On the Road to Perfection? Evaluating Leela Chess Zero Against Endgame Tablebases

by

Rejwana Haque

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

Board game research has pursued two distinct but linked objectives: solving games, and strong play using heuristics. In our case study in the game of chess, we analyze how current AlphaZero type architectures learn and play late chess endgames, which are solved. We study the open-source program Leela Chess Zero in three, four and five piece chess endgames. We quantify the program's move decision errors for both an intermediate and a strong version, and for both the raw policy network and the full MCTS-based player. We relate strong network performance to depth and sample size. We discuss a number of interesting types of errors by using examples, explain how they come about, and present evidence-based conjectures on the types of positions that still cause problems for these impressive engines. We show that our experimental results are scalable by demonstrating the experiments on samples from the five piece tablebase of KQRkq, the difficult endgame of king, queen and rook against king and queen.

Preface

The work presented in this thesis is an elaboration and extension of a research article accepted in Advances in Computer Games 2021 [13], which is coauthored by Professor Martin Müller and Dr. Ting Han Wei. Professor Martin Müller supervised this thesis and Dr. Ting Han Wei participated in the discussion.

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Martin Müller, for his immense support and guidance. I am grateful to him for his kind and supportive attitude throughout the whole journey and for his valuable comments on my thesis.
Secondly, I would also thank our friendly postdoc Dr. Ting Han Wei for his discussion, support and valuable feedback on my thesis. He not only guided me in my thesis but also taught me a lot.
I am grateful to my Mom and Dad for all their support and tremendous sacrifice for me. I am also thankful to my family and friends for always being there for me.
Finally, I would like to give my gratitude to my husband Md Solehin Islam for his encouragement throughout the years of my study and for his unconditional support during my hard days.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Humans make a very large number of decisions using their intelligence every day. Some of the decisions have a short-term single impact. Most of the time these decisions are made with a very clear reason where the person is aware of the consequences of the decision made. However, other decision problems may have a long-term impact involving multiple decisions of subsequent actions. Humans tend to make choices for these types of decision problems based on heuristics and inference. Some of the actions have a small number of choices whereas others may have a very large or even infinite number of choices. The consequence of these subsequent actions can be intractable for the human brain. With the advancement of computational resources, computer systems nowadays are able to execute billions of instructions per second. Artificial intelligence (AI) systems have evolved over time to automate tasks by combining human-like thinking with the computational capacity of these modern computer systems.

Humans make decisions based on a representation of the problem within their brains [46]. Most real-world problems are too complex to represent in computer systems as they involve a huge number of complex rules. Therefore researchers need a simpler ground to compare the performance of an AI system with human intelligence. Strategic combinatorial games have a finite number of states and choices to make. Moreover, these games have simple rules that are easy to implement in a computer system. Therefore, these games are an excellent way to assess the performance of an AI system. Taking advantage of increasing computational resources, groundbreaking AI technologies have been invented which can play difficult (for human) games at superhuman levels. One of the technological breakthroughs in the field of AI is AlphaZero [17], developed by DeepMind. In this work, we investigate how far an AlphaZero approach is from perfect play.

1.1 Motivation of the Research

From zero human knowledge, AlphaZero learns to play at a superhuman level of performance in different strategy games. AlphaZero won against its ancestor AlphaGo Zero, which was a descendent of AlphaGo: the first program to defeat a human world champion in Go. Furthermore, AlphaZero has beaten Stockfish-8, the top chess engine in the 2017 TCEC Championship [17], which is a renowned tournament in the computer chess community. AlphaZero also defeated ELMO, the champion of the 2017 CSA championship in Shogi. From these results it is clear that the method has significantly exceeded the human level of play.

While the AlphaZero algorithm was demonstrated to work for board games, its applications may go beyond. For applications such as drug design where a small deviation from the exact result may cost a huge loss, we need exact solutions. However it is still an open question if AlphaZero-like algorithms can be applied to find exact solutions. To answer this question we have to investigate how these modern programs learn to play, especially in cases where we have exact solutions to compare with. Therefore in this thesis we will analyze how far such a program is from perfect play by comparing it's decision with perfect ones from chess endgame tablebases.

1.2 Research Questions

Our Research is focused on finding answers for the following questions:

- How do stronger and weaker policies differ for selecting moves?

- Regarding the performance of raw neural networks vs networks combined with search, how does search improve move selection?
- How does a stronger neural network improve search results?
- What's the impact of an increased search budget on the performance?
- How well does the engine recognize wins and losses?
- Which is easier to learn, wins or draws?
- In terms of prediction accuracy, what is a good measure of a position's difficulty, and how does the accuracy change with more learning?
- Are there cases where search negatively impacts move prediction? If they exist, why do they occur?

1.3 Contributions of the Thesis

Our work makes the following main contributions:

- We show that the move prediction accuracy of the raw policy network is correlated with the sample size of each decision depth [1] more than with the decision depth.
- We show that in specific cases, the whole engine using networks combined with MCTS can be wrong even when the raw neural network is correct.
- In an AlphaZero style program, the policy and value head output do not always agree. We observe that, even if the policy correctly selects the correct move, the value head is not always correct in estimating the value of the position.
- If the value head error is consistently large in the sub-tree, then search leads to a worse result than no search.

[1]Endgame positions are categorized according to the distance to win or loss. These categories are referred to as the decision depth. More details will be provided in Section 3.1.2.

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 the hardware and software setup along with the network used. We discuss the parameter 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 is given in this chapter.

- **Chapter 5.** ***Experiments on a Five Piece Tablebase:*** In order to observe how our results scale for a larger tablebase, we discuss the experimental results on samples from the KQRkq five piece tablebase in this chapter.

- **Chapter 6.** ***Conclusion and Future Work:*** Concluding remarks about the research are stated and some future research directions based on our findings are proposed.

Chapter 2

Literature Review

In this chapter we review the necessary background materials for our experiments. Furthermore, we discuss related research. In this thesis we focus on two player perfect information zero sum games. So all the terminology we use, is for these types of games.

2.1 Game Tree Search

The game tree is a type of state space representation of game play. It is a directed graph that represents every possible way that the game can be played. The nodes represent the game position and the edges represent moves which lead from one state to another. Each level of the game tree represents a single turn of a player, sometimes also referred to as one ply. From a state every possible legal move for the current player is included as edges; the edges from the next state denote legal moves for the opponent. The root denotes the current game position.Terminal nodes denote the end of the game. There is a true value associated with every terminal node, which is determined by the outcome of the game. In chess, these values can be chosen as +1 for a win, -1 for a loss and 0 for a draw.

A search tree is a tree where leaf nodes may or may not be terminal nodes. A leaf node in a search tree may contain a true value or a heuristic value evaluation depending on whether it is terminal node or not. In the setting above, positive and negative heuristic values are in the range (-1,+1).

The strategy of playing a move which leads to the best possible outcome

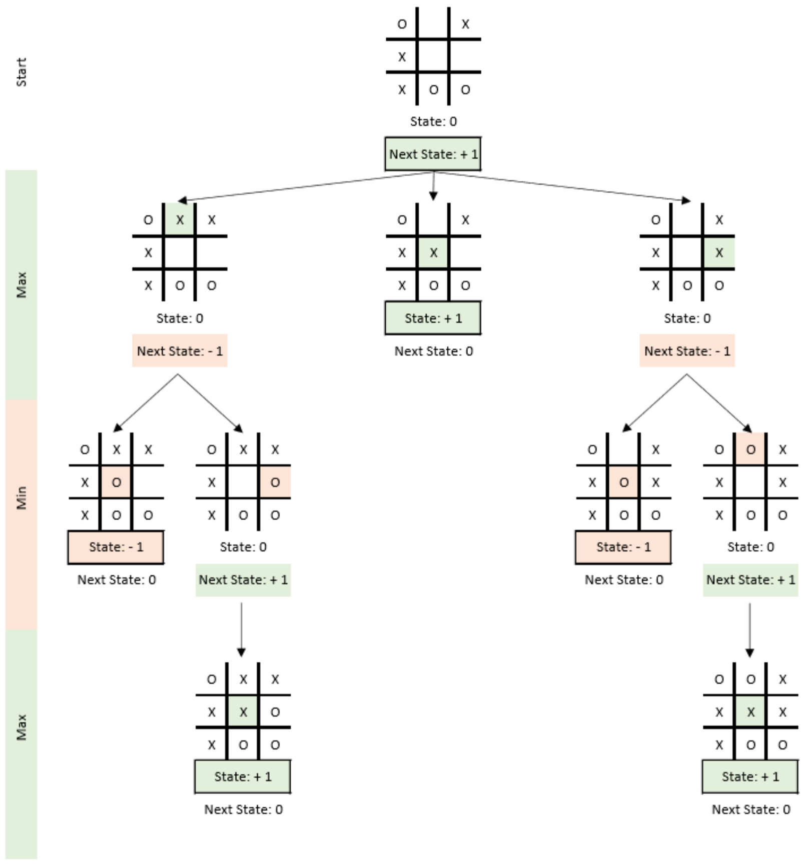


Figure 2.1: Game tree for a Tic-Tac-Toe position.

regardless of the opponent's strategy is called perfect play. A game is called strongly solved, when the perfect play from all legal positions of the game is known [3]. Chess as a whole is an unsolved game. However, endgames of up to seven pieces have been solved and the results are publicly available [1]. When a game is not solved the perfect play of the game is not known [3].

2.1.1 The Minimax Algorithm

To pick the optimal move from a state it is sufficient to determine the minimax value of each state of the game tree. The minimax value of a game tree is the value of the final outcome if both players (i.e. the player to move and the opponent) play perfectly [43].

A simple recursive algorithm called the minimax algorithm can be used to determine the minimax value of all the nodes rooted in the current state from the viewpoint of the current player. The recursion computes the minimax value by proceeding all the way down to the leaves and then backing up to the root.

Since the minimax algorithm performs a complete depth-first exploration of the game tree, with b legal moves at each node and a depth of d, the complexity of the algorithm is $O\left(b^d\right)$ [43]. For nontrivial games like Go or chess this cost is impractical.

2.1.2 Alpha-Beta Search

In the best case, the exponent of the minimax game tree complexity can be cut into half from $O(b^d)$ to $O(b^{d/2})$ by using alpha-beta search [43].

This algorithm maintains two values, *alpha* and *beta* to represent the minimum score guaranteed for the max player and maximum score guaranteed for the min player, respectively. The trick is to ignore the nodes that do not stand a chance to influence the final decision [43]. The algorithm can be easily understood with the example in Figure 2.2. From the example it can be seen that the minimax value of the root node can be determined without evaluating two of the leaf nodes.

2.1.3 Further Improvements

Though the generated tree size can be reduced from $O(b^d)$ to $O(b^{d/2})$, the exponent can not be eliminated. Moreover, cutoffs are not guaranteed. In the worst case, alpha-beta can not prune any nodes and the full tree would need to be searched.

However, if the best move is searched before other sub-optimal moves, alpha-beta search can effectively prune a huge number of game states [50]. Hence, the effectiveness of alpha-beta search depends on the order in which successors are examined. Therefore, move ordering heuristics can be used to further improve the performance of the alpha-beta algorithm.

In real games such as chess or Go, finding the exact solution of the game tree is often not feasible because of the size of the search tree. For example the game tree complexity of chess from the starting position is at least 10^{123}

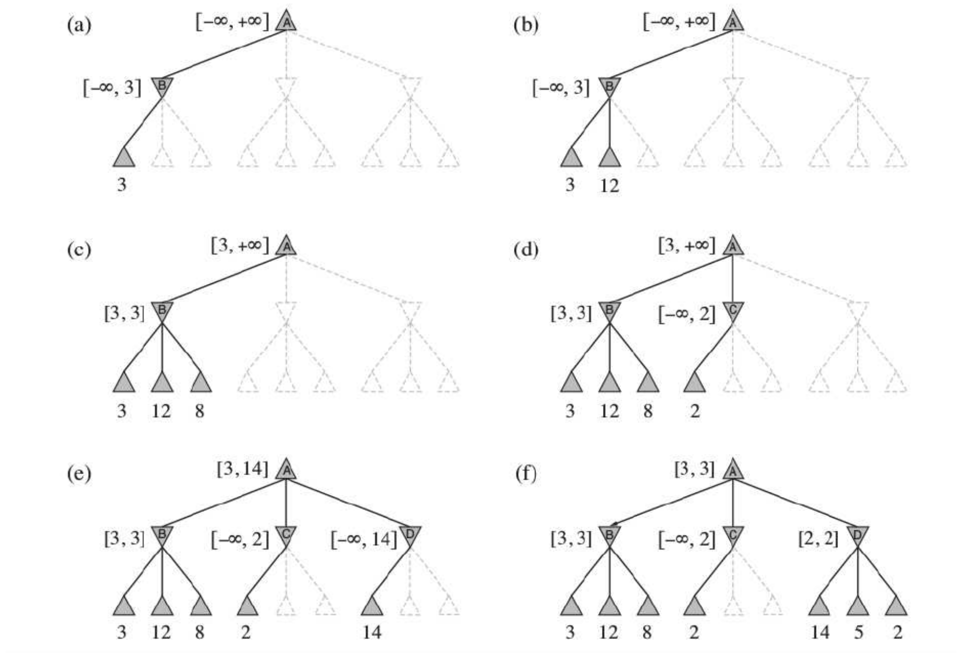


Figure 2.2: Determining the minimax value using alpha-beta search [43].

assuming a depth of 80 and a branching factor 35 [3]. Therefore, depth limited alpha-beta search with a heuristic evaluation function is used in such cases [44]. The search tree is generated up to a certain depth using alpha-beta search. When the maximum depth is reached the leaf nodes are evaluated using a heuristic evaluation function. These functions need lots of domain knowledge in order to estimate the expected value of a node and produce reliable search results. For example, chess experts used features such as the number and type of remaining pieces on the board, and many other 'positional' features, to produce a handcrafted evaluation function which estimates the outcome of a position.

Some features that impact the outcome can still go unnoticed by human experts when designing evaluation functions. An alternative method of heuristic evaluation is Monte Carlo [9] sampling where handcrafted knowledge is not required (though it can definitely help). More on Monte Carlo methods, including Monte Carlo Tree Search, will be provided in Section 2.4.

2.2 The Game of Chess and Computer Chess

Chess is a two player perfect information zero sum game. It is an ancient game which was brought to the western world from India through Persia. Back then it was a royal game, which was a measure of someone's intelligence and sharpness on the battlefield [52]. Later it also became a measure of machine intelligence.

Back in 1770 Kempelen built a chess-playing machine named the Mechanical Turk, which became famous before it was revealed that there was a person hiding inside [52]. Though the machine did not play chess by itself, it gave an idea for making a machine to play chess. After that, inventors tried to build chess playing machines using automata, which was not successful for centuries.

In 1950 Alan Turing proposed an algorithm for computers to play chess, where he acted as the automated agent by following the algorithm [30, 52]. Later in the same year Shannon devised a chess playing program and published a paper [44] that was recognized as a fundamental paper on computer chess [39]. In 1958 the chess program NSS defeated a beginner level human player for the first time [52]. However, it was not until 1997 that a computer program was able to beat a human world champion when Deep Blue [7] defeated Garry Kasparov in a match under standard chess tournament time controls. Deep Blue used an enhanced parallel alpha-beta algorithm for its game tree search [16].

2.2.1 Stockfish: A Strong Modern Chess Engine

After the development of Deep Blue, a number of strong open source chess engines were developed using alpha-beta search and heuristic evaluation functions. Among those, Stockfish [8] is one of the strongest modern open source chess engines. Of the 10 most recent TCEC tournaments, it won 8 times [51]. The engine uses alpha-beta search to find the best move and assigns one single value to each position by evaluating those positions. Initially this evaluation was a heuristic evaluation created by domain experts, where handcrafted features were used. In recent years the developers introduced neural network

Figure 2.3: A chess board [40].

evaluation, which takes the basic board position as input [8]. It can be tuned to play at different skill levels, from 1 (lowest) to 20 (highest). Weaker moves are selected with a certain probability at lower skill levels.

2.3 Endgame Tablebases

Because of the complexity of chess, it has not been possible to strongly solve the whole game. However, researchers have been able to solve chess endgame problems with a reduced number of pieces. The solutions include perfect play information (e.g. minimax optimal outcome, number of ply to win or lose) of all legal game states containing 3 to 7 pieces on the board. The results are stored in databases called endgame tablebases.

As there is a huge number of such positions, storing them requires significant space. If symmetric [19] positions are omitted for compactness, the query performance can be affected. Therefore, to generate these tablebases, one has to trade off between compactness of the tablebase and query speed [19]. In order to compress the tablebase while simultaneously speeding up queries, specialized indexing techniques are used [19].

2.3.1 Sources of Endgame Tablebases

A number of endgame tablebases are hosted on different websites [19, 33, 35]. They differ by storage space and metrics used. One can query these websites to get perfect play results. Moreover, whole tablebases [49] and generator source codes [5, 34] are also available. Syzygy [34] and Gaviota [5] tablebases are free and widely used in recent years, while Nalimov [35] is commercially available.

2.3.2 Probing

In order to get perfect play information from a tablebase, one has to query it. This is called tablebase probing. At first a given query position is checked for legality. After that, the position is mapped to a unique index. Then the result for the index is retrieved and presented for the particular symmetry of the board. Symmetry in chess endgame tablebases will be discussed further in Section 3.1.2.

2.3.3 Metrics

Different types of metrics are used in different tablebases. Each has its advantages. Some metrics provide more exact game paths to finish the game quickly whereas others take less space to store. The most often used metrics are [14]:

- **DTM:** Depth to Mate is the number of ply for a win or loss assuming the winning side plays the shortest way to win and the losing side plays the longest way to lose.
- **DTZ:** Depth to zeroing is the ply count to push a pawn or play a capture.
- **DTZ50:** If 100 consecutive plies of reversible moves are played, the player 'on move' can claim a draw. This is the 50 move rule. DTZ50 is the depth to zeroing in the context of the 50 move rule.

- **DTC:** Depth to conversion is the number of ply until the material value on the board changes, by capture and/or pawn-conversion. This means switching to a different tablebase.

2.4 Monte Carlo Tree Search

Monte Carlo methods were first used in statistical physics to estimate the probability of different outcomes that can not be easily estimated by any other mathematical approach [6, 15]. Later Abramson [2] used such methods to estimate the value of leaf nodes in a search tree. Using the game of Othello, he showed that the estimated value using Monte Carlo sampling was better than the handcrafted evaluation function of domain experts. Since then, Monte Carlo methods have been used in a wide variety of games.

In 2006, Coulom [9] proposed Monte Carlo Tree Search (MCTS), which directly combined Monte Carlo evaluation with tree search. The proposed algorithm starts with only one node at the root and uses random simulation to evaluate the estimated probability of winning. In general the family of MCTS approaches applies four steps per iteration [6]:

- **Select**: Starting from the root a child is iteratively selected to descend through the tree until the most urgent expandable (i.e. non-terminal and has unvisited children) node is reached.

- **Expand**: From the available actions, one or more node(s) are added to expand the tree.

- **Rollout**: Complete one or more random rollouts from a newly added node to produce the value estimate. From the newly added node, rollout plays following a randomised policy until the game ends.

- **Backup**: Backpropagate the rollout result along the path to the root to update the node statistics.

Figure 2.4 shows one iteration of general MCTS approach. The pseudo-code is given in Algorithm 1.

Algorithm 1 General MCTS approach [6]

```
 1: function MCTS($s_0$)
 2:     create root node $v_0$ with state $s_0$
 3:     while within computational budget do
 4:         $v \leftarrow v_0$
 5:         while $v$ is nonterminal do
 6:             if $v$ is not fully expanded then
 7:                 $v \leftarrow$ Expand($v$)
 8:             else
 9:                 $v \leftarrow$ SelectChild($v$)
10:         $v_l \leftarrow v$
11:         $\Delta \leftarrow$ PlayOut($s(v_l)$)
12:         Backup($v_l, \Delta$)
13:     return $a$(BestChild($v_0$))
14: function SelectChild($v$)
15:     return $v' \epsilon$ children of $v$ according to child selection policy
16: function Expand($v$)
17:     choose $a \epsilon$ untried actions from $A(s(v))$    ▷ $s(v)$ is the state of node $v$
18:     add a new child $v'$ to $v$
19:          with $s(v') = f(s(v), a)$                ▷ $f()$ is state transition function
20:          and $a(v') = a$                          ▷ $a(v')$ is the incoming action to $v'$
21:     return $v'$
22: function PlayOut($s(v)$)
23:     while $s$ is nonterminal do
24:         choose $a \epsilon A(s)$ uniformly at random ▷ $A(s)$: action space, $a$: action
25:         $s \leftarrow f(s, a)$
26:     return reward for state $s$
27: function Backup($v, \Delta$)
28:     while $v$ is not null do
29:         $N(v) \leftarrow N(v) + 1$                           ▷ $N(v)$: total visit count
30:         $X(v) \leftarrow X(v) + \Delta(v, p)$   ▷ $X(v)$: total reward, $\Delta(v, p)$ component
    of reward vector associated with current player $p$ at node $v$
31:         $v \leftarrow$ parent of $v$
32: function BestChild($v$)
33:     return best $v' \epsilon$ children of $v$          ▷ The exact definition of 'best' is
    defined by implementation
```

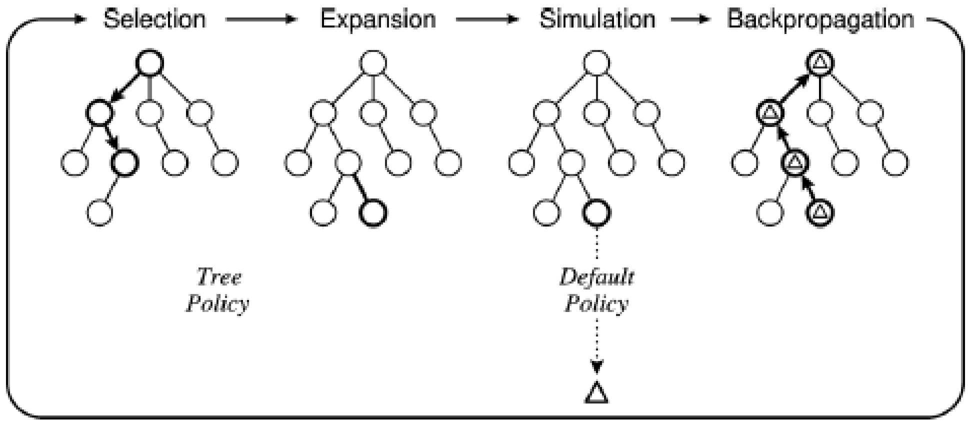


Figure 2.4: One iteration of general MCTS approach [6].

The Upper Confidence Bound (UCB) algorithm is commonly used as the child selection policy. At the end of the search the child of the node that has the highest number of visits is chosen as the move.

2.4.1 UCB for Trees (UCT)

MCTS traverses a search tree by iteratively selecting one child of a node. In a game, a player selects a move from K legal moves that maximizes the reward and minimizes regret. To choose the next move, a player has to exploit the current best move while exploring other moves to find a more promising one. Therefore, the selection can be viewed as a K armed bandit problem [4], where the value of a child node is the state's expected reward.

The most popular algorithm for the multi-armed bandit problem is UCB1 [4], which chooses the action that maximizes the UCB1 value [6]:

$$UCB1 = \bar{X}_j + \sqrt{\frac{2ln(n))}{n_j}}, \tag{2.1}$$

where $\bar{X}_j$ represents the average reward from arm j, n_j is the visit count of j, and n is the overall number of plays so far.

Kocsis and Szepesvari [18] proposed the upper confidence bound for trees (**UCT**) method, which applies a modification of the UBC1 formula at the selection phase of MCTS. Here a child node j is selected to maximize:

$$UCT = \bar{X}_j + C\sqrt{\frac{ln(n))}{n_j}} \tag{2.2}$$

Here C is a constant which controls the level of exploration, n is the visit count of the parent node and the other terms are as in Equation 2.1.

2.4.2 PUCB and PUCT

Given unlimited resources, UCT eventually converges to the optimal move [6]. However for a small search budget it may not converge to the minimax value, due to sampling error or not finding the strongest move in each tree node. As the number of samples increases, the estimated (average) value of each action tends to become more accurate. Without any prior knowledge, every action needs to be sampled at least once for an initial value. This process wastes a significant portion of the search budget on weaker actions, especially for games with large branching factors.

Spending more of the budget on searching promising nodes improves the performance of UCT significantly. Using this idea, Rosin [42] proposed to use a predictor that places a prior weight on the available actions. The search budget is then prioritized for promising actions, thereby reducing the waste of having to sample many poor actions. The proposed algorithm is named Predictor + Upper Confidence Bound (the term PUCT is used to refer to PUCB applied for game trees), which is a modification of the UCB1 algorithm.
PUCB differs from UCB1 as it does not require pulling all the arms once in the beginning. Also the equation is slightly different from Equation 2.1. It selects the action that maximizes:

$$PUCB = \bar{X}_j + c(n, n_j) - m(n, j) \tag{2.3}$$

Here,
$c(n, n_j) = \sqrt{\frac{3log(n))}{2n_j}}$ if $n_j > 0$, otherwise 0
$m(n, j) = \frac{2}{M_j}\sqrt{\frac{log(n)}{n}}$ if $n > 1$, otherwise $\frac{2}{M_j}$

M_j is the prior weight placed on arm j.
All other notations hold the same meaning as in Equation 2.1. Obtaining a predictor to use with PUCB can be done in two ways [42]: offline training that maximizes the weight of the optimal arm, or fitting a probability distribution over all arms, with the target reflecting the probability that a specific arm is optimal.

2.5 The AlphaZero Approach

AlphaZero is a major breakthrough in the field of AI. Without any expert knowledge it can achieve superhuman performance in many challenging games for AI [46]. AlphaZero won against AlphaGo Zero, which defeated AlphaGo, the first program to defeat a world champion in Go. AlphaZero beat Stockfish-8, the top chess engine of the 2017 TCEC Championship [17]. AlphaZero also defeated ELMO, the champion of the 2017 Computer Shogi Association (CSA) championship. In this section we discuss the development of AlphaZero and some open source AlphaZero based approaches.

2.5.1 AlphaGo

AlphaGo [45] was the first computer program that defeated a human professional player in the full sized game of Go. The program uses a *value network* to evaluate a board position and a *policy network* to prioritize moves in the search. The networks are trained using a pipeline of several stages of machine learning as shown in Figure 2.5.

The training starts with the supervised learning (SL) of the policy network from human expert game records. A rollout policy is also trained for repeatedly selecting moves during rollout. The SL policy network was then used to initialize the reinforcement learning (RL) policy network. The RL policy network is further improved using the policy gradient method to maximize the outcome against previous versions of the policy network. Using the RL policy network, a new dataset of self-play games is generated, which is later used to train the value network.

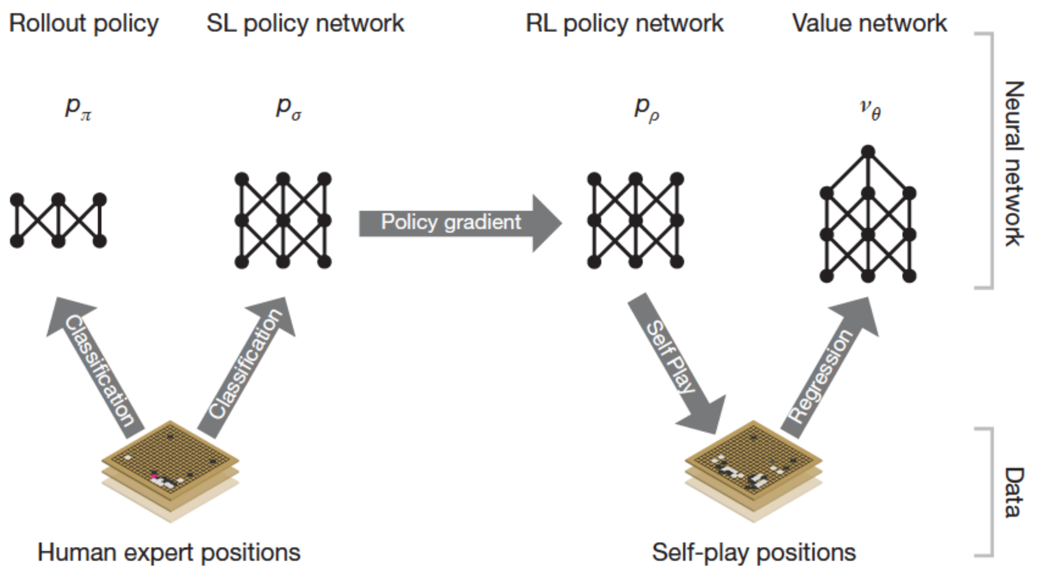


Figure 2.5: AlphaGo network training [45].

The policy and value networks are combined in an MCTS algorithm. Figure 2.6 shows the MCTS in AlphaGo. For selection of an action a variant of the PUCT algorithm is used. At each time step t from state s_t an action a_t is selected such that [45]:

$$a = \arg\max_a \left(Q(s_t, a) + U(s_t, a)\right) \tag{2.4}$$

Here,

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

$Q(s,a)$ is the action value, the average reward for choosing an a from state s. $N(s,a)$ is the visit count, and $P(s,a)$ is the prior probability that the action a is best in s from the SL policy network output.

2.5.2 AlphaZero

After the success of AlphaGo, AlphaGo Zero [48] was developed. Unlike AlphaGo, AlphaGo Zero starts learning from random play without any human knowledge by self-play reinforcement learning. Moreover, it simplifies and improves AlphaGo. It takes only the board position as input features and trains

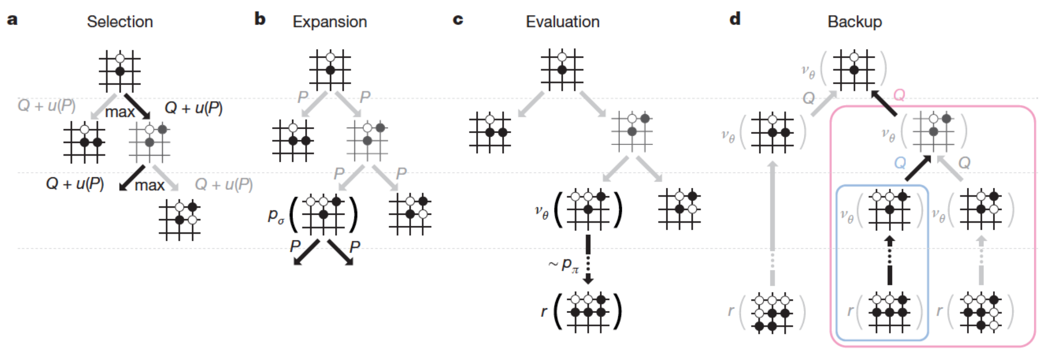


Figure 2.6: MCTS in AlphaGo [45].

a single network with two output "heads" for policy and value. Lastly, it removed the Monte Carlo rollout and uses only neural network evaluation.

AlphaZero [47] is a more generic version of AlphaGo Zero From zero domain knowledge AlphaZero can achieve superhuman level performance in the games of chess and shogi as well as Go. As rotation and reflection can not be applied in chess and shogi, AlphaZero does not take symmetry into consideration as AlphaGo Zero did. Moreover, instead of waiting for an iteration to complete, and evaluating it against the best previous network, AlphaZero maintains only one network that is updated continually [47]. Despite these differences, the training process of AlphaZero is similar to AlphaGo Zero. Figure 2.7 shows an overview of the self-play reinforcement learning in AlphaZero.
The self-play training starts with a random weight initialization of the neural network. The network takes the raw board position as input and returns the value v and a vector of move probabilities $\boldsymbol{p}$ as output. The program plays a game against itself where the moves are selected according to the search probabilities π computed by MCTS. At each time step t, the MCTS execution is guided by the neural network f_θ. Figure 2.8 shows the MCTS used by AlphaZero. It is similar to the one used in AlphaGo, except the rollout is removed and the single value evaluation by the neural network is used.

For each time step t, the state s_t, search probabilities π_t and the final outcome z_t is stored. The neural network parameters θ are adjusted to maximize the similarity between the move probabilities $\boldsymbol{p}$ and the search probabilities $\boldsymbol{\pi}$ while minimizing the error between the predicted value v and the self-play

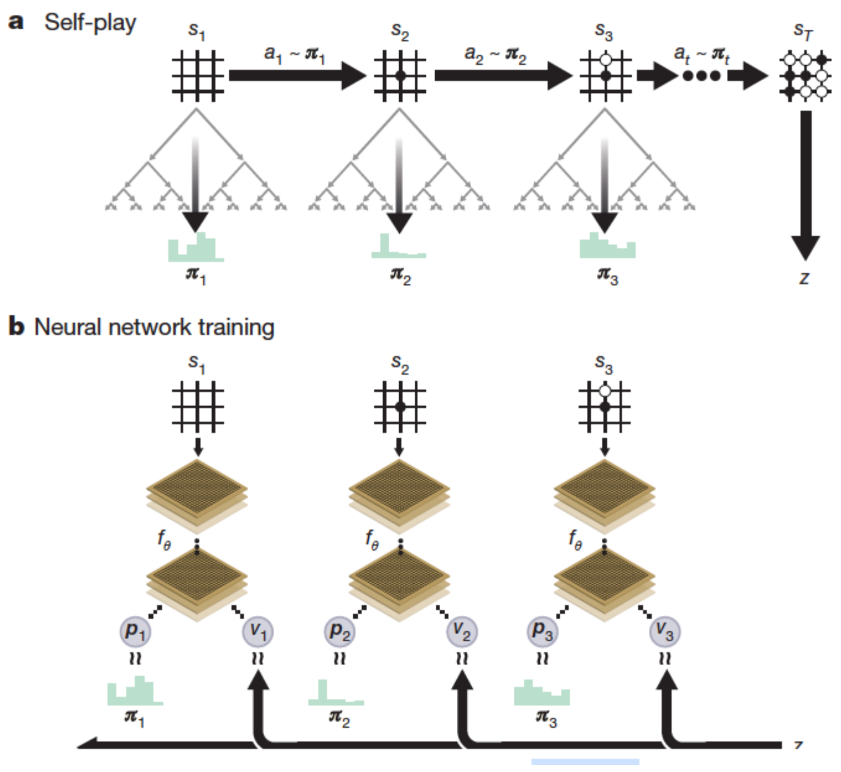


Figure 2.7: Self-play reinforcement learning in AlphaZero [47].

outcome z. The parameters θ are adjusted by gradient descent to minimize the loss function:

$$l = (z - v)^2 - \boldsymbol{\pi}^\top log\boldsymbol{p} + c||\theta||^2 \tag{2.5}$$

2.5.3 Leela Zero

Leela Zero [29] is an open source reimplementation of AlphaGo Zero. It uses distributed resources to train the networks. Members of the community con-

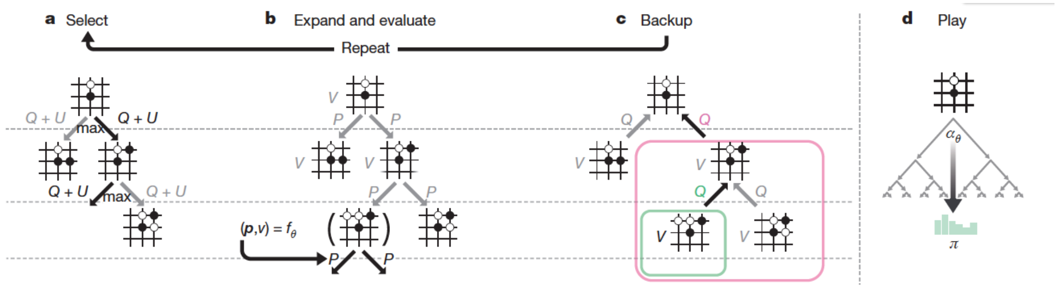


Figure 2.8: Monte Carlo tree search in AlphaZero [47].

tribute their computing resources to generate self-play training data and optimize the neural networks. A server website coordinates the training process. The clients are connected to the server to provide their computing resources.

2.5.4 Leela Chess Zero

Leela Chess Zero(Lc0) is an adaptation of Leela Zero for chess. It is also a distributed effort to reproduce the result of AlphaZero in chess [26]. Like AlphaZero, Lc0 takes a sequence of consecutive raw board positions as input. It uses the same two-headed (policy and value) network architecture and the same search algorithm as AlphaZero. Though the original project was inspired by the AlphaZero project, in 2020 Lc0 surpassed AlphaZero's published playing strength in chess [27].

Over time the developers of Lc0 introduced enhancements that were not in the original AlphaZero. For example, an auxiliary output called the *moves left head* was added to predict the number of plies remaining in the current game [12]. Another auxiliary output called the *WDL* head separately predicts the probabilities that the outcome of the game is a win, draw, or loss [32].

In addition to network heads, Lc0 uses two distinct training methods to generate different types of networks that differ in playing strength. T networks are trained by self-play, as in AlphaZero, while J networks are trained using self-play game records generated from T networks [28]. J networks are stronger and are used in tournament play.

2.6 Related Work

Our work is essentially a systematic measurement of one of the strongest open source game engines for chess. In the original AlphaZero paper [47], the performance was measured in comparison with specialized programs for Go (AlphaGo Zero), chess (Stockfish), and shogi(Elmo). First, AlphaZero was compared with other engines based on its win-draw-loss percentage in a tournament against the baselines. Under the same time settings, AlphaZero surpassed the performance of the specialized programs. Second, the AlphaZero program was evaluated again under reduced time settings against the baselines for chess and shogi. For chess, AlphaZero achieved a higher win-rate as both White and Black with one-third of thinking time than given to Stockfish. The ELO rating of AlphaZero compared to specialized engines was also computed based on tournament results, with one second thinking time. AlphaZero surpassed the ELO rating of all the specialized programs in chess, shogi, and Go.

In terms of comparing game engine or algorithm performance to perfect play, there are also two papers that are worth mentioning. First, in 2006, Lassabe *et al.* [20] proposed using genetic programming to solve chess endgames, rather than the classic method of first using brute force search, then storing the results in a table. The authors combined elementary chess patterns, defined by domain experts, and applied their algorithm on the KRk endgame. The resulting endgame engine was then evaluated by playing 17 reference KRk endgames against the Nalimov tablebase, with the engine as attacker and perfect play as defender. For all 17 specific endgames, the genetic engine was able to achieve the perfect outcome, but was unable to do so with the optimal (least) number of moves. Performance was therefore measured by move count to win. For reference, the average optimal move count was 9 moves, while their algorithm was able to achieve the same outcome in 11 moves.

Second, Romein and Bal [41] first solved awari, then used the solution as a basis to measure performance for two world champion level engines in the 2000 Computer Olympiad. Observing the two programs' line of play in that tournament, the authors counted the number of wrong moves made on each side. The performance was measured as a percentage of right moves played; both engines were able to achieve an accuracy of more than 80%.

Chapter 3

Experimental Setup

In this chapter we discuss dataset preprocessing and parameter tuning for our experiments. First we discuss the dataset used and the steps involved in preprocessing in Section 3.1. In Section 3.2 we describe details of the chess program used and the relevant parameter settings to closely mimic AlphaZero for chess.

3.1 Tablebases and Datasets

To evaluate the performance of AlphaZero in terms of finding exact solutions, a ground truth is required. For that, we used chess endgame tablebases, which contain perfect play information of all positions up to 7 pieces on the board. In this section we discuss how the tablebases are processed for analysis and which tablebases are used.

3.1.1 Datasets

A number of endgame tablebases is available online, as discussed in Section 2.3.1 . Among them, we used the open source Syzygy [33] and Gaviota [5] tablebases as ground truth for the DTZ and DTM metrics.

As these tablebases are compressed using different types of indexing [10, 19], it is not directly possible to iterate through all unique legal positions. Hence, we converted the relevant tablebases to a plain-text format for our experiments.

3.1.2 Dataset Preprocessing

To create our dataset in plain-text format we followed several steps:

Step 1: Generate all unique legal positions

First we generate all unique legal positions using the method proposed by Kryukov [19] for each tablebase used. The positions are stored in the FEN format. We tested each position for uniqueness and legality.

(a) Uniqueness: If multiple positions are in the same equivalence class, only one entry of that class is generated. Positions can be equivalent by board mirroring, rotation, of color swapping. Figure 3.1 shows an example of an equivalence class.

(b) Legality: In chess, a position is legal if and only if:

- It has exactly one king of each color.
- It has no pawn on ranks 1 and 8.
- The side to move can not capture the opponent's king.

Step 2: Create MySQL database

We use a MySQL database to store all plain text tablebase information. For our experiments the transitive dependencies between two endgame tablebases are not required. Therefore this information can be safely omitted. We create a separate table for each different combination of pieces.

Step 3: Probing and storing positional information

To get the perfect play information, we iterate through all the positions generated in step 1. For each position we use the probing tool from python-chess [11] to access the Syzygy and Gaviota tablebases. For each position, the stored features are presented in Table 3.1. We use the term *decision depth* to categorize positions in an endgame tablebase. For winning positions, the decision depth is the DTM score of the position. For drawing positions where losing moves exist, the decision depth is the highest DTM of a losing successor.

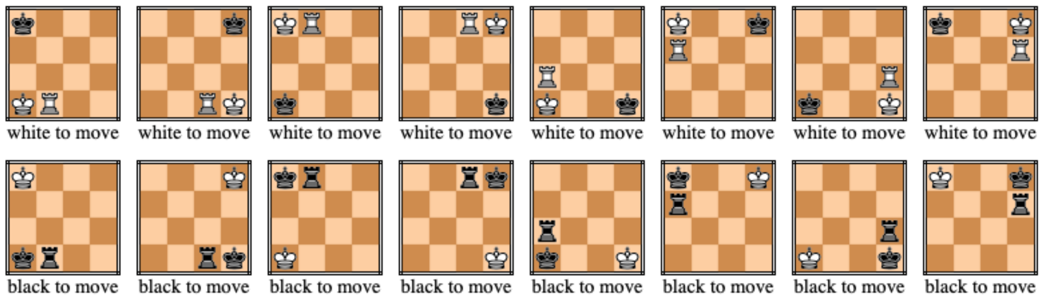


Figure 3.1: Example of equivalence class on a 4x4 board [19].

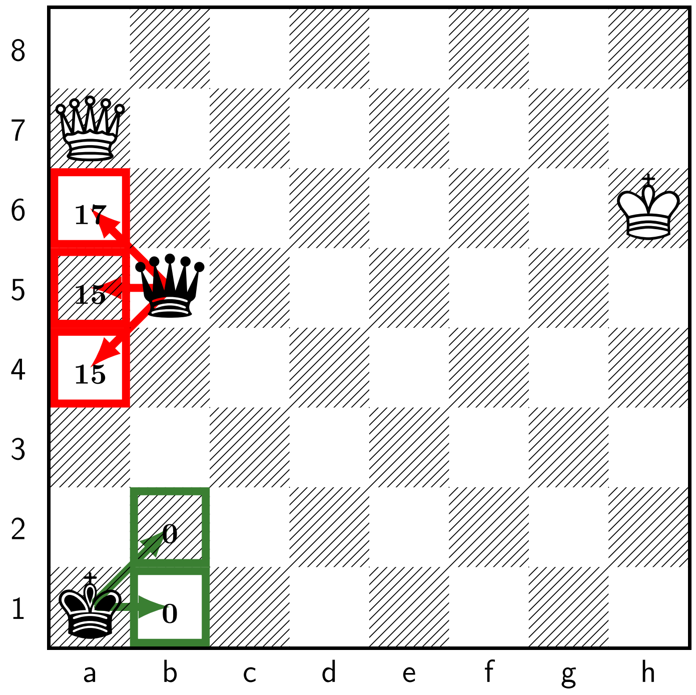


Figure 3.2: Example for decision depth.

Feature	Format
Position	FEN
Winning moves	Universal chess interface (UCI)
Drawing moves	UCI
Losing moves	UCI
WDL score	number
DTZ score	number
DTM score	number
Decision depth	number

Table 3.1: Features stored in the MySQL database.

Figure 3.2 illustrates the definition of decision depth. In the figure it is Black's turn to play. The green squares represent drawing moves, while red represents losing moves. The decision depth of this position is:

$$\textit{decision depth} = \textit{max}\,(15, 15, 17) = 17 \tag{3.1}$$

3.1.3 Tablebase Selection

Due to our limited computational resources, we select a subset of 3, 4 and 5 piece tablebases based on their win-draw-loss statistics to analyze.

- **Simple 3 piece tablebases**

  We selected 3 piece tablebases which contain both winning and drawing positions. These endgames are more balanced in that Black has a chance to lose in the drawing positions.

  We omit tablebases where all positions are drawing, or all the legal moves from almost all positions are drawing. These do not provide interesting situations for analysis. For example we did not study KBk and KNk.

- **Interesting 4 piece tablebases**

  There are two types of 4 piece tablebases: (a) two pieces for each side, (b) three pieces on one side and one piece (the king) on the other. We analyze only the former because balanced situations tend to have more interesting properties. More specifically, the side with just one king has only drawing moves at best, and even those are rare.

Tablebase	Piece		Win		Draw	Selected
	White	Black	White	Black		
KQk	KQ	k	93.7	0	6.3	✓
KRk	KR	k	94.4	0	5.6	✓
KBk	KB	k	0	0	100	
KNk	KN	k	0	0	100	
KPk	KP	k	67.2	0	32.8	✓

Table 3.2: Three piece tablebases with ratio of outcome.

Imbalanced positions like these tend not to be informative about an engine's performance. Next, among all type (a) tablebases, we selected those in which both sides have at least some winning positions. This method of selection ensures spending our limited resources and efforts on a diverse collection of outcomes for each position (i.e winning, drawing, and losing moves).

Tablebase	Piece		% of Win		% of Draw	Selected
	White	Black	White	Black		
KQkq	KQ	kq	21.1	21.1	57.8	✓
KQkr	KQ	kr	80.7	15.7	3.5	✓
KQkb	KQ	kb	86.7	0	13.3	
KQkn	KQ	kn	88.3	0	11.7	
KRkr	KR	kr	14.9	14.9	70.2	✓
KRkb	KR	kb	81.6	0	18.4	
KRkn	KR	kn	28.2	0	71.8	
KBkb	KB	kb	0.0025	0.0025	99.995	
KBkn	KB	kn	0(3 positions)	0(6 positions)	100	
KNkn	KN	kn	0(6 positions)	0(6 positions)	100	
KPkq	KP	kq	4.6	87.9	7.5	✓
KPkr	KP	kr	9.1	77.5	13.4	✓
KPkb	KP	kb	14.9	0	85.1	
KPkn	KP	kn	23	0	77	
KPkp	KP	kp	33.3	33.3	33.4	✓

Table 3.3: Four piece type (a) tablebases with ratio of outcome.

- **Hard 5 piece tablebase**

  In order to observe how our analysis scales to larger tablebases, we conducted one experiment on a 5 piece tablebase: the KQRkq tablebase.

Considering our limited resources, we selected this pawn-less tablebase. It has a high range of decision depths, that ensures that there are difficult positions for human players. We randomly sampled 1% of all unique KQRkq positions, and checked legality for the sampled positions afterwards.

In each selected tablebase, we further restricted our analysis to positions where there are two or more game outcomes among all the legal move choices.

3.2 Leela Chess Zero Settings

We chose Lc0 0.27 [23] as our AlphaZero-style program in our analysis, because it is both publicly available and strong. Lc0 has evolved from the original AlphaZero in many ways, but with the proper configuration, it can still perform similarly.

3.2.1 Parameter Settings for Lc0

A number of parameters can be set for Lc0 [22, 25]. In our experiments we kept the default value for most of them, with the exception of the following:

- **Smart Pruning Factor**
  AlphaZero selects the node having the highest PUCB value during MCTS. Throughout our experiments we observed that the Lc0 search chooses nodes with lower PUCB values occasionally [36]. This is due to smart pruning, which uses the simulation budget differently: it stops considering less promising moves earlier, resulting in less exploration. To preserve the standard AlphaZero behaviour, we disabled smart pruning with the Lc0 option –smart-pruning-factor=0.

- **Number of Execution Threads**
  When analyzing positions with Lc0, the move choices need to be consistent. However, Lc0 is non-deterministic between different runs when using the default number of threads [37]. The contributors confirmed that deterministic multi-threaded search has not yet been implemented.

In order to get a deterministic search result, we set the number of threads to 1 by using the Lc0 option –threads=1.

3.2.2 Backend for Neural Network Evaluation

Lc0 supports a number of neural network backends [31]. Since we used Nvidia Titan RTX GPUs for our experiments, we chose the `cudnn` backend.

3.2.3 Network Used

The Lc0 project saves a network snapshot whenever the network is updated. The Lc0 training process creates a large number of snapshots. For the T60 generation training that we use, there are more than 10,000 snapshots as of October 2021 [24]. For this research we chose two specific snapshots:

- **Strong network:** ID 608927, with (self-play) ELO rating of 3062.00 was the best-performing T60 snapshot up to May 2, 2021. It uses the largest network size (30 blocks $\times$ 384 filters) [21].
- **Weak network:** ID 600060, with a rating of 1717.00, was a network of the T60 generation after 60 updates starting from random play. This snapshot was created before adding the "WDL" and "moves left" auxiliary targets and so it has a smaller size of 24 blocks $\times$ 320 filters.

3.3 Computational Resources

We preprocessed the tablebases with a single-threaded program that uses one CPU core. This includes generating all unique legal positions, creating the MySQL tablebase and probing and storing positional information for all unique legal positions. As the tablebases are compressed, probing them took the most time. For example, creating all unique legal positions of the KQkq tablebase took 3 CPU hours, while probing and inserting the results into a database took approximately 36 CPU hours. Using this as an estimate, probing the full KQRkq tablebase would take approximately 82 CPU days. Therefore we took only a 1% sample from this tablebase.

To test the engine performance we run four engine settings simultaneously. The engine settings are discussed on Section 4.3.2. The program uses CPU for search and Nvidia TITAN RTX GPU for the neural network evaluation. As we set the number of threads to 1, only one CPU core was working for search of each engine. Therefore, the search took the largest amount of time during our main experiment. For example, running the experiment for the KQkq tablebase took 30 days to test all four engine settings.

Chapter 4

Experiments and Analysis

In this chapter we discuss the experiments done and the findings of the experiments. The experiments are closely related to each other. Some of them were built on the findings of previous ones. Therefore, we discuss the experiments in the way they progressed.

4.1 Quick Overview of Experiments

For all of our experiments, each engine plays one move from each position. We define a *mistake* as a move decision that changes the game-theoretic outcome, either from a draw to a loss, or from a win to not a win. Any outcome-preserving decision is considered correct. The engine does not need to choose the quickest winning move. Accuracy is measured in terms of the number or frequency of mistakes, over a whole database.

4.2 Experiment 1: How Does More Learning Improve Playing Strength?

Our experiment starts with evaluating a policy player, which does not use any search to play. It plays the move for which the neural network's policy output is highest. We use the terms *policy* and *Raw NN* interchangeably to refer to this policy player. In this experiment we evaluate the impact of learning on playing strength. We evaluate both networks discussed in Section 3.2.3 for the three and four piece endgame positions.

Table 4.1 shows the total number of mistakes by the weak and the strong policy. In the table, the weak policy has a much higher rate of mistakes than the strong policy. The result suggests that policy approaches perfect play with more training on these positions.

EGTB	**Total Number of positions**	**Weak network**	**Strong network**
KPk	8,596	390	5
KQk	20,743	109	**0**
KRk	24,692	69	**0**
KQkq	2,055,004	175,623	3,075
KQkr	1,579,833	141,104	4,011
KRkr	2,429,734	177,263	252
KPkp	4,733,080	474,896	20,884
KPkq	4,320,585	449,807	6,132
KPkr	5,514,997	643,187	13,227

Table 4.1: Total number of mistakes by strong and weak policy in 3 and 4 piece endgames.

4.3 Experiment 2: How Does Search Improve Playing Strength?

To analyze how search improves accuracy, we compare the search budgets of 0 (raw policy) and 400 simulations per move decision. We call these settings MCTS-0 = policy and MCTS-400. We chose these relatively low settings considering our limited computational resources. We used the same move decision and accuracy measurement as in the previous experiment.

4.3.1 Policy vs Policy+Search

Table 4.2 shows the percentage of errors for MCTS-400. The search result is a substantial improvement compared to the raw policy in Table 4.1. While a strong network by itself can work very well, search strongly improves performance in each case where the network alone is insufficient.

This is because policy networks are trained to increase overall playing strength, while the search on one specific position is more specifically looking ahead at the state space of the position initially guided by the policy. Furthermore, AlphaZero also optimizes its search algorithm during learning.

EGTB	**Total Number of positions**	**MCTS-400 with weak network**	**MCTS-400 with strong network**
KPk	8,596	13	0
KQk	20,743	0	0
KRk	24,692	0	0
KQkq	2,055,004	12,740	36
KQkr	1,579,833	3,750	46
KRkr	2,429,734	6,097	0
KPkp	4,733,080	41,763	423
KPkq	4,320,585	46,981	13
KPkr	5,514,997	60,605	196

Table 4.2: Error rate using MCTS-400 with strong and weak networks on different endgame tablebases.

4.3.2 Varying the Search Effect: Small Number vs Large Number of Nodes

We continued our experiments with increased search budgets of 800 and 1600 nodes, to analyze how deeper search influences accuracy. Table 4.3 shows the number of mistakes using MCTS-0, MCTS-400, MCTS-800, MCTS-1600 for all three and four piece tablebases. Lc0 uses 800 simulations during self-play training of the networks. Therefore we selected budget MCTS-400, 800 and 1600 to evaluate search performance using half, same and double budget of self-play training simulation budget.

The experimental result shows that deeper search consistently helps predicting an accurate move for all these tablebases.

4.3.3 Impact of a Strong Policy on Search

From Table 4.1 it is clear that a strong policy makes a significantly lower number of mistakes than a weak policy. While search can improve the prediction for both strong and weak policy, the stronger policy gives search a more accurate baseline for further improvement. For example in KPk, the total number

EGTB	**Number of mistakes**							
	Weak network				**Strong network**			
	0	**400**	**800**	**1600**	**0**	**400**	**800**	**1600**
KPk	390	13	0	0	5	0	0	0
KQk	109	0	0	0	0	0	0	0
KRk	69	0	0	0	0	0	0	0
KQkq	175,623	12,740	8,999	4,429	3,075	36	24	6
KQkr	141,104	3,750	2,562	1,547	4,011	46	18	1
KRkr	177,263	6,097	1,717	644	252	0	0	0
KPkp	474,896	41,763	28,678	19,066	20,884	423	129	52
KPkq	449,807	46,981	27,395	18,756	6,132	13	5	4
KPkr	643,187	60,605	46,308	35,549	13,227	196	73	26

Table 4.3: Total number of errors using weak and strong network for MCTS with search budgets of 0 (raw policy), 400, 800 and 1600 simulations.

of mistakes of MCTS-400 using the weak policy is still higher than the raw strong policy. As a result a strong policy can lead the search in a more accurate direction than a weaker policy.

Note that, while evaluating the performance of AlphaZero we expect the network to be a fully trained strong network. Therefore, we conducted our next experiments using the stronger network only.

4.4 Experiment 3: Are Wins or Draws Easier to Play?

We added an experiment to find which types of positions Lc0 plays better, wins or draws. We separated the results of our previous experiments based on the perfect play outcome. Table 4.4 shows the $\log_{10}$ error rate of all versions separately for winning and drawing positions. In most cases the fraction of mistakes is higher for winning than for drawing positions.

EGTB	log(% of Error)				
	Type	MCTS-0	MCTS-400	MCTS-800	MCTS-1600
KPk	Win	-1.0968	None	None	None
	Draw	None	None	None	None
KQk	Win	None	None	None	None
	Draw	None	None	None	None
KRk	Win	None	None	None	None
	Draw	None	None	None	None
KQkq	Win	-0.5284	-2.4934	-2.7944	None
	Draw	-1.5623	-3.2713	-3.0952	-3.2713
KQkr	Win	-0.8144	-2.8522	-3.3963	None
	Draw	0.3040	-1.5336	-1.8523	-2.9315
KRkr	Win	-1.6324	None	None	None
	Draw	-2.3773	None	None	None
KPkp	Win	-0.3572	-1.9897	-2.4816	-2.8348
	Draw	-0.3514	-2.2087	-2.8201	-3.4829
KPkq	Win	-2.9277	-5.5310	-5.8734	-5.9703
	Draw	-2.5316	-5.4661	None	None
KPkr	Win	-0.6945	-0.3858	-2.4878	-2.3077
	Draw	-2.9148	-2.7426	-3.5412	-2.8839

Table 4.4: Error rate for winning and drawing move predictions for MCTS with search budgets of 0 (raw policy), 400, 800 and 1600 simulations.

4.5 Experiment 4: How Well Does Lc0 Recognize Wins And Losses at Different Decision Depths?

We conducted a more detailed experiment to find out how accurately the engine differentiates different types of moves (*e.g.* winning, drawing and losing) at different decision depths. As the overall error rate on the 3 piece endgame tablebase is very small for the strong network, we omit these results from this experiment.

With the four piece EGTB we tested again if the engine can differentiate a winning move from drawing and losing moves when the position is winning, and drawing moves from losing when the position is drawing. This time, we separate statistics by decision depth. For this experiment, if the program can play a winning move in a winning position, we assume it can differentiate at least one winning move from all drawing or losing moves. Similarly, for a drawing position, if the engine can play a drawing move we assume it can differentiate at least one drawing move from all losing moves.

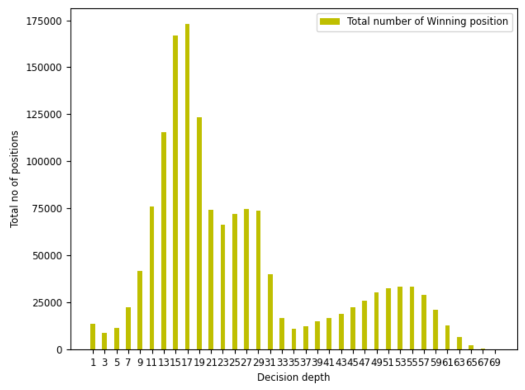
4.5.1 Error Rate of Raw NN at Different Decision Depths

We measure the percentage of errors made by Lc0 at each decision depth. Figure 4.1 shows the total number of positions and the percentage of errors made by the raw NN as a function of decision depth in the KQkr tablebase. The total number of positions and the percentage of errors made by raw NN as a function of decision depth for other four piece tablebases is shown in Appendix A.

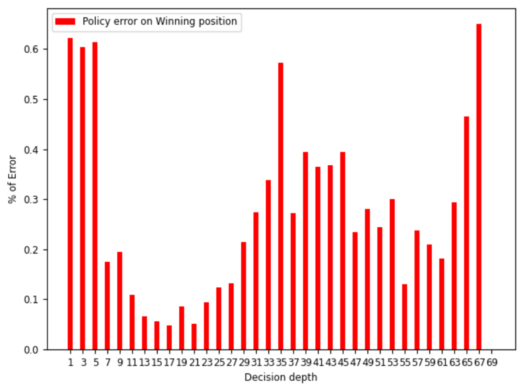
The percentage of mistakes at each decision depth is higher for drawing positions than winning positions for this specific KQkr tablebase. The mistakes by the raw neural network are higher at decision depths where there is only a small number of positions. This holds even for 'easy' positions with low decision depth. Figure 4.2 shows that there is an inverse relationship between the sample size at each decision depth and the error rate of the raw net for each tablebase. Here, the X-axis represents the percentage of samples at each decision depth and the Y-axis shows the error rate at each decision depth on a natural log scale.

4.5.2 Error Rate of MCTS-400 at Different Decision Depths

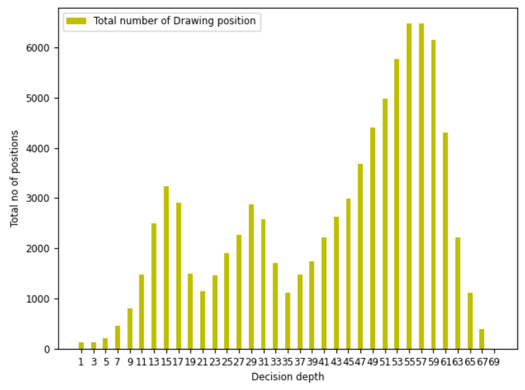
We repeated the experiment of Section 4.5.1 with MCTS-400 instead of the raw NN. As the number of mistakes significantly decreases with more search we only used MCTS-400. Figure 4.3 shows the result for the KQkr tablebase. The result for other four piece tablebases are given in Appendix A The engine only makes mistakes in positions with high decision depths. Search can routinely solve positions with shallower decision depths regardless of the policy accuracy. At higher depths, some errors remain, but there is no simple relation between decision depth and error rate there. Moreover, there is no relationship between search error and sample size at each decision depth.



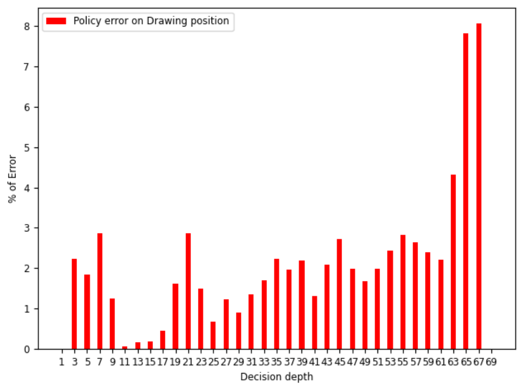
(a) Total number of winning position.



(b) Error rate on winning positions.

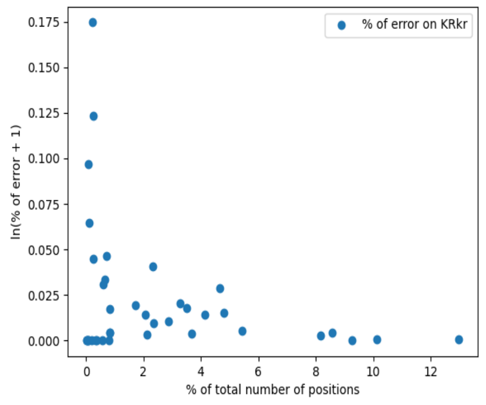


(c) Total number of drawing position.

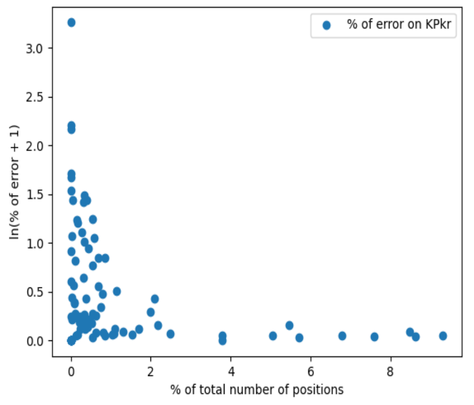


(d) Error rate on drawing positions.

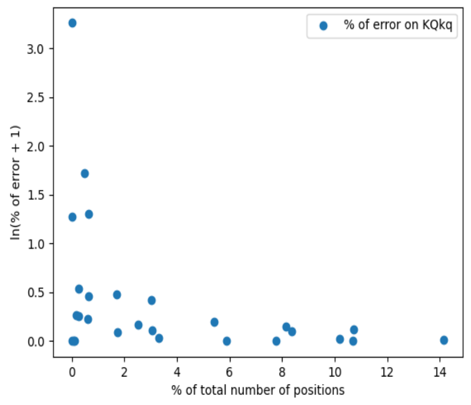
Figure 4.1: Percentage of errors made by the policy at each decision depth for the KQkr tablebase.



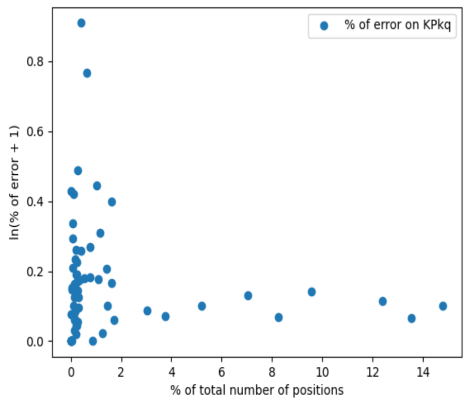
(a) KRkr



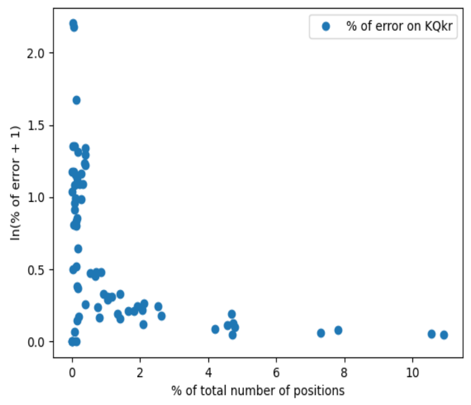
(b) KPkr



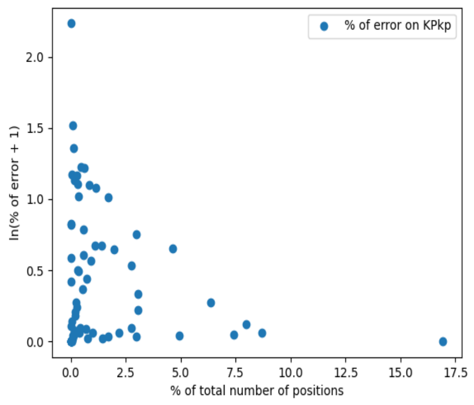
(c) KQkq



(d) KPkq

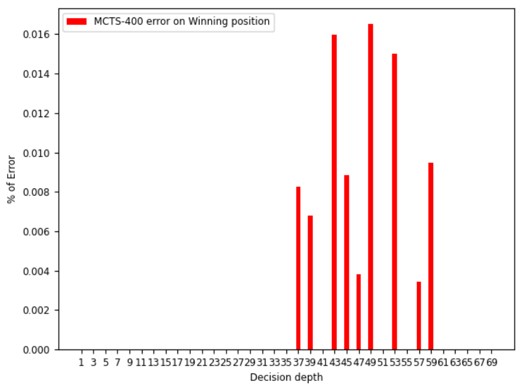


(e) KQkr

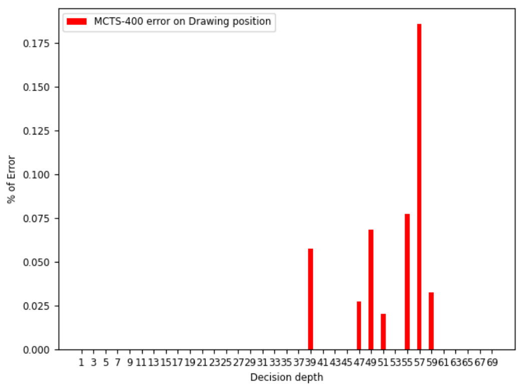


(f) KPkp

Figure 4.2: Percentage of positions at each decision depth vs $\ln(error + 1)$ for four piece tablebases.



(a) Error rate on winning positions.



(b) Error rate on drawing positions.

Figure 4.3: Percentage of errors made by 400 nodes searches at each decision depth for the KQkr tablebase.

4.6 Case Studies of Interesting Mistakes

We stored the positions where Lc0 with 0, 400, 800 and 1600 simulation plays a wrong move in experiment 2. We analyzed positions with different types of errors as follows:

- Policy move wrong
- 400 simulations move wrong
- 800 simulations move wrong
- 1600 simulations move wrong

Comparing these four groups of positions, as expected, in most of the cases where search selects a wrong move the policy is also wrong. Also, in most cases where deeper search is wrong both the policy and the smaller searches are also wrong. Surprisingly there are some cases where the search result is inaccurate while the policy is correct. To find out the reasons for this we investigated some of the errors where policy and search disagree:

- Policy Wrong but Search Correct
- Policy Correct but Search Wrong

4.6.1 Policy Wrong but Search Correct

The most common type of error throughout our experiments is that the policy move is wrong and search corrects this. As mentioned in Section 4.3.1, the search looking ahead combined with the MCTS approximation of minimax makes this possible.

Policy Is Wrong but Smaller Search Is Correct

One such position is given in Figure 4.4. The decision depth of this position is 5. In this position the policy selects a drawing move **Qa1** instead of the winning move **Qg1**. The correct move selected by search is marked by a green cell and the incorrect move selected by the policy is marked by a red cell in the figure. For simplicity we include only these two moves in the Figure. The network's prior probability (policy head) of the incorrect move **Qa1** (0.1065) is higher than for the winning move **Qg1** (0.0974). However, the value head has a better evaluation for the position after the winning move (0.3477) than after the drawing move (0.0067). Therefore **Qg1** becomes the best-evaluated move after only a few simulations.

Figure 4.5 shows details - the changes of Q, U, $Q+U$ and N during MCTS as a function of the number of simulations. At each simulation, the move with the highest UCB value ($Q + U$) is selected for evaluation. The Q value of a node is the average of its descendants' values. The exploration term U depends on the node's visit count N and the node's prior probability. For this example, while the exploration term $U(\textbf{Qa1}) > U(\textbf{Qg1})$ throughout, the UCB value remains in favour of the winning move. An accurate value head can overcome an inaccurate policy in the search.

Policy and All Searches Are Wrong

In Figure 4.6 (decision depth 59), both **Kd3** (green) and **Kd5** (blue) win, but both the raw network and the search choose **Kc3** (red) which draws. **Kd5** has by far the lowest policy (0.2776) and value (0.3855), and its Q and N are consistently low, keeping it in distant third place throughout.

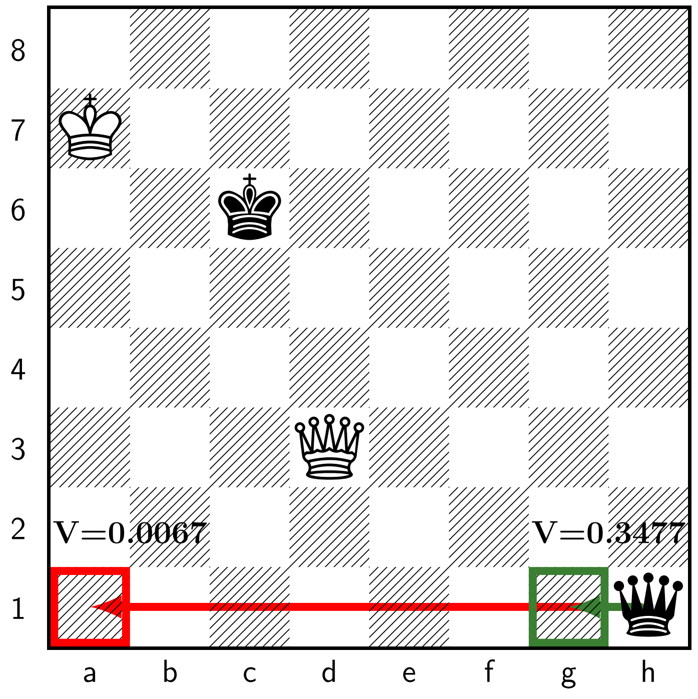
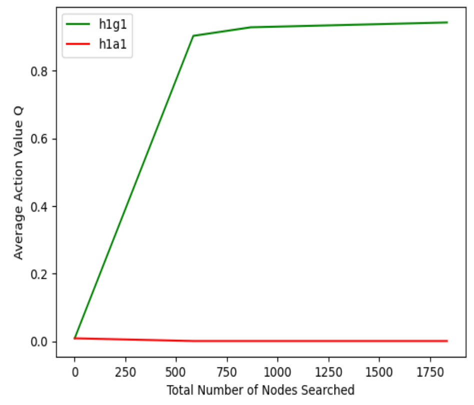
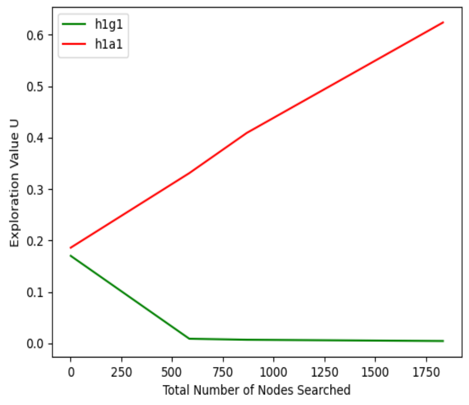


Figure 4.4: A position where policy is wrong but smaller search is correct.

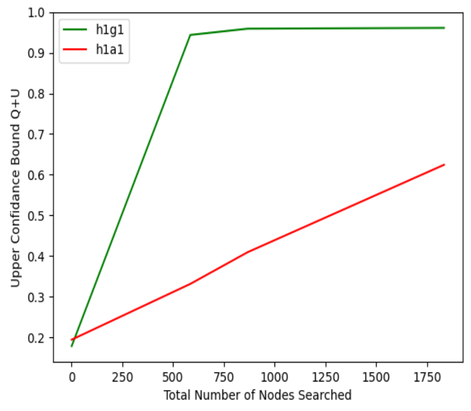
We now observe how the relevant terms in the UCB function affect the engine's choice for the top two moves. Both the initial policy and value are higher for **Kc3** (0.3229 and 0.9144) than for the correct **Kd3** (0.2838 and 0.8501). We extended the search beyond the usual 1600 simulations to see its longer-term behavior. The Q value of **Kc3** remains highest for 6000 simulations, while **Kd3** catches up, as shown in Figure 4.7(a). The UCB values $Q + U$ of all three moves remain close together. MCTS samples all three moves but focuses most on the incorrect **Kc3**. Considering the correct move's Q value is consistently lower before 6000 simulations, its UCB value is propped up by its exploration term U. At 1600 simulations, the inaccurate value estimates play a large role in an inaccurate Q value for **Kd3** and **Kc3**, resulting in MCTS incorrectly choosing **Kc3**. Beyond 6000 simulations, the Q value of **Kd3** keeps improving, and MCTS finally chooses a correct move at about 12000 simulations.



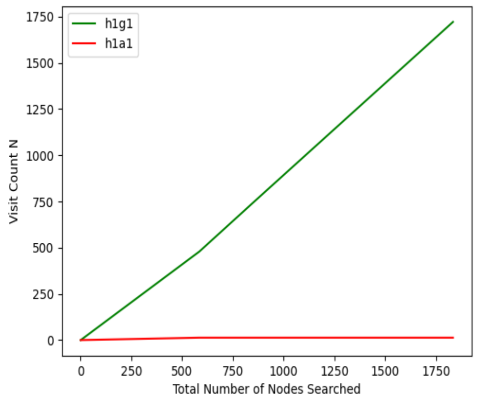
(a) Average action value Q.



(b) Exploration value U.



(c) Upper Confidence Bound Q+U.



(d) Visit count N.

Figure 4.5: The change of Q, U, Q+U and N during the search for the position from Figure 4.4 where the policy is wrong and a smaller search is correct.

4.6.2 Policy Correct, Search Wrong

Surprisingly there are some positions where the policy finds the correct move but search changes to an incorrect move. We show two examples in this section.

Policy Is Correct, Smaller Search Is Wrong

Figure 4.8 shows an example position where only MCTS-400 makes a mistake while both the policy and deeper search choose the correct move. The decision depth of this position is 23. In this case **Kb5** (green) wins while **Kd6** (red) draws. The prior probability of **Kb5** is 0.0728, slightly higher than **Kd6** with 0.0702, but the value for **Kb5** at 0.2707 is slightly lower than for **Kd6** with

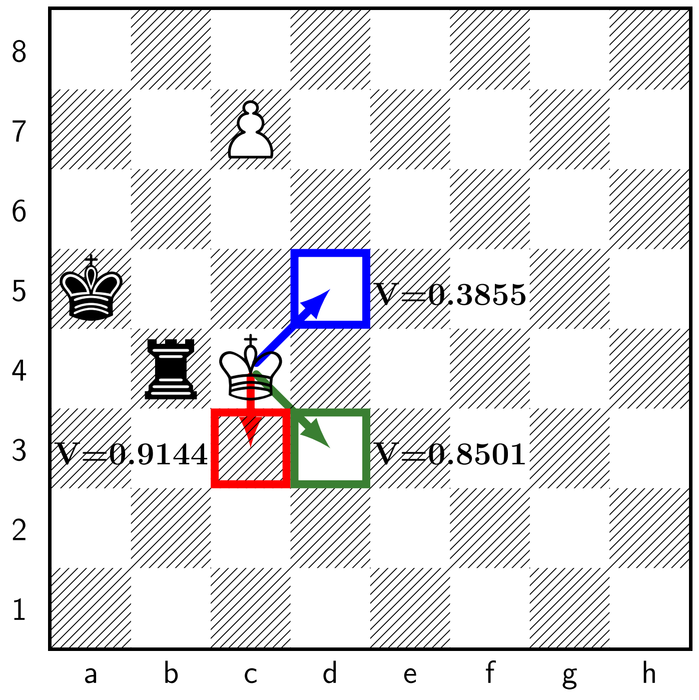


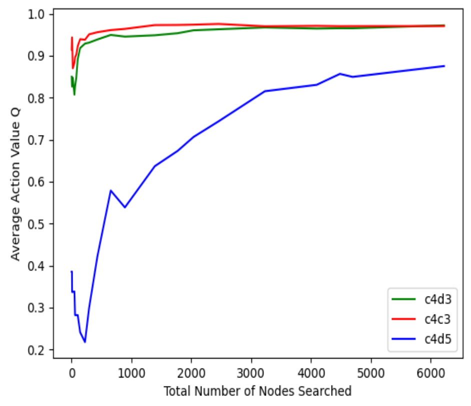
Figure 4.6: A position where policy and all searches are wrong.

0.2834.

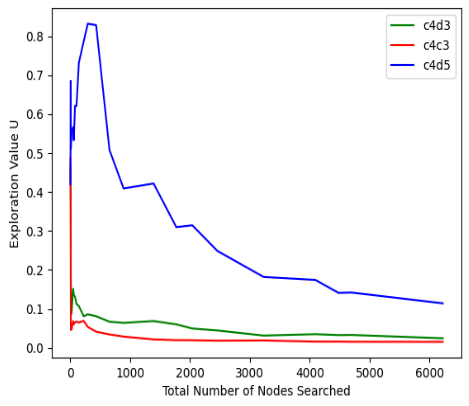
Figure 4.9(a) shows that the Q value of **Kd6** is higher early on due to the value head. Initially, until 400 nodes are searched, the nodes within the drawing subtree are evaluated better than in the winning subtree. As search proceeds, this reverses since the values in the sub-tree of the winning move improve. Consequently, the UCB value of the winning node increases significantly, which in turn increases its visit count. In this example, MCTS overcomes an inaccurate root value since the evaluations of its followup positions are more accurate.

Policy Is Correct But All Searches Are Wrong

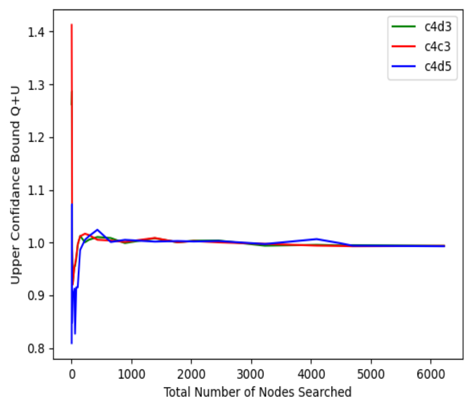
A complete list of such positions is given in Appendix B. In the example shown in Figure 4.10, **d4** (green) wins and decision depth is 33. Up to 1600 simulations, MCTS chooses the drawing move **Kb3** (red). The value of **Kb3** (0.1457) is higher than that of **d4** (0.1053), but the prior probability of **d4** (0.3563) is



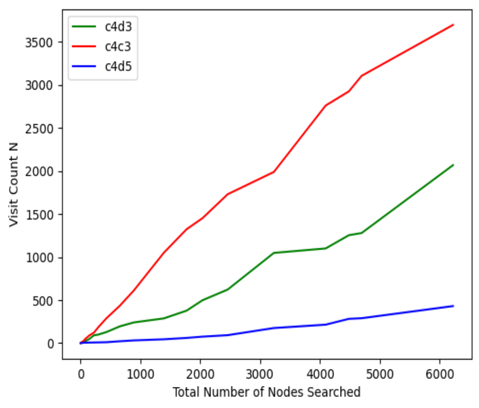
(a) Average action value Q.



(b) Exploration value U.



(c) Upper Confidence Bound Q+U.



(d) Visit count N.

Figure 4.7: The change of Q, U, Q+U and N during the search for the position from Figure 4.6 where policy and all the searches are wrong.

higher than **Kb3** (0.348). Figure 4.11(a-d) shows the search progress. The Q value of **d4** remains lower for longer than in the previous example in Figure 4.8. At around 1500 simulations, the UCB value of the correct move becomes consistently higher. This prompts the search to sample the correct move more. At 2200 simulations, the Q value of the correct **d4** suddenly spikes dramatically.

Figure 4.12 shows a partial line of play from the position. The value of each node is shown on the boards. For the drawing subtree, Lc0 follows the line of perfect play. The value of the nodes on the drawing subtree becomes close to zero searching deeper. For the winning subtree, the values of the winning nodes are low initially, close to a draw score of 0. However, the deeper nodes

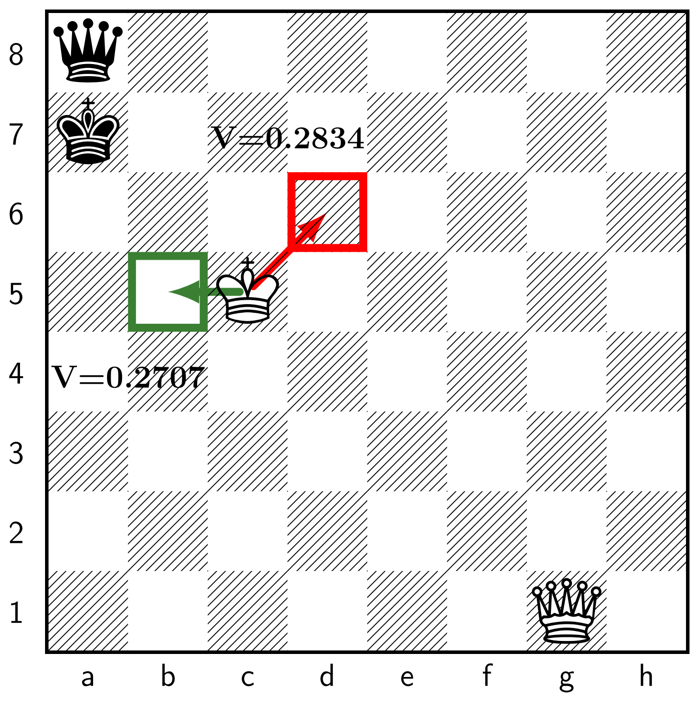
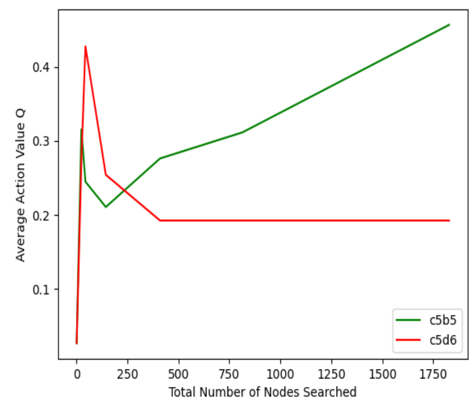


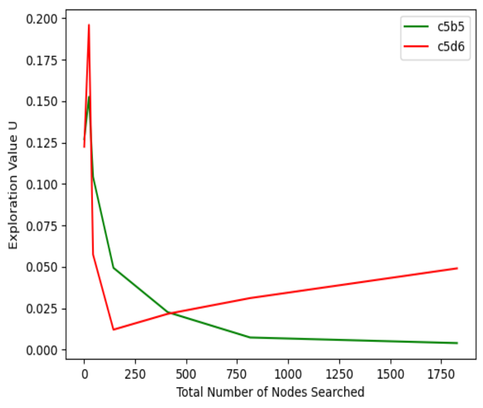
Figure 4.8: A position where the policy is correct but smaller search is wrong.

starting from the marked node in blue are evaluated over 0.99, a confirmed win. At 2200 simulations, the search from the initial position starts seeing these nodes which confirm the win. Finally, from 2700 simulations, the engine selects the winning move.

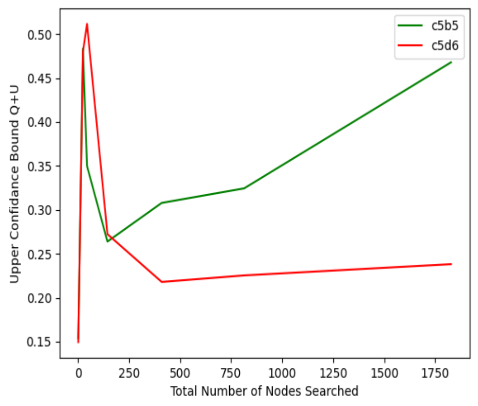
The update rule of Q adds the neural network evaluation of each newly added node in the tree to its ancestors [48]. As a result, the average action value Q depends on how the value head estimates the value of the nodes within the expanded subtree. In every case, the term Q dominates in the UCB value $Q+U$ as N gets larger. As a result, asymptotically the search is guided by the value head rather than the policy head. Therefore, a higher value evaluation error in the value head in the relevant state may lead the search to an incorrect result, as seen in this section.



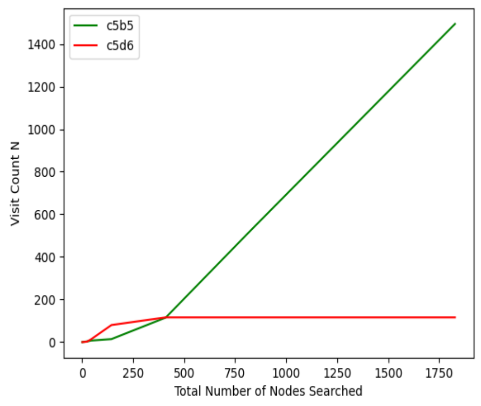
(a) Average Action Value Q.



(b) Exploration Value U.



(c) Upper Confidence Bound Q+U.



(d) Visit Count N.

Figure 4.9: The change of Q, U, Q+U and N during the search for the position from Figure 4.8 where the policy is correct but a smaller search is wrong.

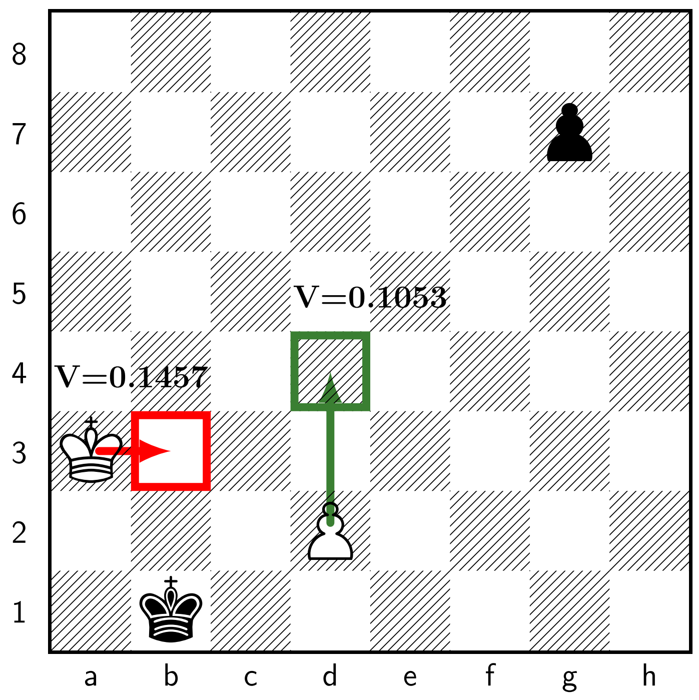
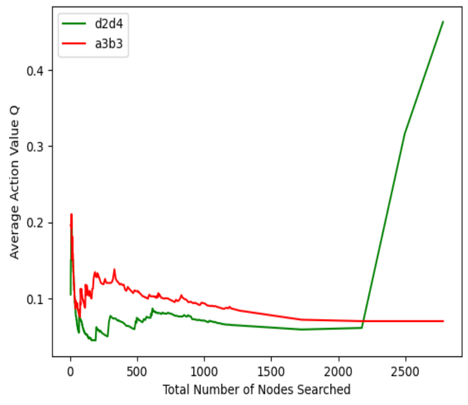
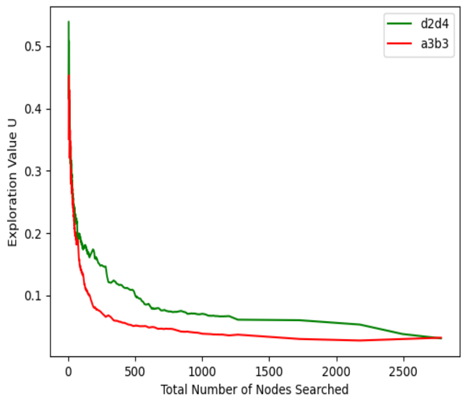


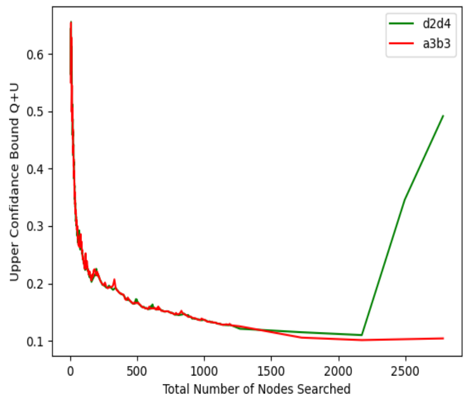
Figure 4.10: A position where policy correct but all the searches are wrong up to budget 1600.



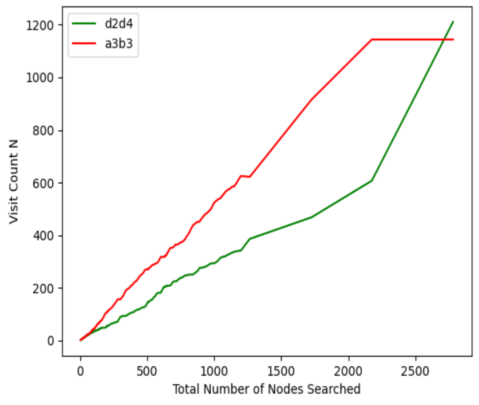
(a) Average action value Q.



(b) Exploration value U.



(c) Upper Confidence Bound Q+U.



(d) Visit count N.

Figure 4.11: The change of Q, U, Q+U and N during the search for the position from Figure4.10 where the policy is correct but all the searches are wrong up to budget 1600.

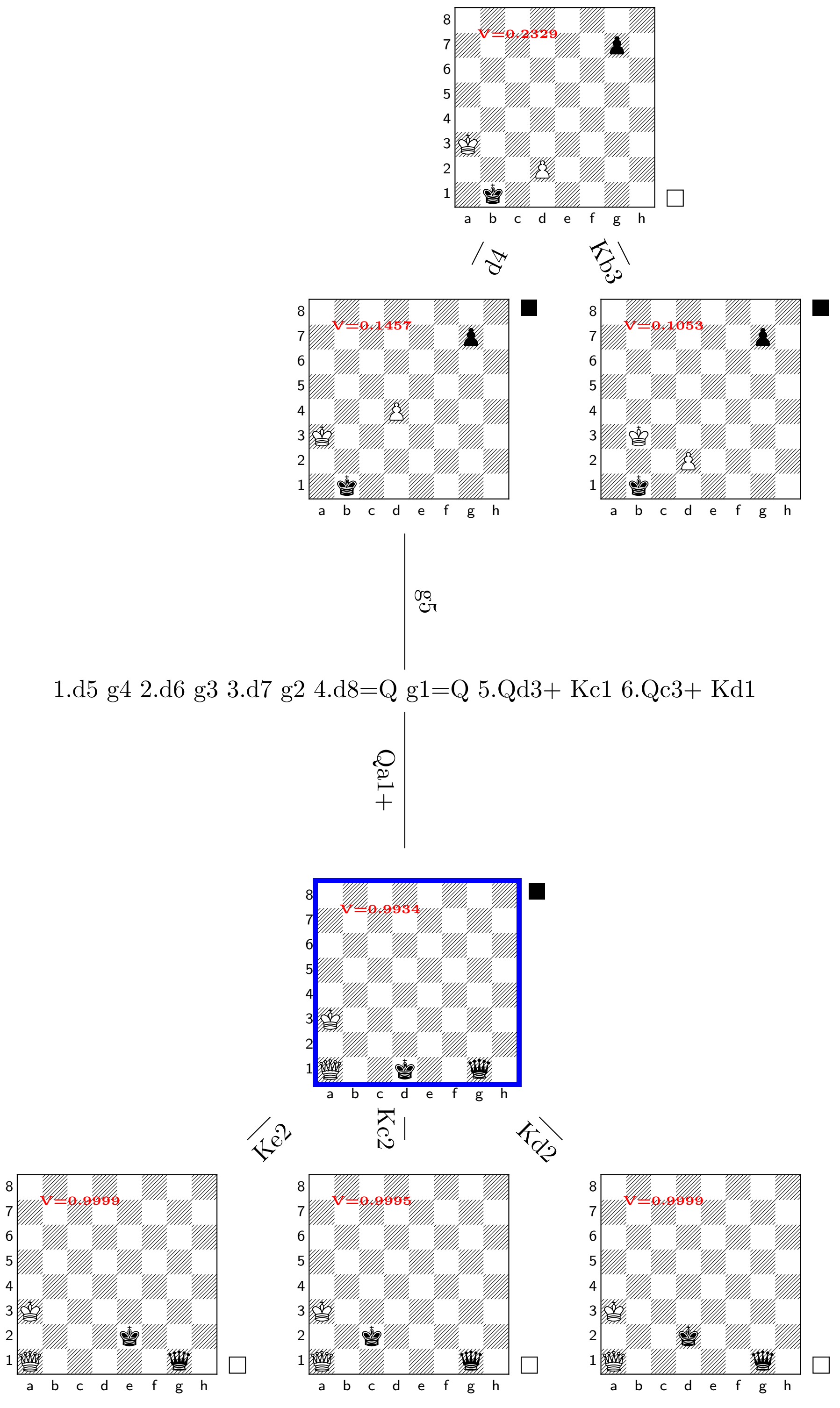


Figure 4.12: Partial search tree for the winning subtree of position from Figure 4.10.

Chapter 5

Experiments on a Five Piece Tablebase

In order to observe how our results scale for a larger tablebase, we conducted experiments on samples from the KQRkq five piece tablebase. Considering our limited resources, we selected only samples of this one pawn-less tablebase. This tablebase has a high range of decision depths, that ensures there are difficult positions for human players. The highest DTM in this tablebase is 131.

To conduct our experiments, we took 2049112 unique positions by randomly sampling 1% from the total of 204911280 unique positions. From these 2049112 unique positions we got 1214165 unique legal positions, which is approximately 1% of the total number of 121812877 unique legal positions in KQRkq. Among the samples taken, there are 683022 winning and 147694 drawing positions with more than one game outcome. In total our test case contains these 830716 positions. The results presented in this chapter are an approximation of the whole five piece tablebase results because of the sampling.

5.1 Performance of the Strong Network on the Five Piece Tablebase

As in Chapter 4 we evaluate the move decisions of Lc0 in KQRkq for both the raw network policy, and for full Lc0 with 400, 800 and 1600 MCTS simulations.

Table 5.1 shows the fraction of mistakes on the sample five piece tablebase. The table shows that the error rate for draws is higher than for wins, unlike the common case of four piece tablebases.

Search Budget	Winning Error	Drawing Error	Overall Error
MCTS-0	1.137	5.186	1.857
MCTS-400	0.040	0.297	0.086
MCTS-800	0.025	0.17	0.052
MCTS-1600	0.011	0.105	0.028

Table 5.1: Error rate in percent on five piece sample tablebase.

5.2 How does the Error Rate Change with Increased Number of Pieces?

To evaluate the change of Lc0 performance with increased number of pieces in Table 5.2 we compare the average error rate on our chosen 3, 4 and 5 piece tablebases. The error rate increases strongly with the increased number of pieces.

Search Budget	Three Piece	Four Piece	Five Piece
MCTS-0	0.0092	0.2306	1.8573
MCTS-400	0	0.0034	0.0857
MCTS-800	0	0.0012	0.0516
MCTS-1600	0	0.0004	0.0278

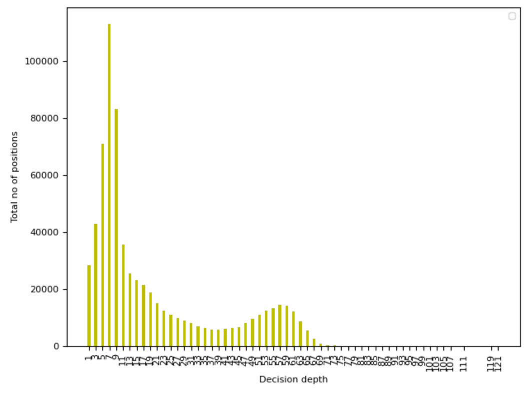
Table 5.2: Average error rate in percent on all tested three, four and five piece tablebases.

5.3 How Well Does the Engine Recognize Wins and Losses at Different Decision Depths?

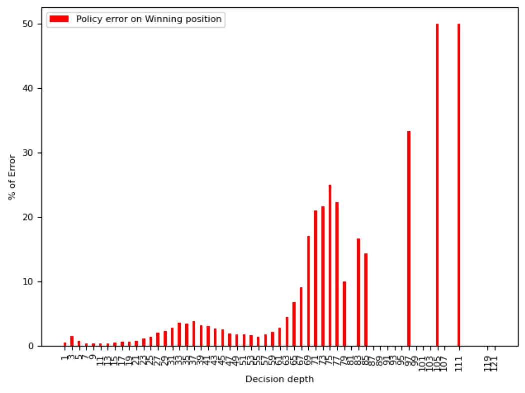
To find out how the error rate changes at each decision depth in the larger tablebase, we extended the Experiment of Section 4.5 to the KQRkq tablebase. As this tablebase has higher decision depths, we evaluated all four: raw NN and MCTS-400 as well as MCTS-800 and MCTS-1600.

5.3.1 Error Rate of Raw NN at Different Decision Depths

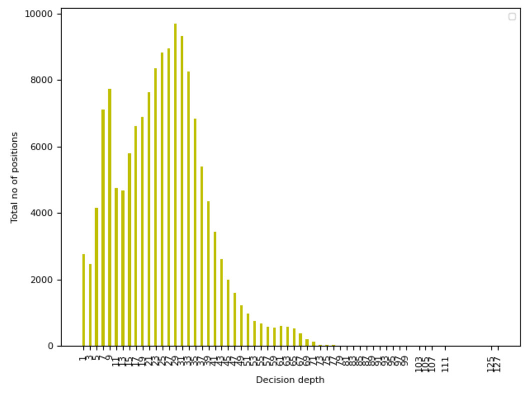
Figure 5.1 shows the total number of positions and the percentage of error for the raw NN at different decision depths. The result is similar to the four piece tablebases. The error rate is still higher for depths where there is only a small number of positions, regardless of the depth of the position. Figure 5.2 shows a similar relationship between the sample size at each decision depth and the error rate of the raw net as for the four piece tablebases in Figure 4.2.



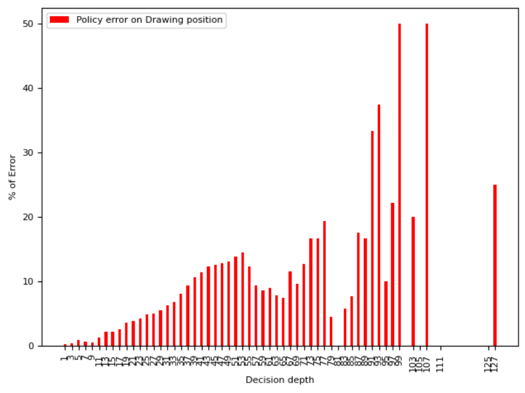
(a) Total number of winning positions.



(b) Error rate on winning positions.



(c) Total number of drawing positions.



(d) Error rate on drawing positions.

Figure 5.1: Percentage of error by the raw policy at each decision depth.

5.3.2 Error Rate of MCTS-400, 800 and 1600 at Different Decision Depths

We evaluate all the search settings used in our experiment (MCTS-400, 800 and 1600) on the five piece tablebase. Similar to the four piece search result in Section 4.5.2 the error rate does not depend on the sample size at each decision

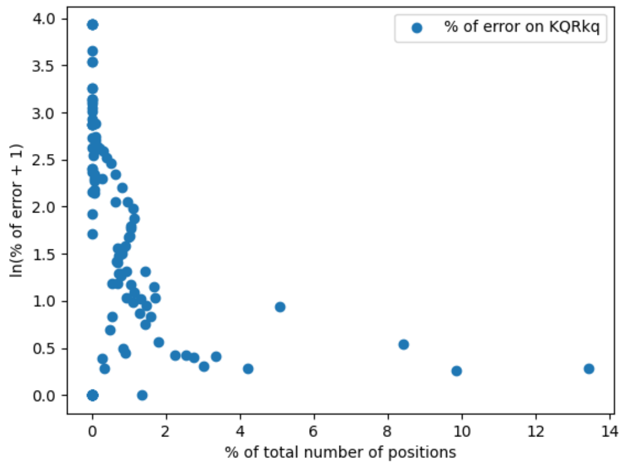


Figure 5.2: Percentage of positions at each decision depth vs $\ln(error + 1)$ for the sample KQRkq tablebases.

depth. Figure 5.3 shows the error rate for all three levels of search at each decision depth. The error rate tends to be higher for higher decision depths. Moreover, large search can correct mistakes done by smaller searches at all decision depths. Again there is no simple relation between decision depth and error rate KQRkq.

5.4 Case Study: How Does Lc0 Play on the Deepest Winning Position?

Figure 5.4 shows the winning position with highest decision depth (131) of all positions in KQRkq tablebase. This position was filtered out from the sample five piece tablebase. We examined Lc0 move selection on this position using both the raw policy and searches. In this position both **Kb3** (green) and **Kb1** (blue) win while **Ka3** (yellow) draws and **Ka1** (red) loses.

The prior probability and value estimation of the child nodes are shown in Figure 5.5. The prior probability of the winning move **Kb3** is the highest. Therefore, the raw NN selects this winning move. The value head estimation of this winning move is also the highest. Therefore, Lc0 selects **Kb3** from the very beginning of search. Figure 5.6 shows the changes of Q, U, $Q + U$ and

N as search proceeds. The average action value Q of **Kb3** remains highest throughout the search. Consequently, this move gets the highest visit count throughout the search. In this case Lc0 clearly differentiates one winning move from drawing and losing moves even though the way to win is long. Furthermore, at 1276 simulations the Q value of the losing move **Ka1** suddenly drops as the search reaches the terminal node. During this search this node is selected only twice as both its policy and value are lowest.

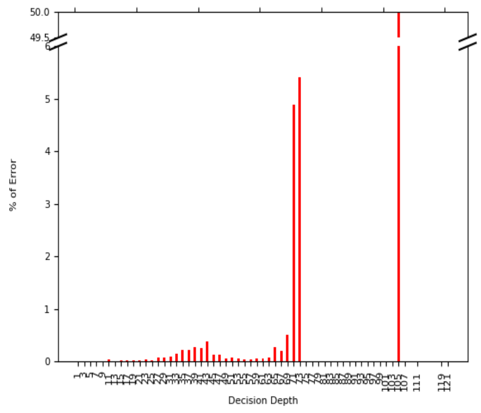
5.5 Position Where the Policy Correct But All the Searches are Wrong

In the position in Figure 5.7 both **Kd4** (Blue) and **Re5** (Green) are winning moves while **Kc4** (Magenta), **Kc6** (Red) and **Kd6** (Yellow) only draw. The decision depth of the position is 43. Here, the policy selects the winning move **Re5**.

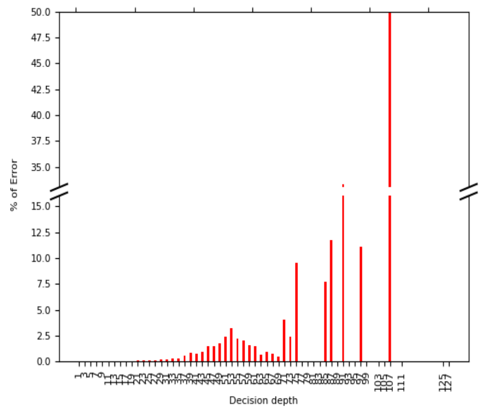
Figure 5.8 shows the prior probability of the moves and the value of the corresponding child nodes. The value of **Re5** is the highest. Therefore, search selects **Re5** initially. However, as shown in Figure 5.9(a) the Q value of the drawing move **Kd6** becomes highest after 36 simulations and remains so for all of MCTS-400, 800 and 1600. Figure 5.9(d) shows that after 1400 simulations another drawing move **Kc6** also achieves a higher visit count and Q value than both winning moves.

Extending the search further, the drawing move **Kc6** becomes the highest visited at 3753 to 80187 simulations. After 80187 simulations, search finally selects the other winning move **Kd4** (Blue).

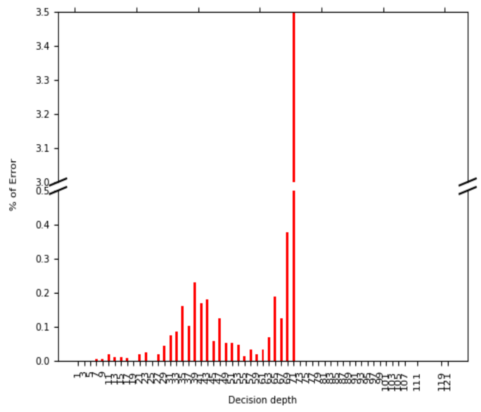
In this particular situation local pathology [38, 53] occurs on the subtree of the game tree, as the search result is erroneous on deeper search with 37 to 80187 simulations, but correct with shallower search up to 36 simulations.



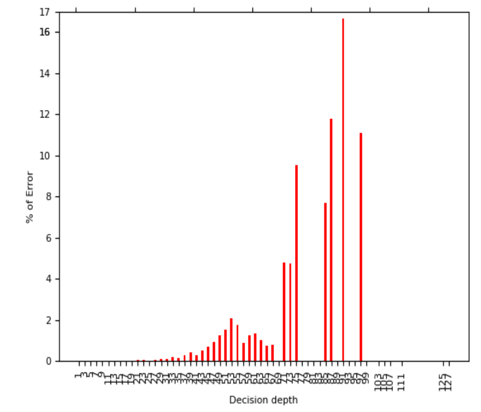
(a) MCTS-400 error in winning positions.



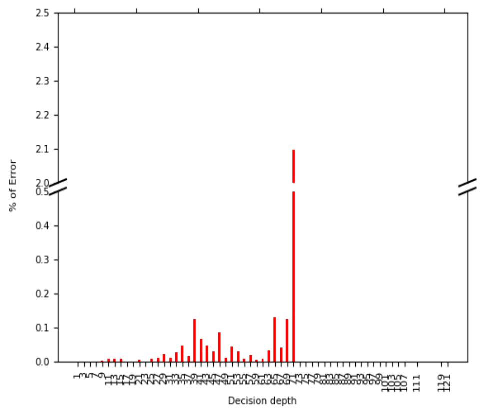
(b) MCTS-400 error in drawing positions.



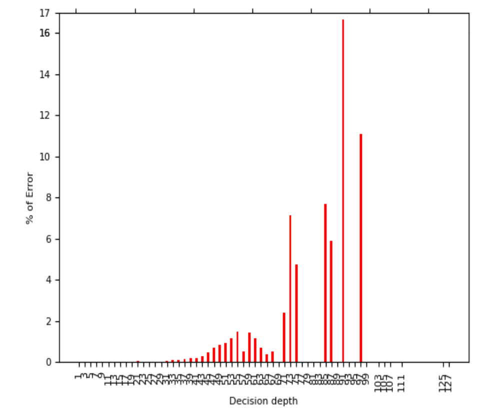
(c) MCTS-800 error in winning positions.



(d) MCTS-800 error in drawing positions.



(e) MCTS-1600 error in winning positions.



(f) MCTS-1600 error in drawing positions.

Figure 5.3: Percentage of error by MCTS-400, 800, 1600 at each decision depth.

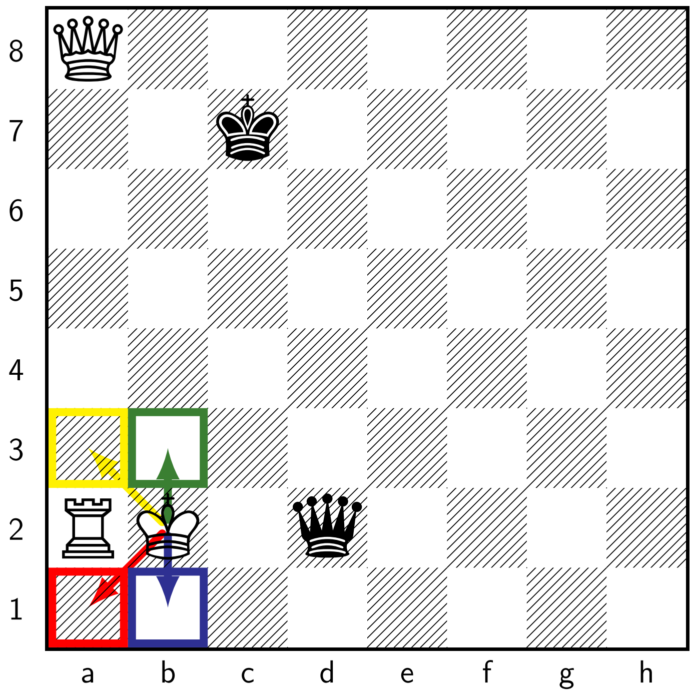
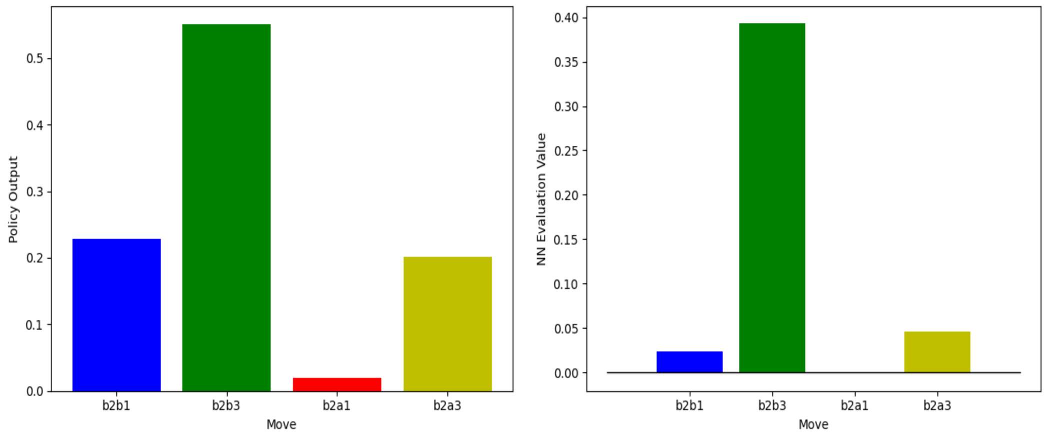
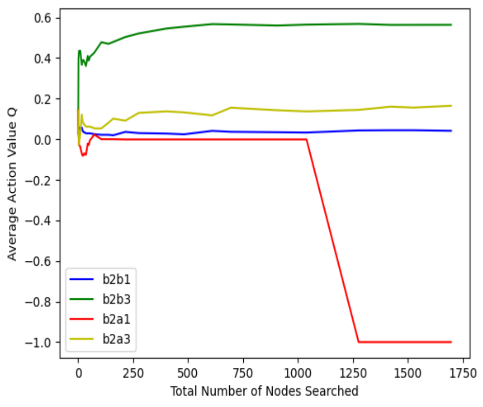


Figure 5.4: Position with deepest win in the whole KQRkq tablebase.

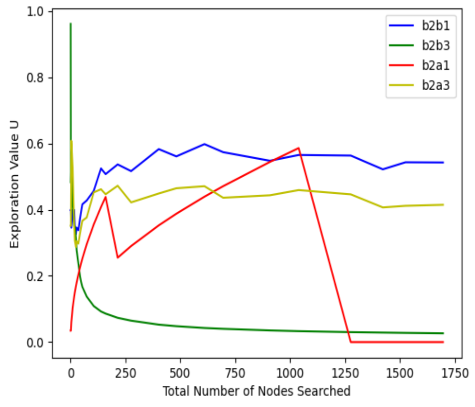


(a) Prior probability of moves. (b) Value evaluation of child nodes.

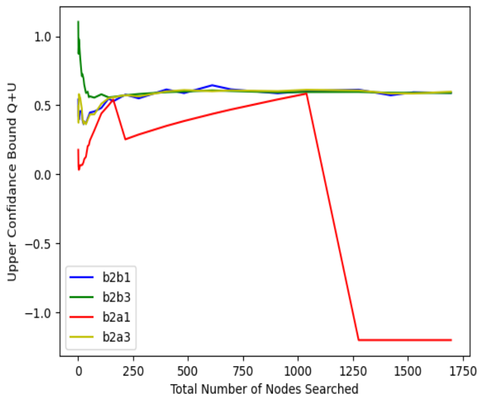
Figure 5.5: Neural network evaluation of the position from Figure 5.4.



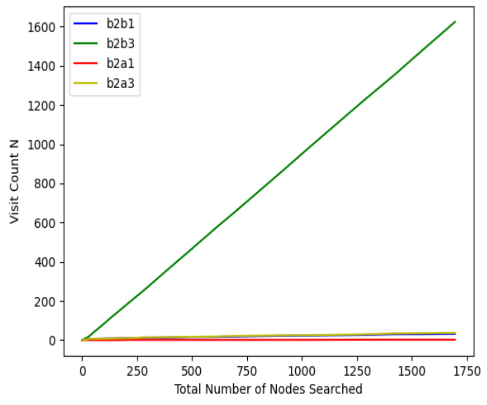
(a) Average action value Q.



(b) Exploration value U.



(c) Upper Confidence Bound Q+U.



(d) Visit count N.

Figure 5.6: The change of Q, U, Q+U and N during the search for the position from Figure 5.4.

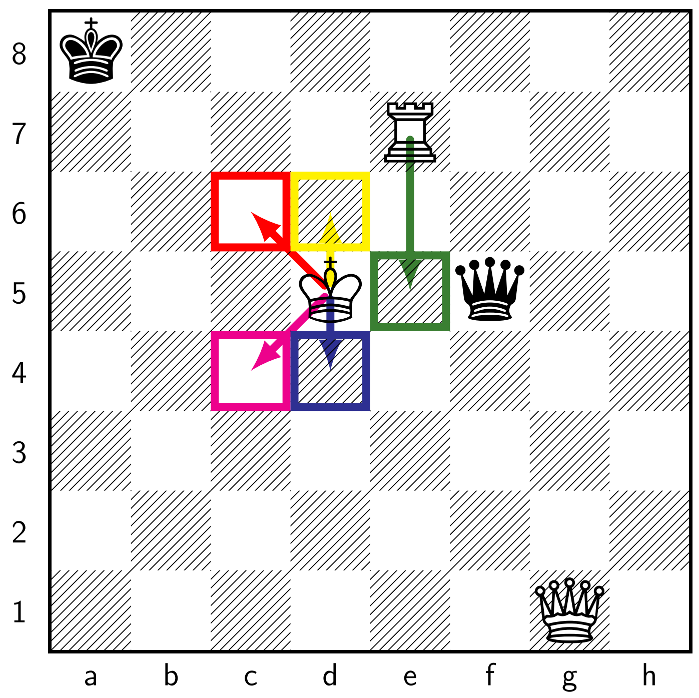
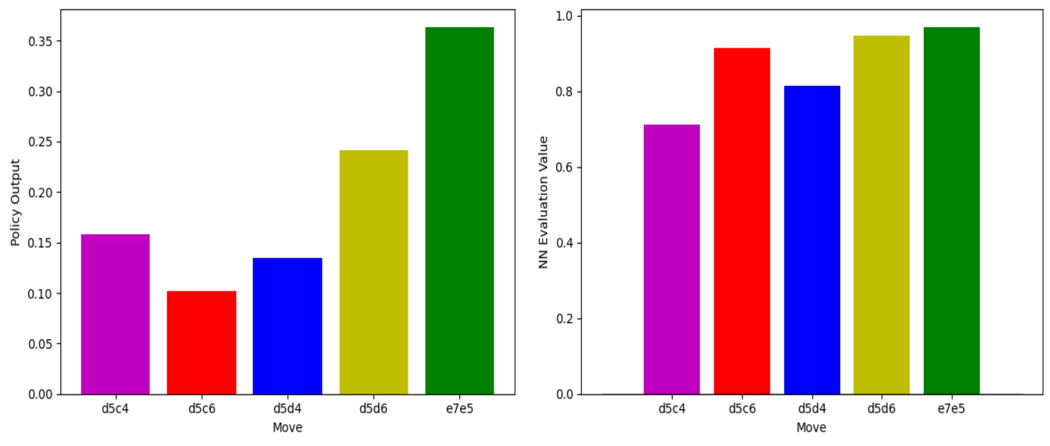


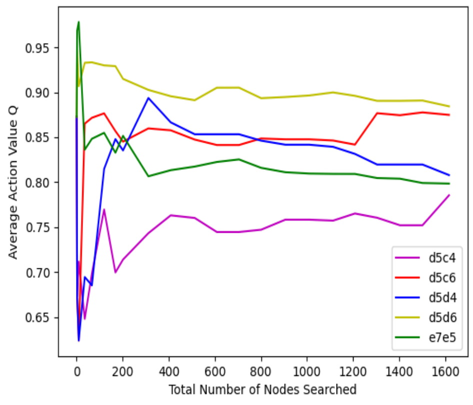
Figure 5.7: Position where the policy is correct but all the searches are wrong up to a budget of 1600.



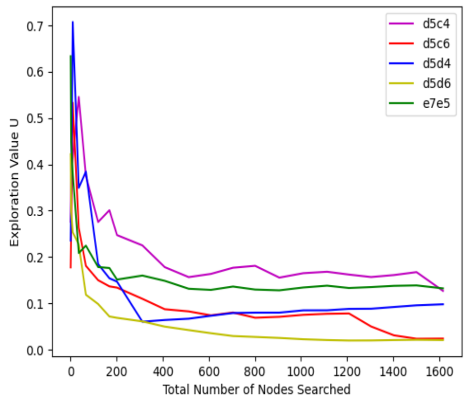
(a) Prior probability of moves.

(b) Value evaluation of child nodes.

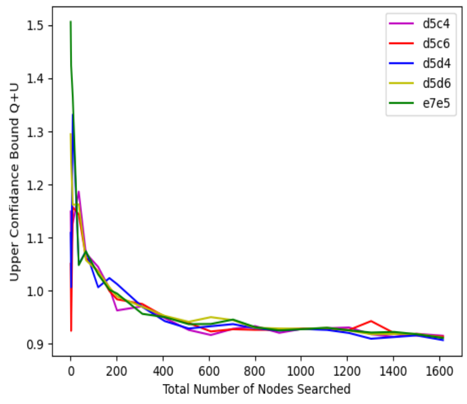
Figure 5.8: Neural network evaluation of the position from Figure 5.7 where the policy is correct but all the searches are wrong up to budget 1600.



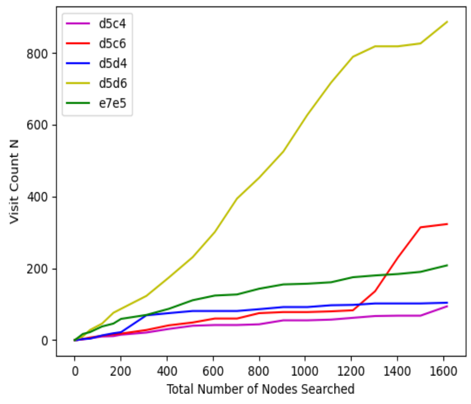
(a) Average action value Q.



(b) Exploration value U.



(c) Upper Confidence Bound Q+U.



(d) Visit count N.

Figure 5.9: The change of Q, U, Q+U and N during the search for the position from Figure 5.7 where the policy is correct but all searches up to a budget of 1600 are wrong.

Chapter 6

Conclusion

In this thesis we investigated the gap between a strong AlphaZero-style player and perfect play using chess endgame tablebases. First, we evaluated perfect play prediction accuracy for the AlphaZero-style Leela Chess Zero program under different settings, including using different NN snapshots, and also between the raw network policy and MCTS with different simulation budgets. We also compared the prediction accuracy for winning and drawing positions. Second, we evaluated the effect of training sample size to prediction accuracy by categorizing positions into different decision depths. We observed the relationship between each decision depth's sample size and the accuracy for that decision depth with both the raw policy and MCTS. We examined specific examples where the engine made different types of mistakes. Finally, we extended our experiment to samples from a larger five piece tablebase and showed results consistent with the four piece tablebases.

The most important findings are:

- NNs approach perfect play as more training is performed.
- Search strongly helps to improve prediction accuracy.
- The number of NN errors decreases for decision depths that have a higher number of samples.
- Search increases the rate of perfect play with shallower decision depths.
- Search corrects policy inaccuracies in cases where the value head accuracy is high.

- Search may negatively impact accuracy in cases where the value head error is high.
- Both raw policy and MCTS performance degrades with increasing number of pieces.

Possible future extensions of this study include:

- Extend the study on samples of other five or more piece endgame tablebases to get the broadest variety of results.
- Perform frequency analyses of self-play training data to measure the number of samples at each decision depth.
- Analyze symmetric endgame positions to verify the decision consistency.
- Examine the value head prediction accuracy more closely and compare it with the policy accuracy.
- Study the case where the program preserves the win, but increases the distance to mate and the case where the program decreases the distance to mate for the losing side.
- Train smaller neural networks to check performance degradation.

References

[1] *7-piece syzygy tablebases are complete*, https://lichess.org/blog/W3WeMyQAACQAdfAL/7-piece-syzygy-tablebases-are-complete, Accessed: 2021-08-26.

[2] B. Abramson, *The expected-outcome model of two-player games*. Morgan Kaufmann, 2014.

[3] L. V. Allis, "Searching for solutions in games and artificial intelligence," Ph.D. dissertation, 1994.

[4] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, no. 2, pp. 235–256, 2002.

[5] M. A. Ballicora, *Gaviota*, https://sites.google.com/site/gaviotachessengine/Home/endgame-tablebases-1, Accessed: 2021-06-06.

[6] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[7] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu, "Deep Blue," *Artificial Intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[8] M. Costalba, T. Romstad, and J. Kiiski, *Stockfish*, 2008-2021. [Online]. Available: https://stockfishchess.org.

[9] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *International Conference on Computers and Games*, Springer, 2006, pp. 72–83.

[10] R. Demontay, *How does Syzygy store its information?* https://chess.stackexchange.com/questions/22207/how-does-syzygy-store-its-information, Accessed: 2021-06-06.

[11] N. Fiekas, *Python-chess*, python-chess.readthedocs.io, Aug. 2014.

[12] H. Forstén, *Moves Left Head*, https://github.com/LeelaChessZero/lc0/pull/961#issuecomment-587112109, Accessed: 2021-08-24.

[13] *On the Road to Perfection? Evaluating LeelaChess Zero Against Endgame Tablebases)*, Lecture Notes in Computer Science, Springer, 2021.

[14] G. Huntington and G. Haworth, "Depth to mate and the 50-move rule," *ICGA Journal*, vol. 38, no. 2, pp. 93–98, 2015.

[15] IBM Cloud Education. (2020). "Monte Carlo simulation," [Online]. Available: `https://www.ibm.com/cloud/learn/monte-carlo-simulation` (visited on 08/20/2021).

[16] R. Kiani and M. N. Shadlen, "Representation of confidence associated with a decision by neurons in the parietal cortex," *Science*, vol. 324, no. 5928, pp. 759–764, 2009.

[17] M. Klein, *Google's AlphaZero destroys Stockfish in 100-game match*, `https://www.chess.com/news/view/google-s-alphazero-destroys-stockfish-in-100-game-match`, Accessed: 2021-06-06.

[18] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *European Conference on Machine Learning*, Springer, 2006, pp. 282–293.

[19] K. Kryukov, *Number of unique legal positions in chess endgames*, `http://kirill-kryukov.com/chess/nulp/`, Accessed: 2021-08-30, 2014.

[20] N. Lassabe, S. Sanchez, H. Luga, and Y. Duthen, "Genetically programmed strategies for chess endgame," in *Proceedings of the 8th annual conference on genetic and evolutionary computation*, 2006, pp. 831–838.

[21] Lc0 authors, *Best nets for Lc0*, `https://lczero.org/dev/wiki/best-nets-for-lc0/`, Accessed: 2021-06-06.

[22] ——, *Engine parameters*, `https://lczero.org/play/flags/`, Accessed: 2021-06-06.

[23] ——, *Lc0*, `https://github.com/LeelaChessZero/lc0/tree/release/0.27`, Accessed: 2021-05-06.

[24] ——, *Lc0 networks*, `https://training.lczero.org/networks/?show_all=0`, Accessed: 2021-06-06.

[25] ——, *Lc0 options*, `https://github.com/LeelaChessZero/lc0/wiki/Lc0-options`, Accessed: 2021-06-06.

[26] ——, *The way forward*, `https://archive.fo/https://blog.lczero.org//2018/06/18/2-the-way-forward/`, Accessed: 2021-06-06.

[27] ——, *What is lc0?* `https://lczero.org/dev/wiki/what-is-lc0/`, Accessed: 2021-08-24.

[28] ——, *What is lc0? (for non programmers)*, `https://github.com/LeelaChessZero/lc0/wiki/What-is-Lc0%3F-(for-non-programmers)`, Accessed: 2021-06-06.

[29] Leela Zero authors, *Leela Zero*, `https://github.com/leela-zero`, Accessed: 2021-06-06.

[30] D. Levy and M. Newborn, *All about chess and computers*. Springer Science & Business Media, 2012.

[31] A. Lyashuk, *Backend configuration*, `https://lczero.org/blog/2019/04/backend-configuration/`, Accessed: 2021-06-06.

[32] ——, *Win-Draw-Loss evaluation*, `https://lczero.org/blog/2020/04/wdl-head/`, Accessed: 2021-08-24.

[33] R. de Man, *Syzygy endgame tablebases*, `https://syzygy-tables.info/`, Accessed: 2021-06-06.

[34] ——, *Syzygy tablebase generator and probing code*, `https://github.com/syzygy1/tb`, Accessed: 2021-06-06.

[35] E. Nalimov, *Endgame Nalimov tablebases online*, `https://chessok.com/?page_id=361`, Accessed: 2021-06-06.

[36] Naphthalin, *Lc0 keeps searching node with lower S.* `https://github.com/LeelaChessZero/lc0/issues/1456`, Accessed: 2021-06-06.

[37] ——, *Same number of nodes search on same position leads to different moves.* `https://github.com/LeelaChessZero/lc0/issues/1467`, Accessed: 2021-06-06.

[38] D. S. Nau, "Pathology on game trees revisited, and an alternative to minimaxing," *Artificial intelligence*, vol. 21, no. 1-2, pp. 221–244, 1983.

[39] M. Newborn, *Computer Chess.* Academic Press, 1975.

[40] R. Nuwer, *A game designer thinks he can improve on chess' 1,500-year-old rules*, `https://www.smithsonianmag.com/smart-news/a-game-designer-thinks-he-can-improve-on-chess-1500-year-old-rules-180948179/`, Accessed: 2021-06-06.

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

[42] C. D. Rosin, "Multi-armed bandits with episode context," *Annals of Mathematics and Artificial Intelligence*, vol. 61, no. 3, pp. 203–230, 2011.

[43] S. Russell and P. Norvig, "Artificial intelligence: a modern approach," 2002.

[44] C. E. Shannon, "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.

[45] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[46] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-Play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[47] ——, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[48] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. Van Den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[49] *Tablebases*, `https://chess.cygnitec.com/tablebases/`, Accessed: 2020-06-10.

[50] E. Thé, "An analysis of move ordering on the efficiency of alpha-beta search," Ph.D. dissertation, McGill University Libraries, 1992.

[51] Thoresen,Martin and Mihailov,Anton and Casaschi, Paolo and Pihlajisto,Ilari and Jonsson,Arto and Gemuh,Matthias and Bernstein,Jeremy, *TCEC*, `https://tcec-chess.com/`, Accessed: 2021-06-06.

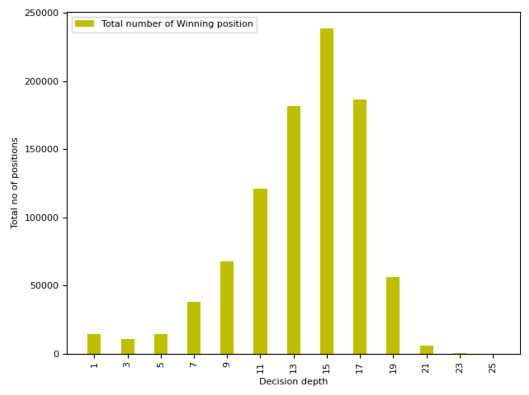
[52] B. Wall, *Computers and chess - a history*, `https://www.chess.com/article/view/computers-and-chess---a-history`, Accessed: 2021-06-06.

[53] B. Wilson, A. Parker, and D. Nau, "Error minimizing minimax: Avoiding search pathology in game trees," in *International Symposium on Combinatorial Search (SoCS-09)*, 2009.

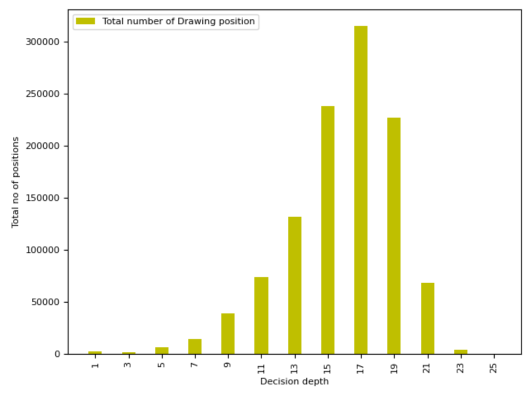
Appendix A

Percentage of Error at Different Decision Depths for Other Four Piece Tablebases

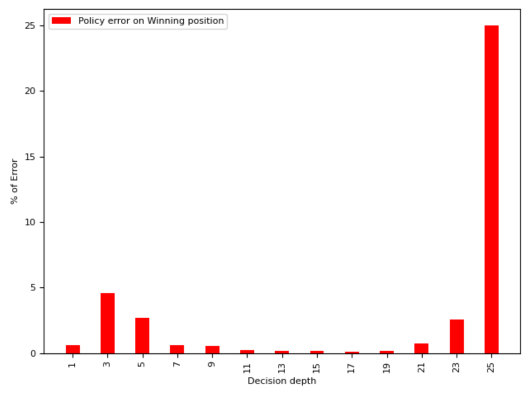
The percentage of error made by the policy and MCTS-400 at each decision depth for the KQkq, KRkr, KPkq, KPkr, and KPkp tablebases are shown in Figures A.1- A.5. Generally, the error rate at each decision depth decreases as the sample size for that depth increases. However, we do not observe any relation between the error rate of MCTS-400 and the sample size. In other words, in most cases, MCTS-400 can correct errors made by the policy at shallower decision depths. In the KPkq and KQkq tablebases, some errors still remain at shallow decision depths. Therefore, there is no simple relation between search error and decision depth.



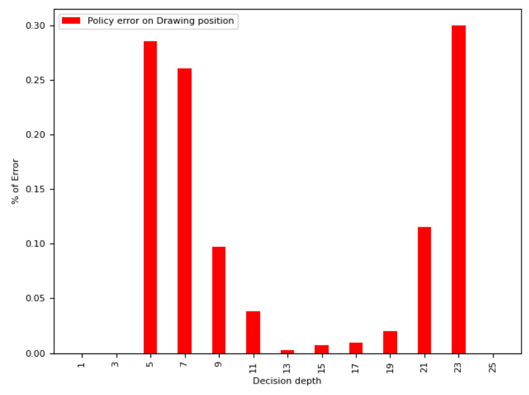
(a) Total number of winning position.



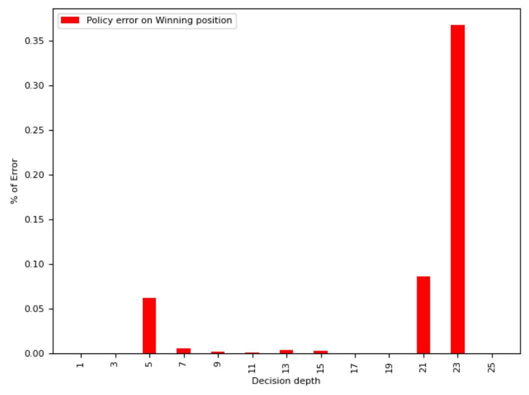
(b) Total number of drawing position.



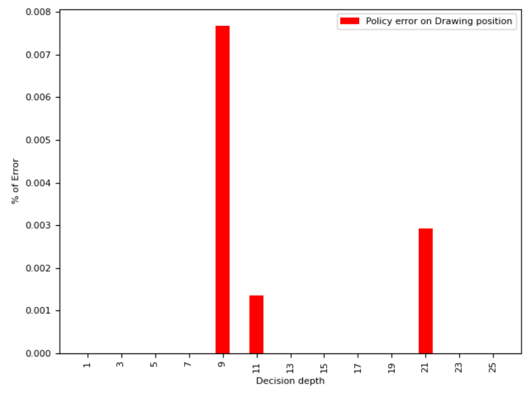
(c) Policy error rate on winning positions.



(d) Policy error rate on drawing positions.

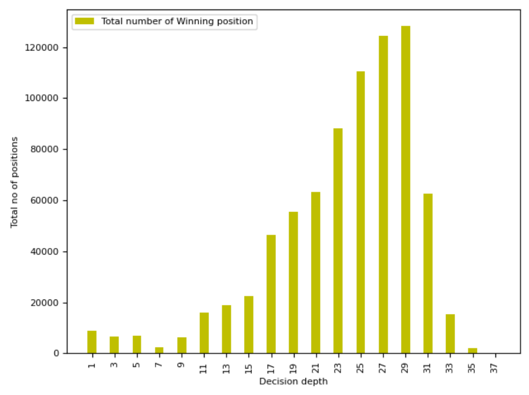


(e) MCTS-400 rate on winning positions.

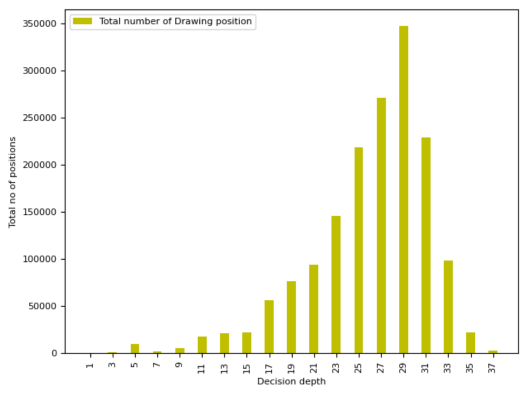


(f) MCTS-400 rate on drawing positions.

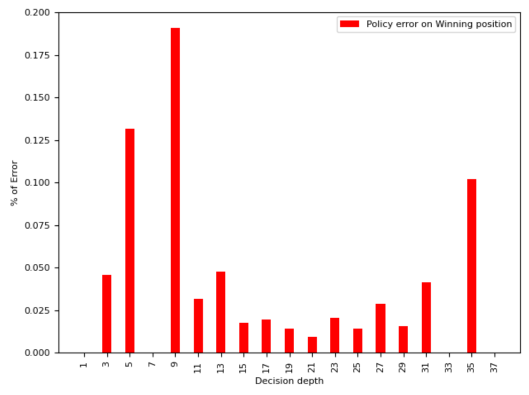
Figure A.1: The percentage of errors made by policy and MCTS-400 at each decision depth for the KQkq tablebase.



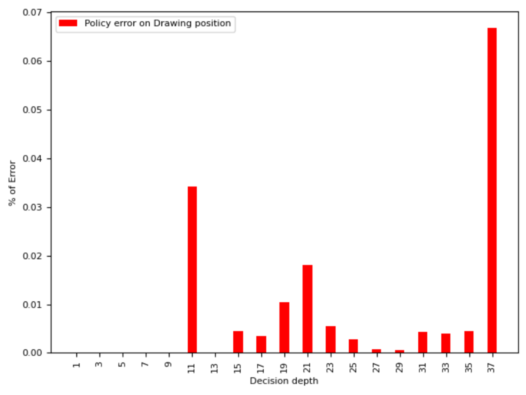
(a) Total number of winning position.



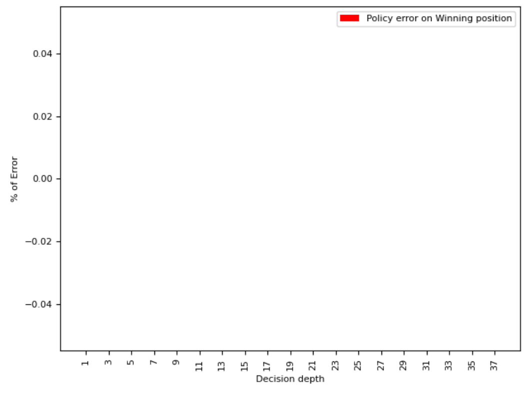
(b) Total number of drawing position.



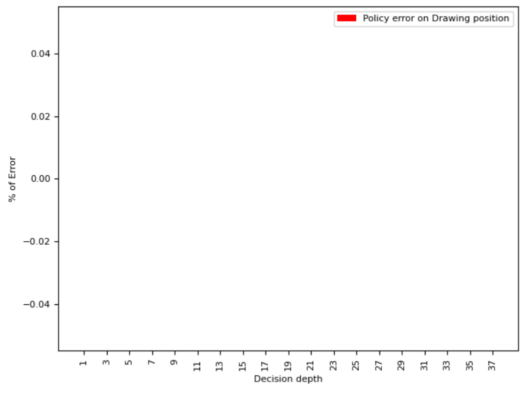
(c) Policy error rate on winning positions.



(d) Policy error rate on drawing positions.

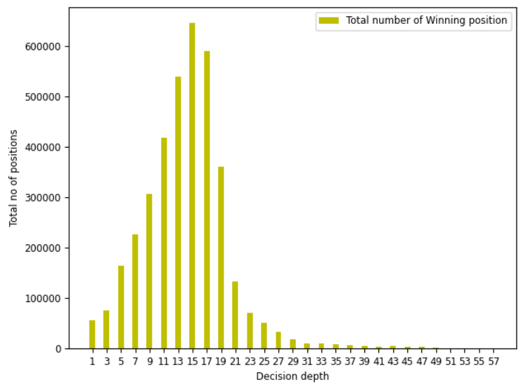


(e) MCTS-400 rate on winning positions.

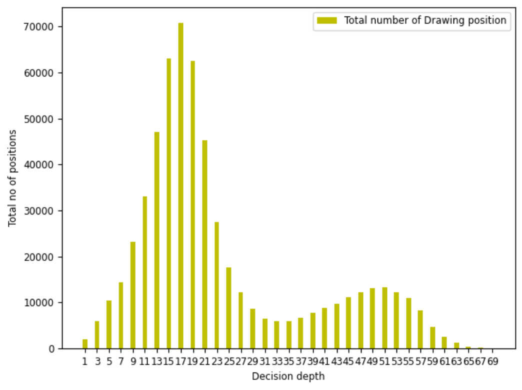


(f) MCTS-400 rate on drawing positions.

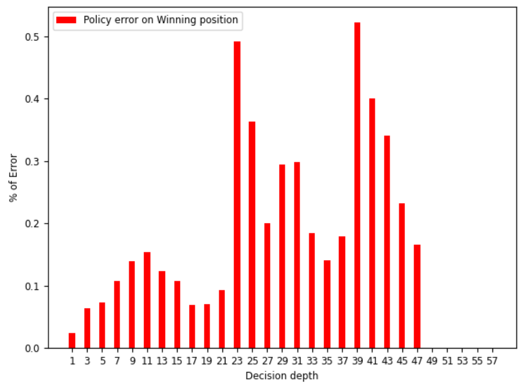
Figure A.2: The percentage of errors made by policy and MCTS-400 at each decision depth for the KRkr tablebase.



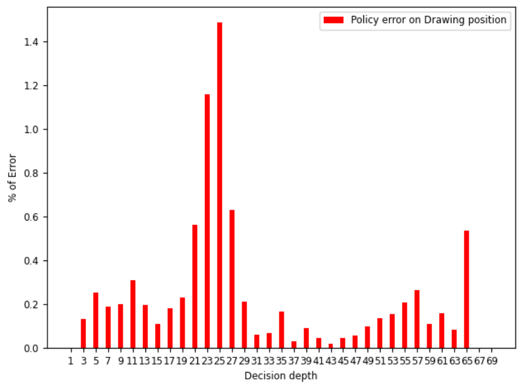
(a) Total number of winning position.



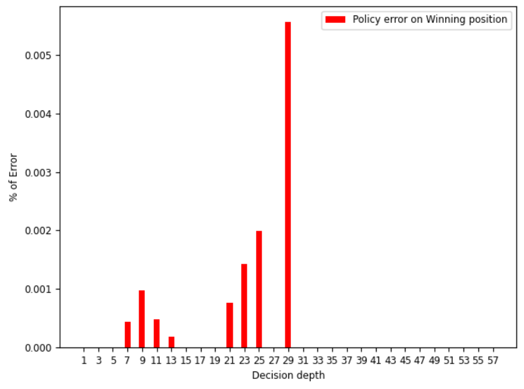
(b) Total number of drawing position.



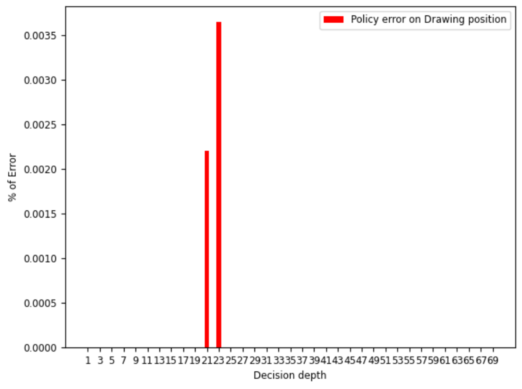
(c) Policy error rate on winning positions.



(d) Policy error rate on drawing positions.

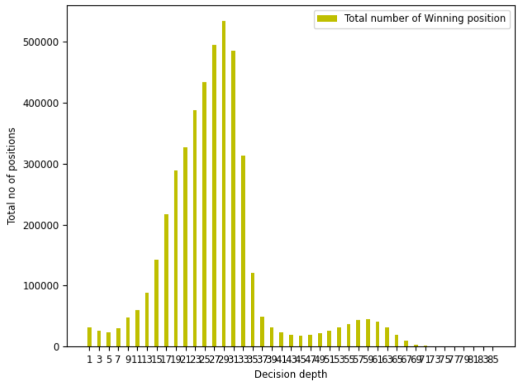


(e) MCTS-400 rate on winning positions.

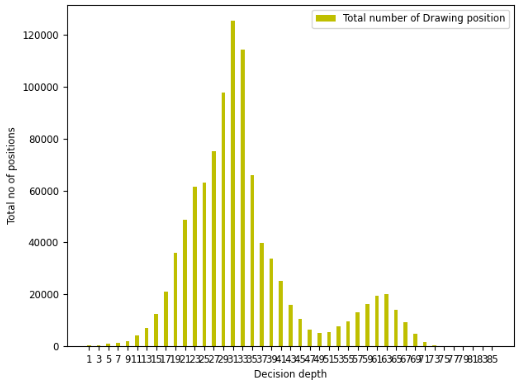


(f) MCTS-400 rate on drawing positions.

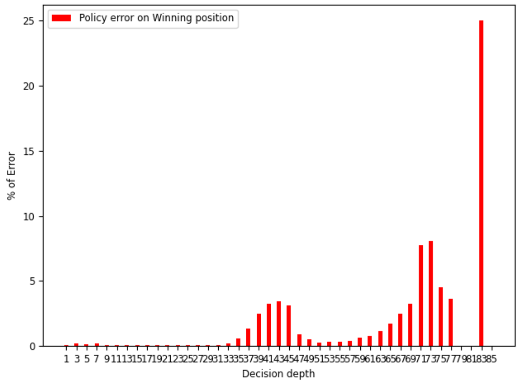
Figure A.3: The percentage of errors made by policy and MCTS-400 at each decision depth for the KPkq tablebase.



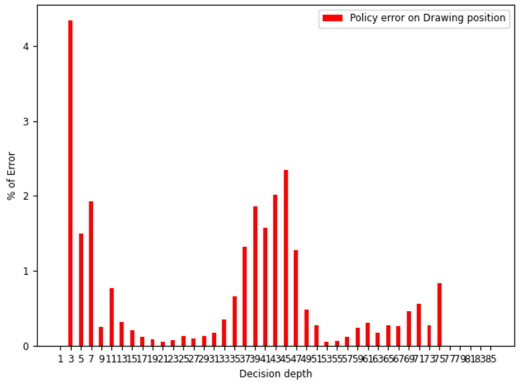
(a) Total number of winning position.



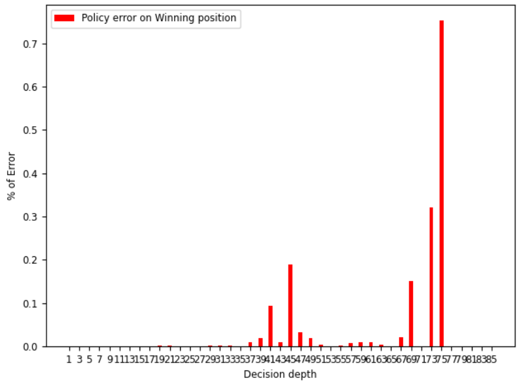
(b) Total number of drawing position.



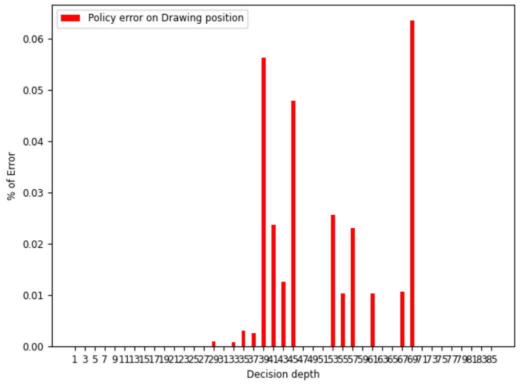
(c) Policy error rate on winning positions.



(d) Policy error rate on drawing positions.

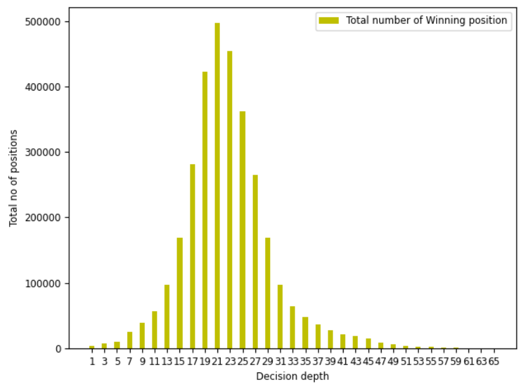


(e) MCTS-400 rate on winning positions.

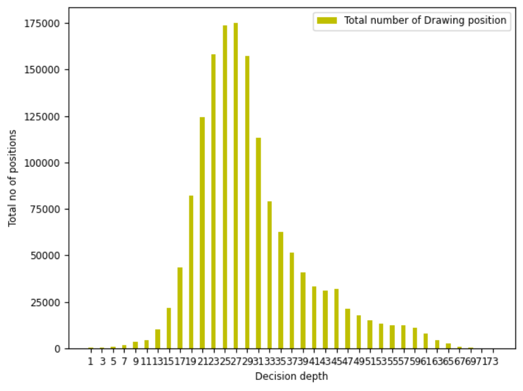


(f) MCTS-400 rate on drawing positions.

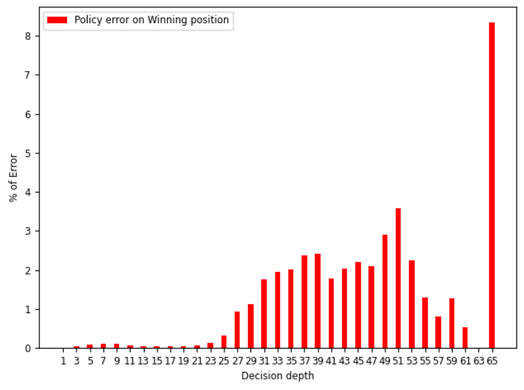
Figure A.4: The percentage of errors made by policy and MCTS-400 at each decision depth for the KPkr tablebase.



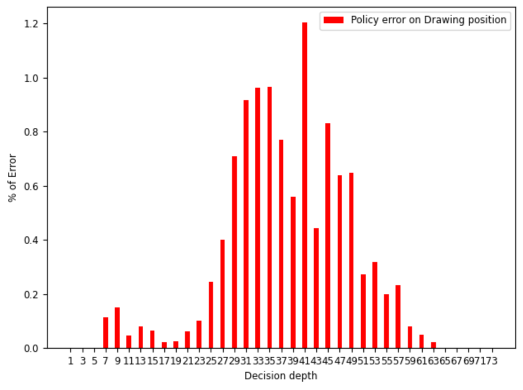
(a) Total number of winning position.



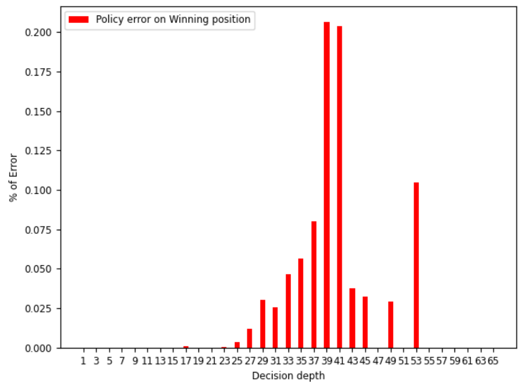
(b) Total number of drawing position.



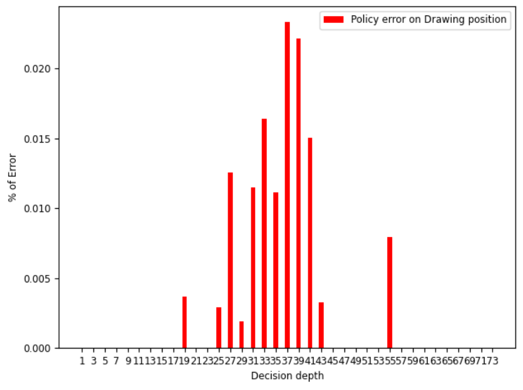
(c) Policy error rate on winning positions.



(d) Policy error rate on drawing positions.



(e) MCTS-400 rate on winning positions.



(f) MCTS-400 rate on drawing positions.

Figure A.5: The percentage of errors made by policy and MCTS-400 at each decision depth for the KPkp tablebase.

Appendix B

Case Study Positions where the Policy Is Correct But All the Searches Are Wrong

Table B.1 includes the list of positions from the chosen four tablebases, where policy is correct but all searches are wrong. Among all the four piece tablebases, these types of positions occur only in the KPkp and KPkr tablebases. The list of positions in the KQRkq tablebase, where policy is correct but all searches are wrong, is given in Table B.2.These types of positions are more common in this five piece tablebase than in four piece tablebases.

Tablebase	Positions
KPkp	8/6p1/8/8/8/K7/3P4/1k6 w - - 0 1
	8/1p6/8/8/1P6/8/8/K5k1 w - - 0 1
	8/1p6/8/8/1P6/8/8/1K3k2 w - - 0 1
	8/1p6/8/8/1P6/8/8/1K2k3 w - - 0 1
	6K1/4p3/7k/8/8/8/1P6/8 b - - 0 1
	6K1/4p3/7k/8/8/1P6/8/8 b - - 0 1
	1k2K3/8/8/1p6/8/8/1P6/8 b - - 0 1
	1k3K2/8/8/1p6/8/8/1P6/8 b - - 0 1
	k5K1/8/8/1p6/8/8/1P6/8 b - - 0 1
	4K3/8/5pk1/8/8/P7/8/8 b - - 0 1
	4K3/2k5/1p6/8/8/8/1P6/8 b - - 0 1
	8/8/6p1/8/8/K7/3P4/1k6 w - - 0 1
	8/1p6/8/8/8/1P6/2K5/4k3 w - - 0 1
	8/1p6/8/7K/8/7k/3P4/8 b - - 0 1
	8/8/3p3K/8/7k/1P6/8/8 w - - 0 1
	8/3p4/7K/8/7k/8/1P6/8 w - - 0 1
	8/3p4/7K/8/7k/1P6/8/8 w - - 0 1
KPkr	8/2P5/8/k7/2K5/8/3r4/8 w - - 0 1
	8/2P5/8/k7/1r1K4/8/8/8 w - - 0 1

Table B.1: Positions in four piece tablebases where the policy is correct but all searches are wrong.

Tablebase	Positions
KQRkq	2q5/8/K7/8/8/1R4Q1/8/k7 w - - 0 1
	k7/4R3/8/3K1q2/8/8/8/6Q1 w - - 0 1
	7k/6R1/K7/8/1q6/8/4Q3/8 w - - 0 1
	6R1/6Q1/8/K4q2/8/8/8/2k5 w - - 0 1
	8/8/8/2K3q1/8/3k4/R4Q2/8 w - - 0 1
	1Q5R/8/8/2K3q1/8/8/8/3k4 w - - 0 1
	7Q/8/8/1K1q4/2R5/4k3/8/8 w - - 0 1
	k7/8/1q6/3K2R1/8/8/8/3Q4 b - - 0 1
	1q6/8/8/2K5/8/Q7/R7/3k4 b - - 0 1
	7R/8/2K5/7q/8/7Q/8/6k1 b - - 0 1
	6R1/8/1K6/1Q6/8/8/3q4/7k b - - 0 1
	8/2R5/K7/3q4/8/8/3k4/6Q1 b - - 0 1
	7q/8/1K6/8/8/8/6QR/4k3 b - - 0 1
	7k/8/1K6/8/7q/8/6R1/5Q2 b - - 0 1
	K7/8/8/7Q/2q5/k7/7R/8 b - - 0 1
	8/8/8/1K6/4q3/8/6QR/1k6 b - - 0 1
	R7/8/8/K7/8/7q/Q7/4k3 b - - 0 1
	8/q7/2K5/8/8/5k2/6R1/6Q1 b - - 0 1
	7R/K7/8/8/8/7Q/8/1q4k1 b - - 0 1
	5k2/8/8/KQ1q4/8/8/8/R7 b - - 0 1
	8/8/8/K7/7Q/8/4qR2/k7 b - - 0 1
	8/8/K7/8/1q6/4k3/5R2/5Q2 b - - 0 1
	8/8/3k4/K6Q/8/5q2/8/7R b - - 0 1
	8/8/K7/8/6q1/8/4QR2/7k b - - 0 1
	7q/1R6/8/2K5/8/2k5/4Q3/8 b - - 0 1
	2q5/8/1K6/1R6/8/4k3/8/Q7 b - - 0 1
	8/7Q/8/1R1K4/5q2/8/8/k7 b - - 0 1

Table B.2: Positions in the KQRkq tablebase where the policy is correct but all searches are wrong.