Intelligent Agents for Othello: Methods, Implementation and Evaluation

Zaizhou Yang[*], **Shengxin Li**[*], **Yiang Ju**[*], **Zhuo Diao**[*], **Guanjie Huang**[*]
School of Information Science and Technology
ShanghaiTech University
{yangzzh, lishx1, juya, diaozhuo, huanggj}@shanghaitech.edu.cn
2021533{015, 019, 088, 046, 181}

https://github.com/mike3090/Othello_CS181

Abstract

Othello, a classic game, serves as a pivotal platform for the development and evaluation of various artificial intelligence algorithms. Traditional approaches to solving Othello often rely heavily on feature extraction, a process which limits their adaptability to diverse scenarios. In this study, we firstly commenced with a literature review of existing methods. Subsequently, we proposed and implemented three distinct methods for training an AI agent to play Othello: Minimax Search with Alpha-Beta Pruning, Deep Q Network (DQN), and Monte Carlo Tree Search (MCTS). Our evaluation results indicate that all three methods surpass the performance of the baseline: Random agent and Greedy agent, with a comparative effectiveness ranking of MCTS > DQN > Minimax. Additionally, we discuss the limitations of each method and suggest potential strategies for enhancing their performance.

1 Introduction

Othello, a board game characterized by its straightforward rules, emerges as an ideal subject for our CS181 project. Our aim is to apply the knowledge acquired in CS181 to develop an agent proficient in playing Othello. We have implemented and explored three distinct algorithms: Minimax Search with Alpha-Beta Pruning, Deep Q Network (DQN), and Monte Carlo Tree Search (MCTS). It is imperative to note that our primary goal is not solely to increase the winning rate but to utilize Othello as a testbed for algorithmic implementation and analysis.

Othello Othello is a two-player strategy game played on an 8x8 board, as depicted in Figure 1, which shows a screenshot of our application at the game's starting position. Players, represented by black and white discs, alternate in placing discs on the board. A move is valid if it traps one or more of the opponent's discs between the disc being placed and another disc of the player's color. These trapped discs are then flipped, changing to the player's color. A player must pass their turn if they cannot make a valid move. The game concludes when both players pass consecutively, and the winner is the player with the majority of discs on the board at game's end. A tie is declared if both players have an equal number of discs.

Contribution The primary contributions of this project are:

[*] All authors contributed equally to this project.

Preprint. Under review.

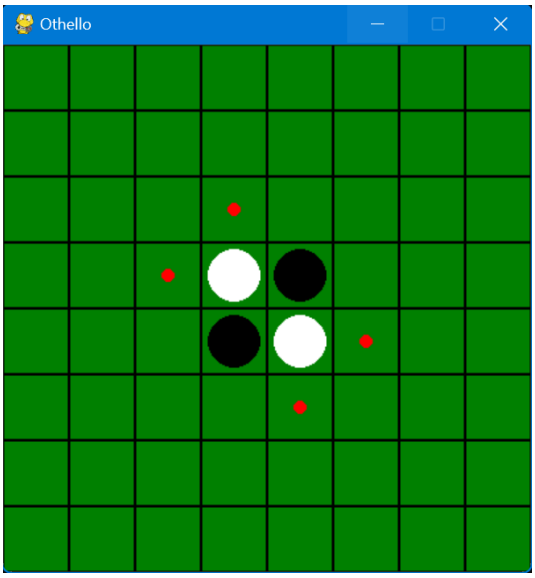

Figure 1: Screenshot of the start position of Othello. The red dots indicate the possible move for the current player (black).

- Literature Review: We conducted a relevant literature review of existing methods and narrowed down to the 3 methods we propose.
- Algorithm Implementation and Training: We proposed, implemented, and trained agents using three strategies: Minimax Search with Alpha-Beta Pruning, DQN, and MCTS.
- Performance Evaluation and Analysis: We conducted a comparative evaluation of the agents against a baseline and within the group, offering insights and discussing potential enhancements.

2 Related Work

Inspired by the advances of AlphaGo[1] for mastering the game of Go provides 2 novel insights for handling the game of Othello: Reinforcement Learning (RL), and Monte Carlo Tree Search (MCTS). By conducting further literature review, results showed feasibility of our methods on Othello. Michiel van der Ree et al.[2] evaluated different strategies and algorithms for training RL agents playing Othello, hence proves feasibility for mastering Othello with RL. Volodymyr Mnih et al. proposed DQN[3] which successfully conducted human-level control over classical Atari games and achieve a level comparable to a professional human gamer across a set of games. Ziyu Wang et al. proposed Dueling DQN[4], a better network structure training both value and action advantage function, which leads to better policy evaluation and outperforms the previous DQN agents. Therefore, We have confidence in the feasibility of the RL method. Meanwhile, Maciej Świechowski et al.[5] reviewed recent modifications and applications of the MCTS method which provides us hints on applying MCTS onto the Othello scenario.

3 Proposed Methods

3.1 Minimax Search with Alpha-Beta Pruning

In the development of our minimax algorithm, we adhered to the foundational structure as taught in our class, incorporating alpha-beta pruning to enhance computational efficiency. The evaluation function, a critical component of the algorithm, is linearly composed of three distinct features:

- Piece Differential: This metric evaluates the net advantage in terms of the number of pieces owned by the chosen color compared to the opponent.

- Edge Control: It assesses the count of pieces that the chosen color possesses along the edges of the board. Edge control is strategically significant in Othello due to the stability and tactical advantages of edge positions.
- Corner Occupancy: This measures the number of corners occupied by the chosen color. Corner occupancy is crucial since corner pieces are unflippable and provide a stable foundation for control of the board.

To optimize the performance of the minimax algorithm, we conducted an extensive experimental phase, running 2000 simulations with various randomly initialized weight configurations for these evaluation criteria. Through this process, we identified and selected the set of weights that demonstrated the highest performance, enhancing the strategic efficacy of our minimax agent. The weights for the 3 features are $w_1 = 2.833, w_2 = 3.661, w_3 = 3.637$ respectively.

3.2 Deep Q Network

Motivated by the principles of Reinforcement Learning, we aimed to implement Q-Learning for problem-solving in the context of Othello. Traditional Q-Learning, however, necessitates the maintenance of a Q-table that records the Q-value for each $(state, action)$ pair. Given the immense state space of Othello, estimated at approximately 10^{28} states, maintaining such a Q-table is computationally infeasible. To address this challenge, we incorporated a Deep Q Network (DQN), wherein the Q-function, $Q(s, a)$, is approximated using a neural network.

3.2.1 Algorithm

A significant aspect of the DQN is its enhancement of learning efficiency through repeated utilization of training data. Contrary to the original Q-learning algorithm, where each data piece is used only once for Q-value updates, DQN leverages experience replay. This technique involves the creation and maintenance of a replay buffer, a repository where data quadruples $(state, action, reward, nextstate)$ sampled from the environment are stored. During the training phase of the Q-network, batches of data are randomly selected from this replay buffer. This approach not only boosts the efficiency of sample usage but also ensures that the samples conform to the independence assumption, a crucial factor for effective training in neural networks.

The pseudocode is as follows:

Algorithm 1 Deep Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode$= 1, M$ **do**
  Initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_t, a_t, r_t\phi_{t+1})$ from D
    Set $y_j = \begin{cases} r_j, & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-), & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ w.r.t. the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  end for
end for

100	-35	10	5	5	10	-35	100
-35	-35	2	2	2	2	-35	-35
10	2	5	1	1	5	2	10
5	2	1	2	2	1	2	5
5	2	1	2	2	1	2	5
10	2	5	1	1	5	2	10
-35	-35	2	2	2	2	-35	-35
100	-35	10	5	5	10	-35	100

Table 1: The reward for each grid.

3.2.2 Environment and Reward

In this project, we define our state space as the current configuration of the board, represented by a vector of length 64 (corresponding to an 8×8 grid). Each element of this vector is assigned a value of $+1$ to represent a black piece, -1 for a white piece, and 0 for an empty grid.

The action space is defined as a set of 65 possible actions, where each of the first 64 actions corresponds to a specific grid location, mapped by the formula $index = 8 \times row + col$. The 65th action represents the scenario of 'No valid position'.

The learning environment includes a 'greedy agent' that alternates turns with the DQN agent, facilitating the training process.

Our reward system consists of two components:

Endgame Rewards Upon the end of a game, the DQN agent receives a reward calculated as $10 \times (\#winner - \#loser)$, where $\#$ stands for the number of pieces owned by the agent. This reward is added if the agent wins and subtracted if it loses.

In-game Rewards During the game, the agent receives rewards based on the strategic value of the grid location chosen for piece placement. Corner grid locations, which offer a strategic advantage as they cannot be flipped once occupied, yield the highest rewards. Edge locations offer moderately high rewards, while the three grids adjacent to each corner incur negative rewards. This is designed to discourage moves that could potentially allow the opponent to capture a corner. The specific Grid-Reward relationship is detailed in Table 1.

3.2.3 Network Structure and Hyper-parameters

We initially employed a Multilayer Perceptron (MLP) with three hidden layers in the Deep Q Network (DQN). The network's input is a 64-dimensional vector representing the state space, while the output layer comprises 65 nodes, each corresponding to a possible action in the action space. The three hidden layers consist of 128, 256, and 128 nodes, respectively, utilizing the Leaky-ReLU activation function[2].

To enhance training efficiency and improve policy evaluation, we integrated a Dueling DQN architecture. This approach involves training two distinct estimators: one for the state value function, denoted as $V^\pi(s)$, and another for the state-dependent action advantage function, $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$. A key feature of Dueling DQN is its ability to update all $Q(s,a)$ pairs concurrently, as an update to $V(s)$ implies an update to all $Q(s,a)$ pairs.

However, the initial formulation of Dueling DQN, given by $Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha)$, posed identifiability challenges, preventing the distinct recovery of V and A from Q. To address this, we modified the architecture to normalize the advantage function by subtracting its mean, thus ensuring identifiability. The modified equation is

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha) - \frac{1}{\mathcal{A}} \sum_{a'} A(s,a';\theta,\alpha),$$

and the structural design of this network is illustrated in Figure 2.

[2]https://pytorch.org/docs/stable/generated/torch.nn.LeakyReLU.html

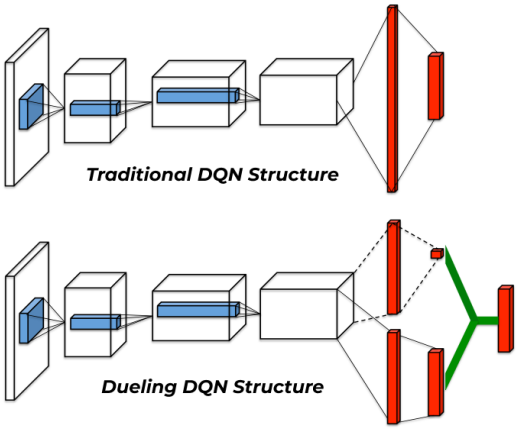

Figure 2: The structure of traditional DQN and Dueling DQN.

Regarding hyper-parameters, the learning rate for network training is set at $lr = 0.01$, with a discount factor $\gamma = 1$, and a learning rate for Q-Learning of $\alpha = 0.8$. Employing an ε-greedy algorithm to balance exploration and exploitation, the initial ε value is set at 0.8, decreasing linearly to 0.01.

3.3 Monte Carlo Tree Search

We employ a classical combination of Monte Carlo Tree Search (MCTS) and Upper Confidence Bounds for Trees (UCT), which significantly enhanced the winning rates of our models. This process initiates from the current game state, denoted as s, which serves as the root node of our search tree. After a number of iterations, the algorithm selects the child node of s with the highest visit count. This node represents the optimal action for the state s, as informed by the extensive search conducted.

Selection and Expansion The selection phase aims to identify a leaf node that is yet to be fully explored. Our policy here is designed to balance between exploration and exploitation. For this, we employ the widely-used Upper Confidence Bound (UCB) policy, defined as:

$$\pi_{UCB}\left(s_i\right) = \underset{a \in \mathcal{A}(s_i)}{\arg\max} \left\{ \frac{V\left(s_i, a\right)}{N\left(s_i, a\right)} + c\sqrt{\frac{\ln N\left(s_i\right)}{N\left(s_i, a\right)}} \right\}$$

where $N(s_i)$ is the number of times the parent node has been visited, $N(s_i, a)$ is the number of times the child has been visited and $V\left(s_i, a\right)$ is the number of winning games derived from the child. c used to balance exploitation and exploration, and we take $c = \sqrt{2}$ in implementation.

Upon selecting an unexplored leaf node, we then proceed to expand the tree by randomly adding a not visited successor state of selected node to the tree.

Simulation Simulations are conducted from the newly added nodes to estimate their potential outcomes. Specifically, black and white sides follow the same default policy until the game ends. We first employ the random policy as default policy:

$$\pi_{\text{random}}\ \left(s_i\right) = \text{rand}\left(\mathcal{A}\left(s_i\right)\right), \forall i$$

Backpropagation In the backpropagation phase, following each simulation, the outcome of the game is used to update the search tree. Starting from the leaf node identified during the selection phase, the reward information is propagated back towards the root node. This process involves updating the rewards and visit counts for each node along the path. Specifically, in our implementation, the value function for a node, $V(s_i)$, is updated as $V(s_i) = V(s_i) + 1$ if the simulation results in a win, and $V(s_i) = V(s_i) - 1$ in case of a loss. The decision for the next move is then based on selecting the child node under the root with the highest number of visits, reflecting the most promising action inferred from the simulations.

	Offensive (X) and its winning rate					
		RAND	Greedy	Minimax	DQN	MCTS
Defensive (O)	RAND	/	41.2% ± 8.2%	71.0% ± 15.1%	80.0% ± 11.8%	87.0% ± 6.4%
	Greedy	45.4% ± 7.2%	/	85.0% ± 6.7%	78.0% ± 14.7%	88.0% ± 6.0%
	Minimax	25.2% ± 7.7%	17.0% ± 13.5%	/	100.0% ± 0.0%	79.0% ± 10.4%
	DQN	21.8% ± 5.6%	26.0% ± 11.1%	0.0% ± 0.0%	/	61.0% ± 17.6%
	MCTS	13.0% ± 9.0%	13.0% ± 10.0%	24.0% ± 14.96%	33.2% ± 12.8%	/

Table 2: The winning rate for each agent being **Offensive** (Black/X; takes turn firstly).

	Defensive (O) and its winning rate					
		RAND	Greedy	Minimax	DQN	MCTS
Offensive (X)	RAND	/	54.6% ± 7.2%	74.8% ± 7.7%	78.2% ± 5.6%	87.0% ± 9.0%
	Greedy	58.8% ± 8.2%	/	83.0% ± 13.5%	72.0% ± 11.1%	87.0% ± 10.0%
	Minimax	29.0% ± 15.1%	15.0% ± 6.7%	/	100.0% ± 0.0%	76.0% ± 14.96%
	DQN	20.0% ± 11.8%	22.0% ± 14.7%	0.0% ± 0.0%	/	66.8% ± 12.8%
	MCTS	13.0% ± 6.4%	12.0% ± 6.0%	21.0% ± 10.4%	39.0% ± 17.6%	/

Table 3: The winning rate for each agent being **Defensive** (White/O; takes turn secondly).

Adaptive iteration Our analysis revealed distinct requirements for computational resources across different phases of the game. In the opening phase, due to the vast number of potential subsequent states, simulating accurate outcomes is challenging, but these early moves are less likely to be immediately critical. In the mid-game, however, each decision becomes increasingly vital, necessitating a higher number of iterations for precise evaluation. Conversely, the endgame, characterized by fewer possible future states, allows for a reduction in the number of iterations without significant loss of accuracy. To accommodate these varying needs, we employed a Gaussian function to dynamically adjust the number of iterations according to the game phase. This adaptive approach ensures a more efficient allocation of computational resources, tailoring the depth of search to the strategic importance of each phase.

4 Evaluation

We established two benchmark agents: a random agent, *RAND*, which selects moves randomly from the available actions, and a greedy agent, *Greedy*, that opts for moves flipping the maximum number of pieces. Additionally, we developed three learning agents—*Minimax*, *DQN*, and *MCTS*—each corresponding to one of our proposed algorithms.

To comprehensively assess the performance of both the learning and fixed agents, we arranged multiple games wherein each agent alternated between offensive (black; taking the first turn) and defensive (white; taking the second turn) roles. Table 2 displays the offensive winning rates for each agent against others, and Table 3 details their defensive winning rates. Each entry in these tables is presented as *mean ± standard deviation*.

The results lead to several key observations:

Comparison with Baselines All three learning agents significantly outperformed both the Random and Greedy baselines, exhibiting at least a 20% lead. This outcome substantiates the efficacy of our proposed methods.

Intra-group Comparison Among our proposed methods, MCTS achieved the highest performance, followed by DQN and Minimax. Notably, the DQN agent won every game against the Minimax agent during testing. This may be attributed to the limited depth of the search tree and the simplicity of the evaluation function used in Minimax, which will be further discussed in the Discussion section.

Impact of Playing Order Our findings reveal that the performance of each algorithm is influenced by the playing order. This variation is particularly pronounced in the case of DQN, where the agent as the Offensive player exhibited a higher winning rate compared to its Defensive role. We hypothesize that this phenomenon is inherent to the nature of Othello itself.

5 Discussion and Future Work

5.1 Minimax Search with Alpha-Beta Pruning

The relative ineffectiveness of the minimax algorithm in our comparative study can be primarily attributed to two factors. First, the simplicity of the evaluation function is a significant limitation. This function's inability to accurately represent the strategic advantage of a given board state in terms of scores restricts the agent's capability to identify and execute the most advantageous moves. Essentially, the evaluation function does not adequately capture the game's complexities, leading to suboptimal decision-making by the minimax agent.

Second, the depth of the minimax search tree was intentionally constrained to a depth of two layers, a decision motivated by considerations of computational efficiency. While this restriction ensures faster processing times, it substantially limits the algorithm's foresight. The minimax algorithm's predictive power is directly proportional to the depth of its search tree; thus, a shallow tree hampers its ability to foresee and plan for future game states. Consequently, this limitation in 'thinking ahead' further impedes the algorithm's performance, particularly when compared to the more robust and far-sighted strategies employed by the other agents.

5.2 Deep Q Network

5.2.1 Reward System

In this project, we implemented a reward system based on both the final score and the position of each piece. This approach, while comprehensive, introduced additional complexity to the learning process. We considered a simpler alternative: awarding a +1 reward for a win, 0 for a tie, and -1 for a loss. This concept, inspired by strategies employed in Intelligent Go Agents, focuses on the end result rather than specific in-game scenarios. Such a streamlined reward system could potentially lead to the development of unexpected but novel, effective strategies and improve the overall winning rate.

5.2.2 Effect of Training Epochs

Our experimentation with the DQN agent involved training across different epochs. Initially, the agent trained for 5000 episodes against the greedy agent, achieving a 78.0% winning rate in the offensive position and 72.0% in the defensive. However, extending training to 10000 and 15000 episodes resulted in diminishing returns, suggesting a possible entrapment in a local minimum, limiting the model's generalizability.

Further trials involved training the DQN agent against itself. An agent trained for 460k episodes significantly outperformed one trained for 50k episodes. Yet, against the greedy agent, the maximum winning rate plateaued at approximately 40%. These outcomes highlight the benefits and limitations of extended training epochs, indicating the potential for improved performance but also emphasizing the challenge of slow training speeds."

5.2.3 Advanced Reinforcement Learning Approaches

DQN, a value-based method in reinforcement learning, excels in learning the value function and deriving policy indirectly. To enhance our approach, integrating policy-based methods, which directly learn the target policy, appears promising.

One such method is the *Actor-Critic* algorithm[6] which involves two networks: the policy network (Actor), which interacts with the environment and refines its policy using policy gradients, and the value network (Critic), which develops a value function based on the Actor's experiences. This dual-network system supports the Actor in evaluating actions and refining its policy, potentially enhancing the DQN's capability to learn more effective strategies and explore a broader range of possibilities.

5.3 Monte Carlo Tree Search

5.3.1 Increasing the number of iteration

Increasing the number of iterations allows the search algorithm to explore a larger portion of the game tree, resulting in more informed decision-making and precise estimates of the value of different moves, as illustrated in Figure 3. However, this also incurs higher computational costs and longer search times. Therefore, a balance must be struck between computational resources and the quality of the solutions obtained.

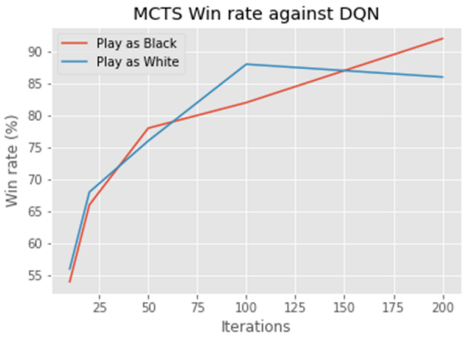

Figure 3: Increase the number of iteraion

5.3.2 Add Opening Book

In the opening phase, to compensate for the expansive nature of the search tree, we initially reduced the number of iterations. By integrating classic openings into our strategy, we can virtually eliminate computational efforts during the initial moves, ensuring not at a disadvantage at the beginning.

5.3.3 Change Default Policy

Besides employing a random default policy, we also experimented with greedy and DQN-based policies. These are defined as follows:

$$\pi_{\text{greedy}}\ (s_i) = \text{greedy}\left(\mathcal{A}\left(s_i\right)\right), \forall i$$
$$\pi_{\text{DQN}}\ (s_i) = \text{DQN}\left(\mathcal{A}\left(s_i\right)\right), \forall i$$

Pairwise comparisons of these default policies yielded the results shown in Table 4.

Duel Condition	Win Rate (left)
π_{random} VS π_{greedy}	60%
π_{random} VS π_{DQN}	36%
π_{greedy} VS π_{DQN}	52%

Table 4: Default Policy Comparation

The data indicates that the performance ranking of default policies is $\pi_{\text{DQN}} > \pi_{\text{greedy}} > \pi_{\text{random}}$. This demonstrates the effectiveness of selecting an appropriate default policy.

5.3.4 Acceleration

To enhance efficiency, one strategy We will consider trying is to extract additional information from the default policy to update the values of nodes in the tree. Typically, only the nodes visited by the tree policy are updated. However, by applying techniques like AMAF (All Moves As First)[7], we can update the values of any nodes encountered during self-play, thereby enriching the tree's information base.

6 Conclusion

In this study, we firstly conducted a literature review, leading to the proposal of three distinct methods for training an agent to play Othello. These methods are: Minimax Search with Alpha-Beta Pruning, Deep Q Network, and Monte Carlo Tree Search. Each method demonstrated efficacy in addressing the Othello challenge and exhibited potential applicability to more complex scenarios. However, it is important to recognize the inherent limitations of these approaches. We acknowledge these constraints and outline plans for future improvements.

References

[1] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016.

[2] Michiel van der Ree and Marco Wiering. Reinforcement learning in the game of othello: Learning against a fixed opponent and learning from self-play. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 108–115, 2013.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015.

[4] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016.

[5] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: a review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, July 2022.

[6] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[7] David P Helmbold and Aleatha Parker-Wood. All-moves-as-first heuristics in monte-carlo go. In *IC-AI*, pages 605–610, 2009.

Appendix

Referenced External Libraries The core algorithms are mostly implemented by ourselves. We refered to these external libraries as listed:

- *PyTorch*[3], for building Neural Network;

[3] https://pytorch.org/

- *Hands-on Reinforcement Learning*[4], which provides a clear explanation of different Reinforcement Learning algorithms;

Division of Labour

- Minimax: Yiang Ju
- DQN: Shengxin Li, Zhuo Diao
- MCTS: Zaizhou Yang, Guanjie Huang
- Presentation and Report: All of us, proudly!

[4]Hands-on Reinforcement Learning