Analyzing KataGo: A Comparative Evaluation Against Perfect Play in the Game of Go

by

Asmaul Husna

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

Research on board games focuses on playing at a superhuman level or finding exact solutions. Recently, Artificial Intelligence (AI) has become really good at playing complex games such as Go. Comparing AI systems to perfect play helps us understand how advanced AI has become. This research explores the performance of KataGo, an AlphaZero-like program, in the game of Go. Our study investigates how different neural networks and search strategies impact KataGo's decision-making abilities when compared against perfect play. In our research, we develop a larger Go endgame dataset labelled with perfect solutions, and examine KataGo's strengths and weaknesses through experiments and analysis. We observe the effectiveness of strong policies in improving move selection, the benefits and demerits of MCTS search enhancements, and the challenges KataGo faces in competing against an exact solver. We further analyse move choices by showing the changes of average action values, lower confidence bound (lcb), winrate, and number of visited node according to MCTS search in KataGo. KataGo has a 90.8% success rate while playing matches against an exact solver in the perfect game dataset.

Preface

This thesis entitled "Analyzing KataGo: A Comparative Evaluation Against Perfect Play in the Game of Go" represents my original and unpublished research work conducted under the supervision of Dr. Martin Müller. We plan to submit a shorter version of this work in the Computers and Games conference (CG 2024).

Acknowledgements

First and foremost, I express my deepest gratitude to the Almighty for helping me throughout this journey.

Then, I would like to thank my supervisor, Dr. Martin Müller, for his incredible support and guidance. I am truly grateful for his kindness, helpfulness, and valuable feedback on my thesis. I also want to acknowledge Dr. Ting Han Wei for his insightful discussions and support.

I want to thank my parents, my sons, Afnan Habib Ayan and Awwab Muhammad Habib, and all my family members and friends, whose love and support have been crucial of my journey.

I am thankful to the University of Alberta, the Department of Computing Science, Alberta Machine Intelligence Institute (Amii), and Digital Research Alliance of Canada for providing the necessary resources and opportunities for my research.

Lastly, I want to express my love and appreciation to my husband, Habibur Rahman, for his continuous encouragement and unconditional support during challenging times.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Board games are considered as a platform for exercising strategic thinking, decision-making, and problem-solving skills. Humans can naturally play board games and enjoy the intellectual challenge they provide. Players must analyze the game's current state, evaluate available options, and anticipate the potential outcomes of their decisions. Humans generally rely on heuristics and inference to make these types of decisions. Nowadays, Artificial Intelligence has revolutionized the way board games are played.

The game of Go has been a fascinating subject of study for both human and AI players. The complexity of the game, with its vast number of possible moves, has made it an excellent platform for testing the limits of human and AI. With the advent of deep neural networks with tree search, in recent years, AI has shown superhuman performance in playing Go, with DeepMind's AlphaGo [20] becoming the first AI program to defeat a top human professional player in 2016. After this success, DeepMind developed Alphazero [21], a program which can master Go, chess, and shogi without human knowledge. Despite their remarkable performance, these AlphaZero-like programs are not perfect and can still make mistakes. We want to understand how these modern programs learn to play the game of Go and explore the limits of their playing abilities. In this work, we investigate the performance of the AlphaZero-like program KataGo against perfect play using Go endgame puzzles [7], and extend these endgames puzzles to facilitate further exploration and analysis.

1.1 Motivation of the Research

AlphaZero surpassed the best human players, and even its predecessors, AlphaGo and AlphaGo Zero. Currently, AlphaZero-like programs dominate board games by achieving remarkable performance without human knowledge. The performance is measured by winning or losing games against opponent players. The success of AlphaZero and its variants in Go has had a profound impact on the field of artificial intelligence and game-playing algorithms. It showcases the power of combining search and deep neural networks with reinforcement learning and self-play to achieve exceptional performance in complex domains.

The potential applications of the AlphaZero algorithm extend beyond games. In areas like self-driving cars, where even a tiny mistake can cause big problems, having accurate solutions is extremely important. However, it is still an open question whether AlphaZero-like algorithms can be adapted to find exact solutions. To address this question, it is essential to explore the learning process of these modern programs. In games, exact solution of an endgame refers to the theoretically optimal, perfect play result that each player can achieve. Recently, Haque, Wei, and Müller analyzed how far a AlphaZero-like program Leela Chess Zero (Lc0) is from perfect play, by comparing its decisions with perfect ones from chess endgame tablebases [9]. They found some interesting mistakes made by the algorithm. Their research motivates us to do a similar analysis on the game of Go. We choose Go as it is the most challenging board game, and unlike chess, the number of endgame positions is not reduced after capturing. One of the main obstacles to doing this research on Go is that there are no endgame tablebases for this game.

Therefore in this thesis, we solve endgame problems using an exact solver [14] and compare an AlphaZero-like program KataGo in Go against perfect play of these problems. We select the endgame puzzles from Berlekamp and Wolfe[7] as the author demonstrated the effectiveness of his theory by setting up a plausible endgame position from which he beat one of the Japanese champions of the game, after which he set up the same position, reversed the

board, and beat the master a second time[1]. This analysis provides insights into the strengths and limitations of KataGo in achieving exact solutions.

1.2 Research Questions

Our research aims to address the following questions:

- What are the differences in move selection between stronger and weaker neural networks?
- How does the addition of a small search enhance move selection compared to using raw neural networks?
- What is the impact of increasing the search budget on the overall performance of the algorithm?
- How good is KataGo at finding the best incentive move compared to finding minimax optimal move?
- How does KataGo perform depending on the size of the endgames?
- What is the performance of KataGo compared to an exact solver when playing matches using endgame starting positions?
- Are there any cases where using deeper search adversely affects move selection compared to using small search?

1.3 Contribution of the Thesis

Our work provides the following key contributions:

- We develop a larger dataset based on the endgame positions in [7] and [13], analyze it, and evaluate the performance of KataGo [27] against perfect play.
- We show how using a small search helps correct errors made by weak and strong neural networks.

[1] Article is available at *https* : *//celebratio.org/Berlekamp_ER/article/*730/

- We demonstrate the behaviour of KataGo from small to large number of searching nodes and identify its strengths and weaknesses.
- We evaluate KataGo's ability to select best incentive move compared to select minimax optimal move. We also show the performance of KataGo in endgames of different size.
- We show a scenario where KataGo loses against an exact solver in a simple endgame.
- We investigate the cases where a small search correct but longer search wrong, as well the cases which are not solved by KataGo's policy and search. We observe that the move with highest policy can be a least favorable move for KataGo after the search.

Ultimately, this research contribute to our understanding of the capabilities of modern programs like KataGo and their potential applicability in domains where exact solutions are crucial. By gaining insights into their performance relative to perfect play, we can evaluate the feasibility of employing AlphaZero-like algorithms in fields such as drug design [10], [30] and safety-critical systems like autonomous vehicles [16], where the cost of deviations from exact solutions is high.

1.4 Organization of the Thesis

The remainder of this thesis is organized as follows:

- **Chapter 2.** ***Literature Review:*** We review the basic concepts underlying this research.
- **Chapter 3.** ***Experimental Setup:*** We describe our dataset and the hardware and software setup along with the neural networks used. We discuss the engine settings which ensure the reproducibility of our results.
- **Chapter 4.** ***Details of Experiments and Analysis:*** The details of all experiments and the experimental results along with our analysis are given in this chapter.

- **Chapter 5.** ***Conclusion and Future Work:*** Concluding remarks about the research are stated and some future research directions based on our findings are proposed.

We use grammarly[2] and chatgpt[3] to improve the writing.

[2]https://app.grammarly.com/
[3]https://chat.openai.com/

Chapter 2

Literature Review

This chapter gives an overview of the essential background material and related work relevant to this thesis. We focus on two-player perfect information zero-sum games such as Go. So all the terminology we use is for these types of games.

2.1 Game Tree Search

Game tree search is a fundamental technique used in artificial intelligence to make good, or even optimal, decisions in sequential games. It involves exploring the possible moves and outcomes of a game by constructing a tree structure known as a game tree. The game tree represents all possible game states and the legal moves that can be taken at each state. Each node represents a specific game state, and the edges represent valid moves that transition from one state to another.

A game tree starts with a root node which represents the initial game state. The tree branches out to represent all possible sequences of moves that the players can make. Each level of the tree corresponds to a player's turn. Each branch represents a different legal move, and the subsequent nodes represent the resulting game states after the move is made, where it is the opponent's turn. Terminal nodes indicate the end of the game, and each has an associated outcome; for example, win (+1) or loss (-1).

A search tree is a subset of the full game tree. A terminal node holds a true game outcome and a non-terminal node can hold a heuristic value. If

we use +1 as evaluation for a win and -1 for a loss, the heuristic values are in the range (-1, +1). Many algorithms, such as minimax and alpha-beta pruning, can be employed to search a game tree and make decisions. These algorithms evaluate the nodes in a search tree with the backed-up value from search algorithms in interior nodes, and eventually in the root node. In the following subsections, we review some of the most popular algorithms.

2.1.1 Minimax and Alpha-Beta Search

The minimax algorithm is commonly used for two-player, zero-sum games, where the goal is to maximize the player's outcome, which is the same as minimizing the opponent's outcome. It is a depth-first search, which assigns values to the leaf nodes of the game tree and then propagates the values upward through the tree. It evaluates each parent node by taking the maximum value of the children when it is the player's turn, and the minimum when it is the opponent's turn.

Figure 2.1 shows an example of minimax game tree search [18]. At the root node of the game, MAX has three possible moves: $a1, a2$, and $a3$. In response to $a1$, MIN can play: $b1, b2$, or $b3$. The game concludes after both MAX and MIN have made one move each. This means the game tree is two moves deep, also referred to as a *ply*. In this game, the terminal states have outcomes that vary between 2 and 14. After analyzing the tree, we find that MAX's optimal move at the root is $a1$, as it leads to the state with the highest minimax value of 3. MIN's best response at node B is $b1$, as it leads to the state with the lowest minimax value of 3.

The naive minimax algorithm performs a depth-first search that explores the entire game tree. In a popular simple model, there are b legal moves at each node, and search reaches a depth of d. As a result, the complexity of the algorithm can be expressed as $O(b^d)$ [18]. However, for nontrivial games such as Go or chess, this computational cost becomes impractically large as they have a large branching factor and depth.

The alpha-beta search algorithm improves the efficiency of minimax, reducing the complexity of the game tree traversal. In the best-case scenario, it

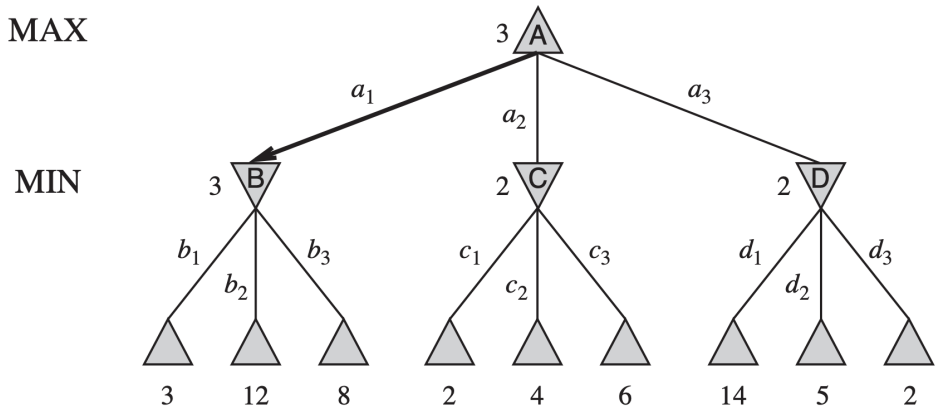


Figure 2.1: A two-ply game tree [18].

can cut the exponential complexity of the minimax game tree from $O(b^d)$ to $O(b^{d/2})$. The key idea is to keep lower and upper bounds (α, β) on the true minimax value, and prune a position if its value v falls outside the window

- $v < \alpha$: We will avoid it. We have a better alternative.
- $v > \beta$: Opponent will avoid it. They have a better alternative.

The algorithm can be easily understood with the example in Figure 2.2. In this example, the algorithm determines the minimax value of the root node without evaluating two of the terminal nodes. This illustrates how alpha-beta search can identify and ignore branches that can not affect the final value of the root.

2.1.2 Monte Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a popular algorithm used to make decisions in games or other domains with high branching factors and uncertainty [6], [8], [20]. It combines elements of random sampling (Monte Carlo) and tree search to build a selective search tree that represents possible states and actions in a given game or problem. The innovative combination of tree search and Monte Carlo policy evaluation played a significant role in the remarkable achievements of successful Go programs. MCTS with an enhanced move and state evaluation algorithm is also used in AlphaGo and AlphaZero [23].

MCTS consists of four steps, repeated as long as there is time left:

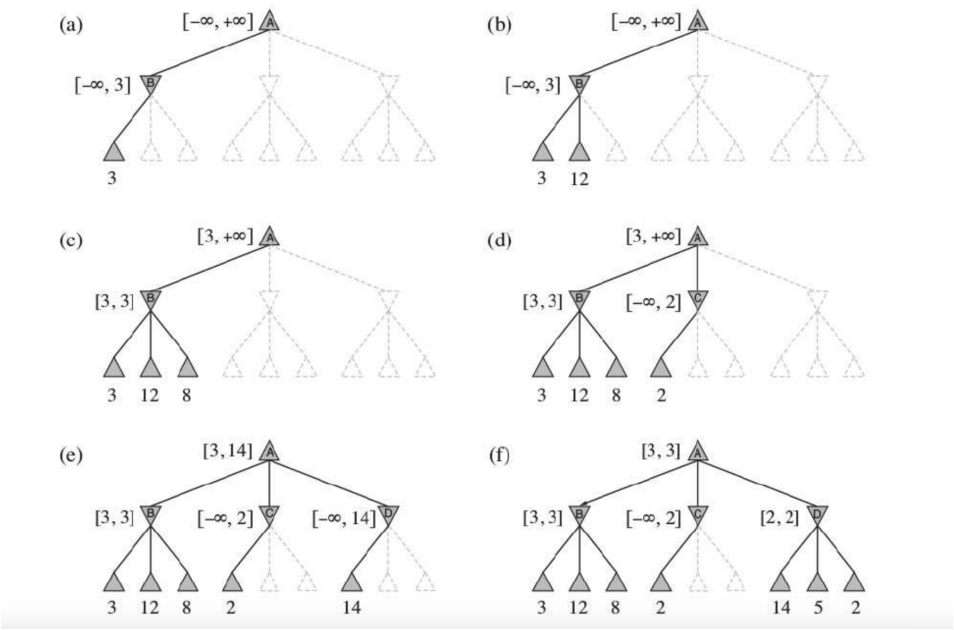


Figure 2.2: Determining the minimax value using alpha-beta search [18].

- **Selection:** Each iteration of the MCTS algorithm starts by selecting a leaf node in the current search tree. Starting from the root node that represents the current state of the game, the algorithm selects successive child nodes.

- **Expansion:** Expansion determines whether nodes should be added to the tree. Unless the leaf node is terminal and gives the game's outcome, the algorithm expands it by adding one or more child nodes to the tree. Each child node represents a state after a valid move.

- **Simulation or Play-out:** After expansion, the algorithm performs a simulation from the newly added child node. The randomized simulation selects actions until the game ends. A simulation is sometimes referred to as a playout or rollout.

- **Backpropagation:** Once the simulation is complete, the results are backpropagated through the tree. Starting from the expanded node, the algorithm updates the statistics of all nodes along the path to the root. Such statistics typically include the number of visits to the node and the accumulated rewards, such as the number of wins.

These steps (selection, expansion, simulation, and backpropagation) are repeated until a termination condition such as time or number of simulations is met. The algorithm gradually builds up knowledge about the game and in the limit, converges towards a solution. Figure 2.3 shows one iteration of the general MCTS approach [8]. By continuously expanding the search tree and refining its estimates of the values of different actions, MCTS can make decisions even in complex domains with large state spaces and uncertain outcomes, such as the game of Go.

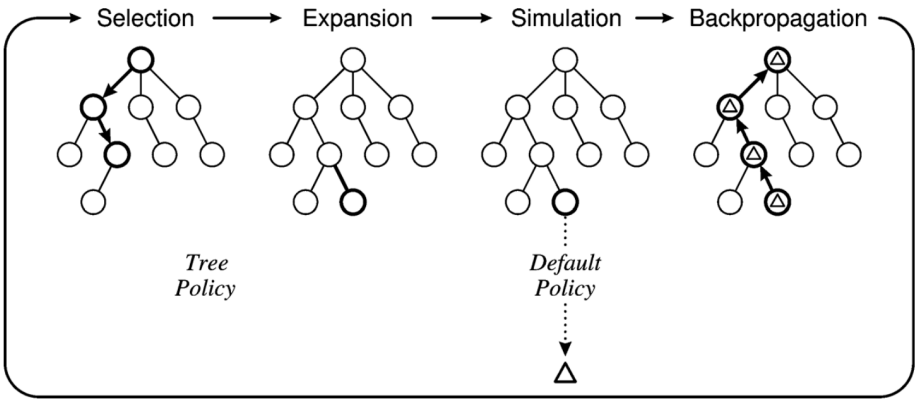


Figure 2.3: One iteration of MCTS [8].

For selecting a child node, the main difficulty is balancing exploration and exploitation effectively. Upper Confidence Bounds for Trees (UCT) [11] is a commonly used strategy within MCTS to achieve this balance. UCT is based on the Upper Confidence Bound (UCB) policy proposed by [5] which is called UCB1. The UCT algorithm assigns values to nodes in the search tree based on a trade-off between two factors: the estimated value of the node (exploitation) and the degree of exploration of less explored nodes. This balance allows the algorithm to explore new possibilities while also prioritizing the exploitation of promising branches.

The UCT formula calculates a value for each child node of a given parent node. The value of child node v, denoted as $UCT(v)$, is computed as:

$$UCT(v) = \overline{X}(v) + C\sqrt{\frac{ln(N(p))}{N(v)}} \tag{2.1}$$

In this formula, $\overline{X}(v)$ represents the estimated value (average reward or win rate) of the child node v, $N(p)$ is the number of times the parent node has been visited, $N(v)$ is the number of times the child node has been visited, and C is a constant that determines the balance between exploration and exploitation. The term $\sqrt{\frac{ln(N(p))}{N(v)}}$ represents the exploration component, while $\overline{X}(v)$ represents the exploitation component.

The exploration component in the UCT formula ensures that less explored nodes are given a higher priority for selection, as the square root of the logarithm of the parent's visit count over the child's visit count is large with less exploration. The exploitation component $\overline{X}(v)$ is updated during the back-propagation phase of MCTS, where the results of simulations are propagated back up the tree.

2.1.3 Solving Vs Playing a Game

Solving a game refers to finding an optimal strategy that guarantees a win or, in some cases, at least a draw, regardless of the opponent's moves. This concept is most applicable to games with perfect information, where all the game states and possible moves are known to the players. The primary goal of solving a game is to find a strategy that will lead to a victory. Achieving a solution for a game typically involves using algorithms, mathematical techniques, and computational power to explore possible moves and counter-moves until a winning strategy is identified. However, it's important to note that not all games are solvable in practice, especially those with a large number of potential positions and complex interactions. Games like Go fall into this category.

Figure 2.4 represents an endgame scenario in Go. We use this endgame position to play matches between an exact solver [14], which can solve this endgame position perfectly, and KataGo, a renowned open source Go program inspired by AlphaZero's approach. Figure 2.5 (a) shows a move sequence from the starting position to the end of the match, where the exact solver plays as white and KataGo with default settings plays as black. At the end of this game, the exact solver wins by 0.5 points. Then in Figure 2.5 (b) we play the match again after interchanging the color of the players. The exact player

beats KataGo by 0.5 points again.

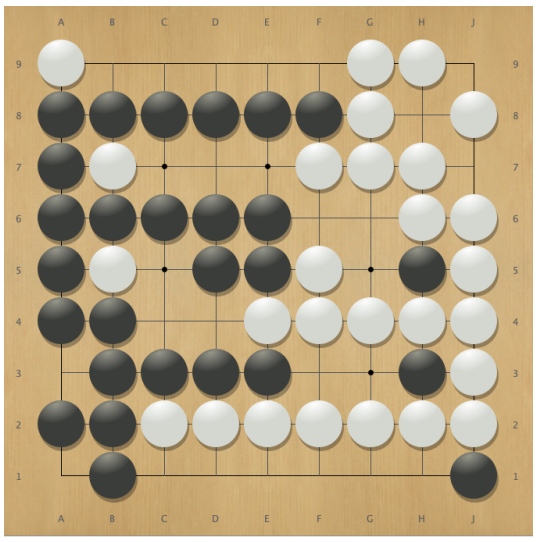


Figure 2.4: An endgame position for playing matches between a solver and a open-source Go bot [7].

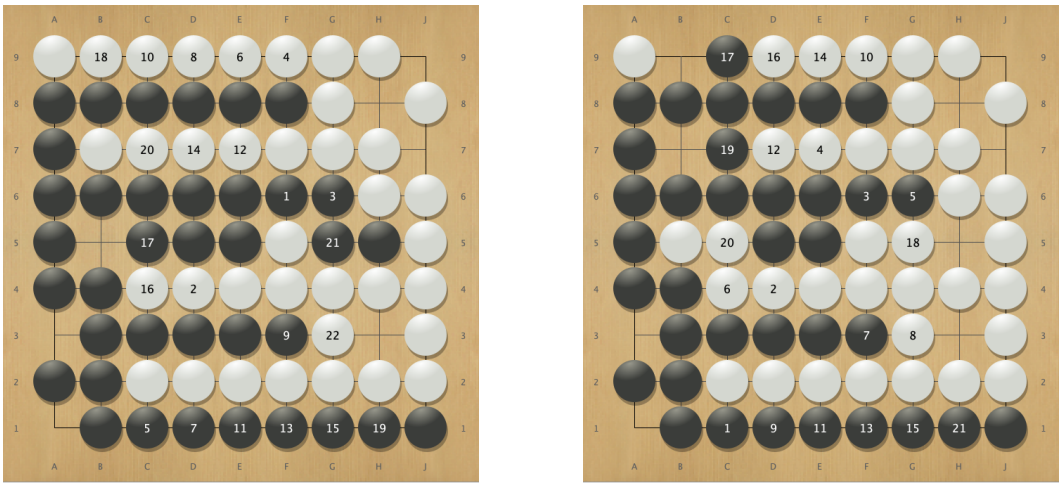
2.2 AlphaGo and AlphaZero

AlphaGo, AlphaGo Zero and AlphaZero, and the open-source program *KataGo* [27] are remarkable artificial intelligence systems that are really good at playing strategic board games. This section explores all of them.

2.2.1 AlphaGo

AlphaGo [20] is the first program where the authors use deep convolutional neural networks for effective move selection and position evaluation functions for the game of Go. A novel combination of supervised and reinforcement learning trains the neural networks. This program made history by defeating Lee Sedol, one of the world's top Go player in 2016.

The schematic representation of the neural network training process is shown in Figure 2.6. Neural networks begin to learn from a training phase, where a supervised learning (SL) policy network, p_σ, is directly trained from expert human moves. Additionally, a fast but less accurate rollout policy, p_π, is also trained to select moves during rollout. The policy networks alternate



(a) Exact solver wins as white against KataGo.

(b) KataGo loses as white against exact player.

Figure 2.5: Two games between exact solver and KataGo from the starting position in Figure 2.4.

between convolutional layers with weights and rectifier nonlinearities. The input they receive is a basic representation of the game board, and they produce a probability distribution of legal moves through a final softmax step.

In the second stage of training, a reinforcement learning (RL) policy network, p_ρ, is initialized using the SL policy network's weights. Through policy gradient learning, this RL policy network is subsequently improved to maximize outcomes (winning more games) by playing games with randomly selected previous versions of the policy network. This RL policy network is used to generate a self-play dataset of games. Finally, a value network v_θ is trained by regression to predict the expected outcome in positions from the self-play dataset.

With the combination of the policy and value networks in an MCTS algorithm, AlphaGo selects actions by lookahead search. Each edge (s, a) of the search tree stores an action value $Q(s, a)$, a visit count $N(s, a)$ and a prior move probability $P(s, a)$. A variant of UCT is used for selection. The tree is traversed by simulation starting from the root state. At each time step t of each simulation, an action a_t is selected from state s_t such that [20]:

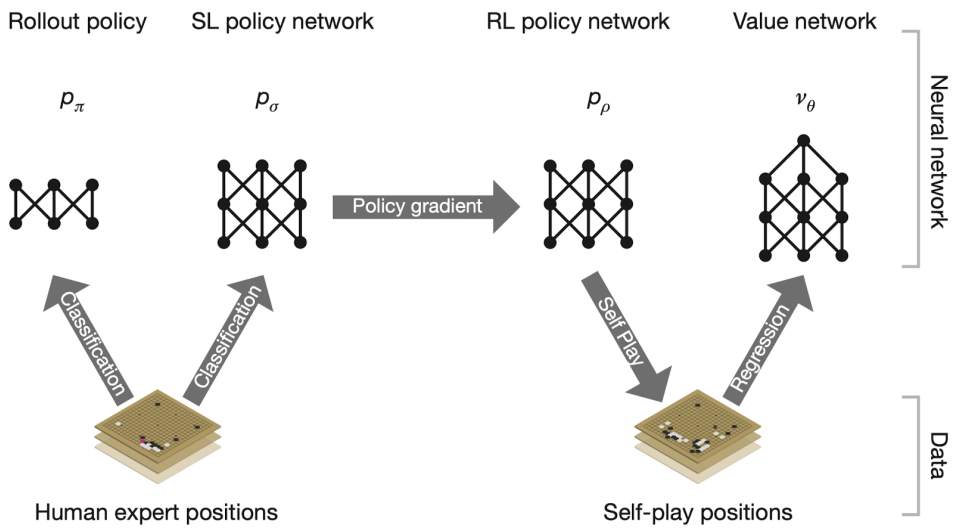


Figure 2.6: A schematic representation of the neural network training in AlphaGo [20].

$$a_t = \arg\max_a(Q(s,a) + u(s,a)) \tag{2.2}$$

where

$$u(s,a) \propto \frac{P(s,a)}{1+N(s,a)},$$

Figure 2.7 shows the process of Monte Carlo Tree Search (MCTS) within the context of AlphaGo [20]. For selection, it picks the edge with highest sum of action value Q and bonus $u(P)$ that depends on prior probability P stored on that edge and is used for exploration. After expansion, the policy network p_σ processes the new node once and the output probabilities are stored as prior probabilities P for each action. At the end of a simulation, the leaf is evaluated using the value network v_θ and running the fast rollout policy p_π. Then the reward r is computed for the current player, $+1$ for winning and -1 for losing. Finally action values Q are updated to keep tracking the mean value of the rewards and expected outcomes in the subtree below that action.

2.2.2 AlphaGo Zero and AlphaZero

AlphaGo Zero [22] is the first AI model for the game of Go that learns entirely from scratch, without any human-provided data or knowledge beyond game

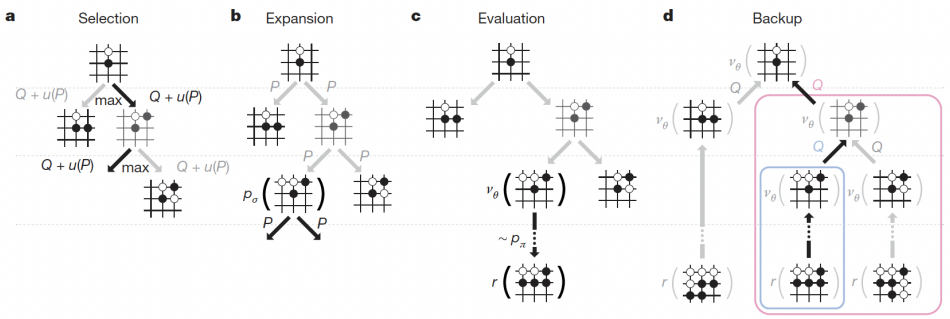


Figure 2.7: Monte Carlo Tree Search in AlphaGo [20].

rules. It differs from AlphaGo in several important aspects. Unlike AlphaGo, it uses only black and white stones from the board as input feature and it has only one neural network, rather than two seperate networks for policy and value. This approach makes the tree search simpler that relies upon only this single neural network to evaluate positions and sample moves without performing any Monte Carlo rollouts.

The neural network is trained using a self-play reinforcement learning algorithm combined with MCTS for each move. Figure 2.8 shows the training pipeline in AlphaGo Zero. During self-play, the program competes against itself, moving through positions $s_1, ..., s_T$ in a game. At each position, MCTS in Figure 2.9 is employed with the most recent neural network f_θ. MCTS is similar to the one used in AlphaGo, except there is no rollout, just a single value evaluation by the neural network. Moves are chosen based on the search probabilities calculated by MCTS. When a terminal position s_T is reached, the outcome is determined according to the game's rules, computing the winner z. The neural network starts with a random weight initialization which receives the raw board position as input and passes it through numerous convolutional layers parameterized by θ. The outputs are both a vector p_t, indicating a probability distribution over moves, and a scalar value v_t, representing the probability of the current player winning in this position. The parameters θ are adjusted to achieve two main objectives: enhancing the similarity between the policy vector p_t and the search probabilities π_t, and minimizing the error between the predicted winner v_t and the game winner z.

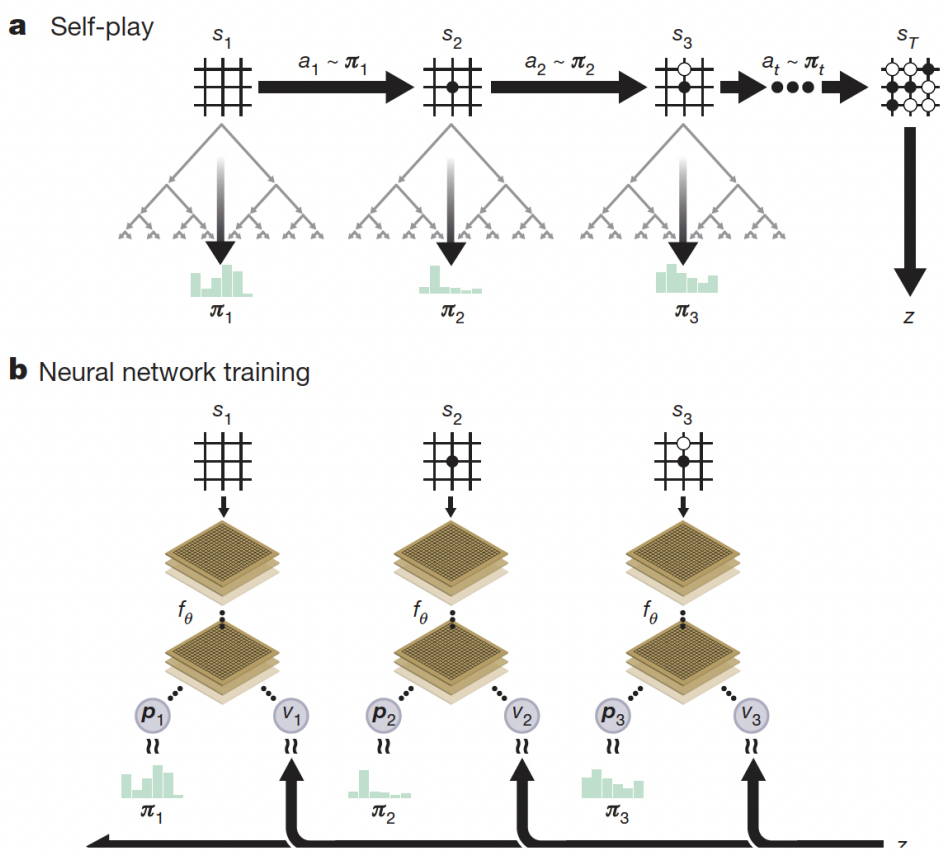


Figure 2.8: **a)** Self-play reinforcement learning in AlphaGo Zero. **b)** Neural network training in AlphaGo Zero [22].

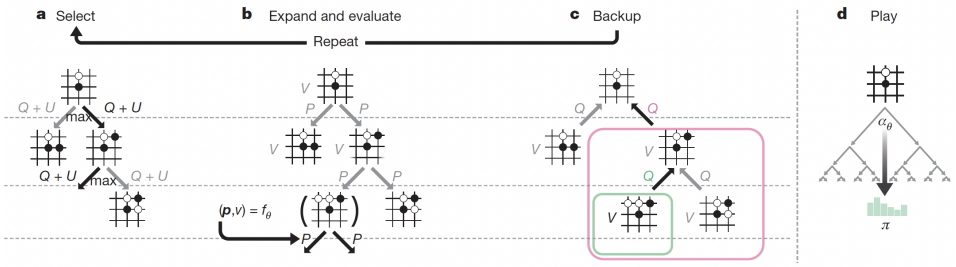


Figure 2.9: Monte Carlo Tree Search in AlphaGo Zero [22].

The AlphaZero [21] algorithm is built upon the foundation of AlphaGo Zero but its capabilities are expanded to achieve superhuman performance in several challenging games. The AlphaZero algorithm differs from the AlphaGo Zero algorithm. AlphaZero focuses on estimating and optimizing the expected outcome rather than the probability of winning. During neural network training, a key difference is that AlphaZero updates the neural network continually without waiting for an iteration to complete. Despite these differences, AlphaZero uses the identical convolutional network architecture found in AlphaGo Zero for chess and shogi as well as Go. Only the input is adapted to each game.

2.2.3 KataGo

KataGo [27] is an open-source Go program which implements the AlphaZero algorithm for the game of Go with several improvements to accelerate the self-play learning. It uses some domain-specific features and optimizations but it still starts from random play and makes no use of outside knowledge except for the game rules.

KataGo makes two important contributions. First, it implements a variety of domain-independent improvements that can be used in other learning methods like AlphaZero. These include techniques to balance data better, improve training by focusing on important parts, and enhance the neural network by adding global pooling layers at various points. Additionally, the authors find the usefulness of adding some game-specific input features to improve learning. The program serves as a case study that there is still a big difference in efficiency between AlphaZero's methods and what can be achieved through self-play. For this, it uses methods specific to Go, such as predicting ownership and scores, which are very helpful.

After these improvements, KataGo surpasses ELF OpenGo's final model, an open-source AlphaZero-like Go project [24], after only 19 days training on fewer than 30 GPUs while ELF required thousands of GPUs over two weeks. It is also stronger than Leela Zero 0.17 [1], another AlphaZero-like open-source Go bot.

2.3 Go Endgame Puzzles

In endgame puzzles in the game of Go, most of the board is filled up, and players focus on small but important local battles to protect their territory and reduce their opponent's. At this stage, there are fewer opportunities for large territorial gains, and the outcome of the game can be decided by just a few points.

Figure 2.10 (left) shows an endgame puzzle from [7]. This puzzle is designed for White to be the first player and if both players (Black and White) play perfectly then White wins by 0.5 points. We use the exact solver [14] to get a correct sequence of moves for this endgame. In Figure 2.10 (right) using Japanese rules to count the score Black has 9 and White has 10. With Komi −0.5, White wins by 0.5 points.

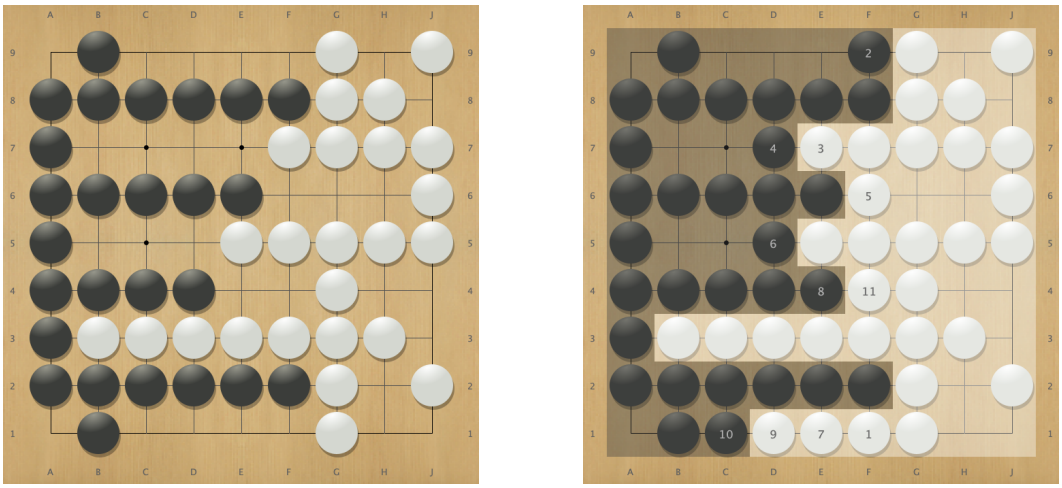


Figure 2.10: An endgame puzzle (left) with White to play and Komi -0.5. The right figure shows an 11 move optimal solution.

2.4 Exact Solver for Go

An exact solver for the game of Go can accurately determine the optimal moves and outcomes for a given Go position. Creating such a solver is a challenging task due to the huge complexity of Go and the large number of possible moves and positions. In our work, we use a solver [13] which can solve a restricted

class of endgame problems with solution lengths exceeding 60 moves. It uses a method called decomposition search [14] for solving combinatorial games. This approach is applied to Go endgame problems to find the optimal move.

2.4.1 Combinatorial Games

Combinatorial games are two-player games, where both players have full knowledge of the game state and available moves. They are turn-based, with players taking alternating turns. We consider only games that have a finite duration. Combinatorial games are structured in a way that allows them to be broken down into subgames, which can be analyzed independently. This decomposition is particularly useful for games that can be viewed as a sum of independent subgames. Each move made by a player corresponds to a move within one of the subgames while leaving all other subgames unchanged. The game concludes when all subgames have reached their end, leading to the final outcome. A well-known example is *Nim* which is played with several heaps of tokens. At each move, the player takes any number of tokens from a single heap. Whoever removes the last token overall is the winner. Figure 2.11 shows an example of a *Nim* position with three heaps and its subgames [14]. Here, each heap is treated as a separate subgame and the overall *Nim* game is the sum of these three heaps/subgames.

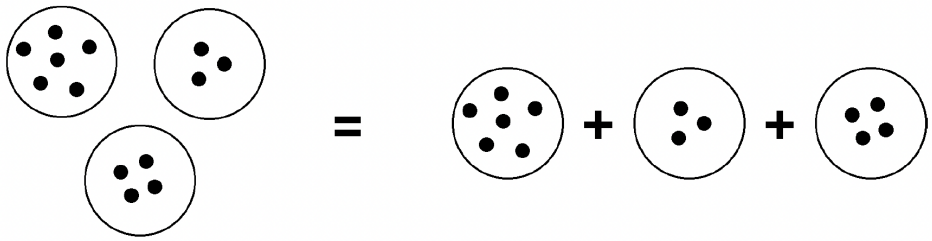


Figure 2.11: A *Nim* position and its subgames [14].

2.4.2 Decomposition Search

Decomposition search is a game-solving framework that utilizes a local search approach called Local Combinatorial Game Search (LCGS) [14]. It aims to find the optimal play in a game that can be broken down into subgames.

At first, this approach identifies subgames within the main game. For the game of Go, this requires game-specific knowledge to partition the game effectively by recognizing safe stones and territories. Safe stones cannot be captured by the opponent. These stones have sufficient sure liberties (adjacent empty intersections that the opponent cannot fill) to ensure their safety. Safe territories are areas on the board completely surrounded by safe stones of one player (either black or white), where the opponent cannot live inside. They are considered as finished subgames with an integer value. Figures 2.12 and 2.13 show the safe stones and territories, and the 7 subgames after decomposing a Go endgame position. Here, the blue squares represent the safe stones and territories, and the capital letters A, B, C, D, E, F, G indicate the subgames which are the connected components of the remaining points of this Go board.

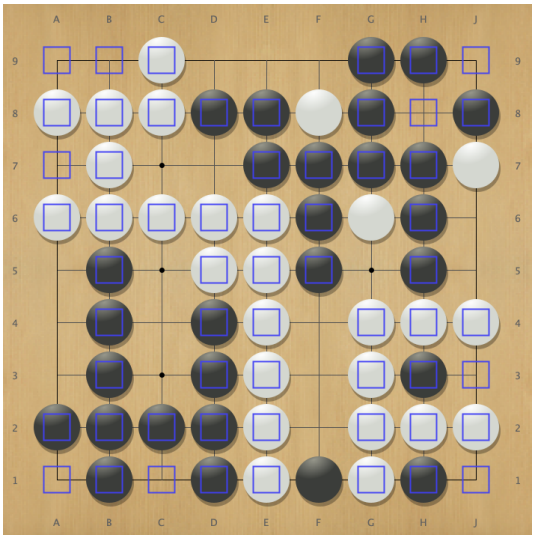


Figure 2.12: Safe stones and territories of a Go endgame position.

Next, decomposition search applies local combinatorial game search (LCGS) seperately on each subgame to find its game graph. LCGS is the main information-gathering step of decomposition search [14]. It constructs a game graph that represents all relevant move sequences that can be played locally. Unlike minimax tree search, LCGS incorporates all possible local move sequences in its analysis as the players can switch between subgames at each move. Therefore all legal local moves are generated for both players until a terminal position is reached. Termination rules determine when to stop the search process, fol-

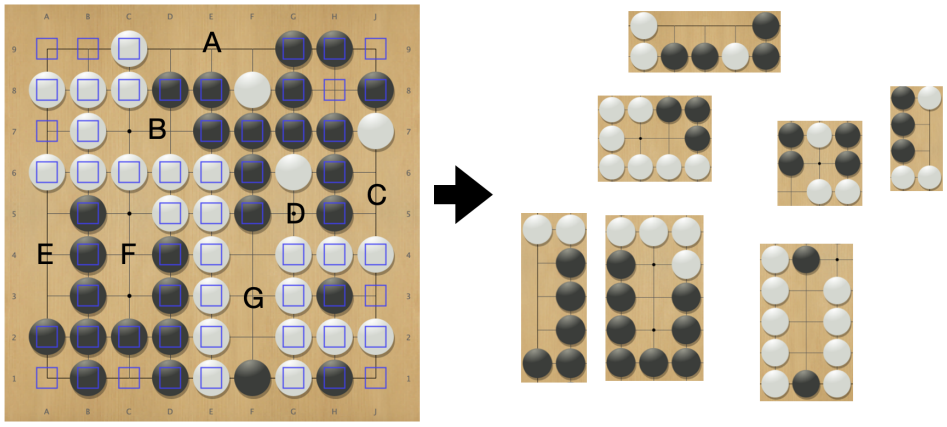


Figure 2.13: Identifying subgames and decomposing the Go board into the subgames.

lowed by local scoring using Japanese rules of all terminal positions. Figure 2.14 shows a local game tree after applying LCGS on subgame B in Figure 2.13. It generates moves for both black and white at the same position. It terminates the search when it doesn't find legal moves. At the last stage, it scores the terminal position using Japanese rules, where a positive score is good for black.

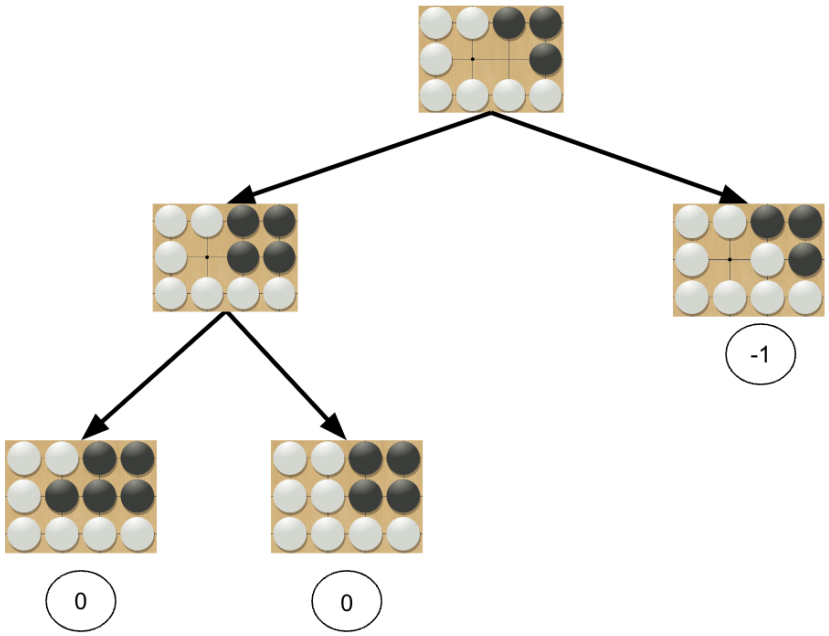


Figure 2.14: Local game tree from local combinatorial game search (LCGS) with evaluation of terminal nodes.

After performing LCGS, the algorithm evaluates the combinatorial game value $C(p)$ of a local game graph using these pre-evaluated terminal nodes. If Black can move to $b_1....b_n$ and white can move to $w_1....w_m$ from a local position p, the evaluation $C(p)$ is determined as follows [14]:

$$C(p) = C(b_1), ..., C(b_n)|C(w_1), ..., C(w_m) \tag{2.3}$$

The last step of decomposition search is to find an optimal move. The focus is on selecting a move that maximally improves the overall position. The improvement is measured by a combinatorial game concept called *incentive* of a move. To find the optimal move, the decomposition search calculates the incentives for all possible moves in each smaller part of the game and chooses the best incentive move.

It often becomes possible to identify a move that dominates all others, thereby determining an optimal move. But sometimes, there are multiple moves that seem equally good. In this case, an optimal move is found by a more complex procedure involving minimax search, which is called minimax optimal. This method considers how the opponent might respond to each move and chooses the one that gives the best chance of winning. In our experiment, we mostly consider incentive optimal. However, if we can't find moves with incentive, we then use the minimax optimal move from the exact solver.

2.5 Related Work

Our research is focused on deeply analysing an AlphaZero-like program in the game of Go against perfect play. We use open-source KataGo as the program [27].

Studies comparing AI systems and perfect play provide valuable insights into the state of AI development. The work by Haque et al. [9] compares Leela Chess Zero (lc0), an AlphaZero-like open source chess program, against perfect play from chess endgame tablebases. The authors calculate move decision errors for both an intermediate and a strong version of lc0's neural network, and for different amounts of MCTS-based search. They find some interesting

cases such as when a move given by the policy (neural network) is correct, but MCTS with a small search gives a wrong move. They further investigate these cases by looking at the changes of action values, and the upper confidence bound of MCTS with increasing amounts of search. Sadmine et al. [19] extend this work by comparing the performance of lc0 with Stockfish, an open source chess program which is totally different than AlphaZero, against perfect play. The authors evaluate the wrong play of both programs with $3, 4$, and 5 pieces. They study the Average Centipawn Loss (ACPL) to identify how much value a player loses while playing a wrong move. They also investigate the behaviour of these engines where there is only one black pawn and one or more stronger white pieces as most of these types of positions give a higher error rate.

Romein and Bal [17] first solved the game of awari and set up a database which contains the minimax values and perfect moves from all reachable positions. With the perfect knowledge from this database, they analyzed the games played by world-champion-level programs: SOFTWARI and MARVIN. The authors found that MARVIN played a correct move in 82% of the cases, and SOFTWARI in 87% of the cases. This relatively high percentage of errors surprised the community, who had assumed that these programs were almost perfect.

Recently, Wang et al. [25] found the vulnerabilities in KataGo by training adversarial policies against it. The adversarial program wins $> 99\%$ of the games against KataGo with no search and $> 97\%$ against KataGo with enough search to be superhuman. These policies trick KataGo into making serious blunders to lose the game instead of playing well. This research could encourage the ML research community to develop more robust training and adversarial defense techniques to produce models needed for safety-critical systems.

Chapter 3

Experimental Setup

In this chapter, we provide our experimental design, including datasets, engine settings, and computational resources. We first describe the datasets in Section 3.1. We discuss specific engine settings, including GTP configuration and Neural Network choices in Section 3.2, and describe computational resources in Section 3.3.

3.1 Datasets

We used the 22 endgame problems, $C.1, C.2, ..., C.22$, introduced by the book "Mathematical Go: Chilling gets the last point" [7]. We call these problems **original problems**. We also used the modified versions of these original problems, which we call **modified problems** [14]. These modified problems were introduced to create equivalent endgame regions separated by stones that can be proven safe by a program. This makes it possible for an exact solver to analyze each small local area independently [15]. Figure 3.1 shows an original endgame $C.4$ (left) and the modified version of $C.4$ (right). The changes make blocks alive and clearly separate the endgames for local search.

There are a few reasons why these problems are chosen:

- The problems are challenging. They require careful analysis and calculation and can be very satisfying to solve. The author of the original problems also demonstrated the effectiveness of his theory by setting up a plausible endgame position from which he beat one of the Japanese

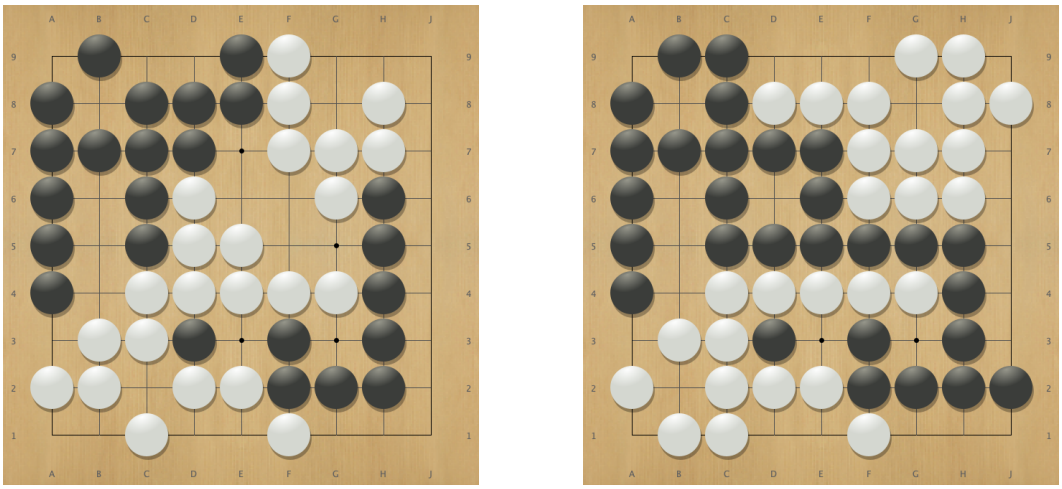


Figure 3.1: An original endgame $C.4$ (left) and the modified version of $C.4$ (right).

champions of the game, after which he set up the same position, reversed the colors, and beat the master a second time[1].

- These problems are more practical than the adversarial perturbation set, where the author creates this set by adding one or two meaningless stones that cause KataGo to make serious mistakes. [12].

- These problems also cover various endgame situations, from simple to complex, which allows testing a program on a range of difficulty levels.

Overall, endgame problems are an excellent choice for evaluating the playing abilities of both machines and humans. Here, white is to play for every problem, and all potential winning moves can be evaluated and determined mathematically. These winning moves are all possible winning moves with perfect play by both players, and we check if our Go engine can find one of them. In Figure 3.1, the only winning move for both the original and the modified problem is $G1$.

3.1.1 Extended Datasets

We use the exact solver to generate one perfect game starting from each state in each modified problem. We adjust the komi when a stone is captured. To

[1]Article is available at $https://celebratio.org/Berlekamp\_ER/article/730/$

validate each perfect game position, we play matches between the exact solver and KataGo. If KataGo suggests any move outside the winning moves given by the exact solver, we verify that it results in KataGo losing the game.

Figure 3.2 shows an example of how we generate the extended dataset from an endgame position. Figure 3.2(a) represents the modified problem $C.7$ with komi -0.5 and white to play. The letter A indicates one of the winning moves of this board position. We play white A to get the next starting endgame position in Figure 3.2(b) with black to play. As there is no captured stone, komi isn't changed here. Similarly we get the third starting position in Figure 3.2(c) by playing black A in Figure 3.2(b). We name this collection of data **perfect games** since the moves follow an optimal line of play. To expand the dataset, we use the modified problems $C.1, C.2, C.3, C.6, C.7, C.8, C.9, C.10$, and $C.21$. Through these extensions, we add a total of 126 test positions to the dataset.

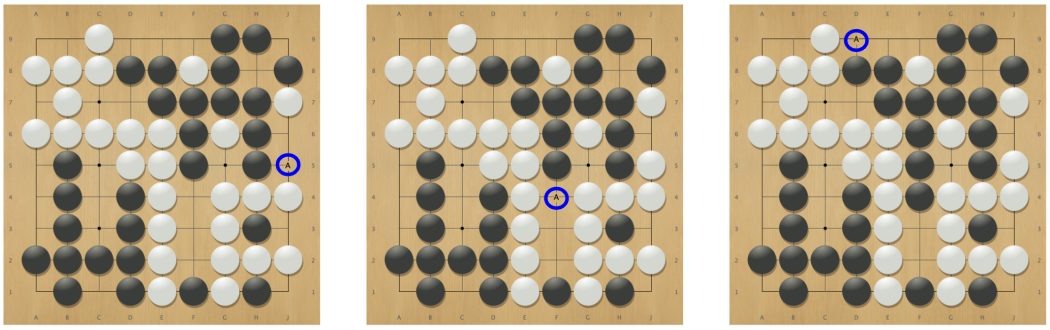


Figure 3.2: Data generation from modified problem $C.7$.

We use GoGui [3] for this procedure. GoGui is a free software package that provides a set of tools for playing Go, and for interacting with Go programs. We attach the exact solver program in GoGui, generate winning moves through the solver, and save the board position as an sgf file. If black captures a white stone, we add -1 to the komi as the captured stone information is not maintained otherwise. On the other hand, if a black stone is captured by white, we add $+1$ to the komi.

To add more 19×19 endgames to our dataset, we utilize modified versions

of problem $C.11$. This problem is particularly challenging, with 29 subgames. Figure 3.3 displays the modified $C.11$ with all its subgames, each labeled with a letter. We use an extension of the subsets of $C.11$ from [15]. As a series of problems of increasing complexity, in these test positions, all other subgames are changed into safe territories, except for the ones we're interested in playing [15]. For example, in the subset of $C.11$ A-E in Figure 3.4, all other original subgames are made safe territories. We utilize a total of 21 subsets from A-A to A-U.

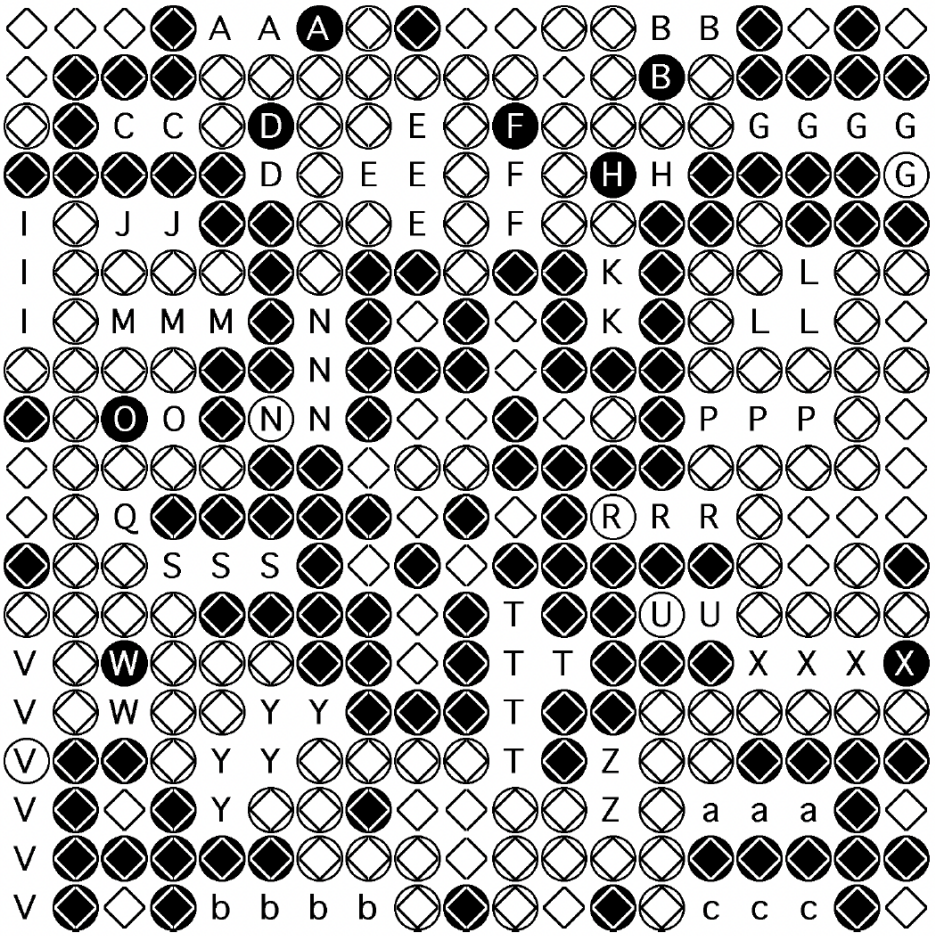


Figure 3.3: Modified $C.11$ with 29 subgames represented by distinct letters. Safe points are marked with a diamond symbol.

Following the procedure outlined above for extending the dataset with perfect games, we apply a similar approach to the subsets obtained from modified versions of problem $C.11$. By utilizing these subsets, we generate additional

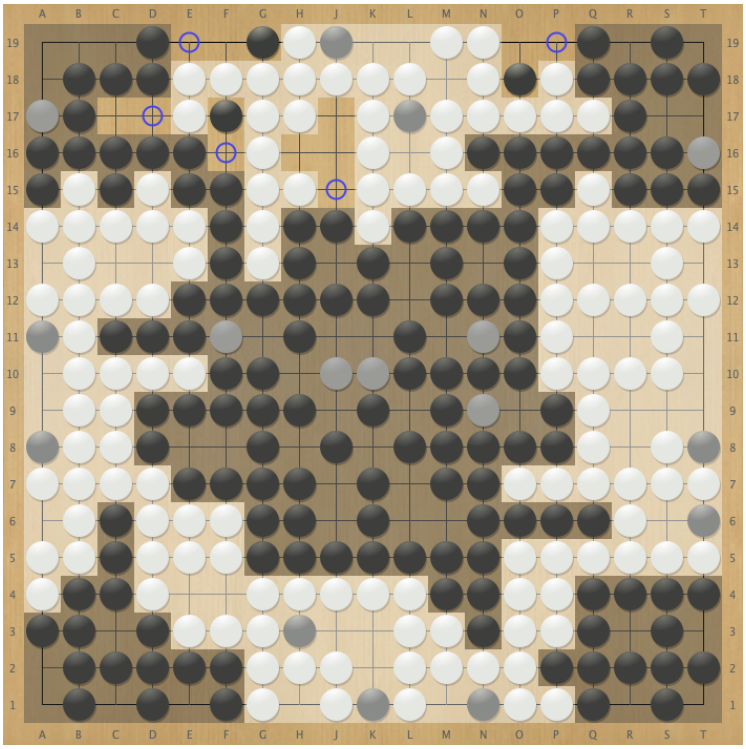


Figure 3.4: The subset A to E of $C.11$. All other subgames are changed to safe territories. The five active endgames are marked with blue circles.

data. Through this process, we obtain a total of 371 endgames from these subsets. We name these endgames **C.11 subsets**.

Table 3.1 shows the number of endgame problems for each dataset. In our test set, we have endgames with board sizes of 9×9, 13×13, and 19×19. All endgame problems are stored in Smart Game Format (SGF) [4], which is a computer file format used for storing records of board games. It is a text-only, tree-based format.

Name of Dataset	**Number of Endgames**
Original	22
Modified	22
Perfect games	126
C.11 subsets	371

Table 3.1: Number of endgame problems in our datasets.

3.1.2 Splitting the Dataset By Unique Incentives and By Endgame Size

We divide perfect games and $C.11$ subsets into categories. We employ two concepts for splitting these test sets to evaluate KataGo's decision-making strength in selecting the best incentive move among all winning moves, and to assess its accuracy across different endgame sizes.

Split based on existence of a dominating incentive

Our extended dataset is perfect games and $C.11$ subsets. We get the optimal moves of these two test sets from an exact solver. It is possible that there is no single dominating incentive in a board position. For this reason, we split them into two sets.

- Set 1: This set consists of the positions where there exists a single dominating incentive. This means that there is at least one move with that incentive provided by the exact solver. We name this set **DI**.
- Set 2: This set includes positions where there is no single dominating incentive. In such cases, there are two or more non-dominating incentives, and search must be used to find the best moves in terms of minimax score. The exact solver returns an empty set of incentive optimal moves, and only minimax optimal moves are considered as winning moves. We call this set **no-DI**

We conduct different tests for **DI** and **no-DI**, which are discussed in detail in Chapter 4.

Split based on number of unsafe points

The complexity of endgame problems varies based on the number of unsafe points remaining on the board. A board with fewer such points is likely to be simpler than one with many. We measure complexity by calculating the number of points which are not considered safe by the exact solver. The formula to calculate unsafe points is:

$$unsafe\_points = boardsize * boardsize - safe\_points$$

Here, safe_points is the number of points considered safe by the exact solver. Figure 3.5 shows a board position with safe points marked by white and dark grey shading. This position has 61 safe points according to the exact solver. The number of unsafe points is $9 * 9 - 61 = 20$.

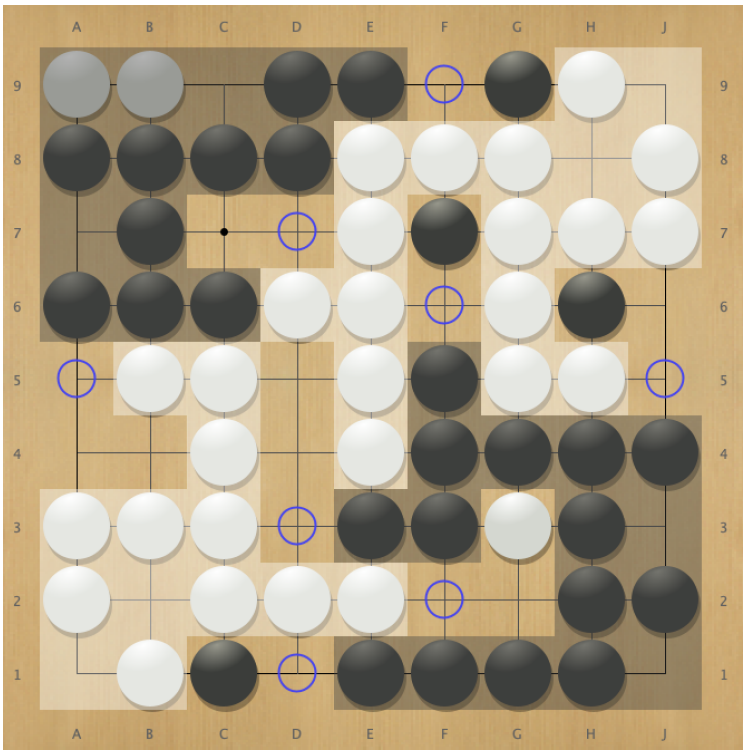


Figure 3.5: An example of a board position with 20 unsafe points.

3.2 Go Engine Settings

We use KataGo version v1.12.4 to analyze our endgame dataset. The reasons to select KataGo as our analysis tool are:

- Since 2021, KataGo has been the strongest open-source Go program available online (available at https://github.com/lightvector/KataGo).
- KataGo is an AlphaZero-like [21] program with many enhancements and improvements [27]. It supports setting the value of komi (including in-

teger values) and a wide variety of rules, including rules that match Japanese rules in almost all common cases.

- KataGo outperforms other available AlphaZero-style open-source Go bots [27]. It also has four backends - OpenCL (GPU), CUDA (GPU), TensorRT (GPU), and Eigen (CPU). So, it can easily run on a wide variety of GPU and CPU-based machines.

3.2.1 GTP Configuration for KataGo

The Go Text Protocol (GTP) [2] is a protocol used by most Go engines and Go servers for playing games between programs. It is also used for the configuration of KataGo's setting [28]. For our experiment, we keep the default values for most of them. The following changes have been made:

- **Rules and Komi**: KataGo sets Tromp-Taylor as its default rule to play the game. We use Japanese rules to match our datasets, and adjust the komi individually for each problem, such that each position is a win for white with 0.5 points with perfect play.

- **MaxVisits**: The term "Visits" refers to the count of search playouts performed in each turn, including the searches conducted in previous turns that are still relevant in the current turn. To illustrate, if KataGo conducted a 200 node search in the previous turn and, after the opponent's response, 50 nodes from that search tree are still valid, setting a visit limit of 200 causes KataGo to explore 150 new nodes, resulting in a final tree size of 200 nodes. We vary the"MaxVisits" parameter to analyse how KataGo performs with both small and large searches. We evaluate the Neural Network without search by setting "MaxVisits = 1". In this case, KataGo will make a decision after evaluating only the root node, and its Neural Network policy, without expanding the tree.

- **Policy Moves**: We determine the policy distribution for the next move of a board position from the Neural Network by using the GTP command "kata-raw-nn". This command provides the output of a raw neural

network evaluation conducted by KataGo including the probability that white wins, the probability that black wins, and the policy distribution for the next move.

- **Move Analysis**: For search-based investigation of a generated move from KataGo, we use the GTP command "kata-genmove_analyze". This command returns information such as winrate, action values, and the lower confidence bound (LCB) for a search.

3.2.2 Choice of Neural Network

The KataGo project has produced a wide variety of neural nets to play the game [29]. In our experiment, to determine the best performance of KataGo, we utilize one of the 'strongest confidently-rated networks', as of October, 2022, known as "kata1-b18c384nbt-s5832081920-d3223508649". This network has 18 blocks and 384 channels that are essential to detect the features. On the other hand, as a weak network we choose a small size "kata1-b6c96-s37368064-d5536083" which has 6 blocks and 96 channels.

3.3 Computational Resources

We use one CPU core to process datasets, which involves turning problems into perfect games and creating expanded datasets using GoGui and the exact solver. To run KataGo program, we use OpenCL GPU as the computational backend since this is the most general GPU version of KataGo and doesn't require a complicated install like CUDA does. Strong policy takes 64.1 seconds to run perfect games (126 endgames) where weak policy takes only 1.2 seconds. The search of KataGo program takes up the most time in our experiments. For example, to run perfect games with MaxVisits = 102400 using strong policy took nearly 5 hours.

Chapter 4

Experiments and Analysis

To measure the performance of an engine such as KataGo, we conduct two types of evaluation: 1. using only the policies from weak and strong raw neural networks without search, and 2. using both the policies and search. In both tests, the engine makes one move for each endgame position. We compare this move with all the optimal moves, both incentive optimal and minimax optimal cases, for that position. If the move is not optimal, we consider it wrong. We measure the success rate as the percentage of correct moves in the entire test set. KataGo's move decisions can vary from one run to another. Sometimes it might provide different outputs compared to a previous run. To account for this variability, we conduct each experiment five times and then calculate the average results. As search of KataGo program takes a significant amount of time (see Section 3.3), we couldn't run it more than 5 times. After running the experiments 5 times, we found that the success rate varied from 20% to 30%. Here, we present the average results, which provide a more accurate performance assessment of KataGo compared to a single run result.

4.1 Experiment 1: Evaluating Network Policies with no Search

We start our experiment by evaluating KataGo's weak and strong raw neural networks mentioned in Section 3.2.2, without using any search to play. From the neural network, KataGo gets the policy distribution for the next move of a board position. It then plays the highest percentage policy move if there is

no search. Table 4.1 shows the total number of correct moves along with the average success rate achieved by the weak and strong policy networks after 5 different runs.

Name of Dataset	**Number of Endgames**	**Total Number of Correct Moves with Avg Success Rate (%)**	
		Weak policy	Strong Policy
Original	22	8.8 (40%)	12.8 (58.2%)
Modified	22	7.8 (35.5%)	13.6 (61.8%)
Perfect games	126	98.8 (78.4%)	118.8 (94.3%)
C.11 subsets	371	355.6 (95.8%)	369.6 (99.6%)

Table 4.1: Total number of correct moves along with average success rate by the weak and strong policy.

In general, the strong policy network demonstrates higher success rates compared to the weak policy across all the dataset, indicating its effectiveness in making optimal moves in the given endgames. We can also observe that the success rate of perfect games and C.11 subsets is higher compared to other datasets. One reason for this might be that the endgames in these datasets have more ways to win, making it easier for KataGo to find a winning move. For example, most original and modified problems have only one or two winning moves, while perfect games and C.11 subsets often have between 4 and 10 winning moves. Another reason could be that the endgame size is gradually reduced when creating these endgames by playing out the games, as in Section 3.1.1. Smaller endgames are usually easier to solve than larger ones.

Let D be the whole dataset, and $M \leq D$ be the set of positions where KataGo makes a mistake. To know how much KataGo's generated move differs from the winning moves, we compare the policy probability of the winning move with highest policy value and KataGo's move on M. We focus on positions in M because if KataGo gives a correct move, it means its highest policy move is one of the winning moves. We only consider the strong network in this test since it makes for fewer mistakes than the weak network. Figure 4.1 shows the data for M, sorted by the policy probability difference. We notice

a significant difference for a few problems, which is quite surprising. To investigate this surprising result further, we show two cases where the difference is exceptionally high.

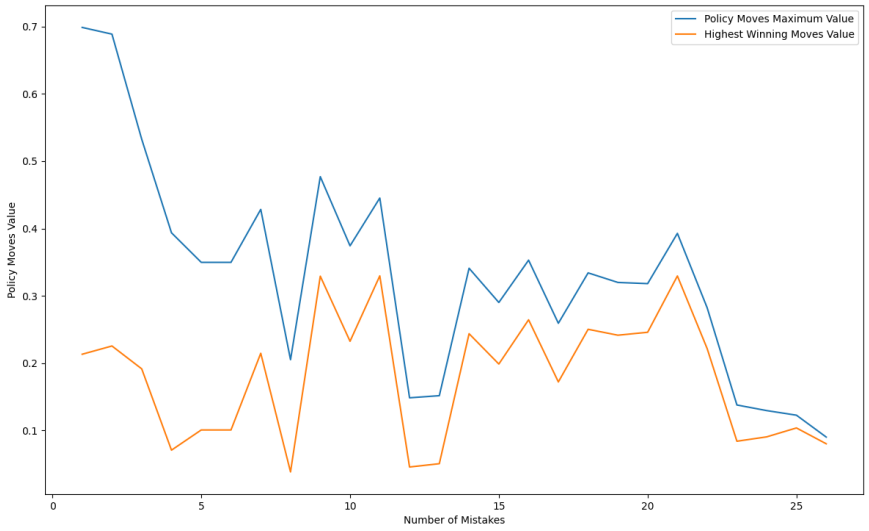
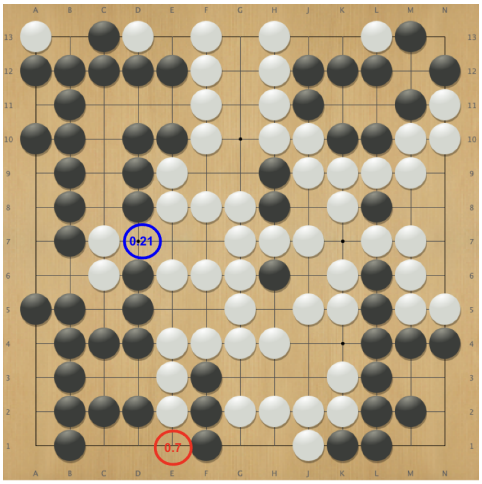


Figure 4.1: Policy probability difference between KataGo's highest policy move and the winning move with highest policy value on test set M.

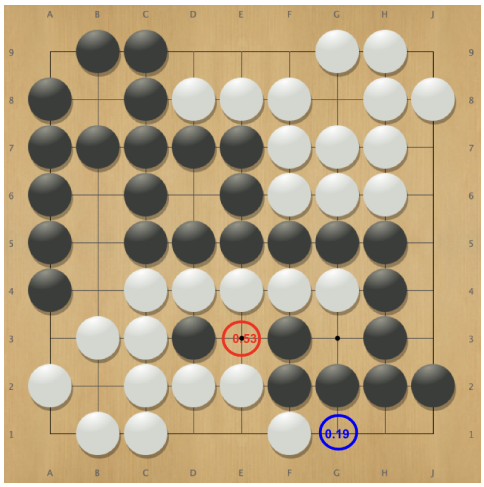
The worst performance of KataGo was on the positions from modified problems $C.22$ and $C.4$, with differences of 0.49 and 0.34 respectively. Figure 4.2 shows these two problems, indicating KataGo's move and the winning move along with their policy probabilities.

4.2 Experiment 2: Evaluating KataGo using both Neural Networks and Search

This experiment investigates the impact of varying the search budget of KataGo when solving the endgames in our whole dataset. We compare the results obtained from the weak and strong raw policies, with a small search using MaxVisits = 100 per move decision. Next, we scale the search budgets from 100 to 102400 to assess the impact of deeper searches on accuracy.



(a) White to play. KataGo's move is $E1$ and the only winning move is $D7$.



(b) White to play. KataGo's move is $E3$ and the only winning move is $G1$.

Figure 4.2: Examples of two endgames where the policy difference between winning move and KataGo's move is 0.49 and 0.34 for (a) and (b), respectively.

4.2.1 Comparing Strong and Weak Policies with Small Search

Table 4.2 displays the average performance of KataGo using weak and strong raw neural networks with MaxVisits = 100. It shows the number of correct moves, success rate, and the improvement over the no-search versions indicated by a ↑ sign. As original and modified problems are challenging, weak policy with small search couldn't improve much. But the search helps the strong network to improve in a significant amount.

Name of Dataset	**Number of Endgames**	**Total Number of Correct Moves with Avg Success Rate (%)**	
		Weak policy + MaxVisits = 100	Strong Policy + MaxVisits = 100
Original	22	9.6(43.6%)↑3.6%	16.2(73.6%)↑15.6%
Modified	22	9(40.9%)↑5.4%	16.8(77.3%)↑14.6%
Perfect games	126	105.8(84%)↑5.6%	123.8(98.3%)↑4%
C.11 subsets	371	361.4(97.4%)↑1.6%	370(99.7%)↑0.1%

Table 4.2: Total number of correct moves along with average success rate and improvement by the weak and strong policies with MaxVisits = 100.

4.2.2 Scaling the Search

We increase our search budget to see how deeper search affects the overall accuracy. We start from MaxVisits = 100 and double it in each step, upto 102400. We utilize all the datasets. For $C.11$ subsets, we only use the one endgame which was not solved by the strong policy in Table 4.2.

Figure 4.3 shows the results of the weak policy with search, across all datasets, of five different runs. For original, modified, and perfect games, the number of mistakes generally decreases. But it's still struggling to improve for the original and modified problems. Unexpectedly the average success rate is only 60% after 102400 search. However, for the $C.11$ subsets, we didn't add the result in this figure because there's only one endgame, and it is never resolved.

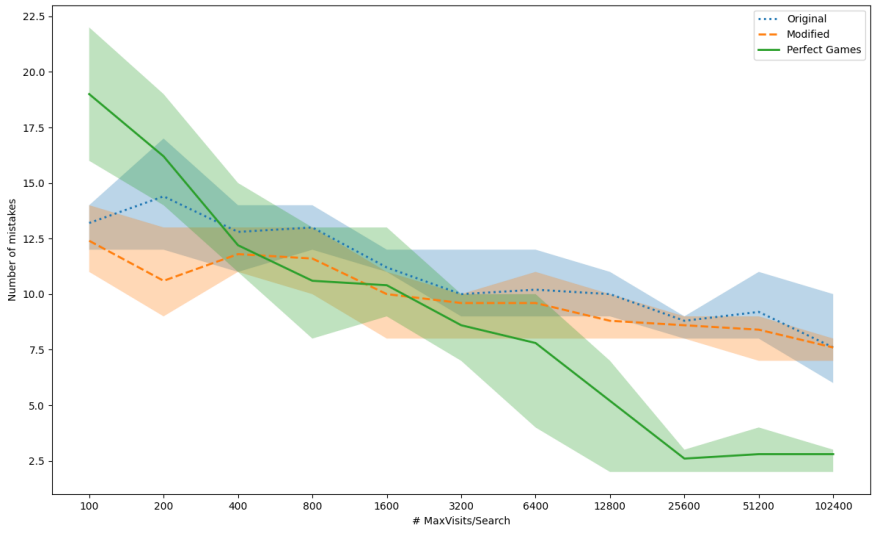


Figure 4.3: Average number of mistakes for KataGo's weak policy with different amounts of search.

The result of the strong policy with search is quite surprising, as shown in Figure 4.4. Initially, the number of mistakes decreases for original, modified, and perfect games. But after reaching a certain point, it starts to rise again. This suggests that for certain endgame scenarios, a shallower search is more effective than a deeper one. We will delve into these interesting cases further

in Section 4.5. The outcome of the $C.11$ subsets is similar to the weak policy. The only difference weak policy gives only one distinct move in every different MaxVisits. For example, we consistently got $G11$ every time we increased the search where the optimal move is $P9$. On the other hand, strong policy gives 4 different moves in different search. This indicates that the strong policy explores more possibilities than the weak one.

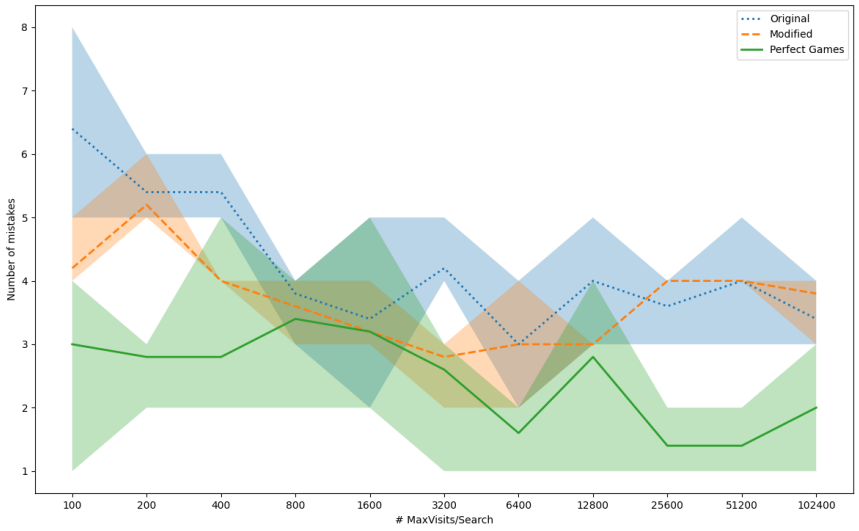


Figure 4.4: Average number of mistakes for KataGo's strong policy with different amounts of search.

4.3 Experiment 3: Evaluating KataGo By Unique Incentives and By Endgame Size

We discussed in Section 3.1.2 how we split our dataset by existence of a unique dominating incentive and by endgame size. In this section, we show the detail experiments, for weak and strong policies, with and without a small search.

4.3.1 Evaluating KataGo By Unique Incentives

From Section 3.1.2 we get two set of data, DI and no-DI, when we split the dataset based on existence of a dominating incentive. We have two different test settings for these two sets.

For DI:

- Check if KataGo's move has the best incentive or not.
- Check if KataGo's move is minimax optimal.

For no-DI:

- Check move within minimax optimal or not as best incentive move is empty here.

Table 4.3 shows the average results for DI with weak and strong policies, and the same policies with a 100 node search. We utilize perfect games and $C.11$ subsets here as we get the winning moves of these test sets from the incentive and minimax calculation of exact solver. From the results of Table 4.3, we find that KataGo struggles more to identify best incentive moves compared to minimax ones. This shows in the success rates of incentive and minimax optimal, where we see a significant difference, ranging from 10 to 40 percent. It seems that KataGo is more focused on finding a path that leads to a victory rather than subtle differences in move strength. Interestingly, we also observe that when we use search with our weak and strong policies, there is more improvement in performance for the weak policy compared to the strong one. This suggests that search has a powerful role in finding the best moves.

We get very few endgames from perfect games and $C.11$ subsets where there is no single dominating incentive. Table 4.4 shows the outcome for no-DI test set. We observe that the results of policies and policies with search are almost identical, except for one extra correct move with the strong policy with search for perfect games. Upon examining the mistaken positions, we find that there's only one winning move in those positions. However, in other positions, there are 6 to 9 winning moves, making those endgames easier to win.

4.3.2 Evaluating KataGo By Endgame Size

We counted how many correct decisions KataGo made and how many endgames there were in total for each endgame size. Then we measured the success rate.

Name of Dataset	**Number of Endgames**	**Total Number of Correct Moves with Avg Success Rate (%)**			
		Weak policy		**Strong Policy**	
		Incentive optimal	Minimax optimal	Incentive optimal	Minimax Optimal
Perfect games	125	70.8(56.6%)	96.8(77.8%)	103(82.4%)	115.8(92.6%)
C.11 subsets	345	190(55.1%)	328.8(95.3%)	251.6(72.8%)	345(100%)

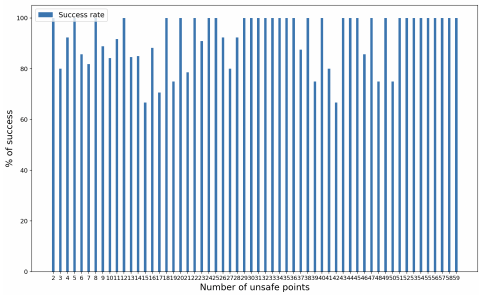
		Weak policy + search = 100		**Strong Policy + search = 100**	
		Incentive optimal	Minimax optimal	Incentive optimal	Minimax Optimal
Perfect games	125	89.2(71.4%)	106.2(85%)	106.2(85%)	122(97.6%)
C.11 subsets	345	235(68.1%)	333.8(96.8%)	254.6(73.8%)	345(100%)

Table 4.3: Total number of correct moves along with average success rate by the weak and strong policy, and policies with 100 search for DI test set, where there exists a single dominating incentive.

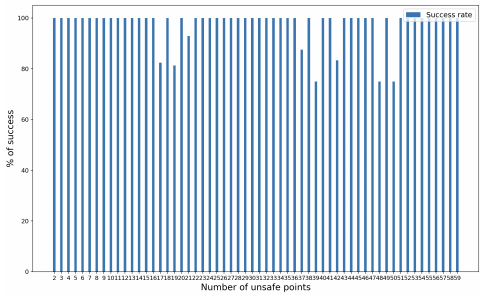
Name of Dataset	**Number of Endgames**	**Total Number of Correct Moves with Avg Success Rate (%)**			
		Policy		**Policy with 100 search**	
		Weak	**Strong**	**Weak**	**Strong**
Perfect games	8	6(75%)	6(75%)	6(75%)	7(87.5%)
C.11 subsets	26	25(96.2%)	25(96.2%)	25(96.2%)	25(96.2%)

Table 4.4: Total number of correct moves along with average success rate by the weak and strong policy, and policies with 100 search for no-DI, where there is no single dominating incentive.

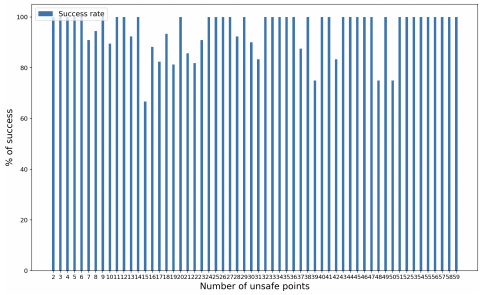
Figure 4.5 displays KataGo's performance using both raw policies and policies with a maximum number of visits set to 500. In Figure 4.5 (a), we see that the weak policy sometimes performs very poorly, even making mistakes in very small endgames. On the other hand, the strong policy, shown in Figure 4.5 (b), only makes mistakes in larger endgames. After performing 100 searches using the weak policy, some of the mistakes in small endgames disappear, as seen in Figure 4.5 (c). The performance of KataGo's strong policy with 100 search, displayed in 4.5 (d), is very strong. There are no mistakes in small endgames, and only a few in large endgames.



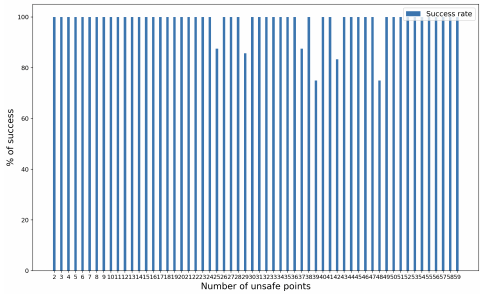
(a) Performance of KataGo's weak policy.



(b) Performance of KataGo's strong policy.



(c) Performance of KataGo's weak policy with MaxVisits = 100.



(d) Performance of KataGo's strong policy with MaxVisits = 100.

Figure 4.5: Percentage of success in terms of total number of endgames across different endgame sizes. The horizontal axis (x-axis) shows the number of unsafe points, while the vertical axis (y-axis) shows the percentage of success.

Figure 4.6 shows a small size simple endgame where KataGo's weak policy makes mistake. The size of this endgame is only 3. It's quite surprising that weak policy could not figure out the winning move in this situation.

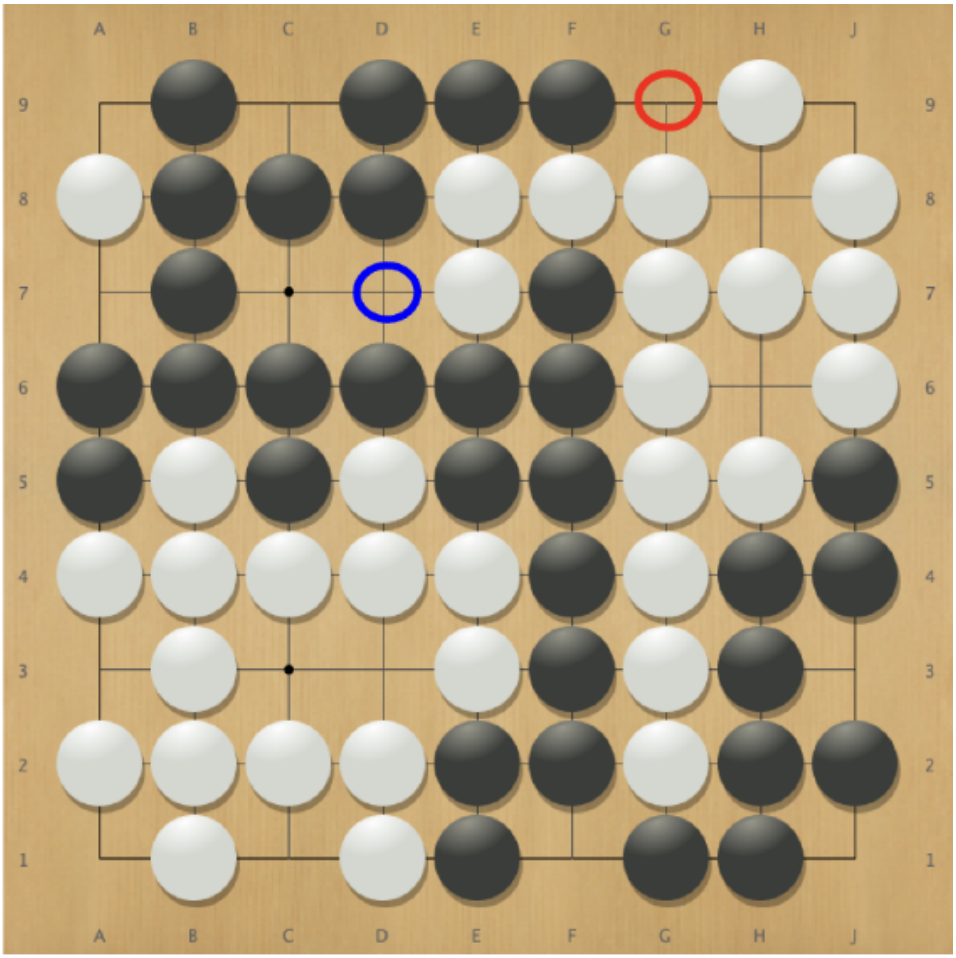


Figure 4.6: A small size (size 3) endgame where KataGo's weak policy makes mistake. Here, $D7$ is the only winning move where KataGo's move is $G9$.

4.4 Experiment 4: Playing Matches between Exact Solver and KataGo

We organize matches between the exact solver and KataGo to show the importance of perfect play. We select the strong policy with MaxVisits = 100. Since the exact solver can solve each state of perfect games and $C.11$ subsets, we utilize these two sets for this experiment. We don't consider endgames with size less than 5.

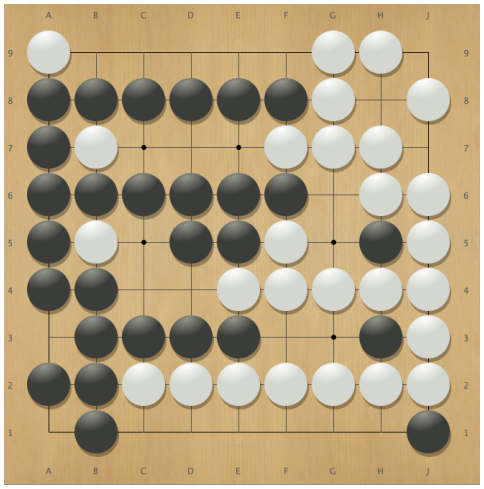
Table 4.5 shows the results of playing matches between the exact solver and KataGo. As expected, the exact solver achieves a winning rate of 100% on all test sets. KataGo loses a few games in the perfect games dataset. However, KataGo manages to win 100% of the matches in the $C.11$ subsets.

Name of Dataset	**Number of Endgames**	**Total Number of Win with Success Rate (%)**	
		Exact Solver	KataGo
Perfect games	120	120 (100%)	109 (90.8%)
C.11 subsets	333	333 (100%)	333 (100%)

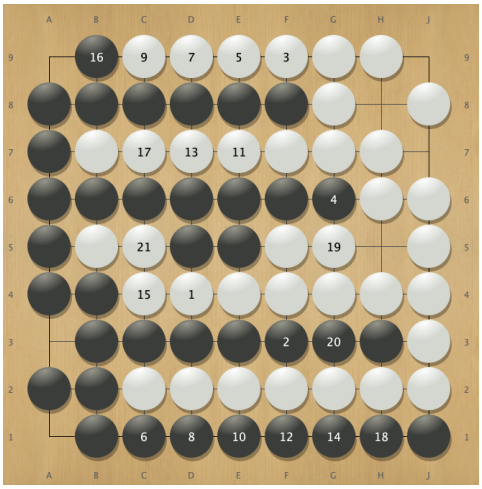
Table 4.5: Total number of wins of KataGo and exact solver along with success rate by the strong policy with 100 search.

Figure 4.7 (a) shows a simple 9×9 board position where KataGo loses as white, but the exact solver wins. If we have a close look on the move ordering of 4.7 (b) and (c), we notice that the exact solver and KataGo have completely different sequences of moves. This difference in move orderings suggests that KataGo might get confused by long corridors on the board.

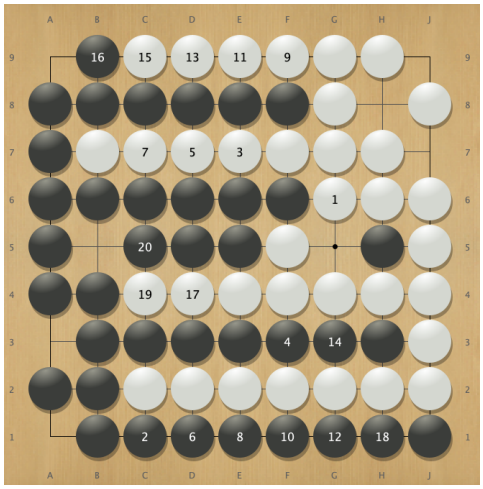
Although KataGo won all the games in $C.11$ subsets as white, it made a mistake when playing as black in one board position. This error ended up benefiting the exact solver, allowing it to win with an extra point. Figure 4.8 (a) shows this position from $C.11$ subsets A-R. In Figure 4.8 (b), we can see the move sequence of the exact solver and KataGo, where exact solver wins by 1.5 as white. Here, the first black move given by KataGo, F16, is not optimal, which helps the exact solver to gain 1 more point. Conversely, the exact solver's first move, P9, shown in Figure 4.8 (c), was optimal, preventing



(a) A simple 9×9 perfect game from modified problem $C.2$.



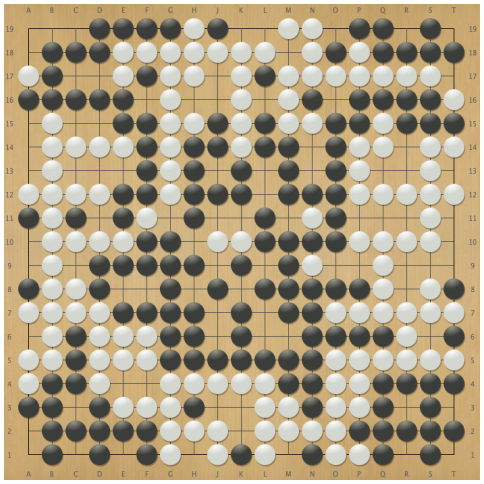
(b) Exact solver wins as white against KataGo.



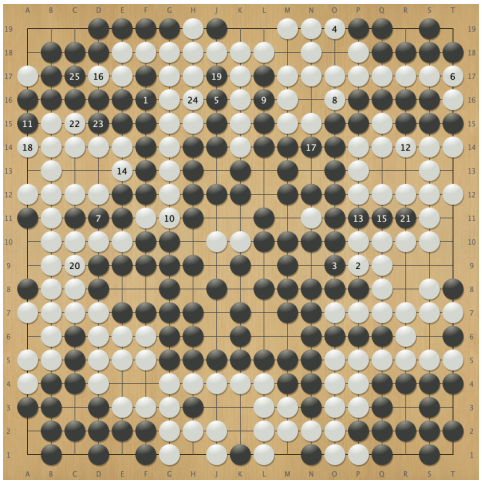
(c) KataGo loses as white against exact solver.

Figure 4.7: Examples of a endgame position where KataGo loses as white but exact solver wins as white.

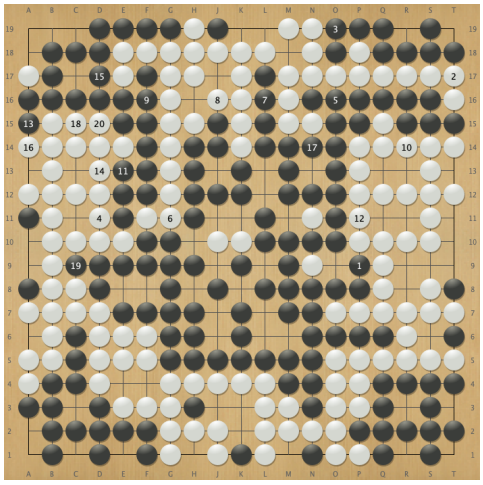
KataGo from securing a 1.5 point win.



(a) A 19×19 board position from $C.11$ subsets A-R.



(b) Exact solver wins as white against KataGo by 1.5.



(c) KataGo wins as white against exact solver by 0.5.

Figure 4.8: Examples of an endgame position where KataGo loses one extra point (b) as black.

4.5 Case Studies of Interesting Mistakes

In our experiments, when a search makes a wrong move, it usually means that the policy is also wrong. Moreover, when deeper search results are incorrect, both the policy and smaller searches tend to be wrong too. Interestingly, there are situations where the deeper search outcome is inaccurate while the

smaller search is correct. In Figure 4.4, we notice that sometimes the number of mistakes rises in larger searches. We also saw a few endgames that remain unsolved even with deep search nodes. For further insights, we run some endgames with more than 200k search and observe the changes of utility (action value), lower confidence bound (lcb), winrate, and number of visited nodes explored during the search process.

4.5.1 Small and Longer Search Both Wrong

We notice that the original problem $C.3$ is not solved by KataGo even with deep search. In different searches, KataGo suggested three different moves: $J13$, $E5$, and $F11$, while the optimal move is $K1$. In Figure 4.9, we can see the policy probability of KataGo's moves indicated by red circles, and the policy probability of the optimal move marked with a blue circle. Interestingly, the probability of choosing the optimal move is lower than for the other moves. Sometimes the bad prediction of policy can be resolved by search. But in this case, even with longer searches, KataGo couldn't find the optimal move.

In Figure 4.10, we see the changes of winrate, lcb, utility and the number of visited nodes during the search process. Initially, $J13$ seems like a promising move based on its policy probability in Figure 4.9, but as the search progresses, it turns out to be the least favorable move. On the contrary, $F11$ starts with a low policy probability but improves during the search, eventually becoming KataGo's preferred move. Although the optimal move, $K1$ rises up initially, later it goes down again. The win rate and lcb of all moves in Figure 4.10 (a) and (b), are below 0.4, indicating a low probability of winning (less than 40%). KataGo estimates that none of these moves are likely to win. It fails to recognize the win with the optimal move, $K1$. All moves have negative utility in Figure 4.10 (c) due to their low chance of winning. Figure 4.10 (d) shows a surprising result. The exploration of all the moves remains low up to 25,600 search, after which only $F11$ shows an increase in the number of visited nodes. This suggests that even after deep search, KataGo views $F11$ as the best move and still overlooks $K1$.

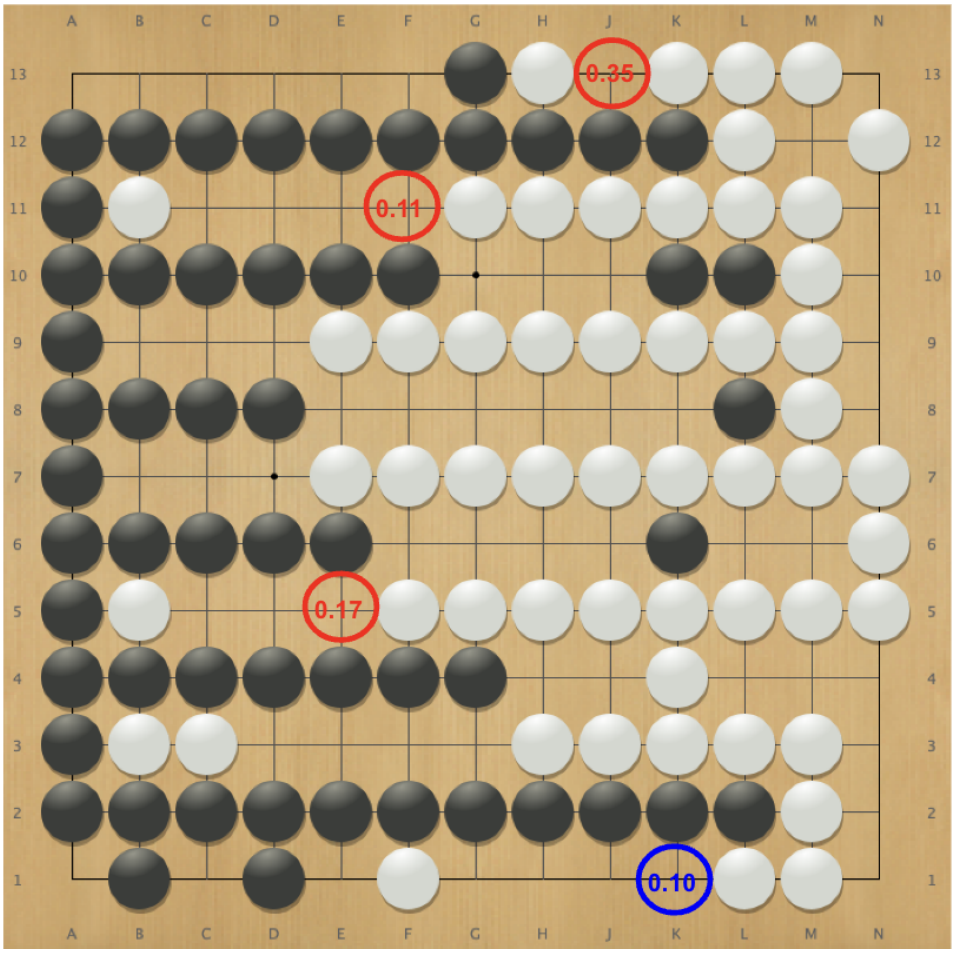
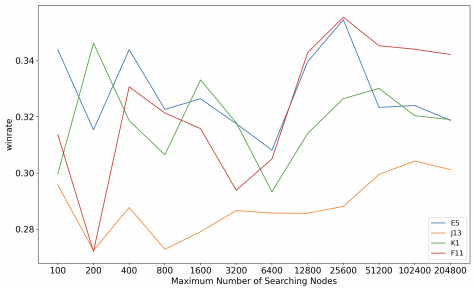
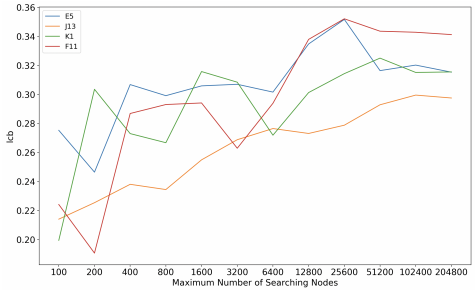


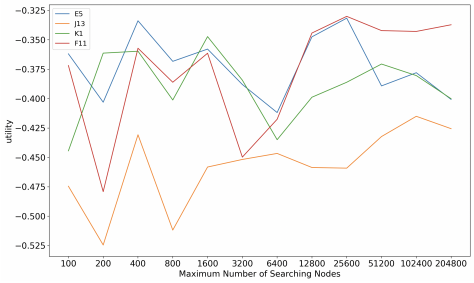
Figure 4.9: The policy probability of KataGo's suggested moves with red circle and optimal move with blue circle in original problem $C.3$.



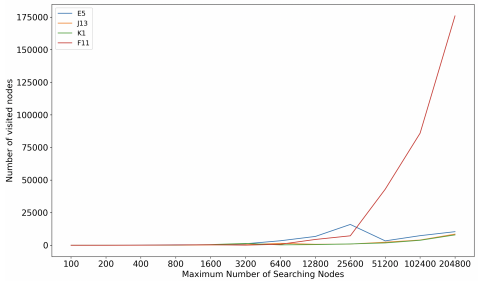
(a) Winrate.



(b) Lower confidence bound (lcb).



(c) Utility (action value).



(d) Number of visited nodes.

Figure 4.10: The change of winrate, lcb, utility, and number of visited nodes during the search of original problem $C.3$ shown in Figure 4.9.

4.5.2 Small Search Correct but Longer Search Wrong

We find an interesting scenario with modified problem $C.11$, where KataGo correctly identifies winning moves with a small search but makes a wrong move after a larger search. The winning moves of this position are $G13$, $L4$, $P9$, and $Q17$. However, KataGo also suggests the losing moves $D11$, $F16$, and $C5$ in different searches. In Figure 4.11, the policy probabilities of the winning moves and KataGo's suggested moves are quite similar.

We can observe the changes of winrate, lcb, utility and the number of visited node in Figure 4.12. Initially, the winning moves are less favorable compared to $D11$, $F16$, and $C5$. After Maxvisits = 1600, there is a significant improvement in the case of $P9$ and KataGo starts selecting this winning move. With deeper searches, the value for $P9$ decreases again, and the losing move $C5$ becomes KataGo's preferred move. Even though the win rate of $D11$ is better than $C5$ in Figure 4.12 (a), KataGo still chooses $C5$ because its decision is not solely based on win rate but also on action value and lcb. Looking at Figure

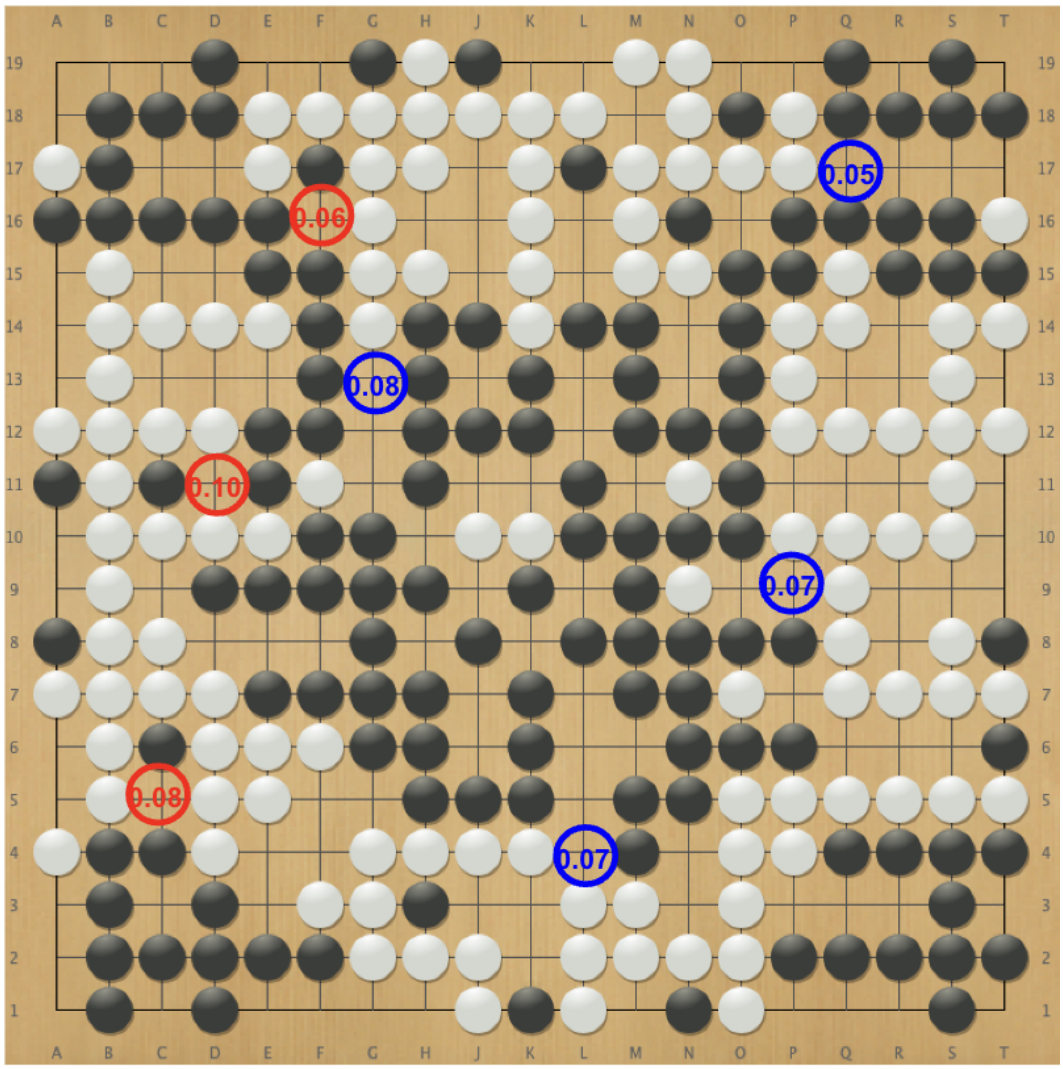
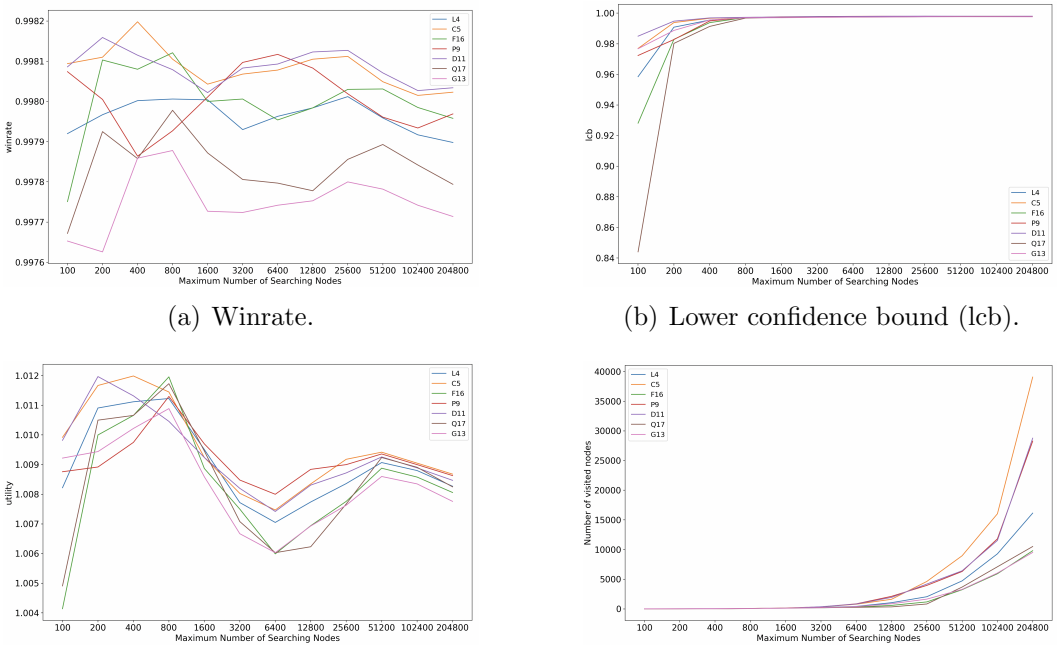


Figure 4.11: The policy probability of KataGo's suggested moves with red circles and winning moves with blue circles in modified problem $C.1$.

4.12 (c), we see that $C5$ has the best utility. However, after a certain point, the lcb of all moves becomes almost the same, as shown in Figure 4.12 (b). The main issue here is that KataGo treats losing and winning moves equally. Even though it picks different moves during different searches, the win rate, utility, and lcb of all losing moves are high, which indicates KataGo's wrong understanding of this position. In Figure 4.12 (d), KataGo explores $C5$ more in the longer search levels. This suggests that sometimes deep search can lead KataGo to make incorrect decisions and to explore more in the wrong move.



(a) Winrate.

(b) Lower confidence bound (lcb).

(c) Utility (action value).

(d) Number of visited nodes.

Figure 4.12: The change of winrate, lcb, utility, and number of visited node during the search of modified problem $C.11$ shown in Figure 4.11.

Chapter 5

Conclusion and Future Work

Through our research, we have gained valuable insights into the performance of KataGo and its ability to play the game of Go at perfect level. We addressed several key questions regarding move selection, the impact of search, and the overall performance of KataGo in different datasets. We observed that the differences in move selection between weak and strong neural networks are significant. However, the addition of a small search significantly improved the performance of the weak and strong policy, highlighting the importance of search algorithms. Increasing the search budget had a positive impact on KataGo's overall performance. However, we also noted cases where deeper search led to incorrect move selections. Our experiments involving playing matches between the exact solver and KataGo showed the importance of perfect play. While KataGo performed well in most cases, there were cases where it made mistakes even in simple endgames, particularly in endgame positions with long corridors that could confuse the algorithm. The network has not learned the correct relative values of those moves, and often prefers simple captures of lesser value. We investigated several interesting cases in detail and found that KataGo faces difficulties balancing exploration and exploitation in these longer searches.

Possible future works based on this study include:

- Develop more complex datasets with perfect solutions with a small number of winning moves.

- Figure out with a deep analysis why KataGo performs worse to find the best incentive moves.
- Investigate and improve KataGo's search algorithms further to improve its decision-making accuracy against perfect play.
- Analyze the possibility to convert KataGo into a solver to solve the endgames [26].

References

[1] -, "Leela Zero," *https://github.com/leela-zero/leela-zero*, 2019.

[2] -, "Go Text Protocol (GTP)," *https://www.gnu.org/software/gnugo/gnugo_19.html*, accessed: 2022-10-11.

[3] -, "GoGui," *https://github.com/Remi-Coulom/gogui*, accessed: 2022-09-11.

[4] -, "Smart Game Format (SGF)," *https://www.red-bean.com/sgf/go.html*, accessed: 2022-09-11.

[5] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, pp. 235–256, 2002.

[6] A. G. Barto, P. S. Thomas, and R. S. Sutton, "Some recent applications of reinforcement learning," in *Proceedings of the Eighteenth Yale Workshop on Adaptive and Learning Systems*, 2017.

[7] E. Berlekamp and D. Wolfe, *Mathematical Go: Chilling gets the last point*. CRC Press, 1994.

[8] 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.

[9] R. Haque, T. H. Wei, and M. Müller, "On the Road to Perfection? Evaluating Leela Chess Zero Against Endgame Tablebases," in *Advances in Computer Games: 17th International Conference, ACG 2021, Virtual Event, November 23–25, 2021, Revised Selected Papers*, Springer, 2022, pp. 142–152.

[10] A. Heifets and I. Jurisica, "Construction of new medicines via game proof search," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 26, 2012, pp. 1564–1570.

[11] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Machine Learning: ECML 2006: 17th European Conference on Machine Learning Berlin, Germany, September 18-22, 2006 Proceedings 17*, Springer, 2006, pp. 282–293.

[12] L.-C. Lan, H. Zhang, T.-R. Wu, M.-Y. Tsai, I. Wu, C.-J. Hsieh, *et al.*, "Are alphazero-like agents robust to adversarial perturbations?" *Advances in Neural Information Processing Systems*, vol. 35, pp. 11 229–11 240, 2022.

[13] M. Müller, "Computer Go as a sum of local games: an application of combinatorial game theory," Ph.D. dissertation, Verlag nicht ermittelbar, 1995.

[14] M. Müller, "Decomposition search: A combinatorial games approach to game tree search, with applications to solving Go endgames," in *IJCAI*, 1999, pp. 578–583.

[15] M. Müller, "Not like other games-why tree search in Go is different," in *Proceedings of Fifth Joint Conference on Information Sciences (JCIS 2000)*, 2000.

[16] A. Riboni, A. Candelieri, and M. Borrotti, "Deep Autonomous Agents Comparison for Self-driving Cars," in *International Conference on Machine Learning, Optimization, and Data Science*, Springer, 2021, pp. 201–213.

[17] J. W. Romein and H. E. Bal, "Awari is solved," *ICGA Journal*, vol. 25, no. 3, pp. 162–165, 2002.

[18] S. J. Russell, *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[19] Q. A. Sadmine, A. Husna, and M. Müller, "Stockfish or Leela Chess Zero? A Comparison Against Endgame Tablebases," in *Advances in Computer Games*, Springer, 2023, pp. 26–35.

[20] D. Silver, A. Huang, C. J. Maddison, *et al.*, "Mastering the game of Go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[21] 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.

[22] D. Silver, J. Schrittwieser, K. Simonyan, *et al.*, "Mastering the game of Go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[23] G. Tesauro and G. Galperin, "On-line policy improvement using Monte-Carlo search," *Advances in Neural Information Processing Systems*, vol. 9, 1996.

[24] Y. Tian, J. Ma, Q. Gong, *et al.*, "Elf opengo: An analysis and open reimplementation of AlphaZero," in *International conference on machine learning*, PMLR, 2019, pp. 6244–6253.

[25] T. T. Wang, A. Gleave, T. Tseng, *et al.*, "Adversarial Policies Beat Superhuman Go AIs," 2023.

[26] M. H. Winands, Y. Björnsson, and J.-T. Saito, "Monte-Carlo tree search solver," in *Computers and Games: 6th International Conference, CG 2008, Beijing, China, September 29-October 1, 2008. Proceedings 6*, Springer, 2008, pp. 25–36.

[27] D. J. Wu, "Accelerating self-play learning in Go," *arXiv preprint arXiv:1902.10565*, 2019.

[28] D. J. Wu, "KataGo GTP Extensions," *https://github.com/lightvector/KataGo/blob/master/docs/GTP_Extensions.md*, accessed: 2021-10-11.

[29] D. J. Wu, "KataGo Networks," *https://katagotraining.org/networks/kata1/*, accessed: 2022-11-04.

[30] X. Yang, J. Zhang, K. Yoshizoe, K. Terayama, and K. Tsuda, "ChemTS: an efficient python library for de novo molecular generation," *Science and technology of advanced materials*, vol. 18, no. 1, pp. 972–976, 2017.