s,a)}$$

c_{puct} is a hyper-parameter controlling the degree of exploration. The choice of the hyperparameter c_{puct} will significantly affect the tendency of the MCTS algorithm to expand nodes, when the value of C is higher, the algorithm tends to expand nodes with high uncertainty. For most optimization methods, selecting an initial value of C within the range of $c \in [1, \sqrt{2}]$ has shown to yield superior results across various scenarios [1]. After conducting practical experiments, we ultimately settled on the hyperparameter value of $C = 1.2$. In the implementation, we will compute $U(s, a)$ for all actions and finally return the action a with the highest evaluated value. The pseudo-code below is shown as below:

Algorithm 2 PUCB

Input: Node n
Output: Selected Action a^*

$a^* \leftarrow arg\max_{a' \in n.\text{actions}} U(n.s, a')$

During the Simulation operation, when the current board is not the final board, we use the neural network's evaluation of the board $v_\theta \in [-1, 1]$ as the reward, with values closer to -1 indicating an advantage for the opponent, and conversely for values closer to 1. However. In the case where the current board represents the final state of the game, we return $r \in \{-1, 0, 1\}$, which respectively represents the opponent's win, draw and our win. The pseudo-code below is shown as below:

Algorithm 3 Simulation

Input: Leaf Node n
Output: Predicted reward r

if GameIsEnd(n.s) **then**
    $r \leftarrow$ OurVictory$(n.s)$
else
    $\vec{p_a}, r \leftarrow$ NeuralNetwork$(n.s)$
    Expand$(n, \vec{p_a})$
end if

And the steps of Selection,Expansion and Propagation are exactly the same as the classical MCTS algorithm, and the pseudo-code is as follows:

Algorithm 4 Selection

Input: Node n
Output: Leaf Node n_{leaf}

while not IsLeaf(n) **do**
    $n \leftarrow$ GetChildNodeByAction$(n,$ PUCB$(n))$
end while

Algorithm 5 Expand

Input: Node n with its action prior possibility $\vec{p_a}$
Output: Node n after child node expanded

for each $action, probs$ **in** $\vec{p_a}$ **do**
  $n_c \leftarrow$ GetNodeByAction(a)
  $n_c.p \leftarrow probs$
  AddChild$(n, n.c)$
end for

Algorithm 6 BackPropagate

Input: Node n with its reward r

while n**is not null do**
  $n.v \leftarrow n.v + 1$
  if $n.parent$**is null then**
    $n.r \leftarrow n.r + r$
    break
  else if IsOurSide$(n.parent)$ **then**
    $n.r \leftarrow n.r + r$
  else
    $n.r \leftarrow n.r - r$
  end if
  $n \leftarrow n.parent$
end while

2) Neural Policy and Value Network: We utilize a neural network f_θ parameterized by θ to process the board state s. This network outputs the continuous value $v_s \in [-1, 1]$ of the board state from the viewpoint of the current player. Additionally, it also generates a probability vector $\vec{p_s}$ representing all possible actions a. This $\vec{p_\theta}$ acts as a stochastic policy that guides the self-play process.

The neural network is initially initialized with random values. After each round of self-play, the network receives training examples in the form of $(s_t, \vec{\pi_t}, r_t)$. Here, $\vec{\pi_t}$ provides an enhanced policy estimate derived from performing MCTS starting at s_t during the self-play which are used for mainly improving the precision of estimating $\vec{p_\theta}$, while $r_t \in \{-1, 0, 1\}$ indicates the game's final result from the current player's perspective, aiming to enhance the accuracy of evaluating v_s.

The neural network is then trained to minimize the following loss function, the loss function is defined combining Mean Squared Error and Cross-Entropy Loss Functions [14]:

$$l = \sum_t \left(v_\theta(s_t) - r_t\right)^2 + \hat{\pi}_t \log(p_\theta(s_t))$$

We employ a neural network that takes the raw board state as its input. This is succeeded by 3 convolutional networks and 2 fully connected feedforward networks, following with 2 connected layers - one that yields v_θ and another that yields the vector $\vec{p_\theta}$. Training is executed using the Adam optimizer [9], incorporating dropout [16] and batch normalization [7].

C. Training Approach

We employ a RL-based algorithm to train the neural network f, primarily using self-play to generate training data. Initially, our neural network is initialized with random weights to start with. The algorithm subsequently undergoes a self-improving iteration as follows:

Self-play: In each iteration, the algorithm engaging in a series of self-playing using MCTS. This process yields a collection of training examples in the form of $(s_t, \vec{\pi_t}, r_t)$.

Training: Subsequently, The algorithm updating our neural network by using the train examples from self-play, minimizing the loss function described in Section III-B2.

Evaluation: To assess the progress, the algorithm pit both the old and new networks against each other in multiple games. If the win ratio of the new network surpasses a predefined threshold, the algorithm replace the best network, otherwise we discard the new network. The algorithm will then move on to the next iteration.

The pseudo-code of the training process is shown below:

Algorithm 7 Policy Iteration through Self-Play

Output: Neural Network Parameter θ

$\theta \leftarrow$ initNeuralNetwork()
$trainExamples \leftarrow []$
while not reach the max number of iterations **do**
  for $e \in [1, \ldots,$ numEpisodes$]$ **do**
    $ex \leftarrow$ executeEpisode(f_θ)
    $trainExamples$.append(ex)
  end for
  $\theta_{\text{new}} \leftarrow$ trainNeuralNetwork(trainExamples)
  if WinRatio$(\theta_{\text{new}}, \theta) \geq$ threshold **then**
    $\theta \leftarrow \theta_{\text{new}}$
  end if
end while

D. Performance Analysis

1) Complexity Analysis: The time complexity of the MCTS search tree is influenced by various factors, making its analysis challenging. In the following discussion, we aim to provide a rough examination on time complexity: The time complexity of MCST can be approximated to:

$$O(N_e \cdot (selection + expansion + simulationb + backup))$$

For the selection part, the average time complexity is

$$O(b \cdot d)$$

Here b is the average branching factor, which is $b \approx 10$ [11], and d is the depth of the tree.

For the expansion part, the time complexity is $O(b)$ since we only need to expand the current node by considering its immediate children at the next level.

For the back-propagation(backup) part, it takes $O(d)$ to recursively go back and update from the left to the root.

For the simulation part, we need to determine whether the current state is an endgame or not, which takes $O(1)$ for a

fixed 8x8 chessboard, and evaluate the nodes based on the neural network. Considering the convolution layer, its time complexity can be described as follows [6]:

$$\text{ConTime} \sim \mathcal{O}\left(\sum_{l=1}^{D} M_l^2 \cdot K_l^2 \cdot Cin_l \cdot Cout_l\right)$$

Here D represents the number of convolution layer, which is 3 in our design. And the M_l represents the length of Feature Map for the l-th convolution layer, while K_l represents the length of the kernel. The Cin_l and $Cout_l$ represent the number of input and output channels, respectively.

Also, the time complexity of fully connected layer can be described as follows [13]:

$$\text{FulTime} \sim \mathcal{O}\left(\sum_{l=1}^{D} C \cdot W \cdot H \cdot N\right)$$

Here D is the Depth of the fully connected layer. C, W, H, and N define the Dimension of the input/output channel, width of input, the height of the input, and the number of outputs respectively.

Since operations like batch normalization and dropout do not significantly impact the forward propagation time [13], the overall time complexity of the simulation process can be approximated as:

$$s_t = O(\text{FulTime} + \text{ConTime})$$

By above analysis, the total time complexity of the MCTS algorithm is

$$O(N_e \cdot (b \cdot d + b + s_t + d))$$

2) Optimality: Utilizing randomness and statistical simulation, the Monte Carlo Tree Search (MCTS) method does not ensure optimal decision-making but rather aims to discover a satisfactory solution within a reasonable timeframe. Nonetheless, three crucial factors influence the quality of decision-making:

Iteration Time N_e: The number of iteration time significantly impacts the final evaluation outcomes, while more simulations lead to more precise results [4].

Iteration Process: The selection and simulation process in each iteration profoundly affect the evaluation of each move. Here we use PUCT algorithm for selection and Neural Network for simulation, which has been shown effectiveness by previous research [10], [14], [15].

Game Process: In the initial stages of the game, the evaluation of nodes heavily depends on the neural network due to the need for deeper search depth. This reliance may result in potential inaccuracy. However, as the game nears its end, the node's evaluation is determined directly by the win/loss outcome, which is notably accurate. Moreover, the search tree size diminishes as the game progresses, allowing more precise results under limit time compared to a larger search tree.

E. Optimization Strategy

The optimization strategy of the method described above focuses on two key components: enhancing the MCTS performance and improving training speed. Here we introduce four specific optimization techniques, the empirical evaluation on the optimizing strategy is shown in Section IV.

1) Enhancing MCTS Performance: **Reusing Search Tree:** Instead of constructing a new search tree for each invocation on MCTS, a more efficient approach involves leveraging information from previous invocation [5], [8], thereby increasing the evaluation accuracy for each node and improve the efficiency.

One method to reuse past search information is using a hash-map to store all visited nodes. While this strategy can improve efficiency by preventing redundant node evaluations, it may become inefficient as it accumulates all the stored nodes throughout the game. Furthermore, utilizing a hash-map can present challenges during back-propagation, especially when multiple parents share the same child, potentially leading to adverse consequences [12].

Alternatively, a viable solution is to implement tree pruning, where the search tree is pruned to the subtree after next invocation on MCTS. By discarding nodes above the current position or on divergent branches, unnecessary nodes are eliminated, enabling the algorithm to retain and reuse information from previous MCTS invocation, thereby enhancing efficiency [18].

Researchers have also proposed utilizing graphs to circumvent duplicate nodes within the search tree [5]. However, the overhead of identifying identical nodes during the search can consume both time and memory resources, rendering it less effective in games like Reverse Reversi due to the low possibility of multiple actions leading to the same state.

Depending on the previous analysis, we choose Tree Pruning as solution to reuse the MCTS search tree. We can implement it by simply modifying the function of creating root node. The pseudo-code is shown as follows:

Algorithm 8 GetRoot

Input: root n for previous invocation on MCTS, current chessboard s

Output: root n current invocation on MCTS

```
1:  if n is null then
2:      n ← Node(s)
3:  else
4:      n ← GetChildByAction(n, a_lastInvoke)
5:      as ← OpponentActions(n, s)
6:      for each a in as do
7:          if IsLeaf(n) then
8:              n ← Node(s)
9:              break
10:         else
11:             n ← GetChildByAction(n, a)
12:         end if
13:     end for
14: end if
```

Optimizing the code speed: Based on the prior analysis, the number of MCTS iterations N_e, plays a crucial role in determining the accuracy of the final evaluation of nodes. Given the constraints of time, the magnitude of N_e depends on the efficiency of the code's execution within the loop. Therefore, enhancing the speed of code execution can profoundly improve the evaluation's precision. To address this,

we adopt Numpy and Numba to improve the code execution speed. Numpy offers a comprehensive set of well-implemented interfaces for calculation, while Numba can improve the code execution speed using JIT technique cooperating with Numpy.

2) Improving Training Speed: **Using Warp-Start strategy:** Warm-Start strategy is used to speed up the initial training of the model [17]. By replacing self-play process with play with other "mature" MCTS opponents, such as random-based MCTS used in our experiment, to obtain training data in the early stage, this strategy can generate higher quality training data, boosting overall training speed.

Training with Multi-processing: As analysis in Section III-C, the training process is divided into three stages: Self-play, Training, and Evaluation. In our implementation, we have optimized the neural network computation speed by leveraging CUDA to fully harness the power of the GPU, resulting in a substantial acceleration of the Training phase. However, the iterative MCTS process still heavily relies on CPU computation, limiting the speedup in the Self-play and Evaluation stages. Employing multi-processing on multi-core CPU can effectively maximize CPU performance, leading to a noteworthy enhancement in the overall training speed.

IV. Experiments

In this segment, we aim to evaluate the performance of the algorithms and optimizing strategy mentioned in Section III. Our experiments will primarily focus on: (a) Comparing DNN-based MCST with alternative models, (b) Assessing the effectiveness of the MCTS acceleration, (c) Assessing the effectiveness the Warm-Start method, and (d) Assessing the training speed of the multi-process training platform.

A. Setup

Configuration:
CPU: 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz
GPU: NVIDIA GeForce RTX 3060 Laptop GPU
RAM: 16G
Operating System: Windows 11 23H2
Python Environment:
Python Version: 3.7.6
Numpy Version: 1.21.0
Numba Version: 0.56.4

B. Results And Analysis

1) Effectiveness evaluation on DNN-based MCST: In this section, we will mainly play the DNN-based MCTS model (with acceleration method adopted) against other models. We have selected DNN-based models with different number of training iterations and compared them with Rnd model, Gdy model, Rnd-MCTS model, Rox-MCTS model and Gdy-AB model. For each comparison, the two models will take turns to compete with each other for a total of 50 times of the game and the final winning percentage will be calculated.The description of the compared models is as follows.

1. Rnd Model: The model will randomly drop pieces at valid points on the board.

2. Gdy Model: The model will greedily choose the point at which the fewest pieces are flipped.

3. Rnd-MCTS Model: MCTS algorithm based on stochastic simulation. For the simulation part, the model will randomly drop pieces on the board until the end of the game and use the winning results as node weights.

4. Rox-MCTS Model: MCTS algorithm based on location-first strategy. The model will drop disc during the simulation to the end based on the location-first strategy, and use the winning result as the node weights. The weights of the positions on the board are modified based on Othello-Roxanne Strategy [2], which combines the Mobility and possible Stability, as shown in Fig 4:

1	5	3	3	3	3	5	1
5	5	4	4	4	4	5	5
3	4	6	6	6	6	4	3
3	4	6			6	4	3
3	4	6			6	4	3
3	4	6	6	6	6	4	3
5	5	4	4	4	4	5	5
1	5	3	3	3	3	5	1

Fig. 4: Modified-Roxanne Strategy

5. Gdy-AB model: Alpha-beta pruning algorithm based on greedy evaluation. and the weight function is determined by subtracting the number of opponent's discs from the number of our discs, $W = N_{oppodisc} - N_{ourdisc}$.

The experiment results of competing the above model are as Fig. 5. The MCTS algorithms all have a time limit of 5s, and the the maximum depth of the search tree for Gdy-AB model is six levels.

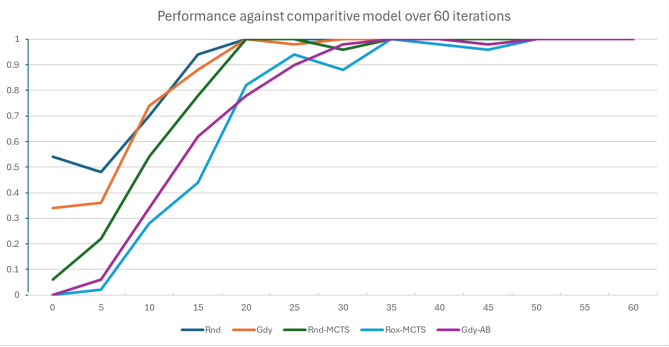

Fig. 5: Performance against comparative models

The experimental results showcase the impressive performance of the DNN-based MCTS model. After just 20 iterations of training, the model's winning rate against the initial three models approached nearly 100%. And until 35 iterations, the DNN-based MCTS consistently achieved close to a 100% win rate against all models, underscoring the remarkable efficacy of the algorithm.

2) Effectiveness of the MCTS Acceleration approach: We selected the neural network after 60 iterations of training and chose the unoptimised MCST model (M0) to perform fifty matches in MCST with Tree reusing (M1), MCTS with Numba

(M2), MCTS with Tree reusing and Numba (M3) alternately in succession, and the results of the win ratio are as Table 2.

Table 2: Win ration in Experiment 2

M0 Competitor	Win Ratio
M1	0.82
M2	0.74
M3	0.86

We found the increment of performance for all the models using optimizing strategies. Further more, to further validate the effectiveness of optimization, we conducted a comparison of the actual number of iterations per invocation of MCTS across various models. We notice that as the game nears its end, the number of MCTS iterations notably increases due to the reduced search tree size, thus we computed the average number of iterations per invocation of MCTS for ten sets of games consisting of 20-40 hands. The experimental results are detailed in Table 3

Table 3: Number of Iteration in Experiment 2

Model	M0	M1	M2	M3
Iteration Times	1657	2641	2159	2832

We note a significant increase in the number of iterations, with the average number of iterations exceeding 2800 for the model that employs both optimisations, whilst further analysis shows that in general, the reuse search tree is able to provide an additional 300-1300 iterations of MCTS search data, while Numba can improve about 25% of the code running speed. The optimised model achieved better performance results with more iterations, demonstrating the effectiveness of the two optimisation strategies described in III-E

3) Effectiveness of Warm Start Strategy: For the Warm Start optimisation, we select the model that has been trained 15 iterations with the Warm Start strategy and the model trained with different iterations without the Strategy for comparison, and each group of comparisons takes turns to play with 50 play times in total. The experimental results of the win rate are as Fig.6

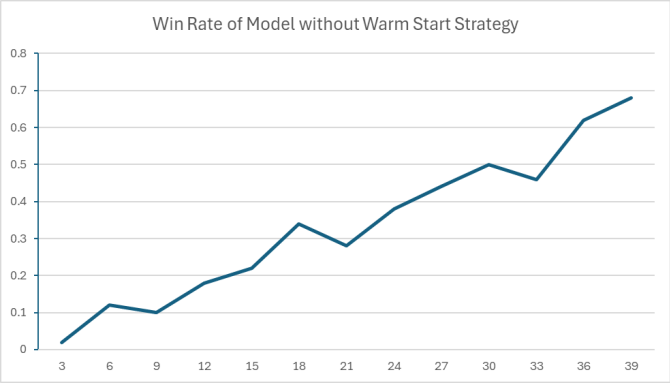

Fig. 6: Win Ratio Of Model Without Warm Start Strategy

We can see that it is not until about 27-33 iterations that the model can reach a win rate close to 0.5 against model with warm start strategy, and the model significantly worse than optimized model for the same number of training, reflecting the effectiveness of the Warm Start Strategy on improving the speed of training at the early stages.

Platform	No multi-processing	8 process
Avg Times (min)	16.18	7.76

4) Effectiveness of Multi-processing: To evaluate the effectiveness of multi-processing in improving the training speed, we conducted ten iterations of training using the training platform without multi-processing and the platform with multi-processes(process number 8), respectively, and counted the average time consumed in each round of training. The results are as shown in Table 5

Table 5
Average Time for one Iteration

We see that with multi-processing turned on, the time taken for each round of training is reduced to about 50% of the original, significantly demonstrated the effectiveness of multi-processing in accelerating training.

When observing the performance panel of the training equipment machine, we can see that the overall CPU utilization in single-process mode is only 20-35%, as shown in Fig 7, indicating the presence of idle CPUs. However, the overall CPU utilization in multi-process mode is between 75-95%, with the majority of CPUs being fully utilized, explaining reason of the improvement of multi-processing on training speed.

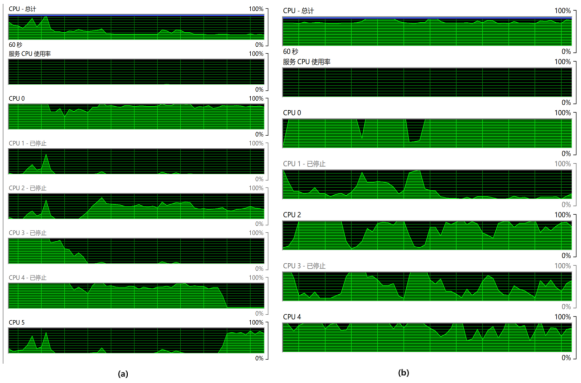

Fig. 7: **CPU usage during training on different platforms:** (a) Single-process platform with total CPU usage of 20-35% with high CPU idle rate. (b) Multi-process platform, total CPU usage 75-95% with low CPU idle rate.

However, the speed bottleneck of training is influenced by various factors. In our experiment, the GPU utilization in multi-process mode remains at 95-100% for a long period, higher than the CPU utilization. Considering that both Self-play and Evaluation stages also require neural networks for valuation, the GPU may be the bottleneck for overall training speed. Additionally, the monitoring panel shows that the CPU core temperature remains at 90-95°C for an extended period, and during the training process, the CPU exhibits periodic frequency decreases and rebounds, indicating that the machine may have undergone automatic downclocking due

to overheating protection, suggesting that the overall training speed may also be limited by the cooling level.

V. Conclusion

In this paper, I propose a Reversed Reversi Agent based on the MCTS algorithm, combining deep neural networks and reinforcement learning methods. I also present four optimization methods to improve the model performance and training speed. Additionally, I analyze the overall complexity and performance of the model and conduct empirical studies to test the effectiveness of the model and optimization strategies. The experimental results show that the model and corresponding improvement strategies have good performance in Reversed Reversi, achieving close to 100% win rate in competition with multiple models. The improvement strategies effectively enhance the model performance and training effectiveness. The overall methods proposed in the paper demonstrate high usability.

Due to hardware limitations, I opted to train the model using a smaller neural network. After extensive training, I observed that the model may have reached saturation, as the loss value stabilizes after a considerable number of training iterations. Upgrading to a larger neural network model and further optimizing the overall code could potentially enhance the model's effectiveness. Moreover, the model approach can be extended and transferred to different board games, different games, or even applied to real-world problem-solving scenarios.

Throughout this project, I delved deep into the knowledge of MCTS algorithm, deep neural networks, and reinforcement learning algorithms. The overall model approach was inspired by the model design and training methods of AlphaZero [14]. This method, which does not rely on human knowledge, has shown remarkable performance. I also referred to optimization and improvement strategies proposed in various papers, learning a great deal of knowledge in the field of artificial intelligence from researchers' results, improving my capabilities. Implementing the Agent Algorithm, multiprocessing training platform, and comparing models (MCTS, AB pruning AI) posed challenges but significantly enhanced my Python programming skills. It also familiarized me with the usage of libraries like Numpy and Numba, providing me with valuable learning experiences.

VI. Acknowledgement

I would like to express my gratitude to Professor Bo Yuan for his exceptional course. Additionally, I am grateful for the guidance provided by SA in the labs. I truly appreciate the opportunity that CS311 has given me to implement this project. The report may be a little long as it contains a lot of pictures and tables, thanks for your patience for reading until the end.

References

[1] F. V. Ameneyro and E. Galvan. Towards understanding the effects of evolving the mcts uct selection policy. *arXiv preprint arXiv:2302.03352*, 2023.

[2] R. Archer. Analysis of monte carlo techniques in othello. 01 2007.

[3] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2–3):235–256, may 2002.

[4] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[5] J. Czech, P. Korus, and K. Kersting. Monte-carlo graph search for alphazero. *arXiv preprint arXiv:2012.11045*, 2020.

[6] K. He and J. Sun. Convolutional neural networks at constrained time cost. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5353–5360, Los Alamitos, CA, USA, jun 2015. IEEE Computer Society.

[7] S. Ioffe and C. Szegedy. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 448–456. JMLR.org, 2015.

[8] S. James, G. Konidaris, and B. Rosman. An analysis of monte carlo tree search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), Feb. 2017.

[9] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[10] K. Matsuzaki. Empirical analysis of puct algorithm with evaluation functions of different quality. In *2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 142–147, 2018.

[11] A. Norelli and A. Panconesi. Olivaw: Mastering othello without human knowledge, nor a fortune. *IEEE Transactions on Games*, 15:285–291, 2021.

[12] A. Saffidine, T. Cazenave, and J. Méhat. Ucd : Upper confidence bound for rooted directed acyclic graphs. *Knowledge-Based Systems*, 34:26–33, 2012. A Special Issue on Artificial Intelligence in Computer Games: AICG.

[13] B. Shah and H. Bhavsar. Time complexity in deep learning models. *Procedia Computer Science*, 215:202–210, 2022. 4th International Conference on Innovative Data Communication Technology and Application.

[14] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv*, 2017.

[15] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[16] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, jan 2014.

[17] H. Wang, M. Preuss, and A. Plaat. Adaptive warm-start mcts in alphazero-like deep reinforcement learning. 05 2021.

[18] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk. Monte carlo tree search: a review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, 2023.