Evaluating AlphaZero in a Strongly Solved Game

by

Bigyan Karki

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Chinese Checkers, a traditional game played on a star-shaped board by 2-6 players, has been a domain for game AI research and has been strongly solved up to a 6×6 board with 6 pieces per player in a two-player game. In this work, we apply the AlphaZero algorithm, known for its success in perfect information, two-player deterministic games like Chess, Shogi, and Go, to Chinese Checkers. Our implementation involved training a custom AlphaZero agent on a 4×4 board with 3 pieces and a 5×5 board with 6 pieces. The primary contributions include a parallelized version of AlphaZero, an evaluation of AlphaZero in perfect information games, exploration of the learning data structure, assessment of learning accuracy on training data, and measurements of generalization on states both similar and random to those observed. While AlphaZero agents have achieved superhuman performance in certain domains, recent analysis on KataGo revealed vulnerabilities to certain strategies that human players would not fall for, indicating potential performance gaps in AlphaZero. Our work studies the nature of training and evaluating agents in simplified variants of Chinese Checkers, identifying a decrease in the accuracy of AlphaZero's learned policy on states outside the training set. Even in smaller variants of Chinese Checkers, adversarial policies were able to leverage these shortcomings, leading to policy mistakes by AlphaZero agent during play. We propose a combination of supervised and self-play training to alleviate these exploitations, aiming to enhance the AlphaZero agent's resilience against adversarial strategies.

This thesis is dedicated to my parents, and brother.

Acknowledgements

I would like to extend my heartfelt gratitude to my supervisor, Professor Nathan Sturtevant, for introducing me to this research area. His advice, guidance, careful editing, and patience during the preparation of this thesis have been invaluable.

I also owe a debt of gratitude to Zaheen Ahmad, a PhD candidate at the University of Alberta, for his invaluable advice and guidance throughout my research. Zaheen worked on building the single thread self play implementation of AlphaZero in Chinese Checkers. I extended his work to parallel thread self play, along with all the experiments done in this thesis.

I am grateful to Antonie Bodley for her assistance in editing my thesis. As a writing consultant hired by my supervisor, her expertise was invaluable in refining the grammar and structure of the manuscript. Although she did not have a background in Computer Science, her contributions significantly enhanced the clarity and readability of my work. I would also like to acknowledge the software tools that aided me in correcting grammar and formatting: Grammarly and ChatGPT.

Finally, my sincere thanks go to the University of Alberta, NSERC, CIFAR, and Amii for their generous funding of my master's program and research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

John McCarthy defined the term artificial intelligence as "the science and engineering of making intelligent machines" [22]. These machines are now commonplace, evident in computers, mobile devices, smartwatches, and beyond. Today's progress in AI, attributed to advanced algorithms and hardware evolution, originated from game-based experiments.

1.1 Games in Artificial Intelligence (AI)

In AI research, games are referred as the "Drosophila of AI" [21]. Analogous to how the fruit fly, Drosophila, aids genetic research due to its study-friendly attributes, games act as a tangible yet complex framework for AI study. Games allow the calibration of AI capabilities, from basic tasks to intricate problem-solving.

Claude Shannon's 1950 work on Chess [25] laid the groundwork for using games to study computational logic and pattern recognition. The finite possibilities in Chess gave researchers a platform to prototype algorithms that attempted to mimic human thinking. This research trajectory extended to other games such as Checkers [24], Connect Four [1], and Go [26], with each presenting distinct challenges and necessitating different algorithmic strategies.

AlphaGo's 2016 success was not merely a high point in AI's game-playing endeavors. It signified strides made in machine learning and neural networks [26]. Given its extensive possibilities, the game of Go was was perceived as a

challenge for computers. Yet, AlphaGo's win against Lee Sedol, the Go world champion, showed that current AI systems can manage high complexities by merging computational power and strategic subtleties [26].

Games serve dual purposes in AI research: as platforms to test algorithms and as tools to probe the depth of the games themselves. A notable illustration is AlphaGo's Move 37 in its second match against the world champion. During the game, on move 37, AlphaGo (playing as white) placed a stone on the fifth line, which is commonly referred to as a "shoulder hit" on a black stone [19]. This was an unusual move, especially at that point in the game. Traditionally, professional Go players would expect a move on the third or fourth lines, which are closer to the edge. The fifth line move was viewed as aggressive and non-traditional for the particular board situation. This unexpected move, initially deemed suboptimal by human standards, ultimately secured its victory. It highlighted both the boundaries of human understanding of the game and the potential insights AI can bring.

This dynamic reveals a mutual influence: games refine AI capabilities, and in turn, AI innovations influence game design and understanding. This interplay fosters progress in both domains.

One of the games that we are interested in is Chinese Checkers.

1.2 Chinese Checkers

Chinese Checkers, a classic board game widely known for its distinctive star-shaped board, dates back to the late 19th century. It is a derivative of the game Halma, which was invented around 1883-1884 [5]. Unlike Halma, which is played on a square board, Chinese Checkers is played on a star-shaped board and can accommodate between 2 to 6 players. The primary objective of the game is to be the first to move all of your pieces across the board into the opposing player's starting position, a task that requires careful strategy and planning. A commonly used Chinese Checkers board is illustrated in Figure 1.1.

Our interest in Chinese Checkers stems from its classification as a two-

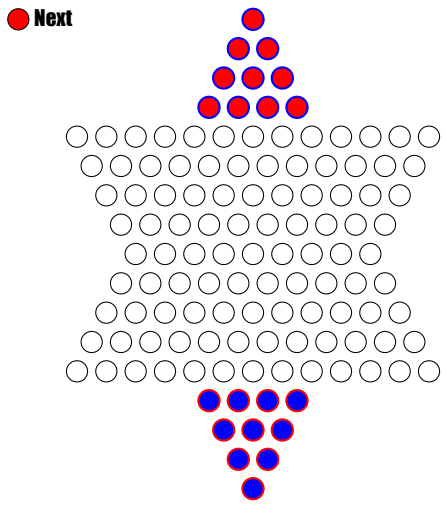


Figure 1.1: Standard 9 x 9 Chinese Checkers Board

player perfect information game. This category of games, which includes renowned games such as Connect-Four [1], Go [26], Checkers [24], and Hex [13], has been the focus of extensive research and study over the years. In these games, all information is available to all players, there is no chance involved, and players take turns, making them good testbed for strategic analysis and algorithmic solutions.

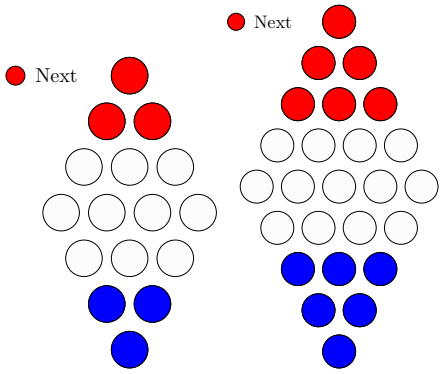


Figure 1.2: The figure on the left is 4×4 board size with 3 pieces, and the figure on the right is 5×5 board size with 6 pieces. There are two players in each marked by red and blue colors. The player whose turn it is to move next is indicated in the top-left corner of each board.

Chinese Checkers, in particular, presents a compelling case for study. A typical board size for the game boasts 1.73×10^{24} states, suggesting that it could potentially be weakly solvable [28]. However, despite this theoretical

possibility, the task of devising a robust proof strategy has proven to be a formidable challenge. This gap in knowledge and understanding forms the first part of our motivation: we seek to contribute to the ongoing tradition of solving perfect information games but aim to do so in a more robust manner for Chinese Checkers. Specifically, our aim is to study smaller solved versions of the game, thereby deepening our understanding of the game's nature and complexity. In this thesis, we looked at 4×4 and 5×5 board sizes with only two players. A starting state of each of the board sizes can be seen in Figure 1.2.

1.3 AlphaZero

The advent of machine learning and, more specifically, the development of the AlphaZero algorithm, have opened up new possibilities for studying and mastering deterministic two-player games with perfect information. AlphaZero's unique approach to learning and decision-making, which combines deep neural networks with Monte Carlo Tree Search, has shown remarkable success in games such as Chess, Shogi, and Go [27]. Given that Chinese Checkers also falls into the category of deterministic two-player games with perfect information, it is reasonable to posit that the AlphaZero approach could be successfully applied to this game. This forms the second part of our motivation: we aim to adapt and test the AlphaZero algorithm on Chinese Checkers, thereby contributing to our understanding of the game while simultaneously providing valuable insights into the effectiveness and adaptability of the AlphaZero learning approach.

1.4 Problem Statement and Research Questions

While Chinese Checkers has been recognized as a perfect information game and potentially weakly solvable, there is a lack of robust proof strategies and practical solutions for playing the game optimally. Moreover, the application of advanced machine learning algorithms, such as AlphaZero, to Chinese Check-

ers has yet to be extensively explored. Therefore, the central problem our research intends to address is to develop a custom-built AlphaZero algorithm and examine its performance and learning process.

The primary research questions guiding our study are:

1. How can AlphaZero be customized and optimized for Chinese Checkers?
2. What is the value of ground truth data in training and evaluating the AlphaZero model for Chinese Checkers?
3. How well does the AlphaZero model generalize to unseen boards or data in Chinese Checkers?
4. How robust is the AlphaZero model against adversarial attacks in the context of Chinese Checkers?

1.4.1 AlphaZero Customization for Chinese Checkers

To address the questions above, our research aims to create a version of AlphaZero that is designed and optimized for Chinese Checkers. We hypothesize that this custom-built version will outperform generic versions of AlphaZero when applied to Chinese Checkers, particularly in terms of processing speed and game performance.

1.4.2 The Value of Ground Truth Data

Ground truth data, which refers to information provided by direct observation as opposed to inferred from a model, is invaluable in the context of our research. 4×4 and 5×5 Chinese Checkers board sizes are strongly solved [28]; as such we have access to a perfect solver. It serves as the benchmark against which our model's predictions are evaluated. In the context of Chinese Checkers, ground truth data include the outcomes of games or optimal moves for a given state determined by existing solvers. The accuracy, reliability, and detail of this data are critical in training and evaluating the performance of our AlphaZero model.

1.4.3 Generalization to Unseen Data

An essential aspect of our research is to assess how well our trained AlphaZero agent generalizes to unseen boards or data in Chinese Checkers. This involves testing the model on new scenarios, or states, that it has not encountered during training. The model's ability to generalize is key to its practical applicability, as it needs to effectively handle a wide range of game scenarios to play Chinese Checkers successfully.

1.4.4 Robustness Against Adversarial Attacks

Finally, we will examine the robustness of our AlphaZero model against adversarial attacks. In the context of game playing, an adversarial attack could involve manipulating the game state or the model's perception of the game state to mislead the model or degrade its performance. Although self play algorithms like AlphaZero have been widely studied, far less work has been done to quantify strengths and weaknesses in AlphaZero learning. Perhaps unsurprisingly, recent work has shown that an agent reliably learn to exploit an AlphaZero agents [31]. But, while this analysis shows that AlphaZero can be exploited, it does not characterize the nature of the original learning, where the weaknesses arise, and whether there is a deeper explanation for the systematic weaknesses in play. We will measure where the weakness arise, exploit those weaknesses, and devise ways to patch them.

1.5 The Importance of this Research

Our research is designed to assess the performance of AlphaZero in the context of perfect information games, with Chinese Checkers serving as the central game environment. This study will not only identify AlphaZero's strengths and vulnerabilities but also enhance our comprehension of Chinese Checkers as a strategic game, thereby enriching the existing literature on perfect information games.

Our work involves creating different versions of AlphaZero that are specifically designed and optimized for Chinese Checkers. This endeavor is expected

to yield a comprehensive analysis of the game in 4×4 and 5×5 board sizes.

We begin Chapter 2 with background concepts and works related to Chinese Checkers and AlphaZero. This is followed by Chapter 3, with a description of the approach taken in the research and implementation details of AlphaZero in Chinese checkers.

In Chapter 4, we provide the results and analysis of our work. Finally, we conclude in Chapter 5 with a summary of our work and future directions of our research.

Chapter 2

Background and Related Work

In this chapter we discuss some of the background materials that are important to our work. Since game research mainly involves solving a game, in the first section, we discuss what it means to solve a game. In the second section, we introduce classical game-solving strategies like MiniMax, Alpha-Beta pruning and Monte Carlo Tree search. In the third section, we discuss deep learning and reinforcement learning algorithms. Finally, in the last section, we delve into the rules of Chinese Checkers and solver-specific definitions for win, draw, and illegal states.

2.1 Solved Games

Perfect play is the behavior or strategy of a player that leads to the best possible outcome for that player, regardless of the opponent's response. However, a perfect play is conditioned on the fact that the game itself is solved. Allis et al. [1] divides solved games into three categories:

1. **Ultra-weakly solved**: In ultra-weakly solved game, game theoretic value of initial position is known. In other words, given a perfect play, what is the best outcome the first player can achieve from the initial position of the game (win, loss or draw). However, the strategy for achieving such a goal is not required to be specified. For example, Hex on any $N \times N$ board is ultra-weakly solved by the strategy stealing argument.

2. **Weakly solved**: A weakly solved game is one where, for the initial position of the game, a strategy has been determined to obtain at least the game-theoretical value of the game for both players. Checkers is one of the weakly solved games. In Checkers, it has shown that there is an strategy that can lead to draw under optimal play [24].

3. **Strongly solved**: A strongly solved game requires a strategy to be identified to obtain the game-theoretic value for all position of the board. In a strongly solved game, a player should be able to play optimally even if mistakes has been made in the past. Chinese Checkers up to 6×6 board sizes and with six pieces has been strongly solved [28].

2.2 Classic Game-Solving Strategies

Classical game-solving algorithms form the bedrock of AI's success in many traditional board games. These strategies rely on exploring game trees, evaluating potential moves, and predicting opponent responses. They provided a structured way to make decisions in deterministic environments, setting the stage for more advanced techniques in modern AI gaming.

2.2.1 Minimax

The Minimax algorithm is a decision-making algorithm that is used for decision making in game theory and artificial intelligence [23]. The algorithm assumes a zero-sum game, where one player's gain is another player's loss. It is used to determine the optimal move for a player, assuming that the opponent is also playing optimally.

Figure 2.1 illustrates a directed acyclic graph that represents possible lines of play of a particular game. In Minimax, two participants, MAX and MIN, sequentially alternate in the multi-layered graph. Every node signifies a state of the game, while each connection between the nodes symbolizes a move. The top layer of the graph comprises a single node owned by MAX, signifying the game's initiation.

Nodes situated at the termini of the graph paths are referred to as leaf nodes. Each leaf node is assigned a utility score for the MAX player. The scoring calculation for non-leaf nodes depends on whether the node is classified as a MAX node or a MIN node. For nodes under the MAX category, the score is the highest score amongst its direct descendants. Conversely, for nodes classified as MIN, the score is governed by the lowest score from its direct descendants. This scoring procedure is formally expressed in Equation 2.1.

$$v(s) = \begin{cases} \text{eval}(s) & \text{if } s \text{ is a leaf} \\ \max_{s'} v(s') & \text{if } s \text{ is a MAX node} \\ \min_{s'} v(s') & \text{if } s \text{ is a MIN node} \end{cases} \tag{2.1}$$

In Figure 2.1, the optimal action in the graph for the maximizing player is to move left towards 3, followed by left again, and then right. However, it's crucial to note that in the context of Minimax, the objectives of the two participants are fundamentally opposed. The maximizing player (MAX) seeks to achieve the highest score, while the minimizing player (MIN) aims to minimize the score. Therefore, optimal actions for the min player would be those that lead to the lowest possible score. The optimal strategy sequence, which is known as the principal variation, is determined based on the assumption that both players evaluate the leaf nodes of the game tree accurately and make decisions that optimally advance their respective objectives—maximization for the max player and minimization for the min player.

2.2.2 Alpha-Beta pruning

Alpha-beta pruning is a search strategy designed to reduce the number of nodes the minimax algorithm must examine in its search tree [23]. In the best case, the exponent of the minimax game tree complexity can be cut in half.

In the alpha-beta pruning procedure, two values known as alpha and beta are used. Alpha signifies the lowest guaranteed score for the player maximizing their score, while beta represents the highest guaranteed score for the player minimizing their score. Initially, alpha is set to negative infinity and beta to positive infinity.

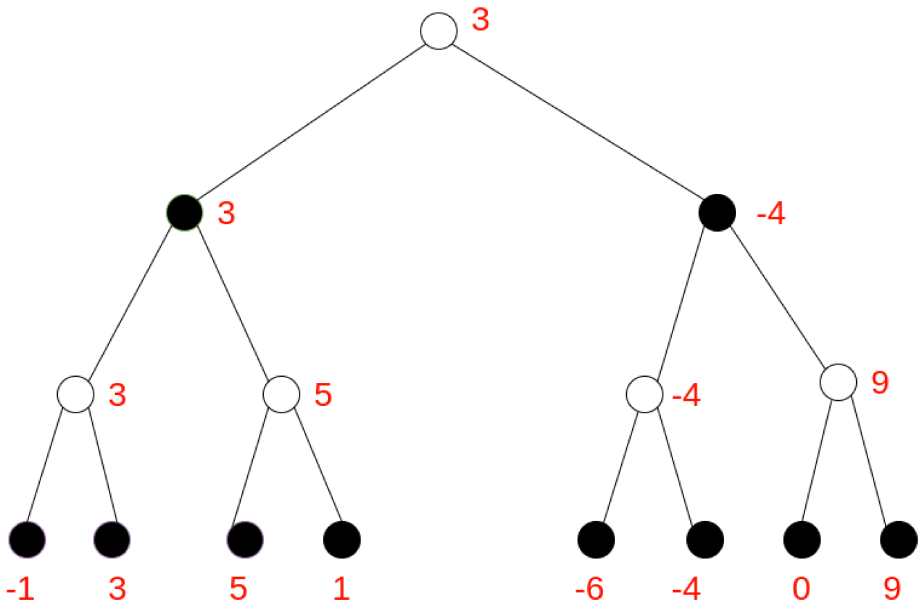


Figure 2.1: The directed acyclic graph (edges are directed downwards) of a game. Each node corresponds to a state and is labeled with that state's minimax value score for player MAX. A white node represents the MAX player, and a black represents the MIN player.

In instances where the beta value (corresponding to the minimizing player) drops below the alpha value (corresponding to the maximizing player), the need to evaluate the remaining branches is eliminated.

When alpha value for the maximizing player exceeds beta value for the minimizing player, evaluating the remaining branches becomes unnecessary since alpha will either remain unchanged or increase further. This is due to the understanding that the alpha value will either increase further or remain unchanged.

In Figure 2.2, the pruned branches, denoted by cross marks, are rendered unnecessary and consequently excluded from further evaluation.

In the case of the 'E' node (representing a maximizing player), its second child, node 'K', is pruned. This occurs because the parent node 'B', a minimizing player, possesses a value of 3. Upon evaluating the 'J' child node of 'E', it becomes apparent that the value of 'E' will be equal to or greater than 5. Since this value exceeds the beta value of node 'B', the subsequent branch is pruned.

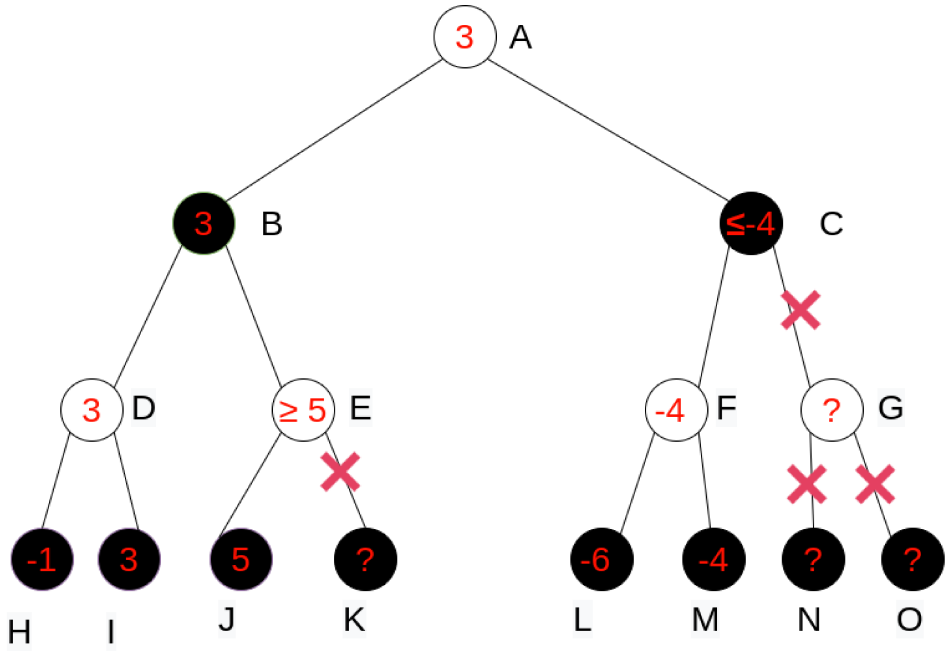


Figure 2.2: Alpha-Beta pruning in action. The pruned branches are denoted by cross marks and are not evaluated.

The utility of alpha-beta pruning lies in its capacity to bypass unnecessary computations, sometimes pruning leaves or entire sub-trees. This optimization saves time, as it refrains from calculating positions that do not influence the ultimate outcome. However, the efficacy of alpha-beta pruning depends on the sequence in which moves are ordered. There can be two cases:

1. **Worst case**: In the worst case alpha-beta works like the minimax algorithm without any pruning. When this occurs, the algorithm's efficiency diminishes, as it fails to prune and expands additional time due to the alpha-beta checks. Typically, in such cases, the most promising move is found on the tree's right side. At worst, the time complexity can be $O(b^m)$, where b is the branching factor, and m is the maximum depth of the tree, representing how many moves into the future the algorithm evaluates [23].

2. **Best case**: Best case is when extensive pruning is executed in the tree, and the best moves are located on the tree's left side. Given that the algorithm employs Depth-First Search (DFS), it first explores the tree's

left side. As a result, it can explore twice as deeply as the minimax algorithm within the same time frame. The time complexity in this optimal case is $O(b^{m/2})$ [23].

Therefore, it is advantageous to order the nodes in the tree to check the best node first. An example is in a game of Chess where a pawn capturing a piece is often a beneficial move and should be explored as a priority [23].

2.2.3 Monte Carlo Tree Search (MCTS)

Another widely used algorithm in game theory is Monte Carlo tree search [4]. MCTS has made impressive progress in Go [7], Chess, Shogi [27], and Hex [13]. It is a heuristic search algorithm utilized primarily in problems characterized by finite and discrete combinatorial spaces.

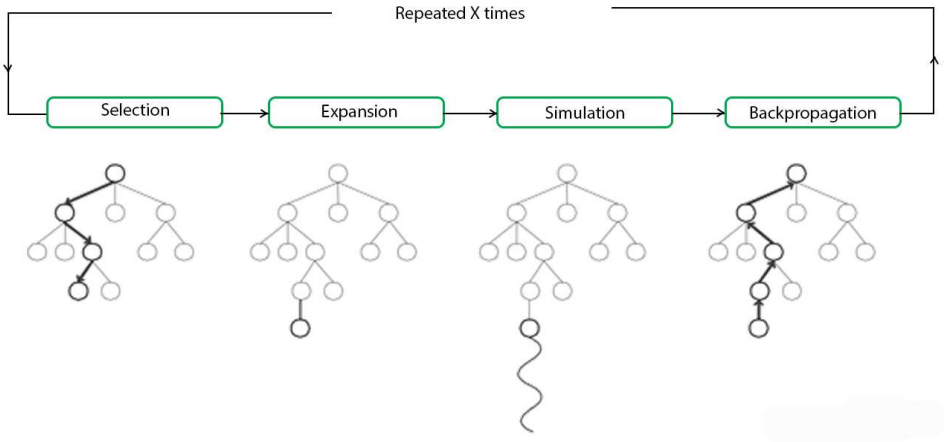


Figure 2.3: Monte Carlo Tree Search has four steps: action selection, state expansion, game simulation and value back-propagation. The step is repeated until a constraint is reached.

The MCTS algorithm revolves around four key phases: selection, expansion, simulation, and backpropagation.

1. **Selection**: The process begins at the root node and traverses the tree based on a policy, such as UCT [16], until it reaches a leaf node of the search tree.

2. **Expansion**: Upon reaching a leaf node, the algorithm extends the search tree by generating one (or more) child nodes unless the leaf node is a terminal game state.

3. **Simulation**: Following the creation of the new node, the algorithm conducts a simulated playout under a certain policy (which could be random) until it reaches a terminal state, yielding a resulting value.

4. **Backpropagation**: The simulated result is subsequently propagated back up the tree, updating the node statistics, such as visit count and cumulative value, from the newly expanded node to the root node.

The algorithm iteratively repeats these four steps for a specified computational budget. The budget could be a time limit or number of repetitions. Once the budget is exhausted, it selects the move leading to the most visited node from the root.

Monte Carlo Tree Search (MCTS) algorithm has an inherent mechanism to strike a balance between exploration and exploitation. Exploration, which involves the search for new and potentially advantageous moves, is implemented through the expansion and simulation stages of the algorithm. In contrast, exploitation, moves with the current highest score, is executed in the selection stage of the algorithm.

This balance is achieved using the Upper Confidence Bound 1 (UCB 1) applied to Trees (UCT) policy [16], a strategy that helps the algorithm to decide whether to explore a less-known but potentially beneficial path (expansion and simulation) or to exploit a path that has already been identified as providing a good return (selection). This aspect of the MCTS algorithm makes it particularly well-suited to decision-making processes in game playing. The UCT formula to select the action is in Equation 2.2.

$$UCT = \frac{w_i}{n_i} + C\sqrt{\frac{\ln N_i}{n_i}} \tag{2.2}$$

where,

w_i is the number of wins after the i-th move,

n_i is the number of simulations after the i-th move,

N_i is the total number of simulations after all moves,

C is the exploration parameter.

The UCT policy is designed to explore less-visited nodes to search for high-reward actions. This is crucial for ensuring that most of the states in a search tree are visited and that the algorithm doesn't prematurely converge on a suboptimal solution.

MCTS works with little or no domain knowledge [4]. However, it has been shown that its effectiveness can often be significantly enhanced through the incorporation of domain knowledge, as well as through improved child node selection and leaf node evaluation strategies [9].

Many improvements have been made to MCTS. One of the significant improvements was by Gelly et al. [10]. They introduced the Rapid Action Value Estimation (RAVE) heuristic, which offers notable advantages in terms of accelerated statistics accumulation and improved performance. Particularly in games where the values of moves exhibit minimal dependence on their order of play, the RAVE heuristic proves highly effective. It can also be effectively parallelized by employing multiple threads or processes to execute it concurrently. Some of the techniques are listed below:

- **Leaf parallelization**: This approach involves executing multiple playouts in parallel from a single leaf node within the game tree [6]
- **Root parallelization**: In this method, independent game trees are constructed in parallel, and the move selection is based on the branches at the root level across these trees [6].
- **Tree parallelization**: This technique entails the parallel construction of the same game tree. To ensure data integrity during simultaneous writes, various synchronization mechanisms can be employed, such as a

single global mutex, multiple mutexes, or non-blocking synchronization [6].

2.3 Deep Learning and Reinforcement Learning in Games

In this section, we move away from search algorithms to learning algorithms. We briefly discuss Deep Learning and Reinforcement learning before studying how they have been used in Chinese Checkers.

2.3.1 Deep Learning

Deep learning uses neural networks to process data and create patterns used for decision making [11]. Deep learning models are composed of multiple layers of artificial neural networks, hence the term 'deep.' Each layer contributes to interpreting the data at different levels of abstraction [17], thereby enabling the model to process complex data. One particular Deep Learning architecture that we are most interested is Convolutional Neural Network (CNN) [20].

A CNN is composed of one or more convolutional layers, often followed by pooling (also known as subsampling or down-sampling) layers, fully connected layers, and normalization layers in various combinations. The convolutional layers are designed to automatically and adaptively learn spatial hierarchies of features, which makes CNNs very efficient for tasks like image classification.

The "convolutional" aspect of the CNN refers to the mathematical operation applied to the input data. This operation involves the use of a filter or kernel, a small matrix of weights, which is passed over the input data to produce what's known as a feature map or convolved feature. Figure 2.4 shows a simple convolution process. During the training process, these weights are learned so that the network can identify important features necessary for the task at hand. More importantly, the same kernel is applied to every patch.

Pooling layers serve to reduce the spatial dimensions (height and width, not depth) of the input volume. It decreases the computational complexity, controls overfitting, and makes the network invariant to small transformations,

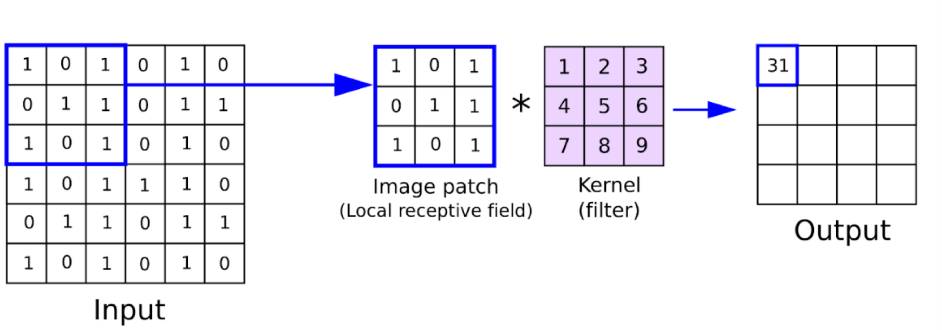


Figure 2.4: Here, a kernal of size 3×3 is applied across a patch of image matrix.

distortions, and translations in the input image [11].

Following the convolutional and pooling layers, the high-level reasoning in the neural network occurs in the fully connected layers. Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular neural networks, and their activations can hence be computed with a matrix multiplication followed by a bias offset.

Normalization layers can be used to improve the convergence, generalization ability, and robustness of the network. One common form of normalization is batch normalization.

He et al. [14] proposed Residual Net (ResNet) that uses shortcut connections to tackle some of the problems existing in CNNs, like gradient vanishing/exploding problems, and also making it possible to train large neural networks.

Most deep learning research is concerned with the following:

1. **Architecture**: How deep and wide a network should be? What and how should networks be stacked?

2. **Optimization**: What optimization techniques, such as Adam [15], or Stochastic Gradient Descent (SGD) [2] should be used?

3. **Hyper-parameter tuning**: What set of hyper-parameters, like learning rate and batch size, should be used?

2.3.2 Reinforcement Learning

Reinforcement learning (RL) is a subfield of machine learning where the learning happens when an agent interacts with an environment.

In reinforcement learning, the agent's environment is modeled by a Markov Decision Process (MDP). An MDP formalizes sequential decision making, where an action influences not just the immediate reward but also the subsequent state and future reward [30]. Mathematically, an MDP can be represented as a 5-tuple (S, A, P, r, μ_0), where S is a finite set of states, A is a finite set of possible actions, P is a transition probability distribution, r is a scalar value reward that an agent receives while transitioning from one state to another, and μ_0 is the agent's initial state distribution. In an MDP, at each time step, an agent is at state s, and chooses an action a according to some policy π. Selecting an action a transitions the agent to a new state s' and the agent receives a real valued scalar reward r. The policy refers to a mapping from state to action $\pi : S \to A$, suggesting which action to take in a given state. The goal of reinforcement learning is to find an optimal policy, one that maximizes the possible sum of rewards, which we generally refer to as return.

Contrary to search algorithms, which focus on investigating a limited segment of a space graph, the goal of RL is to find the exact value of each state based on learned experience. In MDPs, the basis for RL is the set of Bellman Equations [30]. The Bellman equation for the value of a state is defined as follows:

$$v_\pi(s) = \sum_a \pi(s,a) \sum_{s'} \Pr(s'|s,a)(r(s,a,s') + \gamma v_\pi(s')) \tag{2.3}$$

where:

- $v_\pi(s)$ represents the value of a state s under policy π.
- $\pi(s,a)$ represents the probability of taking action a in state s under policy π.
- $\Pr(s'|s,a)$ represents the probability of transitioning to state s' given that the current state is s and action a is taken.

- $r(s, a, s')$ represents the immediate reward received after transitioning to state s' from state s due to action a.
- γ is the discount factor which determines the present value of future rewards.
- $v_\pi(s')$ is the value of the new state s' under policy π.

The Bellman equation for action-value function is defined as:

$$q_\pi(s, a) = \sum_{s'} \Pr(s'|s, a)(r(s, a, s') + \gamma v_\pi(s')) \tag{2.4}$$

where:

- $q_\pi(s, a)$ represents the value of an action a in state s under policy π.

Both Bellman equations serve as the foundational elements for the process of policy evaluation, the objective of which is to accurately compute the value function associated with a designated policy π. The Optimal Bellman Equation is a recursive relation for the optimal policy:

$$v_*(s) = \max_a \sum_{s'} \Pr(s'|s, a)(r(s, a, s') + \gamma v_*(s')) \tag{2.5}$$

$$q_*(s, a) = \sum_{s'} \Pr(s'|s, a)(r(s, a, s') + \gamma \max_{a'} q_*(s', a')) \tag{2.6}$$

Two core approaches in reinforcement learning are value iteration and policy iteration.

Value Iteration is predicated upon the explicit storage of state values, with the iteratively updated values gradually converging to the optimal state values. More formally, Value Iteration methodically utilizes the Bellman Optimality Equation 2.5 to update the value function for each state in each iteration. As this iterative process progresses, the value function approaches the optimal value function v_*, and the policy derived from these optimal state values is the optimal policy.

Alternatively, Policy Iteration involves an intertwined process of policy evaluation and policy improvement. Policy Evaluation refers to the computation of the value function for a given policy, as represented by Equation 2.3, the Bellman Expectation Equation for state-value functions. Subsequent to this, the computed value function is utilized in the Policy Improvement step, indicated by Equation 2.5, the Bellman Optimality Equation for state-value functions. In this step, the policy is improved by making it greedy with respect to the evaluated value function. The entire evaluation and improvement process is repeated until the policy converges to the optimal policy.

Both these algorithms have been proven to converge on the optimal policy under certain conditions [30], and they represent key approaches in the toolkit of reinforcement learning methods.

2.4 Chinese Checkers

In this section, we discuss the rules of Chinese Checkers and the rules defined in the Chinese Checkers solver used in our work.

2.4.1 Rules of Chinese Checkers

On a player's turn in Chinese Checkers, they must move one piece. This movement can be a single step into an adjacent empty spot, a jump over one adjacent piece into an empty spot, or a series of multiple jumps. Figure 2.5 shows a sequence of steps starting from initial positions.

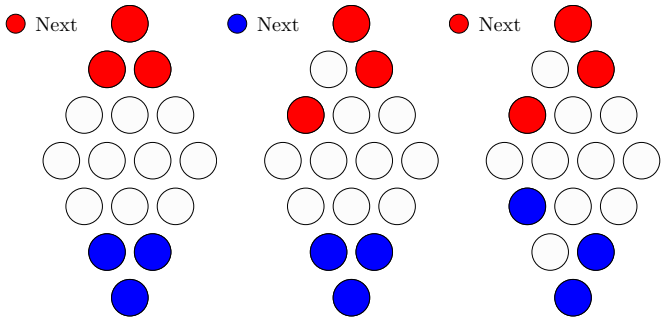


Figure 2.5: Sequence of adjacent moves starting from an initial position.

The second type of move is hops or jumps. These jumps can be over a

player's own pieces or those of other players. Each jump in a multi-jump move must jump over only one piece and land in an empty spot. A piece cannot leap over two or more consecutive pieces, but zig-zag jumps over multiple pieces are allowed. Figure 2.6 shows a single hop from the initial position. Figure 2.7 shows a sequence of hops for the red player.

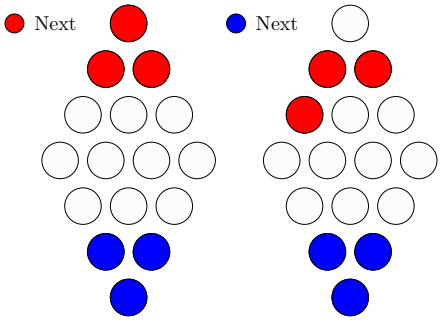


Figure 2.6: Single hop from an initial position.

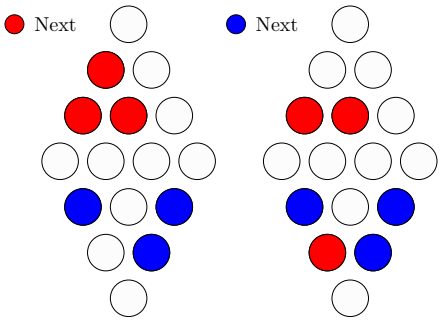


Figure 2.7: Sequence of hops from the red player from the position on the left.

Although the standard board is star-shaped, players are unable to move into the corners of the board, except for their designated start and goal corners. As such, the main gameplay focuses on the center diamond of the board, which can be represented as an m $\times$ m grid as in Figures 2.5, 2.6 and 2.7.

2.4.2 Chinese Checkers Solver

In the standard rules of Chinese Checkers, a player achieves victory by successfully moving all of their pieces into their designated goal area on the opposite side of the board. However, for comprehensive analysis and strategic examination, it is necessary to establish a more precise definition of the winning

conditions. Notably, it has been observed that a player can strategically leave one piece behind in their start area, effectively blocking their opponent from reaching their own goal area.

To handle such situations, Sturtevant proposes two definitions of rules for wins and illegal states to force an end to the game in his solver [28]. We use his solver in our work. As such, it is important to go over these solver specific definitions.

Win definition: *A state in Chinese Checkers is won for player n if player n's goal area is filled with pieces, and at least one of the pieces belongs to player n.* [28]

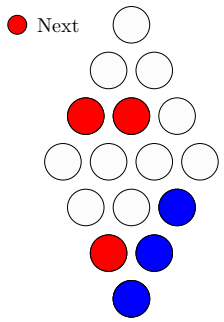


Figure 2.8: Winning position for red player under win definition.

According to this definition, if a player opts to keep some of their pieces in their goal area, the opposing player can strategically surround and navigate around those pieces to secure a victory in the game. However, it's worth noting that in certain instances, particularly in smaller versions of the game, this approach can lead to unintended outcomes, as depicted in Figure 2.8. Such shallow goal states are not found in larger boards. So, such shallow goal states are acceptable.

Still there are some conditions this definition cannot catch when there are six pieces on the board. As such, some board positions are declared illegal.

Illegal states: *(Part 1) A state in Chinese Checkers is illegal for player n if the winning condition is met for player n, and it is player n's turn to move.* [28]

Under this definition, if the winning state is already reached by player n, and it's player's n's turn to move, then it is an illegal state. One such example

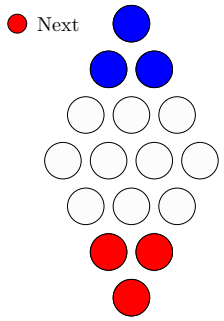


Figure 2.9: Illegal position in the game under part 1 definition.

is presented in 2.9. In the figure, the red player is already at the goal state, and it is red's turn to move. Moving the red piece anywhere else would be a suicidal move causing the other player to win the game. However, it has been shown that this rule is still inadequate and doesn't encompass all states marked illegal.

Illegal states: *(Part 2) A state in Chinese Checkers is illegal if there are one or more unoccupied locations in player n's goal area that are unreachable by player n due to another player's pieces.* [28]

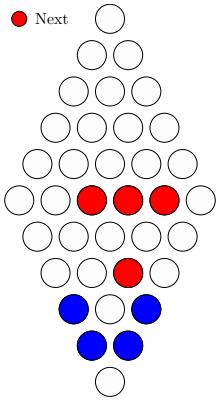


Figure 2.10: Illegal position in the game under part 2 definition

This definition describes board positions where a player blocks spots in the goal area by leaving them empty and surrounding them with their pieces, preventing the opponent from reaching the goal area. In Figure 2.10, the blue player surrounds the bottom tip with its own pieces, and the red player can never get to the bottom tip. Such types of illegal states are seen in larger board sizes with larger numbers of pieces.

Draws: *A game of Chinese Checkers is considered a draw if any board state is repeated during play.* [28]
This rule states a game is a draw if the same board state is repeated during the game. While Sturtevant argues [28] that one repetition of the board state suffices to declare a game draw, we permit six repetitions during training before declaring it so.

2.5 Why is Chinese Checkers interesting?

Our primary focus is on 4×4 and 5×5 board sizes. While a 6×6 board with 2 trillion states have been strongly solved, we used smaller board sizes to allow extensive experimentation in reasonable time frame with available hardware resources. For 4×4 board with 3 pieces, there are 320,320 legal states and for 5×5 board with 6 pieces, there are 9,610,154,400 states. The number of legal board position for the board size can be calculated by using formula below:

$$\binom{n}{k} \times \binom{n-k}{k} \times 2 \tag{2.7}$$

where, n represents the number of all board positions, k represents the number of pieces available for each player. The term $\binom{n}{k}$ represents the number of ways the first player can place their k pieces on n positions on the board. The second term $\binom{n-k}{k}$ calculates the number of ways the second player can place their k pieces on the remaining $n-k$ positions. The whole equation is multiplied by two to account for two possible sequences of play. Either the first or the second player could be the one to place their pieces on the board first.

Despite the smaller size, both boards are important because there are still strategically interesting choices to be made. In the states observed during training in the 5×5 board game, there were in average 12.79 moves to choose from, of which 2.98 were winning moves, 9.70 were losing moves, and 0.14 were draw moves. In other words, most of the move choices are losing moves, and it would be interesting and difficult for an agent to choose winning moves every single time. This would be true for larger board sizes as well. Hence, training

and evaluating agents in smaller board sizes are meaningful.

In this chapter, we discussed some of the background works related to our thesis. In the first section, we discussed Minimax algorithm and an optimization to it in the form of Alpha-Beta pruning. We also introduced the inner working of MCTS tree, which is core to the success of AlphaZero. In Section 2.3, we discussed the use of learning algorithms like deep learning and reinforcement learning. In the last section, we presented the rules of Chinese Checkers, and the solver specific definitions of win, draw, and illegal states.

Chapter 3

AlphaZero applied to Chinese Checkers

In this chapter, we discuss implementation details of the AlphaZero architecture, overall self-play, and the training process. However, before delving into AlphaZero architecture, we take a closer look at the AlphaZero architecture as described in the paper where it was first introduced [27].

AlphaZero has two main process. First is a self play stage. In the self play stage, multiple games are played to generate tuples of the form (S, π, z), where S refers to a state, π refers to a policy and z refers to the outcome of the game. Second, when AlphaZero reaches a certain threshold, a model is trained using the saved tuples. The process is repeated until training converges, or a time limit is reached.

3.1 Self Play Stage

During the self play stage, games are played to generate series of (S, π, z) tuples, which are then used to train the model. We start a game in an initial position S_0 of the board as suggested in Figure 3.1. From the state, a search is performed using Monte Carlo Tree Search (MCTS). We discuss details of how search is performed in the next paragraph. After the search is complete, an action a_0 from the position S_0 is selected. Only forward actions were considered because if backward moves were allowed, it might result in moving into the same states multiple times, causing the game to end in a draw. A

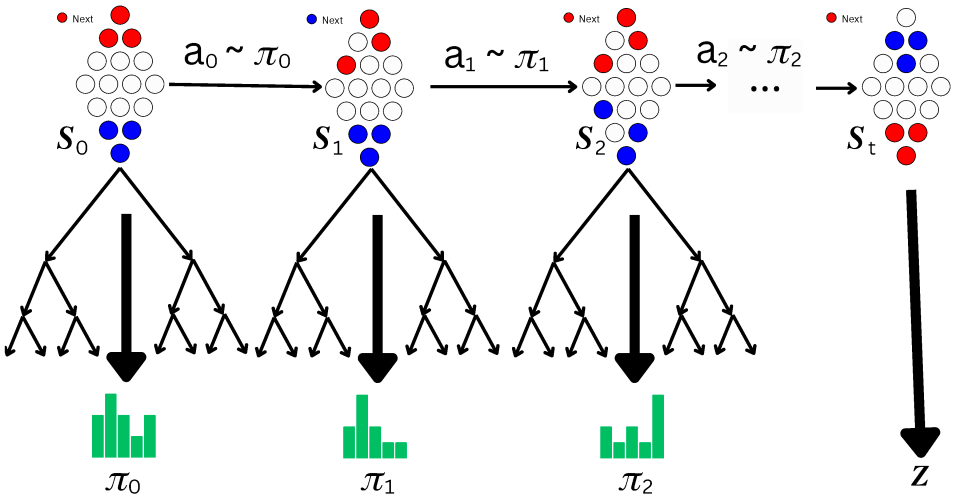


Figure 3.1: Self play in a game of Chinese Checkers

policy π_0 is derived from the node statistics. We will discuss how we convert node statistics to policy later in this section.

After taking the action a_0, we reach a new state S_1. The search step is repeated to select the next action. This process is repeated until we get to a terminal state of the game, when the game is over. This is when we know the outcome of the game: loss, draw or win. A loss is represented as -1, draw as 0, and win as 1. This outcome value is then propagated backwards to each state visited during the game and attached to it to form a (S, π, z) tuple.

Next we discuss how search is performed from a given state. AlphaZero uses a variant of MCTS introduced in the background section, and is visually represented in Figure 3.2. Nodes in the search tree represent different game states, and edges correspond to possible actions. Each edge stores statistics $\{N(s,a), Q(s,a), P(s,a)$, where $N(s,a)\}$ represents number of times a certain state-action has been traversed, $Q(s,a)$ is an estimated backed up value of the action, and $P(s,a)$ denotes the prior probability of selecting that action from a given state.

The search begins with a root node representing the current game state. Given a state, an action is selected that maximizes the PUCT (Polynomial Upper Confidence Trees) score:

$$a = \arg\max_a Q(s,a) + c_{\text{puct}} \cdot P(s,a) \cdot \frac{\sqrt{N(s)}}{1+N(s,a)} \tag{3.1}$$

where c_{puct} is an exploration constant greater than 0. The entire expression $c_{puct} * P(s,a) * sqrt(N(s))/(1+N(s,a))$ forms an exploration term that adjusts the degree to which unexplored actions are favored. The exploration term is denoted as U in Figure 3.2. The more an action has been explored (i.e., the larger $N(s,a)$), the smaller this term becomes, reducing the tendency to select this action purely for exploration. Conversely, if $P(s,a)$ (the network's prior belief in the value of the action) is high, or the state s has been visited many times ($N(s)$ is large), the term increases, encouraging the selection of action a. Thus, the PUCT formula represents a trade-off between exploiting actions known to be good and exploring less certain ones to potentially discover better strategies.

When a new state-action pair with zero visits ($N(s_L, a) = 0$) is traversed, the successor state s' is added as a child node. The leaf node in the tree is represented as s_L. Then, f_θ runs an inference on the state s', where f_θ represents a parameterized function, a convolutional neural network. This step can be seen in step b of Figure 3.2 and is denoted by $f_\theta(s') = (P, v)$. The edge statistics of the legal actions from s are initialized as follows: $N(s,a) = 0, Q(s,a) = 0, P(s,a) = p_a$, where p_a is the probability of taking an action a from state $s^{'}$ as given by the policy network P.

The value estimate is backed up to the pairs traversed in that iteration, updating their action values. Their edge statistics are updated as follows:

$$N(s,a) \leftarrow N(s,a) + 1, \tag{3.2}$$

$$Q(s,a) \leftarrow Q(s,a) + \frac{1}{N(s,a)}(v - Q(s,a)). \tag{3.3}$$

$Q(s,a)$ represents an average of the value estimates, denoted by v, from all states in the subtree rooted at the state-action pair (s,a). Essentially, it predicts the expected outcome from the pair (s,a) by considering the value estimates from the most probable future states that follow this pair. This step is denoted by step c in Figure 3.2.

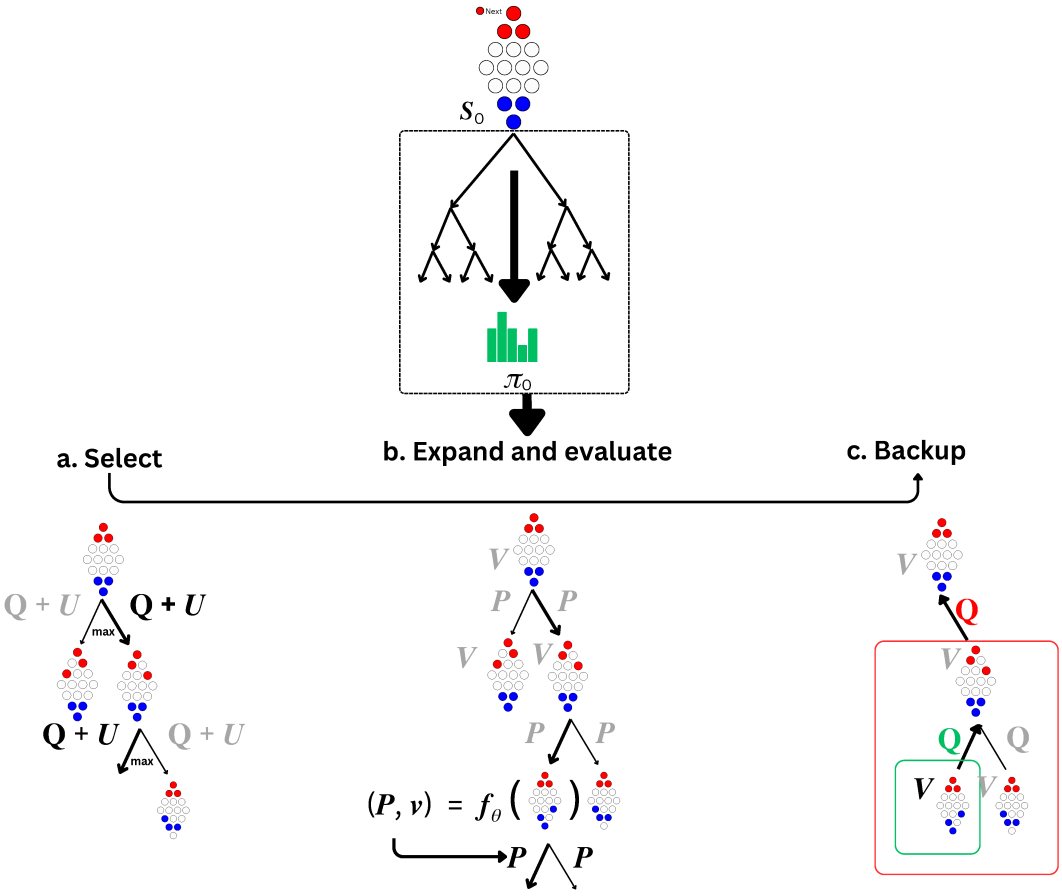


Figure 3.2: MCTS in AlphaZero

In the context of a board game with an $M \times M$ grid, a policy π is derived from the statistics of game nodes. A distribution array is initialized to represent all possible transitions between the cells on the board. The array's size is $(M^2)^2$, where M^2 denotes the total number of cells. This squared value accounts for every potential start ($fromCell$) and end ($toCell$) position of a move.

For a 5×5 board, there are 25 cells yielding $25^2 = 625$ distinct move transitions. However, in practice, there are fewer distinct move transitions because jumps have parity, meaning not all cells are accessible from each position. It also depends on positions of other pieces on the board. The distribution array, initialized with zeros, corresponds to each possible transition.

Node statistics are incorporated into the distribution as follows:

$$\text{distribution}[\text{mapping}[fromCell] \times M^2 + \text{mapping}[toCell]] = N(s, a)$$

Here, *fromCell* and *toCell* are the grid indices, and mapping[*fromCell*] and mapping[*toCell*] convert these indices to the one-dimensional distribution array. The array element corresponding to a move transition is updated state visit counts $N(s, a)$. The distribution array then is normalized so that its sum equals 1.

3.2 Implementation of Self Play games

The self-play process of AlphaZero can be implemented in various ways. In this section, we discuss four approaches we implemented it.

3.2.1 Single Self-play Implementation

One of the naive ways to implement AlphaZero is by playing one game at a time, generating data, and training the model when a certain number of games is played. A high level visual representation of this implementation is shown in Diagram 3.3. In the diagram, a model in the i-th iteration is used to play n games. Each game generates a set of tuples of form (S, π, z), and after n games are played, these tuples are collected into a single dataset. The model i is then trained using this latest dataset and updated to model $i + 1$. This updated model is used in the subsequent game plays in $i + 1$ iteration. The dataset from iteration i is discarded. Instead, a new dataset is created as new games start in the $(i+1)^{th}$ iteration. This process continues until the training time is exceeded.

3.2.2 Single Self-play Implementation with Replay Buffer

One improvement over this naive approach is to introduce a replay buffer [8]. In RL, a replay buffer is a memory storage technique used to store the experiences of an agent as it interacts with an environment. A replay buffer has a fixed length of size N. We fill the buffer with (S, π, z) as they are available from self play games. After we have filled the buffer with N (S, π, z) tuple, we circle back to the first element in the buffer and insert the new (S, π, z) tuple with probability p. In other words, after the buffer is full, the old data are

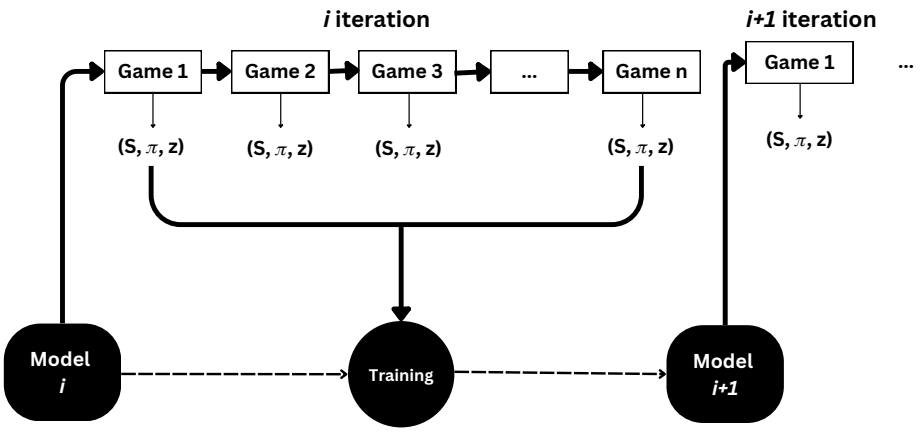


Figure 3.3: Single self play implementation

kept with probability $(1 - p)$. This process has a few advantages:

1. **Data Efficiency:** Each experience collected during the interaction of the agent with the environment can be used more than once, increasing the data efficiency of the learning process.

2. **Reducing Correlation:** By storing and randomly sampling experiences, a replay buffer can reduce the correlation between consecutive training samples. This helps improve the stability of the learning process, as neural networks generally perform better on uncorrelated data.

3. **Balancing Exploration and Exploitation:** By revisiting old states, the replay buffer allows for more balanced exploration and exploitation, facilitating the agent's ability to generalize its learned policy.

4. **Off-policy Learning:** The replay buffer facilitates off-policy learning, where the agent can learn from the experiences of past policies. This can be useful when exploring various strategies or approaches in game play.

Figure 3.4 shows the architecture of the AlphaZero process with an introduction of the replay buffer. A single replay buffer is maintained, and a single game is played. After the game is over, the (S, π, z) tuple is available, and stored in the replay buffer. When the replay buffer reaches a certain threshold, we take a random sample of data from the buffer, and send it for training.

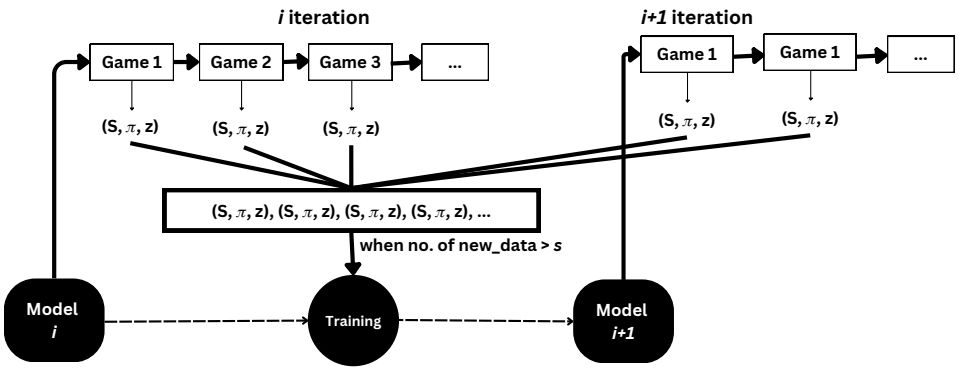


Figure 3.4: Single self play implementation with replay buffer

In our implementation, the threshold was whenever x new (S, π, z) tuples are replaced in the replay buffer. The buffer queue does not have to be full in order to send it for training. However, if the buffer is full, new data are added with probability p from the start of the queue. This process is repeated until a constraint is reached, for example the number of iterations, or time.

Although this approach is an improvement over the naive self play implementation, it is still inefficient. In both approaches, only one game is played at a time, and this approach is slow and under-utilizes computing resources. Most modern computers can run multiple processes, and as such, we can play multiple games at a given time. This is where parallelization comes in.

3.2.3 Self-play Games in Parallel

As discussed above, multiple games can be played at the same time. This process ensures that training data is generated faster, and by extension, training can happen more frequently. Figure 3.5 gives a high level implementation of the parallelization of self play games. The figure shows that n games are played in parallel. As soon as the games are finished, available (S, π, z) are stored in the replay buffer. Since we are playing multiple games in parallel, these data are available sooner compared to the single self play implementation. The process is then similar to the single self play implementation.

Although this approach significantly improves upon single self play implementation, we can optimize it by introducing parallelization during the inference in the MCTS search.

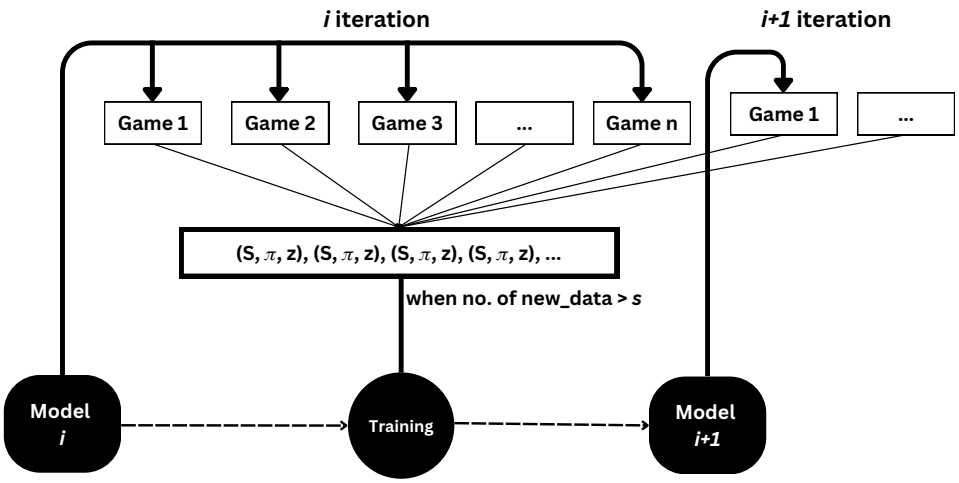


Figure 3.5: Parallel self play implementation with replay buffer

3.2.4 Self-play Games in Parallel with Inference Queue

In step b of Diagram 3.2, only one state of the game is sent to the model for an inference. Inference is a time consuming-process, and we are running inference on just one state at a time. But GPUs can efficiently run inference on many states once.

We optimize the previous approach by introducing a process queue and an inference queue. This is depicted in Figure 3.6. Games are played in parallel, and as such, multiple searches are going on simultaneously. For each game, a local process queue is maintained, as indicated by green box in the Figure 3.6. Also, a global inference queue is maintained, indicated by blue box, which can be accessed by all processes.

While at state S_L, where $N(s_L, a) = 0$ in search, instead of sending the state to inference as in earlier implementations, it is sent it to an inference queue. Once the length of the inference queue equals the number of games, or a certain time limit is reached, all states (S) are batched and sent to the inference process. As inference is a blocking process, playing fewer parallel games allows for quicker dispatch to the inference process, thereby accelerating the overall game play.

The inference process runs an inference model (AlphaZero model) and outputs an array of outcomes (z) and policy distributions (π). The inference

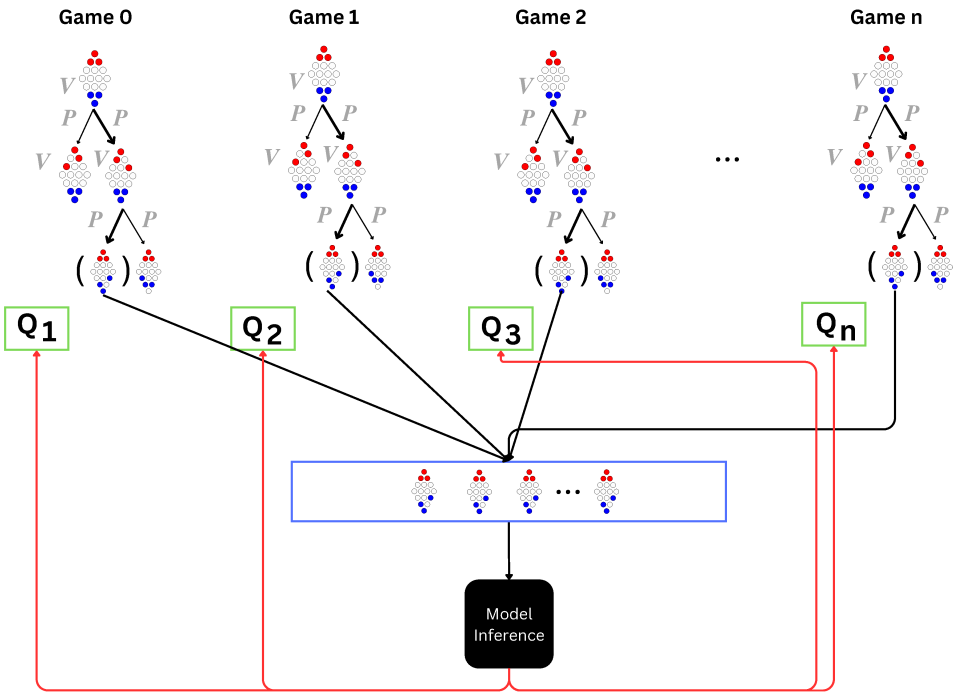


Figure 3.6: Parallel self play with inference queue

model is denoted by a black box in the Figure 3.6. The values that we get back from inference model are then sent to their associated process queue, indicated by the red line. As soon as process queue gets a value, an inference process for that particular search is over, and the values are propagated backwards indicated by step c of Figure 3.2.

In game play, the AlphaZero model performs an inference by taking a state representation S as input and outputs a tuple (z, π). These output values are then backpropagated in the MCTS tree to update the node statistics. This process differs from the inference during training, where the input is the tuple (S, z, π) observed during gameplay, and supervised learning is applied to this data. The details of this training stage are further elaborated in Section 3.3.

3.3 Training Stage in AlphaZero

In this section, we discuss the training stage of the AlphaZero process. First, we discuss neural network architecture of the model and techniques on how exploration is done in AlphaZero.

3.3.1 Neural Network

AlphaZero employs a deep residual neural network with convolutional layers to learn state and policy evaluation. This neural network, denoted as $f_\theta(s) = (p, v)$ in Figure 3.2, is structured similarly to the state-of-the-art deep convolutional networks employed in image recognition. It is adaptable to multiple games due to its general parameters, and it can represent intricate non-linear relationships of the input features.

The network input is a feature vector x, representing a game state s. A game state of Chinese Checkers is represented as a vector, the index of which corresponds to a player's position on the board. For example, a starting position on a 4×4 board with 3 pieces would be represented as $[1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2]$. This state is represented by a stack of two binary matrices, each with dimensions matching the $(n \times n)$ dimensions of the board. The first matrix corresponds to the locations of current players pieces, and the second matrix corresponds to the locations of the other player. For consistency, the current player to play is always represented as the first matrix.

The model then goes through a few convolutional blocks, which includes batch normalization and nonlinear activation function, and multiple residual blocks. Residual blocks, also called ResNet blocks, consist of two sequential convolutional blocks connected via a skip connection. After the resnet blocks are complete, the model branches into two heads: a *policy head* and a *value head*.

The value head is formed by flattening the output from the earlier network. The output then goes through a linear layer of 256 dimensions, ReLU, and a final linear layer that outputs a scalar. The scalar value is passed through the `tanh` function to get the output of the game, indicated by z; either win, draw or loss. We consider anything above 0 to be a win (+1), 0 a draw (0), and anything below 0 a loss (-1).

The policy head is formed of a convolutional block and a fully connected layer, outputting a vector of logits over the actions. To generate a vector of action probabilities p, illegal actions are masked, and a Softmax function

is applied to the residual vector of logit probabilities. Thus, the elements $p_a = Pr(a|s)$ of p determine the probability of selecting action a from state s, and thereby approximating the policy π.

In a specific training sample, represented as (S, π, Z), the neural network utilizes π (the tree policy) as the learning objective for the policy head, and z (the outcome of the self-play match) as the learning target for the value head. Both of these targets are applied to the input s. The parameters of the neural network θ are updated via Adam on the loss function:

$$L = (z - v)^2 - \pi^T \log p + c||\theta||^2 \tag{3.4}$$

where,

- L: the total loss
- z: the actual outcome of the game (1 for win, 0 for draw, -1 for loss)
- v: the predicted outcome by the value head of the network
- $(z-v)^2$: the mean squared error between the predicted and actual game outcome
- π: the search probabilities from the MCTS
- p: the move probabilities predicted by the policy head of the network
- $\pi^T \log p$: the cross-entropy between the predicted move probabilities and the search probabilities
- c: a regularization coefficient
- θ: the parameters of the neural network
- $c\|\theta\|^2$: the L2 regularization term

This procedure optimizes the network's performance by incrementally reducing the loss, or the difference between the network's prediction and the actual result.

After the training process is complete, the updated model is used to play the next iteration of self play games.

3.4 Hyper-parameters in the AlphaZero

The effectiveness of the AlphaZero algorithm is highly dependent on the hyperparameters that guide its learning process. Hyperparameters used in the learning process are as follows:

1. **Simulation Count for MCTS**: This is the number of simulations performed for each move, which affects the depth and breadth of the game tree that is explored before a move is selected. Higher values can lead to better decision-making but increase computation time.

2. **Exploration Constant (C_{puct})**: C_{puct} is used in the PUCT formula 3.1 within MCTS to balance exploration and exploitation, higher values of C lead to more exploration.

3. **Dirichlet Noise Parameter (α)**: This is used to promote exploration during the self-play phase [26].

   At the root of the Monte Carlo Tree Search (MCTS), dirichlet noise is added to the policy probabilities of the network. This nudges the algorithm towards unexplored actions that might not be chosen under the original probabilities, thus ensuring a variety of gameplay.

   The strength and nature of the perturbation are governed by the Dirichlet noise parameter α. With higher α, more exploration is introduced into the system.

   This mechanism aids in creating a diverse dataset for training the deep neural network, contributing to the robustness and generalization ability of the AlphaZero system.

4. **Temperature Parameter (τ):** This controls the level of exploration during the search process.

At each leaf node, AlphaZero uses the probabilities from policy network to guide its search. After performing a certain number of Monte Carlo Tree Search (MCTS) simulations, it generates a new probability distribution over the actions based on the number of times each action was visited during the search.

The temperature parameter comes into play when converting this distribution into a final move. If τ is set to a high value, the move selection will be more random, allowing for a higher level of exploration. If τ is close to zero, the move selection will be almost deterministic, meaning the move with the highest visit count is almost always chosen, emphasizing exploitation of the known good moves.

Specifically, during training, AlphaZero uses a temperature of $\tau = 1$ for the first n moves to promote diverse gameplay and then τ is set close to 0 for the remaining moves. This variation in τ in gameplay during training provides a balance between exploring various strategies early on and then exploiting the most promising moves later in the game.

5. **Learning Rate (η):** This parameter affects how quickly or slowly the model learns from new information. Higher values can lead to faster learning but risk overshooting the optimal solution, while smaller values can lead to more stable learning but risk slow convergence.

6. **Batch Size:** This is the number of game positions used in each training step. Larger batch sizes can lead to more stable learning but require more memory.

7. **Number of Training Steps:** The number of training steps taken for each learning iteration. More steps can lead to deeper learning but require more computation time.

8. **L2 Regularization Parameter (λ):** This helps prevent overfitting by adding a penalty to the loss function based on the size of the weights.

9. **Momentum (β):** Gradient descent algorithms function by iteratively

adjusting the model parameters in the direction of the steepest decrease of the cost (loss) function. However, these algorithms can sometimes be slow to converge, especially in situations where the surface of the loss function is shaped like a shallow ravine with steep sides and a slow, gentle gradient along the bottom. In such cases, the algorithm tends to oscillate across the steep sides of the ravine while only slowly progressing down along the bottom toward the minimum.

Momentum helps address this issue by adding a fraction β of the update vector of the past time step to the current update vector [29]. The momentum hyperparameter β usually takes a value between 0 (no momentum) and 1 (maximum momentum). A common value for β is 0.9. The model converges faster when the momentum hyperparameter is set to high values. However, if the momentum is too high, the algorithm might overshoot the optimal solution, especially near the end of optimization.

In this chapter, we first introduced the overall architecture of AlphaZero. We then discussed ways it can be implemented: single self-play games, single self-play games with replay buffer, and parallel-play games with replay buffer and inference queues. We then discussed the training stage of AlphaZero, and the hyper parameters used in the process. In the next chapter, we will discuss the results of our experiments after we train AlphaZero agent in Chinese Checkers, and evaluate it in novel ways.

Chapter 4

Experiments

In the last chapter, we discussed implementations of various AlphaZero agents, and outlined the rules of the Chinese Checkers game. In this chapter, we discuss the results when we applied AlphaZero in Chinese Checkers.

Experiments were done on three variations of AlphaZero: single self play without replay buffer, single self play with a replay buffer, and parallel play with a replay buffer and inference queue. All of the above experiments were performed on two different Chinese Checkers board sizes: 4×4 with 3 pieces and 5×5 with 6 pieces. Both sizes were chosen because they have been strongly solved [28]. The 4×4 board has 320,320 legal board positions, and 5×5 has 9,610,154,400 legal board positions [28]. For all the experiments, only forward moves were considered during the self play stage.

In this chapter, we outline the hardware and software configurations for our experiments. We then explore the experiments conducted using single-thread self-play and parallel-thread self-play games. We then examine evaluations of the learned agent against fixed opponents, and ground truth data. Furthermore, we identify states where AlphaZero performs poorly, and design an adversarial agent that exploits this vulnerability and pushes AlphaZero agent into losing games. Lastly, we design an experiment to mitigate this vulnerability.

4.1 Experiment Hardware and Software Setup

All variations of AlphaZero code were written on Python 3.8, and the Chinese Checkers board environment was written in C++. Although we wrote the custom implementation of AlphaZero, we used an existing Chinese Checkers environment. The only code added in the C++ implementation was a function named list_to_ccstate that changed state representation from a Python list to the C++ CCState class. We used PyBind11 to create Python bindings of the Chinese Checkers C++ code environment.

We used Jax as our primary deep learning framework. Two main reasons to use Jax over other frameworks were:

1. **Composable Transformations**: Jax enables more sophisticated program transformations like autodifferentiation, vectorization, and parallelization. It makes it easier to apply transformations to parts of computations, providing flexibility in optimization strategies [3]. We have vectorized our code as much as possible and used Jax's automatic parallelization to distribute work to the GPU for inference and training.

2. **Just-In-Time Compilation**: Jax uses XLA for just-in-time (JIT) compilation, which enables efficient execution on accelerators [3]. JIT functions are particularly beneficial when dealing with complex mathematical computations and reduces runtime overhead. We have used JIT functions whenever possible.

In addition to Jax, we also used Jax-based libraries like Haiku to construct neural networks, Optax for optimizers, and Chex for Jax based utility functions. Additionally, we used Python's default multiprocessing module to spawn multiple processes during self play and training.

Finally, we used the Scalene profiler to optimize our implementation. It was used to identify memory leaks in our code, measure the performance of jitted functions, and reduce the number of variable copies.

Most experiments for the 4×4 board were run on a Mac M1 with 8 CPU cores. At the time of the research we were unable to utilize M1 GPU for

calculations. For the 5×5 board, the experiments were run on a lab server which has 32 AMD CPU cores and 2 NVIDIA GeForce RTX 2080 Ti GPUs.

4.2 How do we measure learning?

There are two different ways to measure whether or not the AlphaZero agent is learning. First, we play the learned agent against fixed opponents. The win rate against the opponent can give us a way to measure whether an agent has learned over the training. In our experiments, the AlphaZero agent competed against versions of itself, a random player, and a UCT-based player with various sample sizes, a UCT player with distance evaluation function, and a perfect player. This gives us a robust way to measure learning over iterations. Second, we study and evaluate how well the trained agent generalizes over unseen states. Previous work on solved Chess endgames has suggested that we could measure the learning progression of AlphaZero [12]. We extend this concept to solved versions of Chinese Checkers. For this step, we take the learned model at regular iteration steps and evaluate it against states that were seen during training as well as carefully chosen unseen states.

4.3 Does the model have capacity to learn?

Since AlphaZero employs residual neural network with convolutions layers to learn value and policy functions, we wanted to verify weather the model itself is capable to learning or not. For this, we trained on ground truth in 5×5 board in iterations. For each iteration we randomly sampled 10,000 states from the state space, and trained on their game theoretic values as a supervised learning target. This model was trained for 3,667 iterations (36,670,000 states) over a period of 72 hours. Many random states are uninteresting, with a clear winner that has no losing moves, so the final model was evaluated against states seen by AlphaZero during training, achieving an accuracy of 94%. This experiment gave us concrete evidence that the model is able to learn, and thus was employed during the training.

4.4 Single-thread Self Play Experiment

As discussed in Chapter 3, we first implemented a naive version of AlphaZero. In the naive version, we play a fixed number of games to generate data. Generated data is then used to train and update the model. The updated model is then used to play the next iteration of games. We applied this implementation on the 4×4 board size.

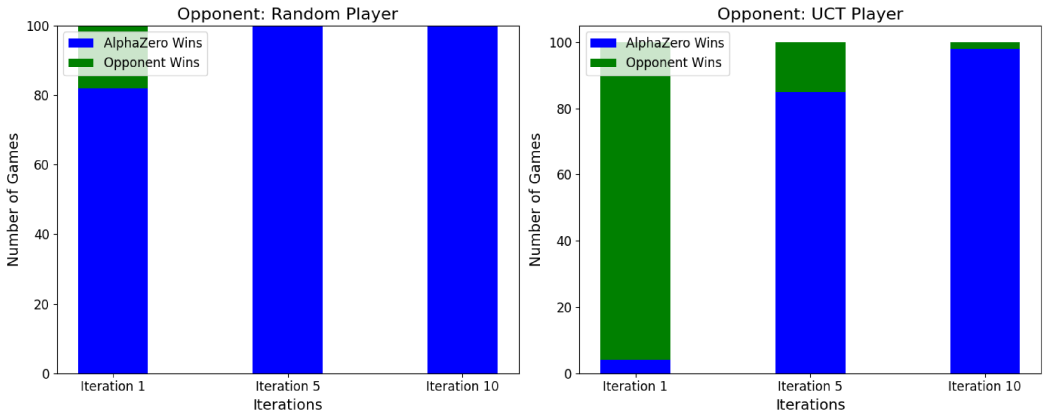


Figure 4.1: Number of games won by AlphaZero agent as first player against random player (left) and UCT player (right).

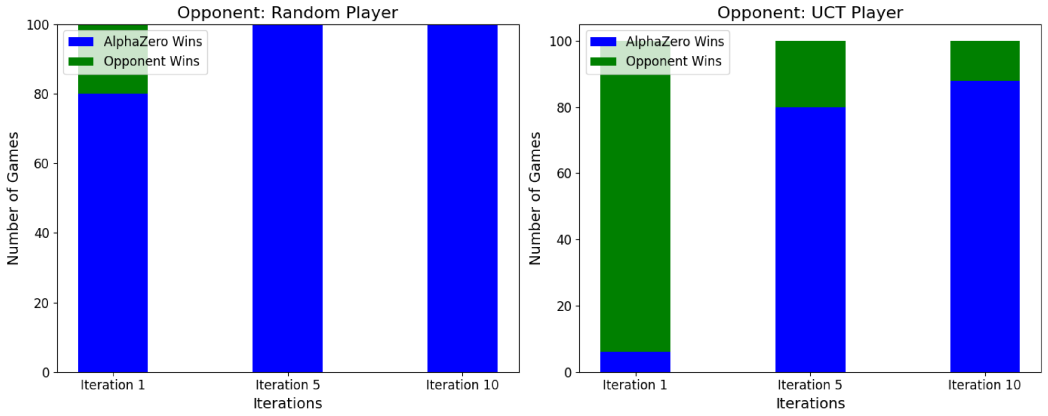


Figure 4.2: Number of games won by AlphaZero agent as second player against random player (left) and UCT player (right).

For this experiment, we played 300 games in each iteration for 10 iterations. The results of this experiment can be seen in Figures 4.1 and 4.2. As a first

player, Figure 4.1 shows that the AlphaZero agent, after the first training iteration, wins 80 games against random player, and 4 games against a UCT player with 100 samples per move. However, after the 10^{th} iteration, it won 100 games against a random player and 98 games against a UCT player. Figure 4.2 shows that the AlphaZero agent, as a second player, wins 81 games against random player, and 6 games against UCT player with 100 samples in the first iteration. However after 10^{th} iteration, it won 100 games against a random player, and 88 games against a UCT player. The increasing win rate against fixed opponents suggested that the AlphaZero agent was able to learn over time. Since Chinese Checkers is a first player win, it also suggests both random, and UCT players are weak players.

4.5 Parallel-thread Self Play Experiments

In this section, we discuss experiments done using the parallel-thread self play architecture. First, we analyze learning progression over training iterations. Second, we play learned agents against fixed opponents. Lastly, we evaluate the learning generalization of the learned agent. All of the three experiments were done on both on both 4×4 and 5×5 board sizes.

4.5.1 Experiments on 4×4 Board Size

For this experiment, we ran parallel self-play games on the 4×4 Chinese Checkers board with 3 pieces, and we also introduced a buffer queue and an inference queue. For this experiment, we used a Mac M1 with 8 CPU cores. We used six cores to spawn up six processes for self play games, and the remaining two were used for inference and training, as described in Section 3.2.4.

Figures 4.3, 4.4, 4.5 and 4.6 show key metrics during parallel-thread self play game during training. The blue line in Figure 4.3 shows the average time to play a single game across all iterations. As we can see, the time taken to finish the game goes down across iterations. This suggests that the agent is able to find better policies such that it is able to finish the game faster.

Figure 4.4 shows the number of games played over 50 iterations. In sub-

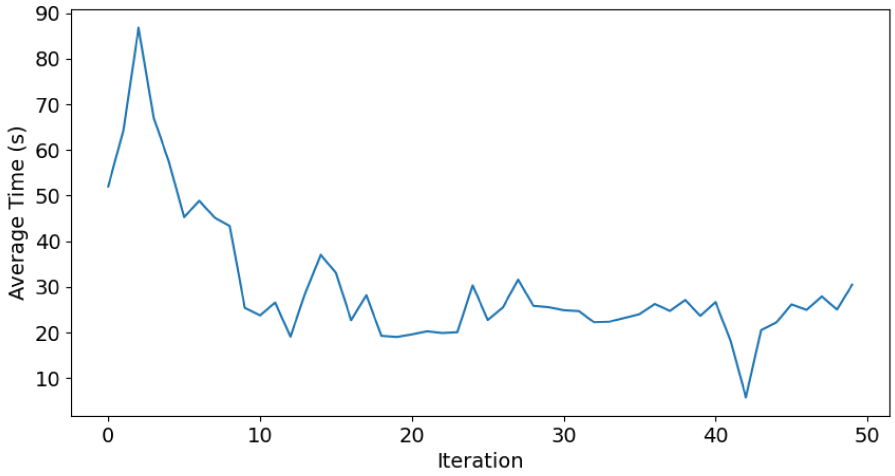


Figure 4.3: Average time required to finish a single game across 50 iterations.

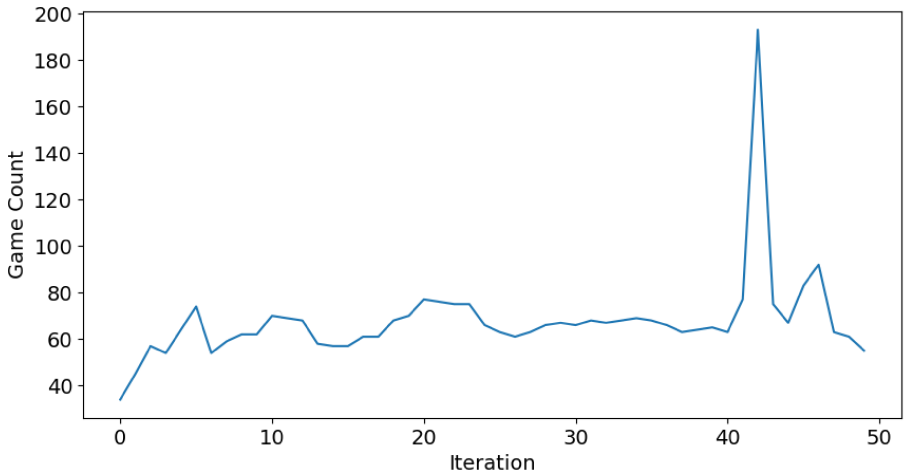


Figure 4.4: Number of games played in each of the 50 iterations.

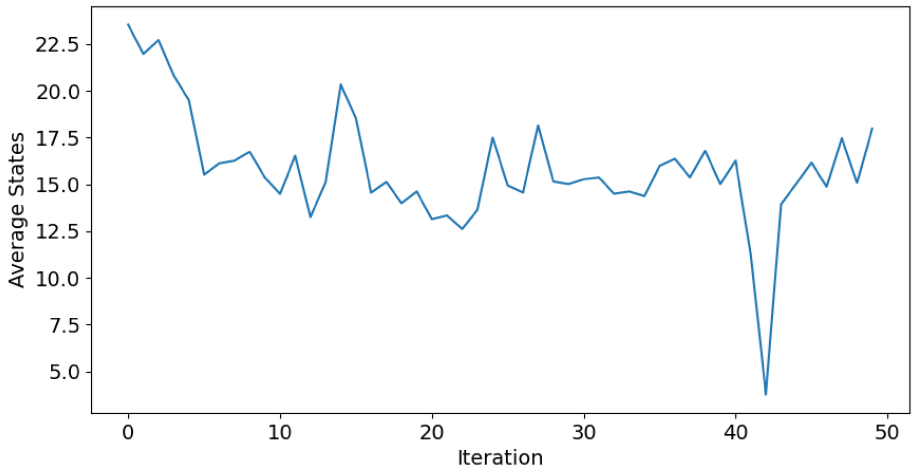


Figure 4.5: Average number of states visited during each of the 50 iterations.

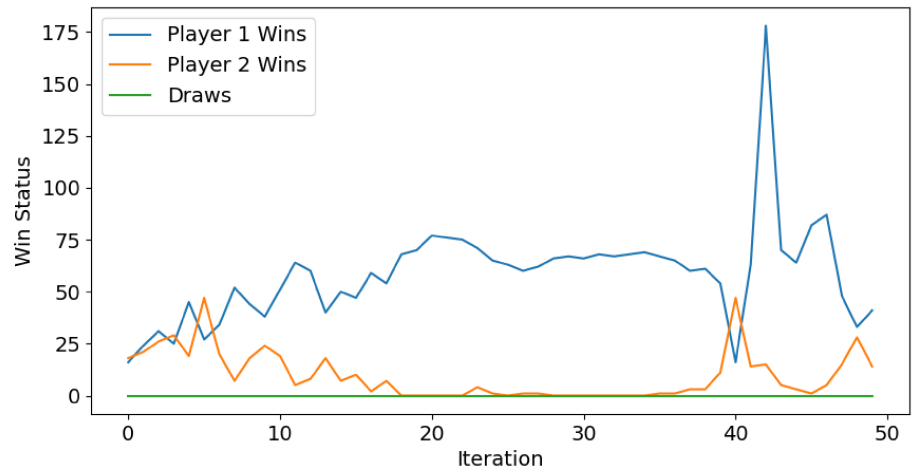


Figure 4.6: Win rates of player 1 and player 2 in each of the 50 iterations.

sequent experiments, an iteration is defined as the completion of training on a new set of (S, z, π) tuples, drawn from replay buffer. As we can see, the game count increases over time. This behaviour is expected as fewer states are visited and inserted into the queue, we need more states to reach constant k. As such, it needs to play more games in order to generate more (S, π, z) tuples.

Figure 4.5 shows the the average number of states visited in a game across iterations. During earlier iterations, more states are visited per game. However, as learning progresses, lesser states are visited per game. It suggests that fewer states are visited in a game, and games are ending faster and in less time as seen in Figure 4.1.

Figure 4.6 shows the player win rates during the self-play process of AlphaZero. The blue line signifies player 1 wins, the yellow line indicates player 2 wins, and the green line represents draws. Before the 5^{th} iteration, player 2 won most of the games. After the 5^{th} iteration, player 1 won the majority of the games against player 2. However, at the 40^{th} iteration, player 2 wins most games against player 1. Yet, player 1 fights back after the 40^{th} to win most games. This "dance" between win rate among players in certain intervals can be seen more vividly in 5×5 board. No draws were observed during game-play. Further evaluations on parallelism will be presented in the section 4.6.

4.5.2 Learned Agent against Fixed Opponent in 4×4 Board

We evaluated the post-training performance of the parallel-thread trained agent against several fixed opponents: UCT-based players with varying sample sizes (100, 200, 1000, 2000), UCT based players that utilizes distance evaluation function with two different sample sizes (1000 and 2000), and a perfect player. We measured the win rate as both player 1 and player 2. Performance metrics were collected every 5^{th} iteration. In Tables 4.1 and 4.3, the number accompanying "UCT" indicates the sample size of the UCT based player. In Tables 4.2 and 4.4, "D" refers to UCT based player that utilizes distance evaluation function instead of rollouts in MCTS and the number associated with

it refers to the MCTS search samples used by the UCT player. A total of 100 games were played, and backward moves were disallowed in all games.

Rollouts, as utilized by a UCT player in MCTS, involve playing out the game to completion from a leaf node of the MCTS tree using random forward moves. The result of this simulated game (win, loss, or draw) is then used to inform the statistical models that govern decision-making, with the outcome being propagated back up the MCTS tree. In contrast, a distance evaluation function, employed by a 'D player', offers an alternative to rollouts. Rather than simulating the game to the end, this function estimates the distance to the goal or terminal state from any given board state. This is advantageous because it provides a direct measurement of progress toward the game's objective. Propagating these distance estimates back through the MCTS tree can result in a more accurate policy for tree expansion and move selection. Furthermore, the use of a distance evaluation function can significantly reduce computational demands compared to executing full rollouts.

"PP" (perfect player) refers to a player that uses the solver described in [28] to choose optimal moves, with the associated number indicating the search sample used by the AlphaZero agent. As both 4×4 and 5×5 boards are solved, a perfect player will always make an optimal move from a winning position, i.e., transitioning to a state that also guarantees a win. However, in a losing position where no available moves lead to a win, the perfect player opts for a random move. It does not employ search in such situations.

The reason why the perfect player does not need to search when in a losing position is that in solved game scenarios, like 4×4 and 5×5 boards, the outcomes of all possible moves are already known to the perfect player. Therefore, if a perfect player is in a losing position and there are no moves that lead to a win, there is no benefit to searching for an optimal move. In such cases, any move will inevitably lead to the same outcome - a loss.

Table 4.1 displays the win rate of a parallel-thread AlphaZero agent as the first player against UCT players with sample sizes of 100, 200, 1000, and 2000. It is noted that in the initial iteration, the AlphaZero agent loses most games against all UCT players. However, after only 15 iterations, the AlphaZero

Iteration	UCT_100	UCT_200	UCT_1000	UCT_2000
1	31	27	10	8
5	96	90	76	64
10	97	97	89	82
15	99	98	94	95
20	99	100	99	96
25	99	100	96	96
30	98	98	99	98
35	100	99	97	96
40	100	98	95	88
45	98	98	98	97
50	100	99	99	98

Table 4.1: Performance of Parallel AlphaZero as Player 1 vs. UCT players (100, 200, 1000, 2000 samples)

Iteration	D_1000	D_2000	PP_128	PP_512	PP_2048
1	2	0	27	32	30
5	44	28	60	65	59
10	55	42	58	61	69
15	64	63	65	68	69
20	73	61	61	67	59
25	77	79	62	63	92
30	84	82	68	61	68
35	88	78	65	70	68
40	69	65	54	59	65
45	86	83	63	62	71
50	91	86	66	69	88

Table 4.2: Performance of Parallel AlphaZero (Player 1) against UCT players with distance evaluation (1000, 2000 samples) and a perfect player.

Iteration	UCT_100	UCT_200	UCT_1000	UCT_2000
1	31	21	7	8
5	89	80	59	55
10	93	94	74	72
15	99	96	92	87
20	100	92	90	86
25	94	97	92	77
30	97	99	93	84
35	99	96	86	84
40	77	81	73	69
45	99	98	88	94
50	99	98	91	94

Table 4.3: Performance of Parallel AlphaZero as Player 2 vs. UCT players (100, 200, 1000, 2000 samples)

Iteration	D_1000	D_2000	PP_128
1	1	0	0
5	12	10	0
10	28	17	0
15	27	37	0
20	37	41	0
25	56	49	0
30	51	38	0
35	57	45	0
40	48	33	0
45	66	50	0
50	71	55	0

Table 4.4: Performance of Parallel AlphaZero (Player 2) against UCT players with distance evaluation (1000, 2000 samples) and a perfect player.

agent manages to win 95 games. By the 50th iteration, the learned agent is capable of defeating all UCT players, even winning 98 out of 100 games against the strongest UCT player. Nonetheless, as discussed in Section 4.4, the UCT players are considered weak opponents. There was a need for stronger agents for comparison with the learned agent.

Table 4.2 presents the win rate of a parallel-thread AlphaZero agent as the first player against stronger agents, namely UCT with a distance evaluation function (UCT_D), and a perfect player. As anticipated, UCT_D outperforms the UCT with rollouts, even winning all games against the first iteration of the learned agent. Interestingly, the perfect player performs worse than UCT_D against earlier iterations of the AlphaZero agent. However, later iterations of the learned agent significantly increase their wins against both UCT_D and the perfect player.

However, as discussed earlier, in Chinese Checkers, the first player has an inherent advantage, and the AlphaZero agent, as the first player, benefits from this. How does the learned agent fare when playing as the second player? Tables 4.3 and 4.4 answer this question. Table 4.3 indicates that the learned agent still manages to win almost all games against the weaker UCT players. Table 4.4 reveals that UCT_D players perform better as the second player against the learned agent, with UCT_D winning 45 games compared to only 14 wins as the first player. The perfect player wins all games against the learned agent, which is expected because, on the solved 4×4, a perfect player starts from a theoretically winning game state and always makes the optimal move.

4.5.3 Evaluation against Ground Truth

In this experiment, we evaluated the generalization property of the learned agent. The aim of this evaluation is to study how learning takes place in AlphaZero. Since the game on the 4×4 board is strongly solved, we had access to a solver [28]. With access to this solver, given a state, and the player, we can get the accurate outcome of the game from the solver.

For this experiment, we made three distinct datasets.

- **Seen state dataset:** This dataset encompasses all states that were experienced directly during the self-play training stage, comprising 7,632 unique states. However, only states whose children have two different possible outcome were sampled. Such states represented a more complex scenario where the model can make mistakes. The aim of evaluating the model on this dataset is to check its capability to remember and apply the knowledge acquired during training. We also wanted to study the inaccuracies in training data and whether the model is able to learn despite these inaccuracies.

- **Neighboring state dataset:** This dataset consists of 30,317 unique neighboring states derived from all the states observed during training. A neighboring state is defined as one that can be reached by executing a single action from any state within our training database. Similar to seen state dataset, only states whose children have two different possible outcome were sampled. None of these neighboring states were directly observed during training. However, they share close relations to the states that were seen. By evaluating the model on this dataset, we aim to assess its ability to generalize to new, yet similar, states.

- **Random state dataset:** Contrary to its name, this dataset is not completely random. It consists of states selected randomly from the game space, but only those where the outcomes of child states differ from their parent states; that is, these are states that are won for the player to move, but they have the potential to make a losing move. If all moves lead to a win or a loss, there are no interesting decisions to make. Also, these states are less frequently seen during gameplay and represent more complex scenarios. Recent work has trained adversarial policies that cause AlphaZero to make serious blunders [31]. These blunders would eventually lead AlphaZero system to lose the game. By evaluating the model on these "hard" states, we aim to test the robustness and adaptability of the AI to unexpected game scenarios. Additionally, this dataset includes states not represented in the seen state or neighboring

state datasets. For 4×4 board size, this dataset contained 10,000 unique states.

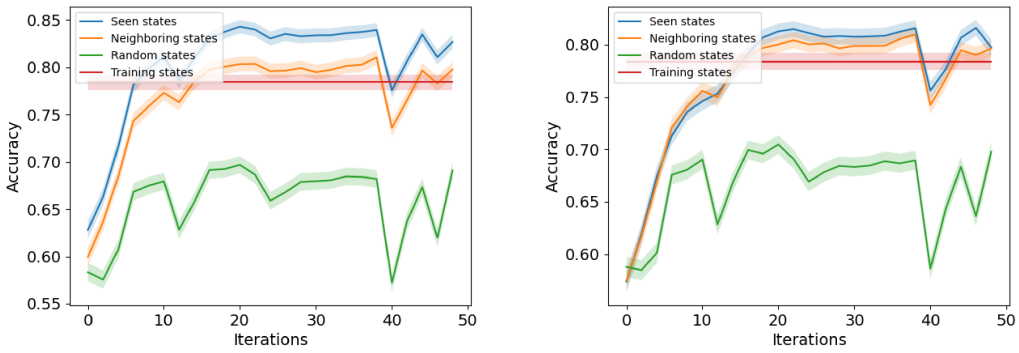
We then assessed the agent's accuracy in predicting game outcomes from states in each dataset. We sampled all states from the training dataset, and 10,000 states from the neighboring and random datasets and compared the AlphaZero value function (z) to the actual outcome from the solver. This allowed us to determine the percentage of accurate evaluations by the model. Figure 4.7 illustrates the results: the blue line indicates accuracy on seen states, the yellow line on neighboring states, green line on random states. Each state was evaluated from the perspectives of both Player 1 and Player 2. For rest of the figures, the left image represents Player 1's perspective, while the right image represents Player 2's.

As learning progresses over training iterations, it can be observed that the model is able to learn the outcome of the seen states, as indicated by increasing blue line. This is expected because it is trained on these data. If the model is learning, it should be increasingly accurate about the outcome of the game from a given state.

Additionally, the orange line denotes outcome accuracy on neighboring states. Unlike seen states, these states were never encountered during training. However, since they are the neighbors of the seen states, it can be assumed that these are closely related to the seen states. The orange line indicates that this assumption might be true. Although the model has never seen these states, it is able to be increasingly accurate about the outcome of the game state. However, it stays below the seen state line because we expect the model to learn more about the seen states rather than unseen states.

The random state accuracy against ground truth is denoted by a green line. The accuracy of random stays increases over iterations. However it stays below the accuracy of seen and neighboring states.

The red line in the Figure 4.7 represents training data accuracy against ground truth. Remember that the training state data is represented as a (S, π, z) tuple. That means for each state, we have access to its outcome from



(a) Player 1 perspective

(b) Player 2 perspective

Figure 4.7: Evaluation of outcome accuracy in seen, neighboring, training, and random states against ground truth in 4×4 board.

the self play game. We compared this outcome against the solvers' outcome. The training sample remains the same over iteration, and by extension, its accuracy should remain constant over each iteration as well. The red line clearly shows that the accuracy is constant around 0.784, indicating that only 78% of our training data is accurate. It is a fact that AlphaZero learns despite the inaccuracies in the training dataset, and we can clearly see this observation as a red line in the Figure 4.7. Figure 4.8 shows the training data accuracy for cumulative states observed up to the current iteration over the course of training iterations. In the figure, it can be seen that training data gets accurate over training iterations.

For the second part of our experiment, we measured the accuracy of AlphaZero policy head. For every position, we compared the moves predicted by AlphaZero policy head to the correct moves given by the solver. This comparison let us calculate the percentage of the model's correct predictions. We then evaluated if the search from the given state would help increase this accuracy. For this experiment, we sampled 1,000 random states from each of the datasets. The results of this experiment are displayed in Figures 4.9, 4.10 and 4.11.

The policy evaluation for observed states is shown in Figure 4.9. As the model progresses, its accuracy improves. Furthermore, adding more samples to the MCTS search enhances the base model's accuracy for both players. This

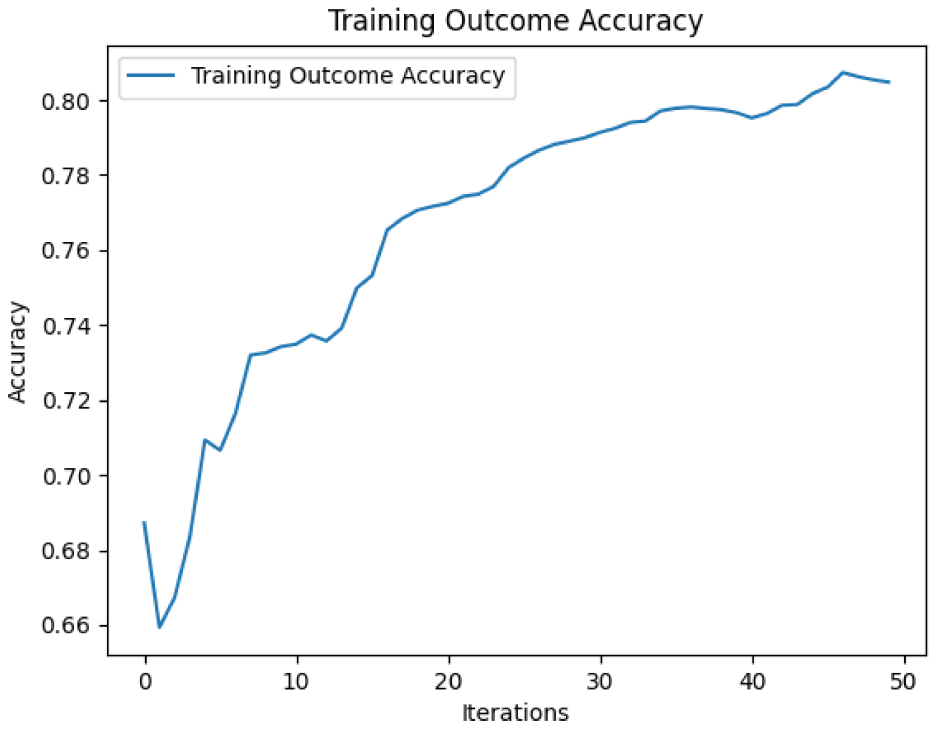


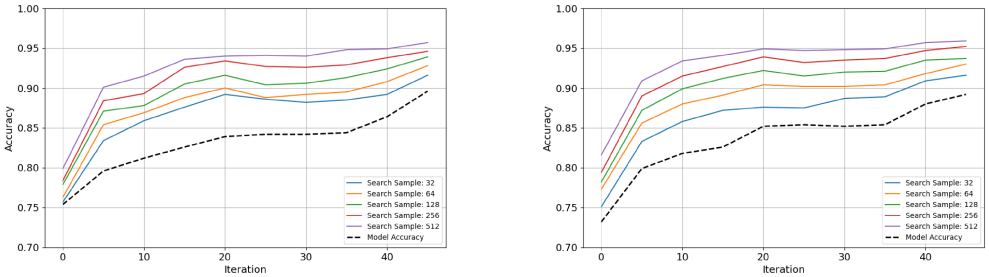
Figure 4.8: Training data accuracy over iteration in 4×4 board.

trend is also evident in both training and random states, as shown in Figures 4.10 and 4.11 respectively. For this experiment, we used a sample size of 32, 64, 128, 256 and 512. In the figures, the dashed line represents the model accuracy, and the rest of the lines represent the model accuracy guided by search.

4.6 Experiments on 5×5 Board

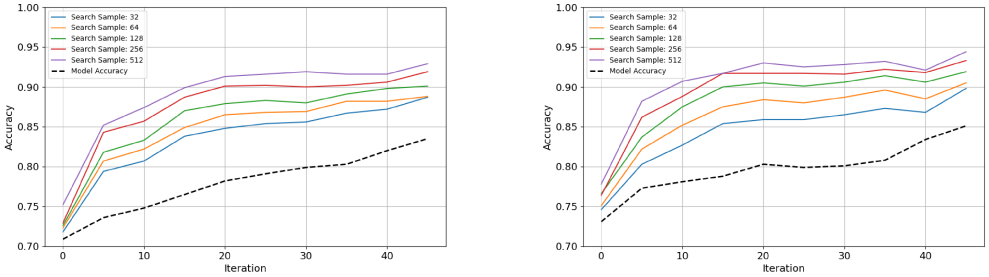
Now, we focus our attention to the experiments conducted on the 5×5 Chinese Checkers board size. We repeat the same experiments from Section 4.5.1 to see if the same trends hold. If it does, we might also expect them to hold on larger board sizes of the game as well.

All our experiments for the 5×5 board were done on the same machine, which had 32 CPU cores and 2 GPUs. We used one main process to start the other processes and to keep important global data. In total, we ran 28 self-play



(a) Player 1 (b) Player 2

Figure 4.9: Evaluation of policy accuracy in training states against ground truth on the 4×4 board.



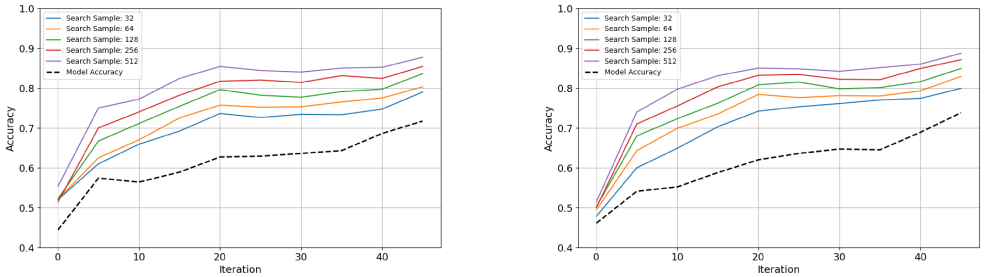
(a) Player 1 (b) Player 2

Figure 4.10: Evaluation of policy accuracy in neighboring states against ground truth on the 4×4 board.

games at the same time using 28 CPU cores. The other two processes worked with the GPUs: one sent data for predictions and the other for training. A remaining CPU core was used to play single-thread self play games, and training. We mainly adjusted settings like buffer size, learning rate, and the constant k. To make a valid comparison, all parameters were kept same except the number of parallel games played. Additionally, both programs ran for a week.

Figures 4.12 shows an average time required to finish a single game across 300 iterations for both single-thread self-play implementation and parallel-thread self-play implementation. Other key metrics during training can be found in appendix A.

Moreover, it can be noticed that the parallel-thread self play implementation is significantly faster than single play implementation. After a week of



(a) Player 1 (b) Player 2

Figure 4.11: Evaluation of policy accuracy in random states against ground truth on the 4×4 board.

training, parallel play implementation reached 301^{th} iteration while single play implementation merely reached 10^{th} iteration. This number is proportional to the number of CPU cores that we utilized to play games. A similar trend was observed when we applied it in 4×4 board size.

4.6.1 Learned Agent against Fixed Agents

We evaluated the post-training performance similar to that in section 4.5.1. Performance metrics were collected every 10 to 70 hours, as shown in Table 4.5. The evaluation was done only until 70^{th} hour because the trained agent consistently won 100% of games as the first player, and approximately 96% as the second player. In the table, "Single AZ" denotes the single-play AlphaZero agent.

Hour	Single AZ	UCT_100	UCT_200	UCT_1000	UCT_2000
10	99	99	96	92	93
20	98	98	99	95	94
30	100	100	97	95	92
40	100	99	100	96	96
50	100	100	100	98	98
60	100	100	99	100	98
70	100	100	100	100	100

Table 4.5: Performance of Parallel AlphaZero as Player 1 vs. Single-thread AlphaZero and UCT players (100, 200, 1000, 2000 samples)

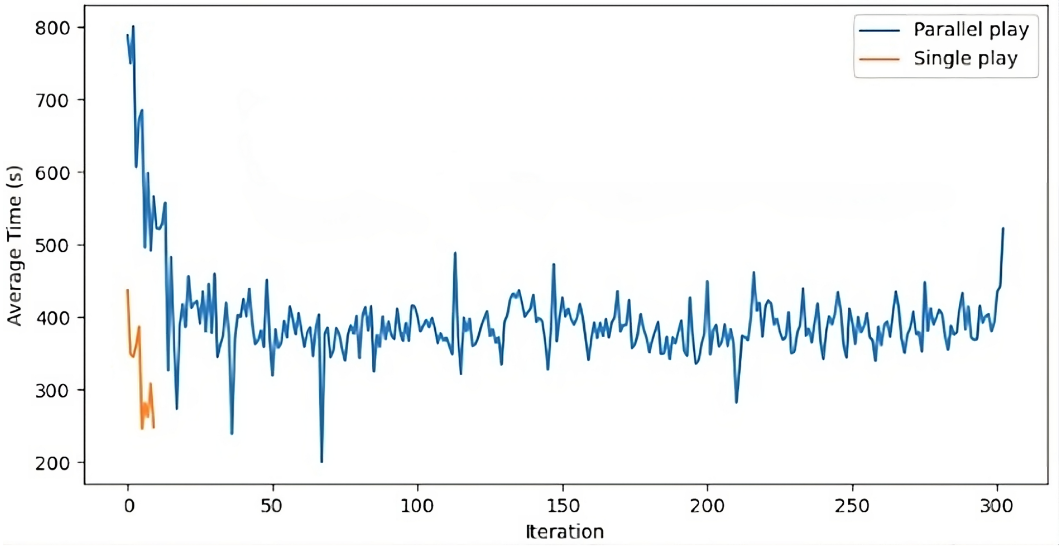


Figure 4.12: Average time required to finish a single game across 300 iterations. After a week of training, the parallel play implementation, indicated by blue line, reached 301^{th} iteration while single play implementation, indicated by orange line, merely reached 10^{th} iteration.

Tables 4.5 and 4.6 shows win rate of AlphaZero agent as first player against single-thread self play agents, various UCT agents and a perfect player. Similar to what we observed in 4×4 board in Section 4.5.1, the parallal-thread AlphaZero agent is able to win all weak UCT agents, and even stronger agents like UCT_D, and perfect players.

Tables 4.7 and 4.8 shows win rate of AlphaZero agent as second player against single-thread self play agents, various UCT agents and a perfect player. The learned is agent is able to beat all weak agents, and stronger UCT_D agents. As expected, a perfect player, as the first player, always wins.

4.6.2 Evaluation on Ground Truth Data

Similar to experiment in Section 4.5.3, this experiment is setup to evaluate the generalization property of the learned agent. The aim of this evaluation is to study how learning takes place in AlphaZero. Since the 5×5 board is also a strongly solved board, we had access to a solver [28]. With access to this solver, given a state, and the player, we can get the accurate outcome of the game from the solver.

Hour	D_1000	D_2000	PP_128	PP_512	PP_2048
10	55	52	34	74	99
20	95	90	80	80	94
30	97	93	93	90	88
40	92	94	84	76	74
50	94	98	89	95	98
60	99	98	82	85	81
70	100	99	96	100	99

Table 4.6: Performance of Parallel AlphaZero (Player 1) against UCT players with distance evaluation (1000, 2000 samples) and a perfect player.

Hour	Single AZ	UCT_100	UCT_200	UCT_1000	UCT_2000
10	93	94	83	79	75
20	100	98	99	83	80
30	100	100	98	92	87
40	100	98	98	94	90
50	100	99	100	96	90
60	100	100	100	95	93
70	100	100	100	97	95

Table 4.7: Performance of Parallel AlphaZero as Player 2 vs. Single-thread AlphaZero and UCT players (100, 200, 1000, 2000 samples)

The dataset for this experiment is similar to the one described in section 4.5.3. However, there are a few differences.

- **Seen state dataset:** This dataset encompasses all states that were experienced directly during the self-play training stage, comprising 412,570 unique states.
- **Neighboring state dataset:** This dataset consists of 4,568,133 unique neighboring states derived from all the states observed during training.
- **Random state dataset:** It consists of states selected randomly from the game space, but only those where the outcomes of child states differ from their parent states. These states are less frequently seen during gameplay and represent more complex scenarios. We randomly sampled 10,000 such states from the game space.

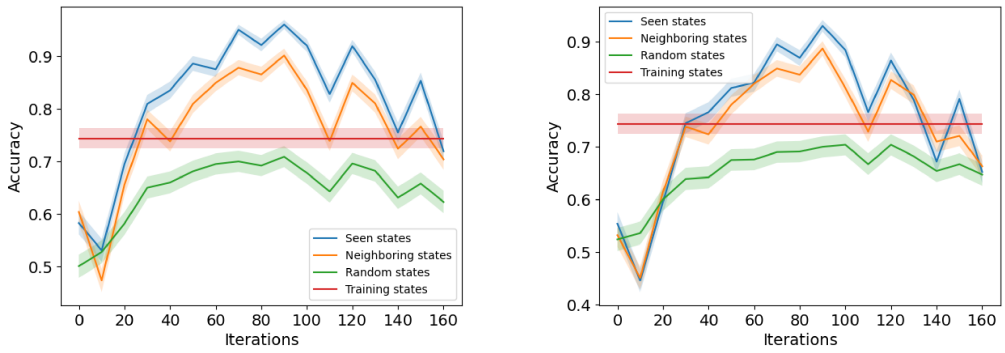
Hour	UCT_Dist_1000	UCT_Dist_2000	Perfect_Player
10	44	40	0
20	87	78	0
30	89	85	0
40	86	89	0
50	92	91	0
60	96	90	0
70	98	92	0

Table 4.8: Performance of Parallel AlphaZero (Player 2) against UCT players with distance evaluation (1000, 2000 samples) and a perfect player.

We then assessed the agent's accuracy in predicting game outcomes from a given state. We sampled 10,000 states from each datasets and compared the AlphaZero value function (z) to the actual outcome from the solver. Only states whose children have two different possible outcome were sampled. Figure 4.13 illustrates the results: the blue line indicates accuracy on seen states, the green line on neighboring states, and the red line on random states.

Similar to Section 4.5.3, training states accuracy outperform neighboring ones, which in turn are more accurate than random states. The training data accuracy is 74%. Figure 4.14 shows the training data accuracy for cumulative states observed up to the current iteration over the course of training iterations. In the figure, we observe an increase in the accuracy of the training data over these iterations. Although the training accuracy increases, it can be seen that its still lower than the outcome accuracy of seen and neighboring states. Despite evaluating in challenging states, the model is increasingly able to evaluate the outcome of the game. However, it can be seen that the model accuracy decreases after 100 iterations. This can be attributed to the fact that the model performs poorly from 100-160 iterations. This can be seen in Figure 4.13. However, we ran a second run of the same experiment, and we did not see the dip. The result is shown in A.4.

For the second part of our experiment, we evaluated the accuracy of AlphaZero policies. For this experiment, we sampled 1,000 random states from each of the datasets. The results of this experiment are displayed in Figures



(a) Starting with player 1 (b) Starting with player 2

Figure 4.13: Evaluation of outcome accuracy in seen, neighboring, training, and random states against ground truth in 5×5 board.

4.15, 4.16 and 4.17.

The policy evaluation for seen states is shown in Figure 4.15. As the model progresses, its accuracy improves, and stagnates around 40^{th} iteration. Furthermore, adding more samples to the MCTS search enhances the base model's accuracy for both players. This trend is also evident in both training and random states, as shown in Figures 4.16 and 4.17 respectively. For this experiment, we used a sample size of 32, 64, 128, 256, 512 and 1024. In the figures, the dashed line represents the model accuracy, and the rest of the lines represent the model accuracy guided by search. The results are similar to that of section 4.5.3. The base model accuracy increases over time, and increasing search sample aids on increasing model's accuracy for seen, neighboring, and random states.

4.7 Exploiting AlphaZero with an Adversarial Agent

Tables 4.5 and 4.6 show that the AlphaZero agent, as first player, wins almost all games against weak and stronger agents in 5×5 board. Additionally, from the experiments in Sections 4.5.3 and 4.6.2, it is clear that the model performed poorly on random states. Can we design an agent that exploits this weakness to steer the AlphaZero agent into these random states and increase its chances

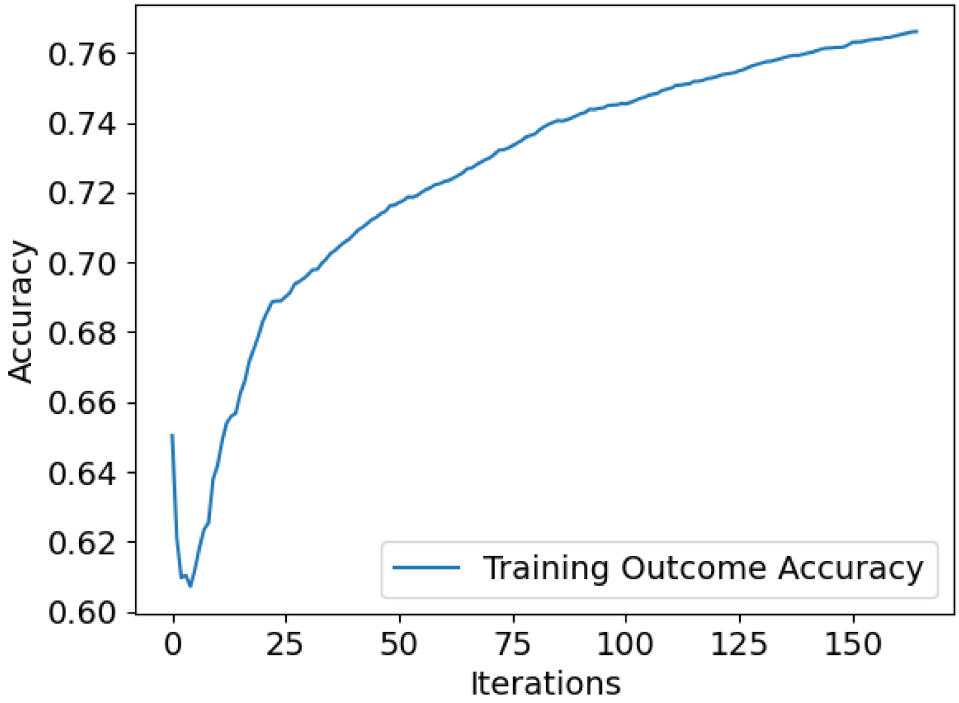


Figure 4.14: Training data accuracy over iteration in 5×5 board.

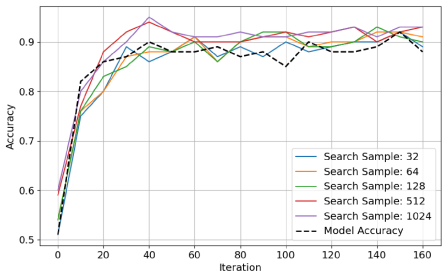
of losing? This question forms the motivation for our next experiment.

As outlined in Section 4.5.2, a perfect player transitions to a winning state when possible; otherwise, it opts for a random move in a losing state. We enhanced the perfect player's strategy to create an adversarial agent.

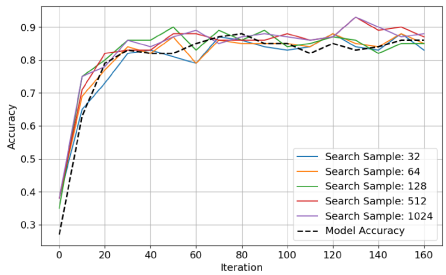
In a winning state, the adversarial agent continues to make optimal moves. In contrast, its approach changes when in a losing state:

- **2-Ply Search:** The agent looks ahead to its next move and the opponent's possible responses, enabling it to plan ahead.

- **Evaluating Moves:** After the 2-ply search, the agent uses a solver to determine whether the potential outcomes are wins or losses for Player 1. It then chooses the move that leads to the most losses for Player 1. If there are several moves with the same likelihood of causing Player 1 to lose, the agent picks one at random. Additionally, we introduced more randomness in the agent's move selection. With a probability of

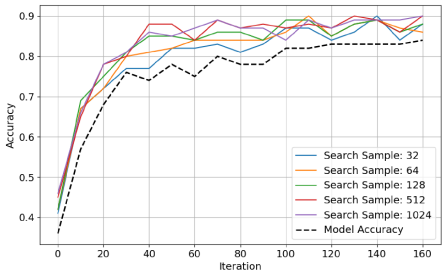


(a) Starting with player 1

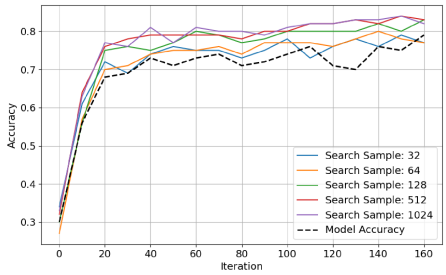


(b) Starting with player 2

Figure 4.15: Evaluation of policy accuracy in training states against ground truth in 5×5 board.



(a) Starting with player 1



(b) Starting with player 2

Figure 4.16: Evaluation of policy accuracy in neighboring states against ground truth in 5×5 board.

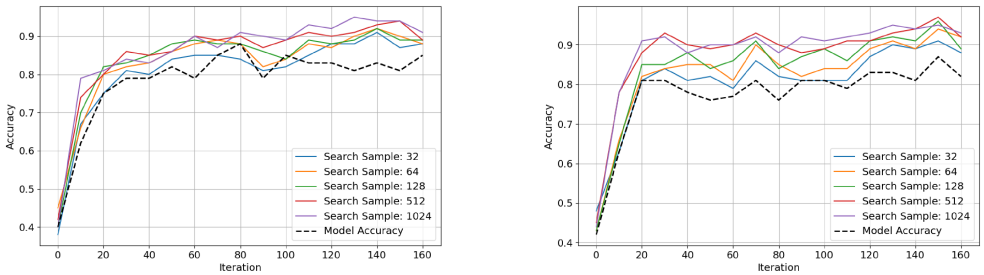
(1-ϵ), the agent selects moves that are most likely to make Player 1 lose. This randomness aims to increase the variety of game states, encouraging Player 1 to face more unpredictable situations.

- **Blocking Player 1's Wins:** The agent also avoids moves that would result in an immediate win for Player 1, effectively preventing Player 1 from securing a quick victory.

This strategy is designed to maximize Player 1's chances of losing, particularly when the agent is at a disadvantage.

For this experiment, we used the best-performing AlphaZero agent from hour 70 (iteration 123) as Player 1 and pitted it against the adversarial player. A total of 100 games were played in total.

In Table 4.9, we observe that the adversarial agent, when playing as player



(a) Starting with player 1

(b) Starting with player 2

Figure 4.17: Evaluation of policy accuracy in random states against ground truth in 5×5 board.

Search Samples	P1 win %	P2 win %	Draws
32	44 ± 10	56 ± 10	0
64	58 ± 10	41 ± 10	1
128	54 ± 10	45 ± 10	1
256	88 ± 10	12 ± 6	0
512	86 ± 7	14 ± 7	0
1024	90 ± 6	7 ± 5	3
2048	93 ± 5	7 ± 5	0

Table 4.9: AlphaZero agent (P1) vs Adversarial agent (P2)

2, wins significantly more games than all previous players. This agent effectively pushes the AlphaZero player into unfamiliar states, leading to errors. Table 4.10 shows that the AlphaZero agent is pushed to slightly higher number of other states in comparison to seen and neighboring states in a game. Table 4.11 shows all the total number of states where AlphaZero agent made mistakes, where the agent made a move to a loosing state from a winning state. With stronger AlphaZero agents, the number of states where the agent makes mistakes decreases. Interestingly, the weaker AlphaZero agents were making more mistakes in neighboring states than other states. However, with increasing number of search samples, the error is corrected by the agent.

In stark contrast, when facing the strongest UCT player with a sample size of 2048, the AlphaZero agent wins all games. While the UCT player does steer AlphaZero towards unfamiliar states, as indicated by other column in 4.12, most of these lead to certain wins for player 1, as indicated by (1-

Search Sample	Total	Seen	Neighboring	Other (NW)
32	16.82	5.41	5.57	5.84 (4.69)
64	16.91	5.73	5.17	6.01 (4.86)
128	16.82	5.34	5.56	5.92 (4.58)
256	16.83	5.32	5.31	6.2 (4.63)
512	16.53	5.96	3.8	6.77 (5.45)
1024	17.57	5.89	4.61	7.07 (4.92)
2048	16.4	5.1	4.98	6.32 (4.22)

Table 4.10: Average number of states visited by AlphaZero agent (player 1) against an adversarial player in a game, and whether the states are in training set, neighboring set, or neither. NW refers to states in Other states where all legal moves are not win states for player 1.

Search Samples	Total	Seen	Neighboring	Other
32	56 (13)	4 (3)	44 (4)	8 (6)
64	41 (11)	4 (2)	28 (3)	9 (6)
128	45 (11)	0 (0)	35 (2)	10 (9)
256	12 (6)	5 (2)	0 (0)	7 (4)
512	14 (7)	0 (0)	3 (1)	11 (6)
1024	7 (6)	1 (1)	1 (1)	5 (4)
2048	7 (4)	3 (1)	0 (0)	4 (3)

Table 4.11: Number of states where AlphaZero agent made mistakes against an adversarial player, summed across all games. Numbers inside parenthesis refer to unique states. For example, the 6 (3) in Seen refers to 6 seen states (among which 3 were unique), where Player 1 made a move that led it to a losing state.

NW) column. The AlphaZero agent only errs in four unique states: once in a seen state and thrice in other states as indicated in 4.12. However, despite these errors, the UCT agent fails to capitalize on them, allowing AlphaZero to recover later in the game.

4.8 Supervised Learning and Self-play Training

In previous section, we were able to create an adversarial agent that was able to push AlphaZero agent into states from where its likely to lose the game. Can we train AlphaZero agent in such way that it is more robust from making

Metric	Total	Seen	Neighboring	Other	1-NW
States Visited	14.14	2.6	1.44	12.1	9.69
Mistakes Made	6 (4)	3 (1)	0 (0)	3 (3)	0 (0)

Table 4.12: Summary of AlphaZero agent's performance against UCT player. The first row represents the average number of states visited by the AlphaZero agent. The second row represents the number of states where AlphaZero agent made mistakes summed across all games.

such mistakes?

One approach is to initialize the model's weights during the first iteration of the game using a pre-trained model. Remember earlier we made a seen state dataset that comprised of states the AlphaZero agent has been trained on. We did a supervised learning on those states using ground truth from solver. We trained a supervised model till 100^{th} iteration, and used the models weight to initialize the first iteration of self-play during AlphaZero training. All the parameters were kept same as before during training.

Search Samples	P1 win %	P2 win %	Draws
32	73 ± 9	24 ± 8	3
64	71 ± 9	26 ± 9	3
128	91 ± 6	9 ± 6	0
256	86 ± 7	12 ± 6	2
512	87 ± 6	10 ± 6	4
1024	88 ± 6	70 ± 5	5
2048	87 ± 7	10 ± 6	3

Table 4.13: Pre-trained AlphaZero agent (P1) vs Adversarial agent (P2)

Table 4.13 shows that the AlphaZero agent, when initialized with supervised model weights, is able to win significantly more games with smaller searches. While increasing the number of search samples did improve the win rates against an adversarial player, the improvement was not significantly greater than that of an agent that did not utilize supervised learning model weights. Although this approach did not address the vulnerability, it suggests that supervised learning, followed by using the learned model weights to initialize the AlphaZero model, can slightly improve the agent's performance.

Moreover, it also suggests this method might improve the agents performance while doing transfer learning from smaller board size to larger board sizes. One can do supervised learning on smaller boards like 5×5, and use the models weights to initialize self-play training in 9×9 board.

4.9 Discussion

In this section, we discuss the key findings of our experiments.

1. **Parallel-thread self-play training is proportional to the number of self-play games**: In Section 4.5, we observed that the parallel play implementation was significantly faster than the single play implementation. After a week of training, the parallel play reached the 301^{st} iteration, while the single play reached only the 10^{th} iteration. This ratio reflects the number of CPU cores utilized to play the games, showing that the number of iterations is directly tied to the available computational resources.

2. **AlphaZero learns despite inaccuracies in the training set**: In Sections 4.5.3 and 4.6.2, we measured that only approximately 75% of the training data was accurate. Yet, AlphaZero was able to learn and win against all opponents (excluding adversarial player) as both the first and second player.

3. **Search contributes to the accuracy of the model**: In Sections 4.5.3 and 4.6.2, we also measured that increasing search sample in the game increased the accuracy of the model.

4. **There are states where AlphaZero performs poorly**: We verified through our experiments in Sections 4.5.3 and 4.6.2 that there are certain states where AlphaZero performs poorly.

5. **We designed an adversarial agent to exploit AlphaZero vulnerability**: In Section 4.7, we designed an adversarial agent that would

push AlphaZero agent into poor preforming states. As a result, the adversarial agent, as a second player, was able to win a significant number of games compared to other agents.

6. **Supervised learning before AlphaZero training can help improve model's performance against adversarial attacks**: We showed in Section 4.8 that performance of the model could be improved when we utilize the seen states to do a supervised learning with ground truth value from solver. We then used the supervised trained model to initialize the AlphaZero model during self-play and training. We showed that the AlphaZero agent, even with smaller search samples, is able to improve its win rates.

In this chapter, we covered the hardware and software setup for the experiment. We defined the concept of learning and discussed experiments on single-thread and parallel-thread self-play on 4×4 and 5×5 board sizes. Additionally, we designed an adversarial agent to exploit AlphaZero vulnerability, and experimented with a mix of supervised and AlphaZero training to create more robust AlphaZero agent.

Chapter 5

Conclusion

In this thesis, our primary motivation was to evaluate AlphaZero in a strongly solved game. We did this by using Chinese Checkers as our primary game environment.

First, we built and trained a custom AlphaZero agent for 4×4 and 5×5 Chinese Checker boards. We created three versions of AlphaZero, namely AlphaZero with single-thread self-play, single-thread self-play with an inference queue, and parallel-thread self-play with an inference queue.

Second, we measured how the learning progressed over time during training in AlphaZero. We did this by taking the learned agent and playing it against fixed opponents. We evaluated the win rate of the learned agent against a random player, a UCT player with various sample sizes, UCT with a distance evaluation function, and a perfect player. The win rate against these opponents gave us a way to measure whether the agent was learning over time.

Third, we measured the value of ground truth data in training and evaluating the AlphaZero model for Chinese Checkers. Since our work was primarily focused on solved 4×4 and 5×5 board sizes, we had access to a solver from which we could obtain the accurate outcome of a given game state. Access to this solver allowed us to implement a perfect player, against whom we evaluated the learned AlphaZero agent. Additionally, we created a dataset comprising states seen during training, neighboring states of the seen states, and carefully chosen random states, and we evaluated these against the ground truth values obtained from the solver. We then assessed the accuracy of the

agent in predicting the outcome of a given state. This was followed by evaluation of game policy both with and without search.

Lastly, we identified states where AlphaZero performs poorly, and designed an adversarial agents that pushed the AlphaZero agent into those states. We then devised a way to patch this vulnerability.

Important findings of our work are listed below:

1. Parallel-thread self-play training is proportional to the number of self-play games.
2. AlphaZero learns despite inaccuracies in the training set.
3. Search contributes to the accuracy of the model
4. There are states where AlphaZero performs poorly.
5. We designed an adversarial agent to exploit AlphaZero vulnerability.
6. Supervised learning before AlphaZero training can help improve model's performance against adversarial attacks.

5.1 Future work

Our work primarily focused on two board sizes: 4×4 and 5×5. Chinese Checkers is strongly solved up to a 6×6 board. As such, future work could extend our study to include this board. Also, future works can extend the work to commonly played board sizes like 7×7 and 9×9.

Although our intent was to evaluate the speed and performance of our custom AlphaZero implementation against other standard versions, we encountered a lack of open-source benchmarks for Chinese Checkers. Our investigation included OpenSpiel, a comprehensive set of tools for reinforcement learning research [18]. Unfortunately, an implementation of Chinese Checkers in OpenSpiel was not available at the time of our research. It is, however, on their development roadmap, allowing future studies to benchmark our method against their forthcoming implementation.

Since we identified and measured AlphaZero's poor performance in certain game states, designed an adversarial agent to exploit this vulnerability, and devised a way to patch this vulnerability, we could not win 100% of the games. Future work can investigate other approaches to mitigate this issue.

References

[1] L. V. Allis, "A knowledge-based approach of connect-four," *J. Int. Comput. Games Assoc.*, vol. 11, p. 165, 1988.

[2] L. Bottou, "On-line learning and stochastic approximations," in *On-Line Learning in Neural Networks*. USA: Cambridge University Press, 1999, pp. 9–42, ISBN: 0521652634.

[3] J. Bradbury, R. Frostig, P. Hawkins, *et al.*, *JAX: Composable transformations of Python+NumPy programs*, version 0.3.13, 2018. [Online]. Available: `http://github.com/google/jax`.

[4] C. B. Browne, E. Powley, D. Whitehouse, *et al.*, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012. DOI: `10.1109/TCIAIG.2012.2186810`.

[5] R. Carlisle, *Encyclopedia of Play in Today's Society*. Thousand Oaks, California: SAGE Publications, Inc., 2009. DOI: `10.4135/9781412971935`. [Online]. Available: `https://sk.sagepub.com/reference/play`.

[6] G. M. .-. Chaslot, M. H. Winands, and H. J. van Den Herik, "Parallel monte-carlo tree search," in *Computers and Games: 6th International Conference, CG 2008, Beijing, China, September 29-October 1, 2008. Proceedings 6*, Springer, 2008, pp. 60–71.

[7] M. Enzenberger, M. Müller, B. Arneson, and R. Segal, "Fuego—an open-source framework for board games and go engine based on monte carlo tree search," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 259–270, 2010. DOI: `10.1109/TCIAIG.2010.2083662`.

[8] W. Fedus, P. Ramachandran, R. Agarwal, *et al.*, "Revisiting fundamentals of experience replay," in *Proceedings of the 37th International Conference on Machine Learning*, ser. ICML'20, JMLR.org, 2020.

[9] S. Gelly and D. Silver, "Combining online and offline knowledge in uct," in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML '07, Corvalis, Oregon, USA: Association for Computing Machinery, 2007, pp. 273–280, ISBN: 9781595937933. DOI: `10.1145/1273496.1273531`. [Online]. Available: `https://doi.org/10.1145/1273496.1273531`.

[10] S. Gelly and D. Silver, "Monte-carlo tree search and rapid action value estimation in computer go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011, ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2011.03.007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S000437021100052X.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[12] R. Haque, "On the road to perfection? evaluating leela chess zero against endgame tablebases," M.Sc. thesis, University of Alberta, Edmonton, Alberta, Spring 2022. [Online]. Available: https://doi.org/10.7939/r3-5s4k-q622.

[13] R. B. Hayward and B. Toft, *Hex: The full story*. CRC Press, 2019.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[15] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].

[16] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293, ISBN: 978-3-540-46056-5.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, ISSN: 0001-0782. DOI: 10.1145/3065386. [Online]. Available: https://doi.org/10.1145/3065386.

[18] M. Lanctot, E. Lockhart, J.-B. Lespiau, *et al.*, "OpenSpiel: A framework for reinforcement learning in games," *CoRR*, vol. abs/1908.09453, 2019. arXiv: 1908.09453 [cs.LG]. [Online]. Available: http://arxiv.org/abs/1908.09453.

[19] N. Leach, "In the mirror of AI: What is creativity?" *Architectural Intelligence*, vol. 1, no. 1, Sep. 2022. DOI: 10.1007/s44223-022-00012-x. [Online]. Available: https://doi.org/10.1007/s44223-022-00012-x.

[20] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[21] J. McCarthy, "Chess as the drosophila of ai," in *Computers, Chess, and Cognition*, T. A. Marsland and J. Schaeffer, Eds., New York, NY: Springer New York, 1990, pp. 227–237, ISBN: 978-1-4613-9080-0.

[22] J. McCarthy *et al.*, "What is artificial intelligence," 2007.

[23] S. J. Russell and P. Norvig, *Artificial Intelligence: A modern approach*. Pearson Education Limited, 2022.

[24] J. Schaeffer, Y. Björnsson, A. Kishimoto, *et al.*, "Checkers is solved," *Science*, vol. 317, pp. 1518–1522, Oct. 2007. DOI: 10.1126/science.1144079.

[25] C. E. Shannon, "Xxii. programming a computer for playing chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 41, no. 314, pp. 256–275, 1950. DOI: 10.1080/14786445008521796. eprint: https://doi.org/10.1080/14786445008521796. [Online]. Available: https://doi.org/10.1080/14786445008521796.

[26] D. Silver, A. Huang, C. Maddison, *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan. 2016. DOI: 10.1038/nature16961.

[27] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[28] N. Sturtevant, "On strongly solving chinese checkers," in Dec. 2020, pp. 155–166, ISBN: 978-3-030-65882-3. DOI: 10.1007/978-3-030-65883-0_13.

[29] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, PMLR, 2013, pp. 1139–1147.

[30] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, Second. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html.

[31] T. T. Wang, A. Gleave, T. Tseng, *et al.*, *Adversarial policies beat superhuman go ais*, 2023. arXiv: 2211.00241 [cs.LG].

Appendix A

Key Metrics During Training on 5×5 Board

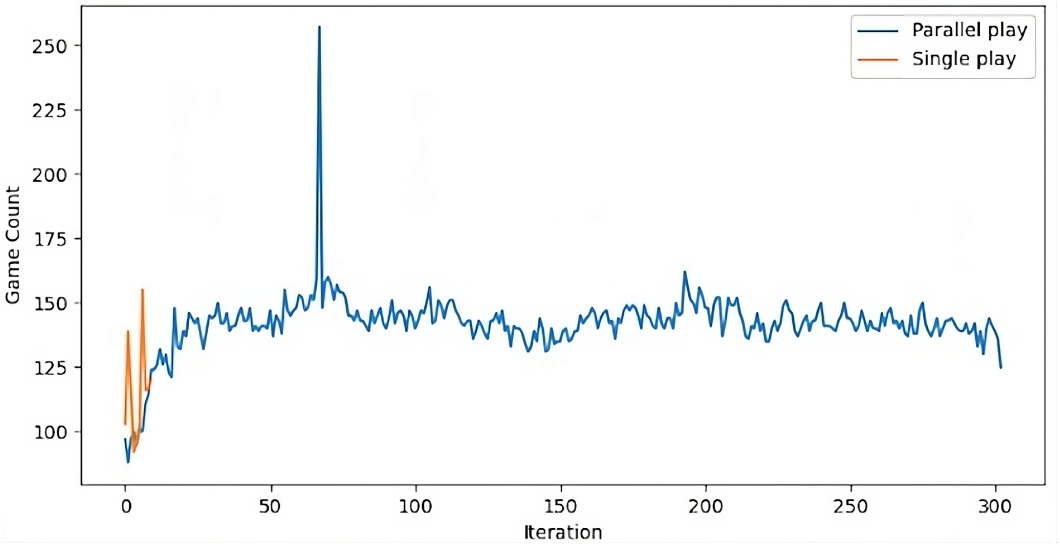


Figure A.1: Number of games played in each of the 300 iterations.

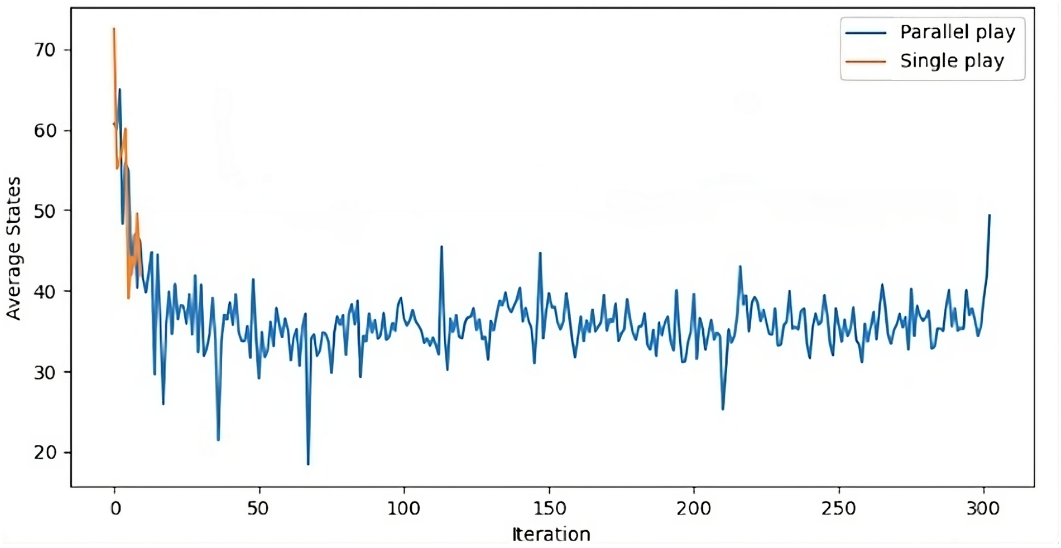


Figure A.2: Average number of states visited during each of the 300 iterations.

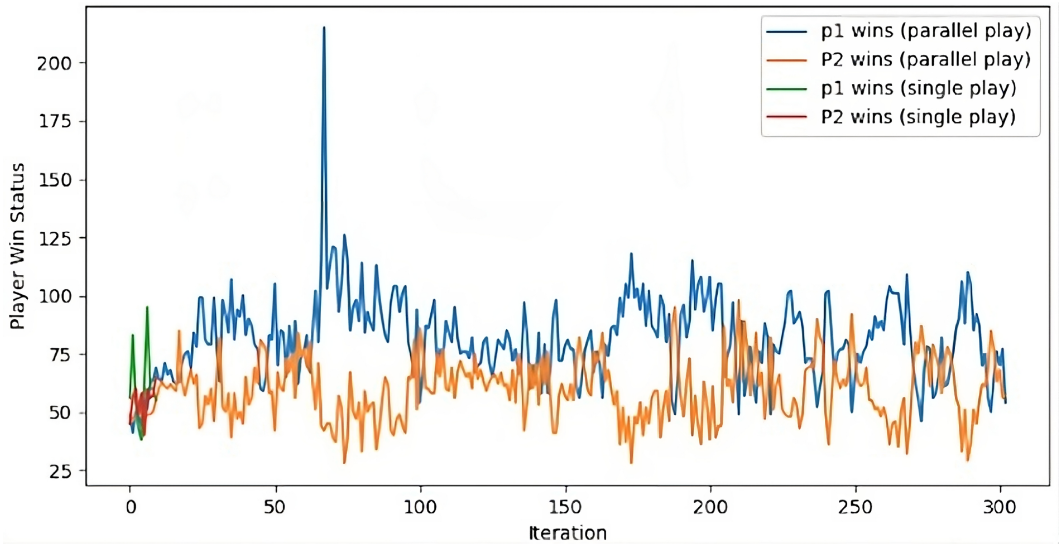


Figure A.3: Win rates of player 1 and player 2 in each of the 300 iterations.

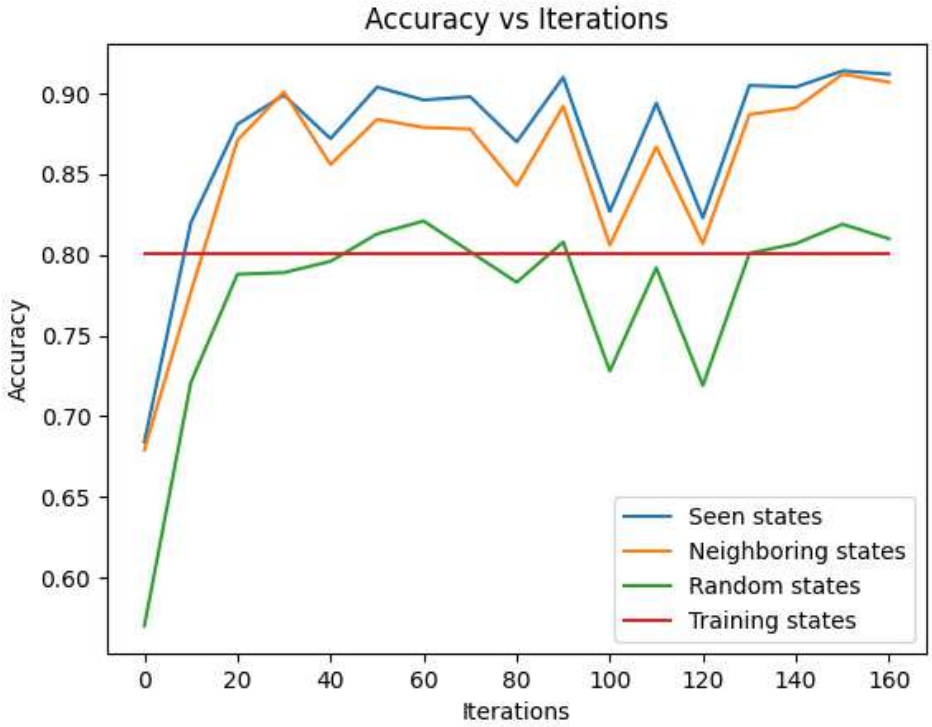


Figure A.4: Outcome accuracy on second run of experiment on 5×5 board.

	4×4 board	5×5 board
Hours	5	72
Parallel self-play games	6	29
Replay buffer size	1,000	10,000
MCTS search sample	128	128
s in %	100	50
Batch size	64	128
Learning rate	1×10^{-5}	1×10^{-5}
Momentum rate	0.9	0.9
Weight decay	0.0001	0.0001

Table A.1: Hyper-parameters used during self play games and training.

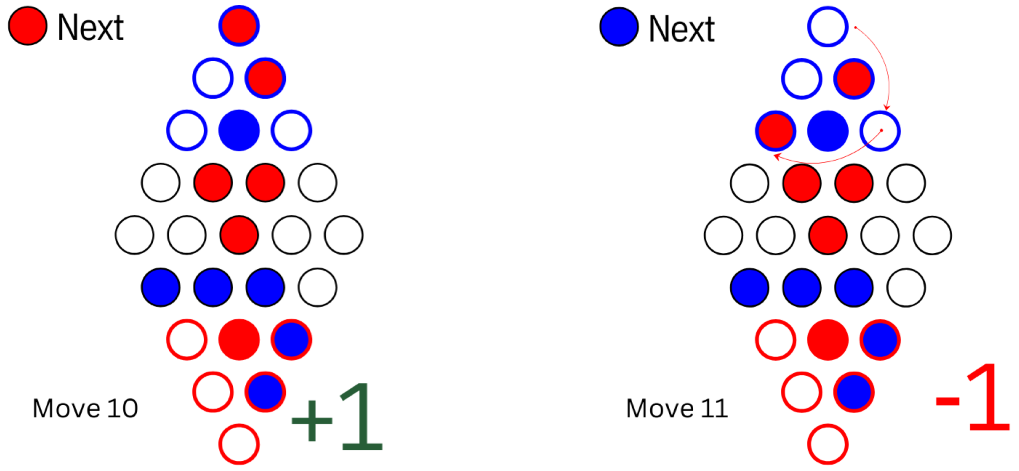


Figure A.5: AlphaZero made mistake on 11^{th} move while playing against Adversarial agent in Game 35.

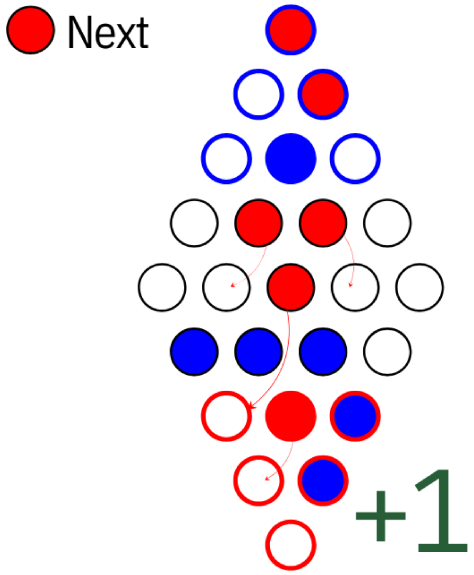


Figure A.6: Potential 11^{th} moves that could have lead AlphaZero to a winning state in Game 35.