Master Thesis

Combining Deep Reinforcement Learning and Monte Carlo Tree Search

Jelle W.M. Jansen

Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Department of Advanced Computing Sciences
of Maastricht University

Thesis Committee:

Prof. Dr. M.H.M. Winands
Dr. J. Kowalski
Dr. K. Driessens

Maastricht University
Faculty of Science and Engineering
Department of Advanced Computing Sciences

February 27, 2023

Preface

This master's thesis was written at the Department of Advanced Computing Sciences, Maastricht University. This thesis describes my research into combining Deep Reinforcement Learning and Monte Carlo Tree Search. The code for this thesis is available on Github, https://github.com/GitHubByJelle/MCTS-NIM. First of all, I would like to thank Prof. Dr. Mark Winands and Dr. Jakub Kowalski for their excellent supervision during this thesis. I want to thank Chiara Sironi, M.Sc., for allowing me to use the server and helping me to set it up. I would also like to thank Dennis Soemers, M.Sc., for assisting me with Ludii and helping me brainstorm. Additionally, I would like to thank Paweł Maka, M.Sc., and Ivo Zenden, M.Sc., for frequently helping me out, as well as Dr. Bram van den Broek, for helping me to get into this master. A special thanks to my parents, René and Ilona Jansen, and my girlfriend, Juliette Kreijns, for their trust and support.

Jelle W.M. Jansen
Maastricht, February 2023

Abstract

The thesis investigates new approaches to improve Monte Carlo Tree Search (MCTS) using Deep Reinforcement Learning. Cohen-Solal introduced the "descent framework", a Deep Reinforcement Learning approach that learns to play by self-play and uses a Convolutional Neural Network (CNN) to convert game positions into a value estimate. Because of its newly proposed Tree Learning, the network tries to predict the value found after searching, instead of the game theoretical value, while requiring less computational resources but still outperforming state-of-the-art algorithms with his newly proposed completed Unbounded Best First Minimax (UBFM) search algorithm. Although state-of-the-art algorithms, such as AlphaGo and Polygames, use MCTS, Cohen-Solal noted that the descent framework did not work well with MCTS.

MCTS is a simulation-based algorithm used in decision-making processes to determine the optimal path in a tree-like search space as in a deterministic two-player zero-sum game with perfect and complete information. The algorithm has four phases: selection, play-outs, expansion, and backpropagation. Since MCTS relies on many simulations, introducing Deep Reinforcement Learning is challenging because of the high computational cost associated with the latter. Similar to state-of-the-art algorithms, such as AlphaGo and Polygames, modifications have been made to various phases of MCTS. The play-out phase was eliminated while backpropagating the estimated values of the expanded node instead. However, a performance boost was achieved by enhancing the UCT (Upper Confidence bounds applied to Trees) function used during selection, enabling the MCTS algorithm to use Network-based Implicit Minimax. Out of the three proposed implicit UCT functions, the UCT function that uses implicit minimax values with a decreasing α value performed best. This implicit UCT function emphasizes implicit minimax values in the initial stages and progressively focuses on backpropagated win rates as the number of visits in a node increases. Finally, a proof-of-concept has been presented that does involve play-outs guided by a heuristic evaluation function and dynamic early termination.

Experiments in the games of Breakthrough and Lines of Action have shown that the proposed algorithm outperformed both the $\alpha\beta$-search and an MCTS algorithm with even higher win rates than the state-of-the-art completed UBFM algorithm. When playing games against the completed UBFM algorithm, the proposed MCTS algorithm using Network-based Implicit Minimax outperformed the completed UBFM algorithm with win percentages of 62.0% and 64.1% in the games of Breakthrough and Lines of Action, respectively.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The thesis investigates new approaches to improving Monte Carlo Tree Search using Deep Reinforcement Learning. This chapter is organized as follows. Section 1.1 discusses Artificial Intelligence in games in general. Next, Section 1.2 explains different search algorithms used to play games before explaining in more detail in Chapter 3. Furthermore, Section 1.3 explains how reinforcement learning is used to enrich these search algorithms before this is discussed in more detail in Chapter 5. After that, the problem statement and research questions attempted to be answered by this thesis are discussed in Section 1.4. Lastly, Section 1.5 provides a general overview of how this thesis is further structured.

1.1 Game AI

Artificial Intelligence (AI) that is able to play games has become increasingly popular in recent years. With many successes in various games, it has become one of the most popular areas of AI research. One of the main challenges in game playing is decision-making: what is the best action in the current state of the game? While not trivial, several search algorithms exceeded the human level of play in multiple games. The $\alpha\beta$ search algorithm [30], including multiple enhancements, has led to Deep Blue [12] defeating the human world chess champion, Garry Kasparov, in 1997. However, in some games, such as Go, the $\alpha\beta$ search algorithm was unsuccessful in reaching an expert level of play. DeepMind changed this by using Monte Carlo Tree Search (MCTS) [21, 31], combined with Deep Reinforcement Learning (supervised and unsupervised learning), in AlphaGo [50]. AlphaGo was able to beat the 9-dan professional, Lee Seedol, without handicap on a full-sized 19×19 board. DeepMind even improved its search engine twice, called AlphaGo Zero [52], and AlphaZero [51].

1.2 Search Algorithms

As mentioned in Section 1.1, search algorithms explore the search space of the current game state to find the best possible action to play. Two famous search algorithms are $\alpha\beta$ and MCTS. $\alpha\beta$ is an enhancement to the minimax search algorithm, eliminating the need to search large parts of the game tree using pruning. The default $\alpha\beta$ uses heuristics to explore the search space of the current game state with uniform search depth. Alternatively, Unbounded Best-First Minimax (UBFM) [32] explores the search space of the current game state with a non-uniform search depth by iteratively expanding the best sequence of actions in the game tree.

However, once the branching factor (number of possible moves) gets too high, or it is too hard to create heuristic functions (as seen in Go), $\alpha\beta$ reaches its limit. MCTS tackles these problems by only focusing on promising paths of the game tree while using simulated play-outs to evaluate a game state. This allows MCTS to work without the need of human knowledge for the creation of a heuristic evaluation function.

By combining both MCTS and minimax, the strengths of both approaches can be used, which leads to stronger play, as seen in several successful attempts. Some approaches use minimax during the MCTS play-out [39, 60] or even replace the play-out (partially) [36, 52, 62]. In contrast, others use minimax to back up the heuristic evaluations implicitly and use these during the selection step of MCTS [33].

1.3 Deep Reinforcement Learning

To improve the search algorithms' performance concerning win rate, it is possible to replace heuristic evaluations with Deep Reinforcement Learning (DRL) to better evaluate game states. Previously, DeepMind combined DRL with MCTS, as mentioned in Section 1.1, to enhance the algorithm's performance. However, in [18], Cohen-Solal proposed an alternative DRL approach called the "descent framework", which could be used with multiple search algorithms. Using this framework, a UBFM based search algorithm was trained that outperformed an AlphaZero-based search algorithm [20].

1.4 Problem Statement and Research Questions

As mentioned several times, DRL combined with search algorithms has exceeded superhuman level of play in multiple instances. However, since the the descent framework was able to outperform the state-of-the-art combination of MCTS and DRL (AlphaZero) [20], the question arose if it was possible to adapt MCTS to use the DRL approaches of Cohen-Solal. This resulted in the following problem statement:

How to improve Monte Carlo Tree Search by using Deep Reinforcement Learning

Even though the state-of-the-art algorithms are based on MCTS [14, 50, 52, 51], Cohen-Solal mentioned that the descent framework resulted in poor performances in combination with MCTS [18, 20]. As discussed in Section 1.2, a promising enhancement to MCTS called MCTS with implicit minimax backups could potentially resolve this issue. As opposed to UBFM, which only uses the evaluation of game states, this enhancement will also use average win rates to make more informed decisions. However, changes need to be made to architecture to make this work, resulting in the first research question:

1. *Can Deep Reinforcement Learning and Monte Carlo Tree Search using implicit Minimax backups be combined?*

Search algorithms usually only perform well on a handful of games. For instance, minimax mainly performs well in games where the path to a win is narrow, while MCTS mainly performs well when it is hard to create heuristic evaluation functions. For example, the UBFM algorithm performs well in Hex [20], while the MCTS algorithm with implicit minimax backups performs well in Kalah, Breakthrough, and Lines of Action [33]. This results in the second research question:

2. *On which perfect-information games does the model perform well?*

The descent framework focuses on using the evaluations of game states as the primary source of information to train the search algorithm. Furthermore, the integration of the descent framework with the enhanced MCTS algorithm has the potential to enhance the performance of the descent framework itself since the integration allows for more informed decisions to be made, resulting in a better-performing DRL process. This results in the third research question:

3. *Can the enhanced MCTS algorithm improve the descent framework?*

Determining the effectiveness of the enhanced MCTS algorithm and its potential to surpass existing techniques requires comparing it to state-of-the-art algorithms. The UBFM algorithm has shown to be a strong performer against state-of-the-art algorithms [18], making it a good benchmark for evaluating the performance of the enhanced MCTS algorithm. This leads to the final research question:

4. *How does the enhanced MCTS algorithm perform against state-of-the-art algorithms?*

1.5 Thesis Outline

The thesis first introduces the reader to the background knowledge required to understand the main part fully. The game environment is discussed in Chapter

2. This includes the games and game rules of the games used during the research, but also the general game system used. In Chapter 3, the search algorithms are explained in more detail, followed by a more detailed explanation of Deep Learning in Chapter 4. Then, Chapter 5 reviews various approaches to combine search algorithms with Deep Reinforcement Learning. In Chapter 6, the main part of this thesis is discussed, combining Deep Reinforcement Learning and MCTS with implicit minimax backups. This is followed by the configuration and setup of the experiments in Chapter 7. The thesis concludes with a discussion of the problem statement and the four research questions and wraps up with possible future research in Chapter 8.

Chapter 2

Game Environment

The game used during the research and experiments is a deterministic two-player zero-sum game with perfect and complete information. First, the general game system used is discussed in Section 2.1. Besides that, this chapter also provides an overview of the games used during the experiments. Section 2.2 discusses the game Breakthrough, and Section 2.3 discusses Lines of Action.

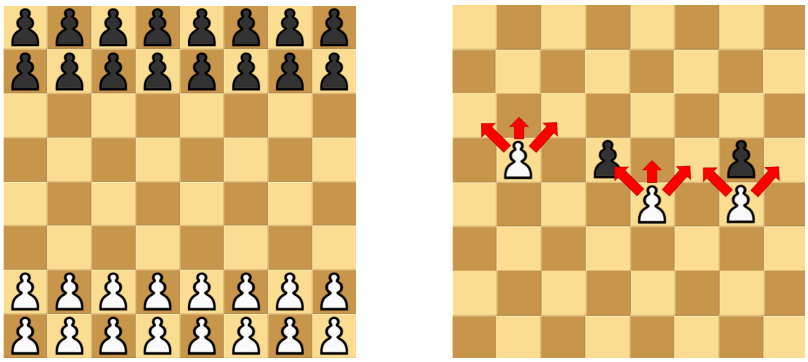
2.1 Ludii

Rather than implementing the game, a general game system called Ludii is used to play games. Ludii was selected because it is equal to or better than other advanced General Game Systems [42]. Ludii features an interface for implementing external agents, which can then be imported into Ludii's GUI and used to play a Ludii game. Ludii offers more than 1,000 implemented Ludii games, as well as search algorithms, which can be used as benchmark algorithms. The custom search algorithms can be evaluated conveniently since Ludii provides programmatic access to Ludii's game as well [9].

2.2 Breakthrough

Dan Troyka invented Breakthrough in 2000, which won the 8×8 Game Design competition in 2001, organized by the Abstract Games magazine [28]. The game can be played on various board sizes but is most common on an 8×8 chessboard with initially 16 pieces on its two back ranks for both players (see Figure 2.1a). Both players take turns to move. Each turn, a player must move one of her pieces, where the player is not allowed to pass.

Pieces may move one square forward into an empty square, one step diagonally into an empty square, or one step diagonally into a square occupied by a piece of the enemy player. In the last scenario, the enemy piece is captured and removed from the board to no longer be part of the game (see Figure 2.1b).



(a) Initial board (b) Legal moves

Figure 2.1: Breakthrough

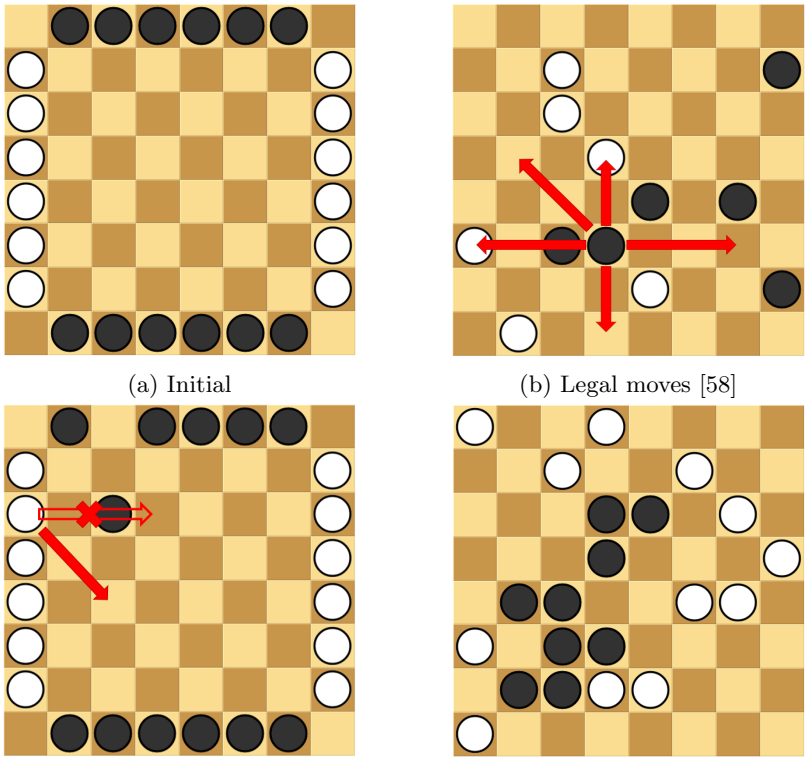
So, a piece is not allowed to move into a square with a friendly piece or capture a piece on the square in front forward. The player whose piece reaches the opponent's back rank or captures all enemy pieces wins. A draw is impossible [28].

2.3 Lines of Action

Lines of Action was invented by Claude Soucie around 1960 and described by Sid Sackson in 1969 [45]. Similar to Breakthrough, the game is played on an 8×8 board. Initially, 12 black pieces from player one are positioned in two ranks along the top and the bottom of the board, while the 12 white pieces from player two are positioned at the two files at the left and right of the board (see Figure 2.2a). Both players take turns to move. Each turn, a player must move one of her pieces, where the player is not allowed to pass.

Pieces move orthogonally or diagonally, advancing the same distance as the number of pieces (both friendly and opponents) on the line of their movement, see Figure 2.2b. The pieces may only jump over friendly pieces. Thus it is not allowed to jump over enemy pieces, see Figures 2.2c. However, an enemy piece can be captured and eliminated from the game by landing on its square (see Figure 2.2b).

The player who first connects all their pieces into a contiguous unit, with connections being either orthogonal or diagonal, wins the game (see Figure 2.2d). A player who is reduced to a single piece wins the game since, by definition, all his pieces are united. If a move (resulting from a capture) leads to both players having all their pieces connected in a contiguous unit, the game is a draw. If a player cannot move, this player losses.



(a) Initial

(b) Legal moves [58]

(c) Jump over opponent pieces

(d) Terminal game position

Figure 2.2: Lines of Action

Chapter 3

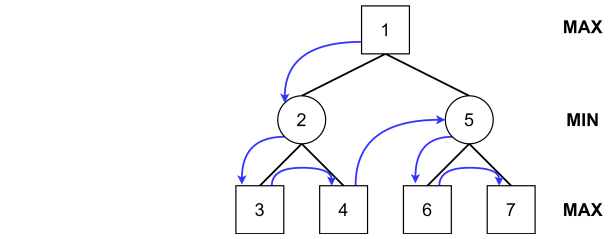
Search Algorithms

All deterministic two-player zero-sum games with perfect and complete information can be represented using a game tree, which can be explored using a search algorithm. A node in the game tree represents the game state, whereas the branches represent all legal actions for that specific game state. The levels of the game tree alternate between the two players, where each level represents a ply [46]. The search algorithm aims to find the best-performing action of the current game state. This chapter starts by describing two different search methods in Section 3.1, depth-first and best-first search. These methods can be used for searching and traversing a tree data structure. This is followed by sections describing search algorithms used to play games in this thesis. Section 3.2 described two different minimax search algorithms based on depth-first search and best-first search, together with their enhancements. Next, in Section 3.3, Monte Carlo Tree Search (MCTS) is explained, together with the enhancements used to improve the algorithm. Lastly, in Section 3.5, a combination of MCTS and minimax, using implicit minimax backups, is discussed.

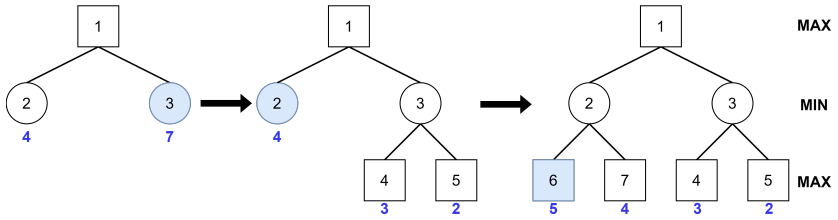
3.1 Methods for Traversing a Tree

Both depth-first and best-first search are methods to traverse a tree data structure. Depth-first search starts in the root node and explores as far as possible across each branch before backtracking, resulting in a uniform exploration of the tree data structure. Best-first search starts in the root node and recursively expands the node with the best score, resulting in a non-uniform exploration of the tree data structure.

Depth-first search usually expands more nodes than best-first search. However, best-first search requires an exponential amount of space since all nodes need to be kept in memory, while depth-first search only requires space linear in the maximum search depth. See Figure 3.1a and 3.1b for examples. The maximizing player (player one) is indicated by a square, while a circle indicates the minimizing player (player two).



(a) Depth-First Search



(b) Best-First Search

Figure 3.1: Traverse order of the game tree

The number of edges from the node to the tree's root node is called the depth of the tree, where the root node has a depth of 0. The average number of children a node has is called the branching factor.

3.2 Minimax Search

The minimax search algorithm [57] is a recursive algorithm and is most commonly used for choosing the best move in a two-player game. Minimax determines an approximated value, after a certain number of moves, with perfect play according to an evaluation function. Since in zero-sum games, one player wins while the other player loses, one player wants to maximize her gain, while the other player wants to minimize her gain of that player. Because of this, minimax alternates between maximizing and minimizing the score recursively for each ply in the tree [38]. This can also be seen in Figure 3.1, where a square indicates the maximizing player, and a circle indicates the minimizing player.

Minimax uses depth-first search by default [38] (see Algorithm 1, which shows the pseudocode of minimax), but can also be used with best-first search [32]. Both combinations are discussed in Subsections 3.2.1 and 3.2.2, respectively.

3.2.1 $\alpha\beta$ Search

When minimax is combined with depth-first search, the algorithm explores all nodes until the specified depth. However, as the search depth of minimax in-

Algorithm 1 Minimax search algorithm

```
1: function MINIMAX(state, depth, maximizing)
2:     if depth = 0 or node = terminal then
3:         return evaluate(state)
4:     end if
5:     if maximizing then
6:         score = -∞
7:         for all actions in state do
8:             child = APPLY(state, action)
9:             score = MAX(score, MINIMAX(child, depth-1, False))
10:        end for
11:    else
12:        score = ∞
13:        for all actions in state do
14:            child = APPLY(state, action)
15:            score = MIN(score, MINIMAX(child, depth-1, True))
16:        end for
17:    end if
18:    return score
19: end function
```

creases, the number of nodes explored increases exponentially. The $\alpha\beta$ search algorithm [30] is a significant enhancement of minimax since it seeks to decrease the number of nodes explored with minimax by shallow- and deep-pruning large portions of the game tree.

This is done by terminating the evaluation of a move if at least one possible option proves the move is worse than a previously examined move. One move is enough to end the evaluation since optimal play is assumed. The algorithm keeps track of the moves by using a lower bound (α) and upper bound (β), for the maximizing and minimizing player, respectively. Even though $\alpha\beta$ search prunes many subtrees, it returns the same value as minimax. See Algorithm 2 for the pseudocode of this algorithm. Many enhancements to this algorithm exist. Only the enhancements implemented in Ludii are discussed in the rest of this subsection.

Move Ordering

$\alpha\beta$ search relies on searching the best move first. If the best move, according to the evaluation function, is examined first at every node, the minimax value is obtained from a traversal of the minimal game tree [38]. Instead of exploring b^d nodes, where b is the branching factor and d is the depth of the tree, Knuth and Moore [30] prove that only $b^{\lceil \frac{d}{2} \rceil} + b^{\lfloor \frac{d}{2} \rfloor} - 1$ nodes need to be explored. An example of the savings of this algorithm can be seen in Table 3.1. For this example, a branching factor of 35 is used, equal to the average branching factor of chess [11].

Algorithm 2 $\alpha\beta$ search algorithm

```
1: function AlphaBeta(state, depth, α, β, maximizing)
2:     if depth = 0 or node = terminal then
3:         return evaluate(state)
4:     end if
5:     if maximizing then
6:         bestValue = -∞
7:         for all actions in state do
8:             child = Apply(state, action)
9:             value = AlphaBeta(child, depth-1, α, β, False)
10:            if value ≥ bestValue then
11:                bestValue = value
12:                α = Max(α, bestValue)
13:                if bestValue ≥ β then
14:                    break                          ▷ β cut-off
15:                end if
16:            end if
17:        end for
18:    else
19:        bestValue = ∞
20:        for all actions in state do
21:            child = Apply(state, action)
22:            value = AlphaBeta(child, depth-1, α, β, True)
23:            if value ≤ bestValue then
24:                bestValue = value
25:                β = Min(β, bestValue)
26:                if bestValue ≤ α then
27:                    break                          ▷ α cut-off
28:                end if
29:            end if
30:        end for
31:    end if
32:    return bestValue
33: end function
```

Depth	b^d	$b^{\lceil \frac{d}{2} \rceil} + b^{\lfloor \frac{d}{2} \rfloor} - 1$
1	35	35
2	1225	69
3	42875	1259
4	1500625	2449
5	52521875	44099
6	1838265625	85749
7	64339296875	1543499
8	2251875390625	3001249

Table 3.1: Savings of traversing minimal game tree compared to minimax, with a branching factor of 35

Selecting the best move for each node is a difficult task since, in most cases, the best move is yet to be discovered. Because of this, traversing the minimal game tree is mostly theoretical. In practice, more nodes will be explored before obtaining the minimax value.

Iterative Deepening

One technique for ordering the moves of the root node is by using Iterative Deepening [22, 54]. Iterative Deepening first executes search at depth 1, then depth 2, then depth 3, and in like manner. Even though it is not immediately apparent why this would be more efficient since more searches need to be completed ($b^1 + b^2 + b^3 + \dots$), it speeds up the search since the moves of the root node can be ordered based on the information of the previous search [63]. This algorithm can be repeated until no time is left for further search or when the whole tree has been traversed.

Transposition Tables

In many board games, a reoccurrence of a game position, called transposition, can occur. As an example of a transposition during a chess game, see Figure 3.2. The position shown in Figure 3.2 can be reached by making the moves: 1. e4 e5, 2. Nf3 or 1. Nf3 e5, 2. e4. To prevent the $\alpha\beta$ search algorithm from searching the same subtree twice, a transposition table can be used [4]. If the transposition table contains the same game position, with similar or higher depth, the stored value can be used for that position, resulting in less search.

The transposition table stores the most important information of the search process for each game position. These are

- Value
- Type of value (flag: exact value, lower bound, upper bound)
- Best move
- Search depth
- Hash value (discussed later)

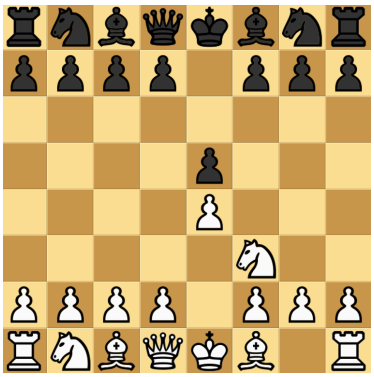


Figure 3.2: Legal game state during chess

Algorithm 3 shows how the transposition table can be added to $\alpha\beta$ search to keep track of the data. On top of recognizing transpositions, the transposition table can also be used for ordering the moves in combination with Iterative Deepening, such that the "best" move is played first, not only for the root node but for all stored nodes.

Zobrist hashing

Instead of saving the entire board, Ludii [9, 42], the general game system used, uses Zobrist hashing [64] to create a hash value of n-bits for each game position. These hash values are then used to keep track of the transpositions.

Zobrist hashing uses XOR (indicated with $\oplus$). The following properties for the $\oplus$-operator with a random sequence r_l of n-bit integers can be assumed [64]:

1. Commutative: $r_1 \oplus r_2 = r_2 \oplus r_1$.
2. Associative: $(r_1 \oplus r_2) \oplus r_3 = r_1 \oplus (r_2 \oplus r_3)$.
3. Inverse: $r_1 \oplus r_1 = 0$.
4. If $s_i = r_1 \oplus r_2 \oplus \cdots \oplus r_i$ then s_i is a random sequence.
5. s_i is distributed uniformly.

At the program's initialization, an array of pseudorandom numbers must be created for all distinct piece and board square combinations. For example, in chess, there are m different pieces ($m = 12$, 6 for white, and 6 for black) and n different squares on the board ($n = 64$). This means that $m \times n = 768$ pseudorandom numbers (i.e., $r_{m,n}$) must be created to express all board positions. Additionally, one pseudorandom number needs to be created to encode the side to move. The hash value of the initial game position can be created by XOR-ing all random numbers linked to the initial board set-up:

$$r_{\text{white rook},a1} \oplus r_{\text{white knight},b1} \oplus r_{\text{white bisschop},c1} \oplus \cdots \oplus r_{\text{black rook},h8}$$

Algorithm 3 Transposition table added to the $\alpha\beta$ search algorithm

```
1: function AlphaBetaWithTT(state, depth, α, β, maximizing)
2:     α_old = α
3:     entry = retreive(state)                          ▷ Transposition table look-up
4:     if entry.depth ≥ depth then ▷ If position is not found, entry.depth = -1
5:         if entry.flag = Exact then
6:             return entry.value
7:         else if entry.flag = LowerBound then
8:             α = Max(α, entry.value)
9:         else if entry.flag = UpperBound then
10:            β = Min(β, entry.value)
11:        end if
12:        if α ≥ β then
13:            return entry.value
14:        end if
15:    end if
```

Perform AlphaBeta(state, depth, α, β, maximizing) as seen in Algorithm 2. Additionally, besides the bestValue, the bestMove needs to be saved as well (see lines 11 and 24 of Algorithm 2).

```
16:    if bestValue ≤ α_old then
17:        flag = UpperBound
18:    else if bestValue ≥ β then
19:        flag = LowerBound
20:    elseflag = exact
21:    end if
22:    store(state, bestMove, bestValue, flag, depth)   ▷ Stores the
information in the TT
23:    return bestValue
24: end function
```

Because of the inverse property, the hash value can be incremented, e.g., when making a move, a piece can be removed by XOR-ing with $r_{\text{piece,from square}}$ and be added to the new position by XOR-ing with $r_{\text{piece,to square}}$:

$$\text{hash value new} = \text{hash value old} \oplus r_{\text{piece,from square}} \oplus r_{\text{piece,to square}}$$

Of course, more "difficult" moves, such as capturing, require more operations. But can (in most cases) still be achieved incrementally.

Terminal and leaf evaluation

Evaluating terminal nodes (i.e., nodes of terminal game positions) for two-player zero-sum games can be done by using the following score: +1 if player one wins, 0 for a draw, and -1 if player two wins [18].

Reaching the terminal game positions of a game requires too much searching. Because of this, many leaf nodes need to be evaluated using an evaluation function.

Evaluation functions are needed to give an estimated value of the game's position. These functions have to correlate with the true (game theoretical) value. The stronger the correlation, the more valuable the evaluation function. Please note that in the $\alpha\beta$ framework, all values will be calculated with respect to the first player.

For $\alpha\beta$ search, a linear heuristic evaluation function is used most commonly because these can be calculated relatively fast. Heuristic features are determined, multiplied by a weight, and summed to calculate a heuristic value. Ludii, the general game system used, implemented a wide variety of heuristic features (called "heuristic terms") that can be used in the heuristic evaluation function. These are [8]:

- **centreProximity**
  Defines a heuristic term based on the proximity of pieces to the center of a game's board.
- **componentValues**
  Defines a heuristic term based on the values of sites that contain components owned by a player.
- **cornerProximity**
  Defines a heuristic term based on the proximity of pieces to the corners of a game's board.
- **currentMoverHeuristic**
  Defines a heuristic term that only adds weight for the player whose turn it is in the current game state.
- **influence**
  Defines a heuristic term that multiplies its weight by the number of moves with distinct "to" positions that a player has in a current game state, divided by the number of playable positions that exist in the game.

- **influenceAdvanced**
  Defines a heuristic term that multiplies its weight by the number of moves with distinct "to" positions that a player has in a current game state, divided by the number of playable positions that exist in the game. Compared to "influence", this is a more advanced version that will also attempt to gain non-zero estimates of the influence of players other than the current player to move.
- **intercept**
  Defines an intercept term for heuristic-based value functions per player.
- **lineCompletionHeuristic**
  Defines a heuristic state value based on a player's potential to complete lines up to a given target length [6].
- **material**
  Defines a heuristic term based on a player's material on the board and in their hand.
- **mobilitySimple**
  Defines a simple heuristic term that multiplies its weight by the number of moves a player has in a current game state.
- **mobilityAdvanced**
  Defines a more advanced mobility heuristic that attempts to also compute non-zero mobility values for players other than the current mover.
- **nullHeuristic**
  Defines a null heuristic term that always returns a value of 0.
- **ownRegionsCount**
  Defines a heuristic term based on the sum of all counts of sites in a player's owned regions.
- **playerRegionsProximity**
  Defines a heuristic term based on the proximity of pieces to the regions owned by a particular player.
- **playerSiteMapCount**
  Defines a heuristic term that adds up the counts in sites corresponding to values in Maps where Player IDs (e.g., 1, 2, etc.) may be used as keys.
- **regionProximity**
  Defines a heuristic term based on the proximity of pieces to a particular region.
- **score**
  Defines a heuristic term based on a Player's current score in a game.
- **sidesProximity**
  Defines a heuristic term based on the proximity of pieces to the sides of a game's board.

- **threatenedMaterial**
  Defines a heuristic term based on the threatened material (which opponents can threaten with their legal moves).
- **threatenedMaterialMultipleCount**
  Defines a heuristic term based on the unthreatened material (which opponents cannot threaten with their legal moves).
- **unthreatenedMaterial**
  Defines a heuristic term based on the unthreatened material (which opponents cannot threaten with their legal moves).

Depending on the game, a selection of heuristic features can be made to create a heuristic evaluation function. Ludii provides a fine-tuned selection of features for a variety of games. For example, Breakthrough uses a weighted combination of *lineCompletionHeuristic*, *mobilitySimple*, *influence*, *ownRegionsCount*, *centreProximity*, *cornerProximity*, *sidesProximity*, *playerRegionsProximity*, *regionProximity*, *material*, *unthreatenedMaterial*, *threatenedMaterial* and *threatenedMaterialMultipleCount*, while Lines of Action only uses *centreProximity*. However, a more simplistic but game-specific heuristic evaluation function may also produce good results [33]. For example, the Breakthrough heuristic evaluation function proposed in Maarten Schadd's thesis [47] assigns each piece a score of 10 and the furthest row achieved as 2.5. The heuristic evaluation functions can be easily scaled with a slope and a sigmoid function.

Besides heuristic evaluation functions, a Neural Network (see Section 4 for more information) can also be used to evaluate leaf nodes. Even though Neural Networks can create better evaluation functions [50] (as seen in AlphaZero), a significant increase in time is required to calculate the evaluation value. Techniques to determine the weights of these Neural Networks are discussed in Section 5.

For each leaf evaluation function, a choice must be made between the correlation with the true value and the computational complexity. If it takes too long to determine the leaf evaluation, less search will be performed, but if the leaf evaluation has a low correlation with the true value, a non-optimal search will be performed.

3.2.2 Unbounded Best-First Minimax

As mentioned in Section 3.2, minimax can also be used with the best-first search method. Korf and Chickering [32] were the first to propose the best-first minimax search algorithm, but many other variants have also been designed. Cohen-Solal designed a variant of the best-first minimax search algorithm, called Unbounded Best-First Minimax (UBFM) [18], and enhanced the UBFM algorithm to perform better in combination with DRL. This led to the descent algorithm, which is explained in more detail in Section 5.1. See Algorithm 4 for the pseudocode of the UBFM algorithm.

Note that similarly to Iterative Deepening, this search can continue until no time is left. As seen in Algorithm 4, all explored nodes are stored using a

Algorithm 4 Unbounded Best-First Minimax search algorithm

```
1: function UBFM_ITERATION(state, TT, first_player)
2:     if state is terminal then
3:         return evaluate(state)
4:     else
5:         if state not in TT then
6:             for all actions in state do
7:                 child = APPLY(state, action)
8:                 values[child] = evaluate(child)
9:             end for
10:            entry = STORE(state, values)  ▷ Store the information in the TT
11:        else
12:            entry = RETREIVE(state)    ▷ Retreive information from the TT
13:            action = BEST_ACTION(state, entry, first_player)
14:            child = APPLY(state, action)
15:            values[child] = UBFM_ITERATION(child, TT)
16:            STORE(state, values)
17:        end if
18:        action = BEST_ACTION(state, entry, first_player)
19:        child = APPLY(state, action)
20:        return values[child]
21:    end if
22: end function

23: function BEST_ACTION(state, entry, first_player)
24:    if first_player then
25:        return  arg max_{action∈Actions}  entry.values[APPLY(state, action)]
26:    else
27:        return  arg min_{action∈Actions}  entry.values[APPLY(state, action)]
28:    end if
29: end function

30: function UBFM(state, maxTime, firstPlayer)
31:    startTime = time()
32:    while time() - startTime < maxTime do
33:        UBFM_ITERATION(state, TT, first_player)
34:    end while
35:    return FINAL_MOVE_SELECTION(state, TT, first_player)
36: end function
```

Transposition Table (see Section 3.2.1 for more information). Since the children are selected in minimax fashion (see BEST_ACTION) the evaluation of all states is with respect to the same player, player one.

ϵ-greedy

Selecting only the best action during the search will result in low exploration and high exploitation. The search algorithm will mainly focus on what it already knows while not improving its knowledge for long-term benefit. Because of this, instead of selecting the best action, ϵ-greedy can be used. This algorithm takes a random action with probability ϵ while taking the best action with probability $(1 - \epsilon)$. See Algorithm 5 for the pseudocode. Algorithm 5 could be used in UBFM_ITERATION (see Algorithm 4) by replacing BEST_ACTION on line 13 and 18.

Algorithm 5 ϵ-greedy selection algorithm

```
1: function EPSILON_GREEDY_ACTION(state, entry, first_player, epsilon)
2:     if UNIFORM(0,1) < epsilon then                    ▷ Uniform number
3:         return RANDOM(actions)                        ▷ Selected uniformly
4:     else
5:         return BEST_ACTION(state, entry, first_player)
6:     end if
7: end function
```

Safest child

Alongside this enhancement, a more robust final action selection can be achieved by selecting the most visited child of the root node instead of selecting the best action (see Algorithm 6). This is because the most visited child has been selected most often for having the best action during the search. Cohen-Solal showed in [18] that this led to better play. This requires to use of Algorithm 6 instead of FINAL_MOVE_SELECTION in Algorithm 4 on line 35. The UBFM algorithm with this enhancement is denoted as UBFM$_s$. Take note, that the STORE function in lines 10 and 16 of UBFM_ITERATION also requires an additional save of the visit count.

Algorithm 6 Selecting the safest move for final move selection

```
1: function SAFEST_MOVE_SELECTION(state, TT)
2:     entry = RETREIVE(state)
3:     return  arg max_{action∈Actions}  entry.visits[APPLY(state, action)]
4: end function
```

3.3 Monte Carlo Tree Search

Like UBFM, Monte Carlo Tree Search is a best-first search method [21, 31]. However, instead of using evaluation functions to approximate the value of a game state, the algorithm is guided by Monte Carlo simulations [59]. Based on these approximated values, the algorithm gradually builds a tree by focusing on the more promising parts of the search tree, using the current game state as the root node of the search tree, as long as there is time left for search. The algorithm consists of four phases: selection, play-out, expansion, and back-propagation [15, 59]. After the search is completed, the final move needs to be selected. The four phases are shown in Figure 3.3.

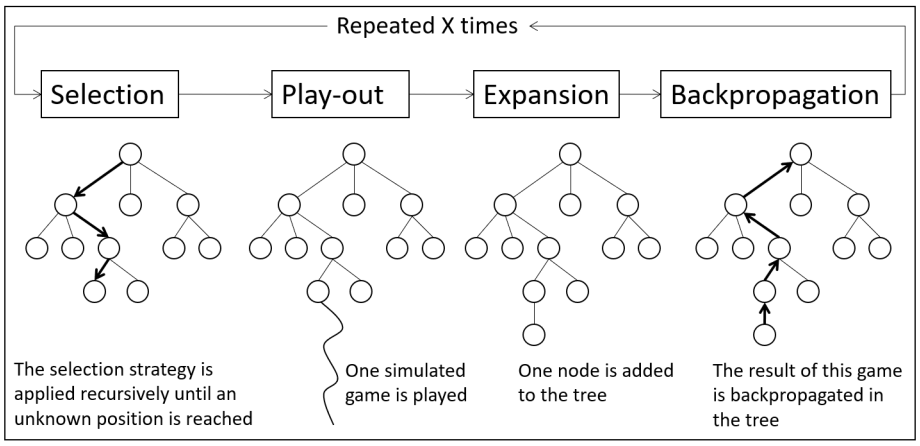


Figure 3.3: Four phases of Monte Carlo Tree Search (adapted from [15])

3.3.1 Selection

During the selection phase, the algorithm traverses down, starting in the root node, until a not yet fully expanded node is explored. A node is deemed fully expanded if it contains all child nodes.

The selection phase manages the balance between the exploitation of promising nodes seen during previous simulations and the exploration of less promising nodes, which have yet to be visited often during these previous simulations [59].

Upper Confidence Bound

Various selection algorithms have been presented [7]. The most well-known selection algorithm is based on the Upper Confidence Bound (UCB1) algorithm [1], called Upper Confidence Bound applied to Trees (UCT) [31]. Child c of node p with index i is selected that maximizes Equation 3.1

$$UCT = \frac{w_i}{n_i} + C \times \sqrt{\frac{ln(n_p)}{n_i}}, \tag{3.1}$$

where w_i is the number of wins in node i for the player, n_i is the number of visits in node i and n_p is the number of visits in node p. C is a parameter constant that controls the exploitation and exploration within the algorithm. A lower value for C results in higher exploitation, while a higher value results in higher exploration. This value needs to be tuned experimentally [59].

RAVE

By considering "all-moves-as-first" [10], the simulations can acquire faster results. For each given game position p, AMAF assigns each move a a value, where each move is considered as important as the first move. For every simulated game S_t played from a given game position p, in which move a has been played, $S_t(p, a) = 1$ if the player who played move a won the simulated game and 0 if the player who played move a lost the simulated game. The AMAF value is then the average over t, which allows AMAF to be computed incrementally [59].

Rapid Action-Value Estimation (RAVE) [25] combines UCT with AMAF values, aiming to increase the amount of information when the number of visits at a node is small. RAVE has many different implementations for different games [7]. For this thesis, the Ludii implementation has been used, where child c of node p with index i is selected that maximizes Equation 3.2.

$$UCT^{GRAVE} = (1 - \beta) * \frac{w_i}{n_i} + \beta * AMAF_i + C \times \sqrt{\frac{ln(n_p)}{n_i}}, \tag{3.2}$$

where the AMAF value of move a at node p are stored in child node i ($AMAF_i$). In Ludii the β is calculated as $\beta(n_p, n_i) = \frac{n_p}{n_p + n_i + B * n_p * n_i}$, where B is a bias constant.

A problem of RAVE is that nodes close to the leaf nodes have a low visit count, which results in not only less accurate AMAF values but also less accurate UCT_{GRAVE} values [53]. Generalized Rapid Action-Value Estimation [13] solved this by using the AMAF value of its ancestor if the visit count is lower than a given threshold, which does require more memory when being used. Since Ludii only has a GRAVE implementation and often leads to similar or better results [53], GRAVE has been used instead of RAVE.

Progressive Bias

Another proposed enhancement for the UCT formula (see Eq. 3.1) is called Progressive Bias (PB) [15]. PB adds domain knowledge by using an evaluation function. With PB, child c of node p with index i is selected that maximizes Equation 3.3.

$$UCT^{PB} = \frac{w_i}{n_i} + C \times \sqrt{\frac{ln(n_p)}{n_i}} + \frac{H_i}{l_i + 1}, \tag{3.3}$$

where H_i is the estimated value according to the evaluation function for node i, and l_i is the number of losses for node i. As a result, nodes with many losses are not biased for too long, while nodes with few losses remain biased.

Progressive History

A disadvantage of PB is that it requires an evaluation function [39]. Progressive History [48] solves this by replacing H_i with a history score, which is often used to order moves in $\alpha\beta$ search [30]. With the history score, the assumption is made that moves that are good in some game states are also good in other game states. The number of games and the total scores for each move performed during simulations are stored for each player (e.g., in a "from-to" table). Using this information, the average score for a particular move of all simulated games can be calculated, called the history score.

Both Progressive History and RAVE bias their moves based on their historical performance. However, while Progressive History stores the data for each player separately in a global table, RAVE keeps track of the AMAF values in each node. This means that Progressive History requires less memory since it does not grow with the size of the tree (as RAVE does), but it also means that its history heuristic is less accurate than RAVE's history heuristic. To prevent old data, the history table is cleared after a move is played in the actual game [59].

3.3.2 Play-out

After a node that is not part of the tree yet has been selected, the play-out phase starts. In this phase, simulations are used to determine an approximation of the value of the game state. The moves during this play-out can be selected according to different simulation strategies. Many simulation strategies exist [59]. The strategies used in this thesis are discussed.

Simulation strategy

Most commonly, during the play-out phase, random moves (drawn uniformly from all legal moves) are performed repeatedly until a terminal game state is reached. Even though playing random moves does not reflect realistic game-of-play, many simulations can be performed since it is computationally cheap. By having many simulations, a reasonable approximation of the value of the game state can be made.

Instead of taking random moves, the play-out can also be performed greedy or ϵ-greedy, guided by an evaluation function. This will result in simulations with a more realistic game of play.

However, this requires the addition of an evaluation function that increases the time needed to perform the play-out. Another simulation strategy that could be used is the Move-Average Sampling Technique (MAST) [2]. Like Progressive History, MAST assumes that moves good in one position are also good in another. It keeps track of the average results of play-outs in which a certain move is played and stores this in the global memory. The implementation used [55] selects the best history score in an *epsilon*-greedy fashion during the simulations [59].

Early termination

However, by evaluating states using an evaluation function, the play-outs will take considerably more time. Because of this, early termination can be used to compensate for this. Two different techniques exist, fixed-depth early play-out termination and dynamic early termination. Fixed-depth early termination plays k moves when a leaf node has been reached. Dynamic early termination periodically checks if the evaluation of the game state encountered during the play-out exceeds a predefined threshold. If so, the play-out is terminated [33].

3.3.3 Expansion

A new node(s) is added to the tree during the expansion phase. Most commonly, only the node selected during the selection phase, together with its value from the play-out phase, is added.

3.3.4 Backpropagation

During this last phase, the results from the play-out are backpropagated through the tree. Most commonly, each node keeps track of the sum of wins (i.e., w_i) and the number of visits (i.e., n_i), as seen in Equation 3.1.

All nodes seen during the selection phase (including the selected node) are updated. For all these nodes, the visit count is incremented by 1. In Ludii, the win count increments by 1, 0, or -1 if the play-out phase results in a win, draw, or loss for player τ, respectively. For a two-player game, the values are propagated back in a negamax fashion [30]. This is called the Monte-Carlo backpropagation. After backpropagating all values, the updated nodes can then be used again during the selection phase.

Early termination

When early termination during play-out is used (as described in Subsection 3.3.2), the final state (win or loss) of the play-out may be unknown. Because of this, the backpropagated value is handled differently.

For fixed-depth early termination, the value of the last game state of the play-out, determined by the evaluation function, is backpropagated instead (scaled between $[-1, 1]$). For dynamic early termination, 1 (a win) is backpropagated

if the evaluation value is above the threshold, and -1 (a loss) is backpropagated if the evaluation value is below the negated threshold [33].

3.3.5 Final Move Selection

The four phases repeat until no time is left. After the search is completed, the move used during the actual game needs to be selected. A few final move selection strategies can determine the best child [15].

- **Max child**
  The child with the highest win ratio ($\frac{w_i}{n_i}$).
- **Robust child (similar to safest child)**
  The child with the highest number of visits (n_i).
- **Secure child**
  The child that maximizes a lower bound. For example, the child that maximizes $\frac{w_i}{n_i} + \frac{A}{n_i}$, where A is a parameter (e.g. 4) [59]. Especially used when MCTS-Solver (see Section 3.4) is enabled.

In this thesis the robust child will be used by default.

3.4 Monte Carlo Tree Search Solver

Even though MCTS, as described in Section 3.3, is unable to prove game theoretical values, it can converge to the game theoretical values because of the UCT formula (see Eq. 3.1). Unfortunately, this does not work well in sudden-death games (like Lines of Action), where the main line towards the win is narrow since the MCTS does not converge fast enough to the game's theoretical values. For this reason, MCTS-Solver was proposed [61]. It modifies the backpropagation and selection, which will be described in Subsection 3.4.1 and 3.4.2, respectively.

3.4.1 Backpropagation

Besides backpropagating the values $\{1, 0, -1\}$, MCTS-Solver additionally backpropagates the game theoretical values (∞ for a win, and $-\infty$ for a loss), based on the player to move. If the parent node has a child with a proven win (game-theoretical value of ∞), the parent node is a proven win as well. However, the parent node is only a proven loss if all children are a proven loss.

3.4.2 Selection

When the parent node has a child with a proven win (game-theoretical value of ∞), no selection and play-out need to occur. When a parent node has one or more children that are proven losses, the proven losses cannot simply be ignored since it can result in over- and underestimation of the value of parent node [61]. Based on experiments in [61], the most effective selection is as follows: when

Eq. 3.1 is used, children with a loss will never be selected. However, when the visit count of the node is below a threshold, moves are selected according to a simulation strategy that allows proven losses to be selected. When a child is selected with a proven loss, -1 (a "normal" loss) is backpropagated to the rest of the tree.

3.5 MCTS using Implicit Minimax Backups

Many different suggestions have been made for all four phases of MCTS. Many of them even combined minimax with MCTS. Some approaches use minimax during the play-out phase [39, 60], while others even replace the play-out phase (partially) [36, 52, 62]. In contrast, by using minimax during the play-out phase, it can also be used during the other phases. Monte Carlo Tree Search with heuristic evaluations using implicit minimax backups ($\text{MCTS}_{\text{implicit}}$) [33] uses minimax of heuristic evaluations during the selection and backpropagation phases. The other two phases stay the same as described in Section 3.3.

3.5.1 Selection

Instead of only using the win count w_i and visit count n_i, $\text{MCTS}_{\text{implicit}}$ additionally uses v_i^τ, the implicit minimax evaluation with respect to player τ. This new value at node i maintains a heuristic minimax value built from the evaluations of subtrees below node i. Similar to UCT (Equation 3.1), the proposed selection consists of an exploration and exploitation part (see Equation 3.4) while also selecting the child c with index i of node p that maximizes the equation.

$$UCT^{IM} = (1-\alpha)\frac{w_i}{n_i} + \alpha v_i^\tau + C \times \sqrt{\frac{ln(n_p)}{n_i}}, \tag{3.4}$$

where α is a parameter constant that weights the influence of the heuristic minimax value.

Initially, when a node is expanded for the first time, the implicit minimax value, v_i^τ, is determined by performing a 1-ply minimax search.

3.5.2 Backpropagation

The win count w_i and visit count n_i are updated as described in Subsection 3.3.4. Additionally, the implicit minimax value, v_i^τ, is updated using the minimax backup rule based on the children's implicit minimax values. The complete pseudocode of $\text{MCTS}_{\text{implicit}}$ can be seen in Algorithm 7. Please note that the terminal evaluation in line 22 of Algorithm 7 is always with respect to the same player since it is being used on a two-player zero-sum game [33].

Algorithm 7 MCTS using implicit minimax backups

function SELECT(state) state Let A' be a set of actions from the current state maximizing Equation 3.4
    return Random action from A' ▷ Selected uniformly
end function

function UPDATE(state, reward)
    $w_{state} = w_{state} + \text{reward}$
    $n_{state} = n_{state} + 1$
    $v^{\tau}_{state} = \max_{\text{action} \in \text{Actions}} v^{\tau}_{\text{APPLY}(state,\ action)}$
end function

function SIMULATE(parent, parent_action, state)
    if some child of state not in tree **then**
        EXPAND(state)
        for all actions in state **do**
            child = APPLY(state, action)
            $\text{values}^{\tau}_{\text{child}} = \text{evaluate(child)}$ ▷ Determine heuristic value
        end for
        $v^{\tau}_{\text{state}} = \max_{\text{child} \in \text{Children}} (\text{values}^{\tau}_{\text{child}})$ ▷ 1-ply minimax back-up
        reward = PLAYOUT(state)
        UPDATE(state, reward)
        return reward
    else
        if state is terminal **then**
            return evaluate(state) ▷ Terminal evaluation
        end if
        action = SELECT(state)
        child = APPLY(state, action)
        reward = SIMULATE(state, action, child)
        UPDATE(state, reward)
        return reward
    end if
end function

function $\text{MCTS}_{\text{IMPLICIT}}$(state, maxTime)
    startTime = time()
    while time() - startTime < maxTime **do**
        SIMULATE(-, -, state)
    end while
    return safest child ▷ Child with the highest visit number
end function

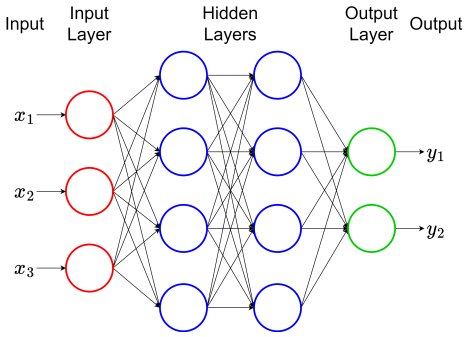
Chapter 4

Deep Learning

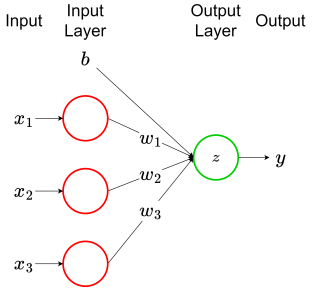
Deep Learning (DL) is a subdiscipline of Machine Learning that is mainly based on Artificial Neural Networks [44, 49], also known as Neural Networks (NNs). In recent years, multiple successful suggestions using DL have been made to improve the quality of search algorithms [18, 50]. This chapter describes the basic DL techniques used for these search algorithms. Section 4.1 describes the default NNs. In Section 4.2, a different class, better suitable for the spatial domain, Convolutional Neural Networks (CNNs), are described. Lastly, Section 4.3 briefly discusses the training technique used to train the NNs for search algorithms.

4.1 Neural Networks

Neural Networks are composed of nodes that are built in a layer-wise structure. Directed weighted edges connect the nodes (also known as artificial neurons or neurons). When all nodes of a layer are connected to all nodes of the next layer,



(a) Artificial Neural Network



(b) Linear regression visualized using artificial neurons

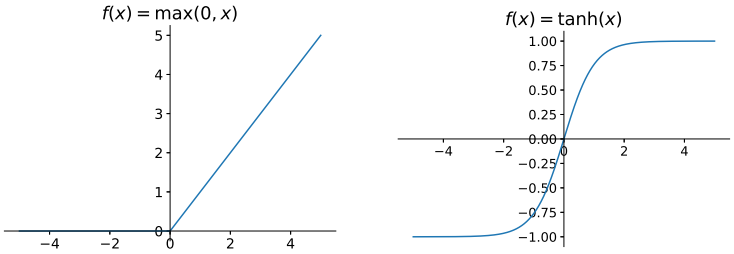
Figure 4.1: Neural Networks

the layer is called fully connected. Each network consists of an input layer, one or more hidden layers, and an output layer. See Figure 4.1a for an example of a network with fully connected layers only. If a NN has multiple hidden layers, it is called a Deep Neural Network (DNN).

Each node can be seen as its own linear regression model (see Figure 4.1b). Equation 4.1 shows how the inputs $\vec{x} = \{x_1, x_2, x_3\}$ determine the weighted sum z based on the weights $\vec{w} = \{w_1, w_2, w_3\}$ and bias node b.

$$z = w_1x_1 + w_2x_2 + w_3x_3 + b = \vec{w}^T\vec{x} + b \tag{4.1}$$

The weighted is also known as activation. A (usually non-linear) activation function is used to determine the final output of the node y. For this thesis, two different activation functions are used, Rectified Linear Unit (ReLU) [27], and hyperbolic tangent (tanh) [34] (see Figure 4.2 for visualizations). By combining multiple linear regression models in the NN, the network can express non-linear functions, which can significantly improve the expressivity power (the ability to approximate functions [37]) of the leaf evaluation function.



(a) ReLU (b) tanh

Figure 4.2: Activation functions

4.2 Convolutional Neural Networks

Based on the represented problem, a different class of Artificial Neural Networks needs to be used. One such class is CNN [24, 40]. CNNs are mainly used for problems represented in the spatial domain, such as image classification [17].

An image consists of multiple pixels with a fixed width and height. In the case of RGB images, all pixels contain three values. This means that images can be represented using three matrices (called channels) with the same width and height as the original image.

However, CNNs can also be used to evaluate game states. Instead of having an image as input, a matrix is created with the same width and height as the board. In this case, the channels represent the position of the pieces for both players (see Figure 4.3). When each player only has one type of piece, only two channels are needed (one for each player). However, when a single player has

multiple pieces (such as in chess), a distinct channel needs to be created for every piece for each player. To use the spatial input in the NN, a convolutional layer is needed, creating a CNN.

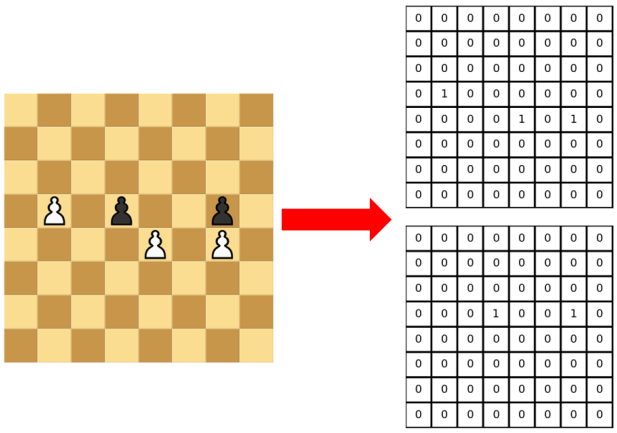


Figure 4.3: Conversion of the game state to CNN input

4.2.1 Convolutional Layer

During the forward pass of a convolutional layer, a filter with a predefined size is shifted along the input matrix. The step size of this shift is called the stride. For each step, the dot product between the filter and the input matrix is computed, resulting in a two-dimensional matrix representing an activation map of the filter used [26, 40]. This is called convolution. See Figure 4.4 for an example.

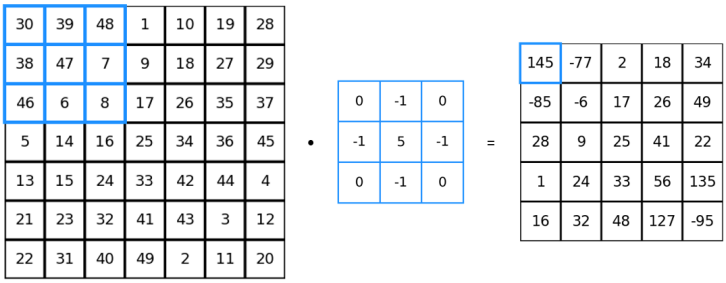


Figure 4.4: Convolution

Note that the size of the output matrix decreases in size after applying convolution (see Figure 4.4). To prevent this problem, zeros can be added at each side of the matrix, called zeros padding. When padding equals three, additional zeros are added to the matrix three times. See Figure 4.5 for an example.

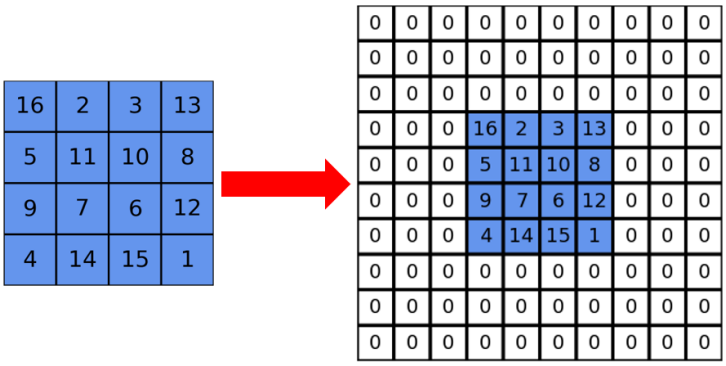


Figure 4.5: Padding of three

As mentioned before, a single filter produces one single two-dimensional activation map, regardless of the number of input channels [40]. Besides the possibility of changing the stride and the size of the filter, the number of filters used can also be changed. A new matrix with multiple channels is created as output by stacking the activation maps of multiple filters.

4.3 Reinforcement Learning

The NNs used in search algorithms can be trained in different ways. When a labeled dataset, e.g., games of experts, is available, the dataset can be used to train the NN. This is called supervised learning. However, this network will (most certainly) not exceed the expert level of play. To surpass this expert level of play, the network can be trained using reinforcement learning, with a technique called self-play [3].

In self-play, a search algorithm using a NN will play against itself while updating its weights incrementally [18]. In chapter 5, the approach used in this thesis to combine self-play and search is discussed.

Additionally, both methods can be combined by pre-training the NN using supervised learning and proceeding with the training using self-play. There are different ways to pre-train the network. DeepMind used games of experts [50], while Cohen-Solal used terminal states [18]. However, a randomly initialized network can also be used to start self-play.

Chapter 5

Combining Search Algorithms and Deep Reinforcement Learning

Successful attempts have been made to combine search with DRL (e.g., AlphaGo [50]), as mentioned in Section 1.3. Many methods of these methods are based on supervised learning or require much computational power. In [18] Cohen-Solal compared various unsupervised DRL techniques that required less computational power, were able to be trained on less computational recourses while still outperforming state-of-the-art methods. In [20], he referred to this framework as the "descent framework", named after the modified UBFM search algorithm (see Subsection 3.2.2) called descent, which is designed to produce better data for training. The descent framework is a combination of the association of multiple building blocks: a search algorithm, an action selection algorithm, a terminal evaluation function, and a procedure for selecting the data to be learned, which are discussed in Sections 5.1 to 5.4, respectively. During training, the descent framework uses experience replay to make the training more robust, which is explained in Section 5.5. Section 5.6 explains how a combination of the building blocks described can be used for training, together with the pseudocode of the training algorithm. Lastly, Section 5.7 describes an enhancement of the search algorithm, called the completion technique, to improve search and learning even more.

5.1 Search Algorithm

By replacing the heuristic evaluation function of a search algorithm with a NN, all search algorithms can be used to train the weights of a NN. However, depending on the search algorithm, different data will be produced for training (as explained in Section 5.4)

5.1.1 Descent Minimax

To produce even higher quality training data, in [18] Cohen-Solal designed a modified UBFM search algorithm called descent minimax or, more succinctly, descent. Descent combines UBFM with end-game simulations, providing values close to the optimal game path, which can be used in the learning process (according to the NN used). Similar to UBFM, descent selects the best child. However, while UBFM only extends one leaf, descent recursively selects the best child until a leaf node is reached, which can be seen as an MCTS play-out guided by an evaluation function. When a terminal state is reached, the values are updated in a minimax fashion. See Algorithm 8 for the pseudocode of the descent algorithm. Descent makes use of the same methods as UBFM, which are described in Subsection 3.2.2.

The terminal states are evaluated with a terminal heuristic (see Section 5.3). The leaf nodes are evaluated using a NN, which can be batched together for faster calculations. Similar to UBFM, both the terminal and leaf evaluations of the descent search algorithm will be with respect to the first player since the best action selects the children in a minimax fashion. Please note that in [18] Cohen-Solal used the descent search algorithm for training and the UBFM search algorithm for playing.

5.2 Action Selection

Similar to MCTS, the action selection of descent during the play-out (lines 20 and 24 of Algorithm 8) and final move selection (line 34 of Algorithm 8) can be changed.

5.2.1 Action Selection during Play-out

Different action selection methods were proposed to be used during the play-out in the descent framework: best child, ϵ-greedy, softmax distribution, and ordinal distribution.

Best child

The most straightforward approach for the end-game simulations is to select the child with the best-estimated value, similar to UBFM (see Subsection 3.2.2).

ϵ-greedy

However, to encourage exploration, ϵ-greedy can also be used in descent (see Subsection 3.2.2).

Softmax distribution

A disadvantage of ϵ-greedy is that it does not differentiate the action in terms of probabilities (except for the best action) [18]. Because of that, another dis-

Algorithm 8 Descent Minimax search algorithm

```
1: function DESCENT_ITERATION(state, TT, first_player)
2:     if state is terminal then
3:         value = evaluate(state)                          ▷ Terminal evaluation
4:         STORE(state, value)                ▷ Store the information in the TT
5:         return value
6:     else
7:         if state not in TT then
8:             for all actions in state do
9:                 child = APPLY(state, action)
10:                if child is terminal then
11:                    values[child] = evaluate(state)      ▷ Terminal evaluation
12:                    STORE(child, values[child])
13:                else
14:                    values[child] = evaluate(child)      ▷ Leaf evaluation
15:                end if
16:            end for
17:            STORE(state, values)
18:        end if
19:        entry = RETREIVE(state)   ▷ Retreive the information from the TT
20:        action = SELECT_ACTION(state, entry, first_player)
21:        child = APPLY(state, action)
22:        values[child] = DESCENT_ITERATION(child, TT)
23:        STORE(state, values)
24:        action = SELECT_ACTION(state, entry, first_player)
25:        child = APPLY(state, action)
26:        return values[child]
27:    end if
28: end function

29: function DESCENT(state, maxTime, firstPlayer)
30:    startTime = time()
31:    while time() - startTime < maxTime do
32:        DESCENT_ITERATION(state, TT, first_player)
33:    end while
34:    return FINAL_MOVE_SELECTION(state, TT, first_player)
35: end function
```

tribution is often used, the softmax distribution [5]. It creates a probability for each action, where the best value has the highest probability of being selected (see Eq. 5.1).

$$P(v^\tau) = \frac{e^{v_i^\tau}}{\sum_{j=1}^{N} e^{v_j^\tau}}, \tag{5.1}$$

where N is the number of children in state s, $i \in \{0, 1, \ldots, N-1\}$, v_i^τ is the estimated value of the state after playing action a_i in state s, and $P(v^\tau)_i$ is the probability of selecting action a_i.

Ordinal distribution

In [18], Cohen-Solal proposed an alternative distribution that does not depend on the values (like softmax), but on the order of the values, see Eq. 5.2.

$$P(c_i) = (\epsilon' + \frac{\epsilon}{N-i}) \cdot (1 - \sum_{j=0}^{j<i} P(c_j)), \tag{5.2}$$

where N is the number of children in state s, $i \in \{0, 1, \ldots, N-1\}$, c_i is the i-th best child of the state, ϵ is the exploration constant, and ϵ' is the exploitation constant $(1 - \epsilon)$.

Even though the softmax and ordinal distribution can differentiate the actions in terms of probability, Cohen-Solal mentioned that ϵ-greedy performed best [18]. Because of this, similar to [18], ϵ-greedy is used by default in this thesis.

5.2.2 Final Move Selection

Similar to UBFM, the descent algorithm can select the best child or safest child during the final move selection (see Subsection 3.2.2). The safest child is used by default in this thesis.

5.3 Terminal Evaluation

In the descent search algorithm, non-terminal nodes are evaluated using a NN. A different evaluation function must be used when a terminal state is encountered (in either the selection or play-out). As mentioned in Subsection 3.2.1, the most common score, +1 if player one wins, 0 for a draw, and -1 if player two wins, is used. This is used by default in this thesis.

However, in [18], Cohen-Solal also proposed different terminal evaluation functions called reinforcement heuristics. The reinforcement heuristics allows the reinforcement process to use general or dedicated knowledge. The best-proposed reinforcement heuristics are the additive depth heuristic (Subsection 5.3.1) and the score heuristic (5.3.2).

5.3.1 Additive Depth Heuristic

The additive depth heuristic considers how many moves it takes to reach a terminal game state. The heuristic returns l if player one wins and $-l$ if player two wins, with $l = P - p + 1$, where P is the maximum number of moves, and p is the number of moves played. By using this heuristic, the search algorithm prefers to win as quickly as possible while postponing losses. This heuristic hypothesizes that a state close to the end of the game has a more accurate value than a state further away and that the game's duration is easy to learn. Under this assumption, the search takes less risk with this heuristic to try to win as soon as possible and lose as late as possible. If P is unknown, l can be calculated by using the estimated maximum number of moves, $\tilde{P}$, with $l = \max(1, \tilde{P} - p)$.

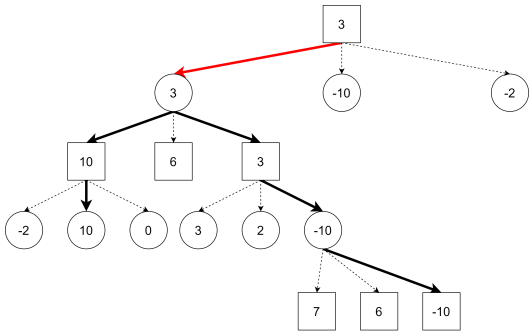
5.3.2 Score heuristic

As in other works, a natural reinforcement heuristic can be used to add additional information to the terminal evaluation function [41]. This is called the score heuristic. For example, in Othello, the goal is to have more pieces than the opponent at the end of the game. The score heuristic is the number of own pieces minus the number of pieces of the opponent. Please note, that this heuristic is not possible for games where no natural score heuristic exists (such as Breakthrough). By using the score heuristic, the NN will learn to contain more information than just an approximation of the state since it will instead contain an approximation of the scores of states.

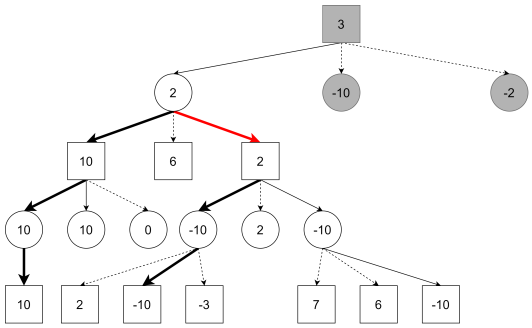
5.4 Data Selection for Learning

A set of pairs consisting of states (inputs) and values (outputs) are required to train the weights of a NN. These pairs can be generated using the search tree from a search algorithm that plays games against itself.

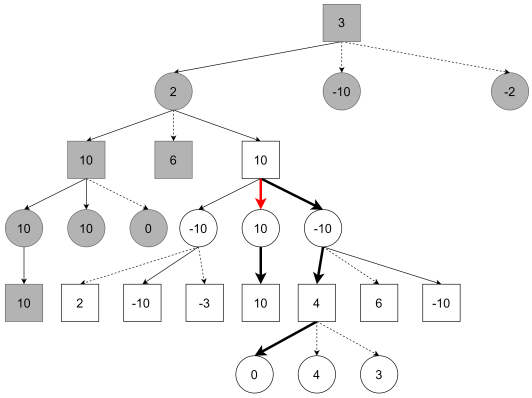
For example, see Figure 5.1, in which the descent algorithm plays a short game against itself. For each ply, the descent algorithm has enough time to perform two non-deterministic play-outs (ϵ-greedy, indicated by the bold solid lines), after which the best child is selected (indicated with the red line). The leaf nodes of each parent are indicated with a dashed line, while the interior nodes and terminal nodes are indicated with a solid line. A grey color indicates the previously explored nodes that cannot be reached anymore. For readability, the score of each node (game state) is an integer between $[-10, 10]$, where 10, 0, and -10 are a win, draw, and loss for player one, respectively. Similar to Figure 3.1, the maximizing player (player one) is indicated by a square, while a circle indicates the minimizing player (player two).



(a) Ply one



(b) Ply two



(c) Ply three

Figure 5.1: Search tree of the descent search algorithm playing a game against itself until a proven win for the first player is found

The search tree made by the search algorithm is used to generate a dataset. It is worth noting that if a different search algorithm is used, the resulting search will be different, which in turn means that the generated dataset will be different as well.

In [18] Cohen-Solal proposed three different data selection strategies to cre-

ate a set of pairs from such search trees, which he called terminal learning, root learning, and tree learning. Reported results showed that tree learning outperformed the other learning strategies in most games.

For training, the values of the pairs need to be with respect to the same player (player one, as seen in Figure 5.1). This allows a NN to create an evaluation function for this player without any modifications to the architecture of the network. The evaluation value of the other player can be calculated by multiplying the outputted value of player one by -1.

5.4.1 Terminal Learning

In AlphaGo, the set of pairs used during the learning phase is $D = \{(s, o) \mid s \in \mathbf{R}\}$, where $\mathbf{R}$ is the set of states of the sequence of the actual game, and o is the final outcome of the game (e.g., $\{1, 0, -1\}$) [50], see Figure 5.2 (the selected nodes are indicated by the green color, together with the target value). In [18], this approach is called terminal learning. It focuses on learning the value of the terminal state for the sequence of the actual states of the game. That is why all selected nodes (green nodes) contain the same (terminal) value in Figure 5.2.

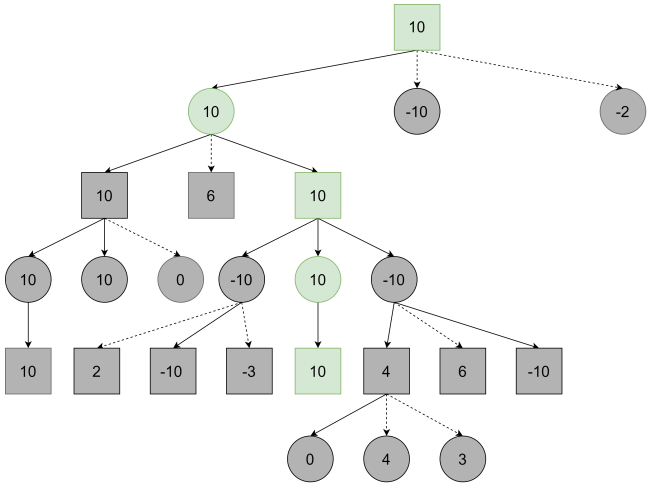


Figure 5.2: Terminal learning

5.4.2 Root Learning

The set of pairs used during the learning phase of root learning is $D = \{(s, v) \mid s \in \mathbf{R}\}$, where v is the backpropagated value of state s in the search tree. Since the backpropagated value is used, the selected nodes (indicated in green) can contain different values (see Figure 5.3). By using the backpropagated values as targets, the NN learns the value of the search tree instead. In other words, the network will learn the value it finds after searching.

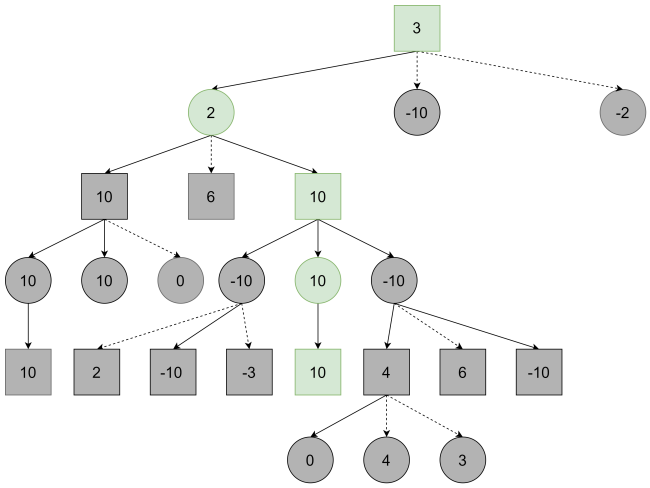


Figure 5.3: Root learning

5.4.3 Tree Learning

Using only the sequence of states of the actual game will result in a set with a relatively small number of pairs since much information used to decide the action to play is discarded, as seen in Figures 5.2 and 5.3. By including the explored game states of the search algorithm, more information can be used for training. The set of pairs used during the learning phase is $D = \{(s, v) \mid s \in \mathbf{T}\}$, where $\mathbf{T}$ is the set of states of the partial game tree of the game (all states encountered during search), excluding the non-terminal leaf nodes (see Figure 5.4). This approach is called tree learning since it uses (almost) all values of the search tree as targets. Like root learning, tree learning learns the backpropagated values (see Figure 5.4).

Instead of only using the states selected during the actual game, significantly more pairs are used for training when all encountered game states of the search tree are used, as seen in Figure 5.4. Since tree learning generates significantly more pairs, it gives the advantage of not having to run games in parallel (as done in AlphaZero [51]) since enough data is created [20].

5.5 Experience Replay

Using all data of a single game during the update of the weights of the NN can result in unstable training because of the autocorrelation between the iterations. For this reason, the descent framework uses a replay memory technique called experience replay [35].

Contrary to using all data of a single game, experience replay stores multiple games into a replay memory. It then samples from the replay memory to create a minibatch of experience, used for updating the weights of the network. This

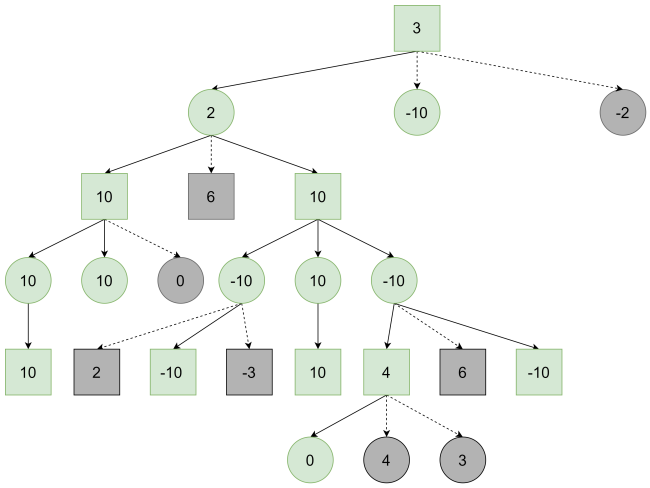


Figure 5.4: Tree learning

tackles the problem of autocorrelation leading to unstable training by making the problem more like a supervised learning problem which results in faster-converging NNs [35].

In [18], the size of the replay memory used is fixed. This means that only a fixed number of pairs (see Section 5.4) can be stored. However, the number of pairs created in each game can differ much based on the game played, the time given to determine a move, the search algorithm used, the system used, the randomness, etc. This means that in some cases, many previous games are stored in the replay memory, while in other cases, not even a full game is, making it hard to find a suitable size. Thus, in this thesis, the size of the replay memory is based on the number of games (e.g., storing the last three games). See Algorithm 9 for the pseudocode of this replay memory technique.

If the replay memory D contains more games than the maximum number of games, EXPERIENCE_REPLAY(D) starts by removing the data from the old game in the replay memory. It then samples from the replay memory to create a minibatch of experience d, which can be used for training. Contrary to the work of Cohen-Solal [18], the size of this minibatch is dynamic as well since it is based on a percentage of the replay memory (e.g., 4%).

Algorithm 9 Experience replay

```
1: function EXPERIENCE_REPLAY(D)
2:     if number of games in D > maximum number of games then
3:         D = REMOVE_OLD_GAME(D)
4:     end if
5:     d = UNIFORM(D)
6:     return D, d
7: end function
```

5.6 Training the Neural Network

By creating a combination of the building blocks, the NN can be trained into a value network by playing games against itself and updating the weights incrementally, which is depicted in Algorithm 10.

Algorithm 10 Descent framework

```
1:  function TRAINING()
2:      D = ∅
3:      TT = ∅                                          ▷ Transposition table
4:      while time left do
5:          s = INITIAL_GAME_STATE()
6:          while state not terminal do
7:              TT = SEARCH(TT)                          ▷ Self-play
8:              a = FINAL_MOVE_SELECTION(TT, s)
9:              D = ADD_NEW_DATA(TT, s) ▷ Add data based on data selection
10:             s = APPLY(s, a)
11:         end while
12:         D = ADD_TERMINAL_DATA(TT)
13:         D, d = EXPERIENCE_REPLAY(D)
14:         UPDATE_NETWORK(d)
15:     end while
16: end function
```

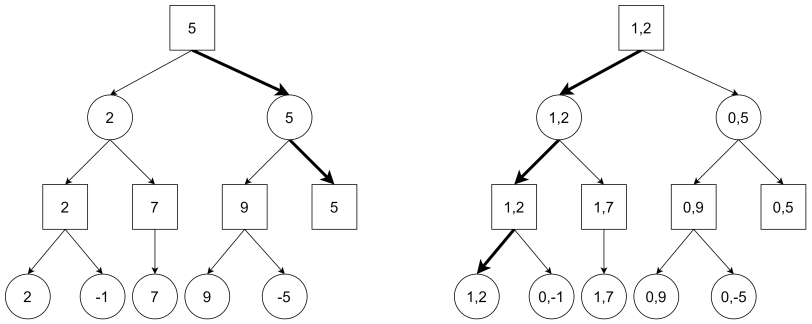
When the training starts, an empty dataset that can be seen as replay memory and a transposition table to keep track of the explored states during the search are created. As long as there is time left for training, a new game is created. The search algorithm used (see Section 5.1) plays one single game against itself. Each ply, before the actual move is performed (see Section 5.2), the stored data in the transposition table during the search is added to the replay memory D based on the data selection method used (see Section 5.4). The final (terminal) state is added to D when a terminal state is reached. This is followed by applying experience replay (see Section 5.5), where a minibatch of experience is created (d). This minibatch of experience is used to update the weights of the NN.

Please note that the descent framework only trains a value network (a neural network that estimates the values belonging to a given state), while AlphaGo Zero and Polygames train both a value network and a policy network (a neural network that indicates the probability of playing a move given a state) [14, 52].

5.7 Completion

Relying solely on leaf and terminal evaluation during the search can lead to incorrect outcomes, especially when an outcome of a leaf evaluation function can exceed the outcome of terminal evaluation (such as reinforcement heuristics,

see Section 5.3). For example, when state s is preferred over state s', based on the estimated value, even though s' is resolved state (a state with proven win or loss). Choosing s over s' is an error since the guarantee of winning can be lost. See Figure 5.5a for an example in which the guaranteed win is given away.



(a) Without using completion (b) When using completion

Figure 5.5: Preventing stepping away from the guaranteed win, by using completion (inspired by [18])

In [18] Cohen-Solal proposed a "solver" technique (similar to MCTS-Solver, see Section 3.4) that takes this into account [19]). The so-called completion technique consists of the combination of a completion value $c(s)$ and a resolution value $r(s)$ for each state s. The completion value, $c(s)$, of a leaf node, is 1 when the leaf node is terminal and a win for player one, -1 when the leaf node is terminal and a win for player two, or 0 when the leaf node is a draw or non-terminal. For non-leaf nodes, the completion value is calculated in a minimax fashion. The resolution value, $r(s)$, of a leaf node is 1 if it is terminal and 0 if not. For non-leaf nodes, the resolution value is 1 if $|c(s)| = 1$, and is the minimum resolution value of the children otherwise.

Instead of only using the estimated value of the state, $v(s)$, in the minimax framework, states are compared based on the lexical order of the pairs of $(c(s), v(s))$ (see Figure 5.5b). Using the resolution of states during the final move selection reduces the duration of games since actions with guaranteed wins are played, and therefore, a priori the duration of the learning [18]. See the pseudocode of an iteration for the completed descent and UBFM algorithm in Algorithm 11 and Algorithm 12, respectively, together with the pseudocode of both search algorithms in Algorithm 13, and the methods used in these Algorithms in Algorithm 14

Additionally, to encourage more exploration, completed ϵ-greedy can be used during the selection and play-out. In this approach, the best move is selected with probability $(1 - \epsilon)$, and a random move with the same completion value as the best move is selected with probability ϵ. This has been used by default in this thesis.

Algorithm 11 Iteration of the completed descent search algorithm

```
function COMPLETED_DESCENT_ITERATION(state, TT, first_player)
    if state is terminal then
        resolution = 1
        completion = classic_terminal(state)
        value = evaluate(state)
        STORE(state, resolution, completion, value)  ▷ Store the information in the TT
    else
        if state not in TT then
            for all actions in state do
                child = APPLY(state, action)
                if child is terminal then
                    resolutions[child] = 1
                    completions[child] = classic_terminal(child)
                    values[child] = evaluate(child)
                    STORE(state, values[child])
                else
                    values[child] = evaluate(child)              ▷ Leaf evaluation
                end if
            end for
            entry = STORE(state, resolutions, completions, values)
            action = COMPLETED_BEST_ACTION(state, entry, first_player)
            child = APPLY(state, action)
            completion, value = completions[child], values[child]
            resolution = BACKUP_RESOLUTION(state, entry)
            STORE(state, resolution, completion, value)
        end if
        if state is unresolved then
            entry = RETREIVE(state)
            actions = UNRESOLVED_ACTIONS(entry)
            action = COMPLETED_BEST_ACTION_DUAL(state, entry, actions, first_player)  ▷ Update visits
            values[child] = COMPLETED_DESCENT_ITERATION(child, TT)
            entry = RETREIVE(state)          ▷ Updated values after iteration
            action = COMPLETED_BEST_ACTION(state, entry, first_player)
            completion, value = completions[child], values[child]
            resolution = BACKUP_RESOLUTION(state, entry)
        end if
    end if
    return value
end function
```

Algorithm 12 Iteration of the completed UBFM search algorithm

```
1: function COMPLETED_UBFM_ITERATION(state, TT, first_player)
2:     if state is terminal then
3:         resolution = 1
4:         completion = classic_terminal(state)
5:         value = evaluate(state)
6:         STORE(state, resolution, completion, value)  ▷ Store the information in the TT
7:     else
8:         if state is unresolved then
9:             if state not in TT then
10:                for all actions in state do
11:                    child = APPLY(state, action)
12:                    if child is terminal then
13:                        resolutions[child] = 1
14:                        completions[child] = classic_terminal(child)
15:                        values[child] = evaluate(child)
16:                        STORE(state, values[child])
17:                    else
18:                        values[child] = evaluate(child)          ▷ Leaf evaluation
19:                    end if
20:                end for
21:                entry = STORE(state, resolutions, completions, values)
22:            else
23:                entry = RETREIVE(state)
24:                actions = UNRESOLVED_ACTION(entry)
25:                action = COMPLETED_BEST_ACTION_DUAL(state, entry, actions, first_player)  ▷ Update visits
26:                values[child] = COMPLETED_UBFM_ITERATION(child, TT)
27:                entry = STORE(state, resolutions, completions, values)
28:            end if
29:            action = COMPLETED_BEST_ACTION(state, entry, first_player)
30:            child = APPLY(state, action)
31:            completion, value = completions[child], values[child]
32:            resolution = BACKUP_RESOLUTION(state, entry)
33:            STORE(state, resolution, completion, value)
34:        else
35:            entry = RETREIVE(state)
36:            value = entry.value
37:        end if
38:    end if
39:    return value
40: end function
```

Algorithm 13 Completed descent and UBFM search algorithm

```
1: function COMPLETED_DESCENT(state, maxTime, firstPlayer)
2:     startTime = time()
3:     while time() - startTime < maxTime do
4:         COMPLETED_DESCENT_ITERATION(state, TT, first_player)
5:     end while
6:     return COMPLETED_SAFEST_MOVE_SELECTION(state, TT, first_player)
7: end function

8: function COMPLETED_UBFM(state, maxTime, firstPlayer)
9:     startTime = time()
10:    while time() - startTime < maxTime do
11:        COMPLETED_UBFM_ITERATION(state, TT, first_player)
12:    end while
13:    return COMPLETED_SAFEST_MOVE_SELECTION(state, TT, first_player)
14: end function
```

Algorithm 14 Definitions of methods used in Algorithm 11 and 12

function COMPLETED_BEST_ACTION(state, entry, first_player)
  if first_player **then**
    return $\underset{\text{action}\in\text{Actions}}{\arg\max} \begin{pmatrix} \text{entry.completions}[\textsc{Apply}(\text{state, action})], \\ \text{entry.values}[\textsc{Apply}(\text{state, action})], \\ \text{entry.visits}[\textsc{Apply}(\text{state, action})] \end{pmatrix}$
  else
    return $\underset{\text{action}\in\text{Actions}}{\arg\min} \begin{pmatrix} \text{entry.completions}[\textsc{Apply}(\text{state, action})], \\ \text{entry.values}[\textsc{Apply}(\text{state, action})], \\ \text{-entry.visits}[\textsc{Apply}(\text{state, action})] \end{pmatrix}$
  end if
end function

function COMPLETED_BEST_ACTION_DUAL(state, entry, actions, first_player)
  if first_player **then**
    return $\underset{\text{action}\in\text{actions}}{\arg\max} \begin{pmatrix} \text{entry.completions}[\textsc{Apply}(\text{state, action})], \\ \text{entry.values}[\textsc{Apply}(\text{state, action})], \\ \text{-entry.visits}[\textsc{Apply}(\text{state, action})] \end{pmatrix}$
  else
    return $\underset{\text{action}\in\text{Actions}}{\arg\min} \begin{pmatrix} \text{entry.completions}[\textsc{Apply}(\text{state, action})], \\ \text{entry.values}[\textsc{Apply}(\text{state, action})], \\ \text{entry.visits}[\textsc{Apply}(\text{state, action})] \end{pmatrix}$
  end if
end function

function BACKUP_RESOLUTION(state, entry)
  if $|\text{entry.completion}| = 1$ **then**
    return 1
  else
    return $\underset{\text{action}\in\text{Actions}}{\min}$ (entry.resolutions[APPLY(state, action)])
  end if
end function

function COMPLETED_SAFEST_MOVE_SELECTION(state, entry, first_player)
  if first_player **then**
    return $\underset{\text{action}\in\text{Actions}}{\arg\max} \begin{pmatrix} \text{entry.completions}[\textsc{Apply}(\text{state, action})], \\ \text{entry.visits}[\textsc{Apply}(\text{state, action})], \\ \text{entry.values}[\textsc{Apply}(\text{state, action})]] \end{pmatrix}$
  else
    return $\underset{\text{action}\in\text{Actions}}{\arg\min} \begin{pmatrix} \text{entry.completions}[\textsc{Apply}(\text{state, action})], \\ \text{-entry.visits}[\textsc{Apply}(\text{state, action})], \\ \text{entry.values}[\textsc{Apply}(\text{state, action})]] \end{pmatrix}$
  end if
end function

Chapter 6

Monte Carlo Tree Search using Network-based Implicit Minimax

As described in Section 3.3, $\text{MCTS}_{\text{implicit}}$ can improve the performance of MCTS by using an evaluation function during selection and play-out. Instead of using a heuristic evaluation function, it is also possible to replace this with a NN. As mentioned in Section 3.2, using a NN is computationally more expensive than using a heuristic evaluation function. Therefore, changes have to be made to the MCTS architecture, as seen in AlphaGo [50]. Many variants have been suggested to create a working MCTS architecture that works with the high computation time of NNs. Changes have been made to the selection, play-out, and backpropagation phase, which are discussed in Sections 6.1, 6.2 and 6.3, respectively (all the other suggested variants are discussed in Section 7.5). The architecture described in this chapter, MCTS using Network-based Implicit Minimax, is denoted as MCTS_{NIM}. An additional MCTS architecture for future work is discussed in Section 6.4. Besides using $\text{MCTS}_{\text{implicit}}$ for game-playing, it could also be used during training, as discussed in Section 6.5.

6.1 Selection

As discussed in Section 3.3 and 3.5, many MCTS algorithms changed and enhanced their UCT selection function to improve their performance. Two of these models are AlphaGo Zero, and Polygames [14, 52], which changed their UCT function to be more suitable when using a neural network (see Eq. 6.1 and 6.2, respectively).

$$UCT^{AlphaZero} = \frac{w_i}{n_i} + P_i \times \frac{1}{1+n_i}, \text{ and} \tag{6.1}$$

$$UCT^{Polygames} = \frac{w_i}{n_i} + P_i \times \frac{n_p}{n_i}, \tag{6.2}$$

where P_i is the probability value from the policy network from the move from parent node p to child node i.

This thesis proposes improvements to the original implicit UCT function (Eq. 3.4) used in the MCTS_{NIM} algorithm. Three variations of the implicit UCT function, each with enhancements to either the exploration, exploitation or both are proposed. The modifications are discussed in Subsections 6.1.1 and 6.1.2 for the exploration and exploitation, respectively.

6.1.1 Exploration

While UCT^{IM} (see Eq. 3.4)) uses a fixed constant to scale the exploration, both AlphaGo Zero and Polygames use policy networks to scale the exploration instead. Since the descent framework only generates a value network (as mentioned in Section 5.6), a policy network cannot be added to UCT^{IM} when using the descent framework.

However, since all children are calculated during the one-ply search of $\text{MCTS}_{\text{implicit}}$, something similar can be achieved by using a softmax function (see Eq. 5.1) to convert the estimated values and implicit minimax values of the children into probabilities and use these probabilities to scale the exploration. To encourage exploration even more, a temperature can be added to the softmax, which makes the probabilities more equal in the beginning while making them more different when the number of visits increases (see Eq. 6.3).

$$P(v^\tau, \mathrm{T}) = \frac{e^{v_i^\tau/\mathrm{T}}}{\sum_{j=1}^{N} e^{v_j^\tau/\mathrm{T}}}, \tag{6.3}$$

where N is the number of children in state s, $i \in \{0, 1, \ldots, N-1\}$, v_i^τ is the estimated value of the state after playing action a_i in state s, T is the temperature, and $P(v^\tau)_i$ is the probability of selecting action a_i. The substitution of the exploration constant C in Eq. 3.4 with the softmax function that includes a temperature (as defined in equation 6.3) results in an enhanced implicit UCT function equation, as shown in Eq. 6.4.

$$UCT^{exploration} = (1-\alpha)\frac{w_i}{n_i} + \alpha v_i^\tau + P(v^\tau, \frac{C}{n_p})_i \times \sqrt{\frac{ln(n_p)}{n_c}}, \tag{6.4}$$

where $P(v^\tau, \frac{C}{n_p})_i$ is the softmax value with temperature $\frac{C}{n_p}$ of child i based on the estimated values of all children v^τ

6.1.2 Exploitation

$\text{MCTS}_{\text{implicit}}$ uses a fixed α value during search (see Eq. 3.4). This means that the search algorithm is not able to differentiate between good and bad estimates

of the MCTS score ($\frac{w_i}{n_i}$). This is solved by using the number of visits of a node and decreasing the α value linearly over time, such that in the beginning, UCT is more dependent on the implicit minimax value, while later on, the UCT is more (or fully) dependent on the MCTS score (see Eq. 6.5).

$$a_i = \max\left(\alpha_{\min}, \alpha_{\text{init}} - sn_i\alpha_{\text{init}}\right), \tag{6.5}$$

where α_{init} is the initial influence of the estimated values, $\alpha_{\min}$ is the minimum bound of influence of the estimated values, and s is the slope used to decrease α.

Combining this enhancement with the original implicit UCT function (Eq. 3.4) results in a second enhanced implicit UCT function, as shown in Eq. 6.6.

$$UCT^{alpha} = (1-\alpha_i)\frac{w_i}{n_i} + \alpha_i v_i^{\tau} + C \times \sqrt{\frac{ln(n_p)}{n_i}}, \tag{6.6}$$

where α_i is the influence of the estimated value calculated by Eq. 6.5. The exploitation and exploration enhancements can also be combined to create a third enhanced version of the implicit UCT function. This new equation is shown in Eq. 6.7.

$$UCT^{combined} = (1-\alpha_i)\frac{w_i}{n_i} + \alpha_i v_i^{\tau} + P(v^{\tau}, \frac{C}{n_p})_i \times \sqrt{\frac{ln(n_p)}{n_c}}, \tag{6.7}$$

6.2 Play-out

When using a greedy or ϵ-greedy play-out, most of the evaluations, so also computations when using NN, take place during the play-out phase of the MCTS algorithm. Hence most commonly, modifications are made to the play-out phase [18, 20, 51, 52]. After trying multiple variants (see Section 7.5), as also shown in previous research [52], performing no play-out results in good-performing MCTS search algorithms when using a NN. Therefore, no play-out is performed in this architecture. Rather than selecting the first node encountered during play-out for expansion, the modification involves selecting the best-performing child, determined by the UCT selection function, to expand (as done by the MCTS implementation of Ludii [9, 8, 42]).

6.3 Backpropagation

Since, most often, a non-terminal game position is selected during the selection phase, and no play-out is performed, it is not always possible to backpropagate a win or a loss. Because of this, similar to fixed-depth early termination (as seen in Subsection 3.3.4), an estimated value is backpropagated instead (scaled between $[-1, 1]$). AlphaGo Zero and Polygames backpropagate the estimated

value of the neural network for the expanded game position [14, 52]. This will also be used by MCTS_{NIM}.

Figure 6.1 depicts a visualization of the modified architecture of MCTS_{NIM}. Specifically, the figure shows the original selection phase, the use of no play-out, the addition of the implicit minimax method when adding a new node, and the backpropagation of both the implicit minimax value and the estimated value of the expanded node.

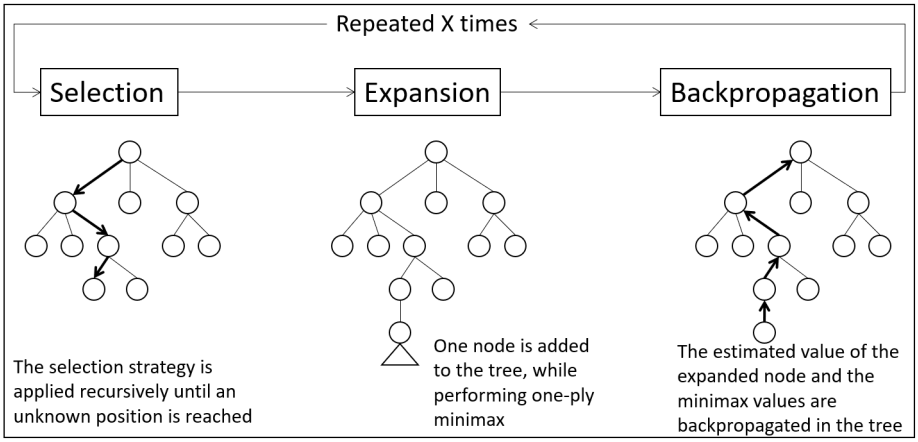


Figure 6.1: Three phases of MCTS_{NIM} (adapted from [15])

6.4 Proof of Concept

The $\text{MCTS}_{\text{implicit}}$ algorithm in [33] uses heuristic ϵ-greedy play-outs with dynamic early termination. However, since this play-out is computationally too expensive when using a NN, no play-out is performed in the proposed MCTS_{NIM} architecture (see Sections 6.1 till 6.3). Nevertheless, it is also possible to perform play-outs with a smaller policy network, as done in AlphaGo [50]. Because of the limited time, training a second policy network was not possible. Therefore a proof of concept has been made for a second architecture, denoted as MCTS_{POC}.

This is achieved by using the neural network as an evaluation function for the selection phase (as done in Section 6.1) while using Maarten Schadd's evaluation function (as described in Subsection 3.2.1) for the ϵ-greedy play-outs with dynamic early termination. As opposed to the proposed MCTS_{NIM} architecture, it turned out that this architecture performed better with a fixed α value (see Appendix C). Therefore, MCTS_{POC} uses Eq. 6.4 by default.

6.5 Training with $\text{MCTS}_{\text{implicit}}$

Descent and $\text{MCTS}_{\text{implicit}}$ have much in common. Both search algorithms have a selection phase, an expansion phase, a play-out phase, and a minimax backpropagation. During training, descent only gets guided by the minimax backups. It ignores the move that wins the most as long as it does not have the highest estimated value. This can be a problem when you have a move that is slightly better than another move, according to the evaluation, while it results in more losses. $\text{MCTS}_{\text{implicit}}$ solves this problem by also using the backpropagated estimated values during selection. This could result in faster converging and better-performing NNs, since more informed decisions could be made. Because of this, four different $\text{MCTS}_{\text{implicit}}$ implementations have been made for training.

The first $\text{MCTS}_{\text{implicit}}$ implementation was developed to be as similar to completed descent as possible. It uses the original implicit UCT function (Eq. 3.4) and an ϵ-greedy approach for play-outs guided by a neural network. The child with the most visits is selected for play, while all terminal and non-leaf game states are added to the training data.

The second $\text{MCTS}_{\text{implicit}}$ implementation is more similar to completed UBFM, as it does not use play-outs. Instead, the backpropagated value is an estimated value of the expanded game position. Everything else remains the same as in the first implementation.

Additionally, using a trained network by MCTS_{NIM} itself could potentially also increase its performance since the network learns to predict the values found after searching with MCTS_{NIM} more specifically. Therefore, for the third and fourth implementations, the enhanced implicit UCT function (Eq. 6.7) is used instead of the original implicit UCT function. Both implementations are otherwise similar to the first and second implementations, respectively.

Chapter 7

Experiments

Several experiments have been performed to test the performance of the MCTS_{NIM} algorithm while using a NN trained with the completed descent search algorithm. First, the experiments and the setup of the experiments are described in Section 7.1. This is followed by Section 7.2, which explains the implementation details, and Section 7.3, which discusses the parameter selection for the implemented search algorithms. Section 7.4 shows the results of the performed experiments against the benchmark models, but also against the state-of-the-art completed UBFM algorithm. Section 7.5 discusses the additional variants that have been suggested to create the proposed architecture (as seen in Chapter 6), and Section 7.6 describes and shows the results of the experiments when using a NN trained with $\text{MCTS}_{\text{implicit}}$ (as described in Section 6.5) instead of completed descent. Finally, Section 7.7 shows the results of the experiments on the proof of concept architecture for future work (see Section 6.4).

7.1 Setup

To test the performance of MCTS_{NIM}, the search algorithm with its enhancements will be compared against the two benchmark models discussed in Subsection 7.1.1, while the CNN as shown in Subsection 7.1.2 is used by MCTS_{NIM}. For the first experiments, the CNN is trained by the completed descent search algorithm described in Subsection 7.1.3. The hardware used for training and the experiments is described in Subsection 7.1.4.

7.1.1 Benchmark Models

The Ludii framework [9, 42] offers several implemented search algorithms included with heuristic evaluation functions. For this thesis, two Ludii implementations have been selected as benchmark algorithms.

The first algorithm is $\alpha\beta$ search, with move-ordering, iterative deepening,

and transposition tables (denoted as $\alpha\beta_{\text{bench}}$). The heuristic evaluation used is the evaluation function fine-tuned by the Ludii team (as shown in Subsection 3.2.1).

For the second algorithm, an MCTS implementation was used. To select the best MCTS implementation, several parallelized MCTS implementations (with 6 threads) have been compared to a non-parallelized MCTS implementation with Eq. 3.1 for selection, random play-outs, and Monte-Carlo backpropagations (denoted as $\text{MCTS}_{\text{default}}$) in 100 games of Breakthrough. The Progressive Bias implementations use Maarten Schadd's evaluation function (as described in Section 3.2.1). Table 7.1 shows the performance of the tested MCTS implementations with a confidence interval of 95%. Please note that the confidence interval can only be calculated if there are more than four wins [29]. MCTS with Progressive Bias and MAST is the best-performing tested MCTS algorithm in Breakthrough. Because of this, MCTS with Progressive Bias and MAST will be used as the second benchmark model (denoted as $\text{MCTS}_{\text{bench}}$).

Bot 1	**Bot 2** $\textbf{MCTS}_{\textbf{default}}$
MCTS_ProgressiveBiasGRAVE_MAST	88.0 ±6.4
MCTS_ProgressiveBias_MAST	**98.0**
MCTS_ProgressiveHistoryGRAVE_MAST	46.0 ±9.8
MCTS_UCB1GRAVE_MAST	74.0 ±8.6
MCTS_UCB1GRAVE_Random	24.0 ±8.4
MCTS_UCB1_MAST	96.0
MCTS_UCB1_Random	83.0 ±7.4

Table 7.1: Win percentage of several parallelized MCTS implementations with different enhancements against $\text{MCTS}_{\text{default}}$ for 100 games of Breakthrough

7.1.2 CNN Architecture

The architecture used is similar to the CNN architecture used by Cohen-Solal [18]. The following abbreviations, inspired by [23], are used to describe this CNN architecture:

- **Zero Padding Layer:** Z-(size of padded rows and columns)
- **Convolutional Layer:** C-(number of filters)-(size of the filters)-(stride)-(activation function)
- **Fully Connected Layer:** F-(number of nodes)-(activation function)
- **ReLU:** R
- **Hyperbolic tangent (tanh):** T

The architecture, as shown in Figure 7.1, is structured as follows: Z-1 → C-64-3-1-R → C-64-3-1-R → C-64-3-1-R → F-100-R → F-1-T.

Please note that in [18], the final activation function of the NN got changed when using the reinforcement heuristics. However, as done in this thesis, the architecture of the CNN stays unchanged since the reinforcement heuristics get scaled between $[-1, 1]$ instead.

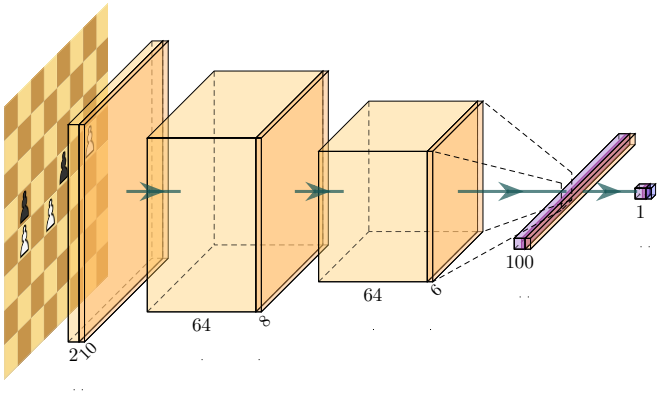


Figure 7.1: CNN architecture

7.1.3 Training of Neural Network with Descent

To speed up training, the shown CNN is first pre-trained on one million random terminal game states, after which the CNN is trained by using the descent framework. It uses the completed descent search algorithm with ϵ-greedy exploration policy ($\epsilon = 0.05$), the safest selection policy combined with tree learning, and the classic evaluation function. The descent algorithm gets 1 second per move to collect the data, resulting in approximately 1000 games per day when also updating the NNs. Data augmentation has been performed by mirroring the game positions horizontally. Instead of using the experience replay with a specified number of data pairs, an experience replay that is based on the last 3 games with a sampling rate of 0.04 has been used (see Section 5.5) since it performed better than other amounts of games after 12 hours of training (see Appendix A). Cohen-Solal trained his network for 30 days [20]. However, due to the time constraints of this thesis, the network used for the experiments will be trained for 36 hours only (unless otherwise stated).

7.1.4 Computational Specifications

The following hardware has been used for the performed training and experiments: two GPUs, both NVIDIA RTX A5000, CPU AMD EPYC 7453 28-Core, and 251GB RAM on RedHat. The code is implemented in Java (version 19), using DeepLearning4J (version 1.0.0-M2.1) with CUDA for the Deep Learning implementations.

7.2 Implementation Details

Several decisions have been made to increase the performance or optimize the code for completed UBFM, completed descent, MCTS_{NIM}, and MCTS_{POC}, which are described in this section.

7.2.1 Completed UBFM and Completed Descent

Similar to [18], for completed UBFM and completed descent, an ϵ-greedy exploration policy ($\epsilon = 0.05$) has been used during the search, while the selection policy of the final move is based on the safest child. On top of that, all children of a node are evaluated by the network in a single batch. Instead of iterating through all children to select the best one, a sorted list has been used. When a new value is found, the respective child will be repositioned in this list. Lastly, contrary to [18], the explored search tree is reused. This is done by only deleting the game states in the transposition table which have not been seen during the last three searches.

7.2.2 MCTS_{NIM}

Like UBFM and descent, MCTS_{NIM} reuses its tree for the next search while also evaluating multiple children batched. To increase the number of iterations, tree parallelization [16] has been used. Since multiple threads have been used, each thread received its own neural network for evaluations. When a child is not yet expanded, the initial MCTS score ($\frac{w_i}{n_i}$) is set to the MCTS score of the parent. Since multiple threads are used, a virtual loss (of 1) is added to this score for each entered thread [16]. The used implicit UCT function, together with the selected parameters, are discussed in Subsection 7.4.1.

7.2.3 MCTS_{POC}

MCTS_{POC} is implemented similarly to MCTS_{NIM}, where the only difference is the play-out and backpropagations phase as discussed in Section 6.4. The heuristic ϵ-greedy play-outs with dynamic early termination used a threshold of 0.4 as in [33].

7.3 Parameter Selection

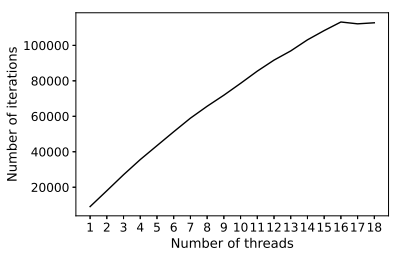
As seen in [20] and [33], the parameters used in the UCT function can significantly impact the performance of an MCTS algorithm. Therefore, the parameters used by MCTS_{NIM} need to be adjusted precisely. On top of that, the number of threads used for the parallelized MCTS algorithm also needs to be tuned.

In the most optimal situation, the parameters used by MCTS_{NIM} would be tuned using an optimization technique in combination with a large number of games (to prevent large confidence intervals) for each suggested implicit UCT

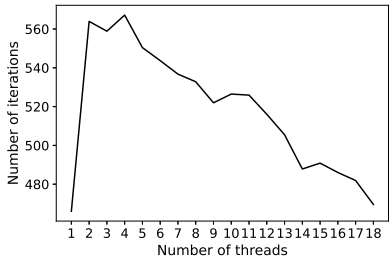
function individually. However, only a limited amount of time is available to tune the parameters, which results in a limited amount of parameters to try. Therefore, 100 games against $\alpha\beta_{\text{bench}}$ are run for each parameter only. Since an optimization method requires too much time, the parameters are hand-tuned with an approach inspired by Powell Search [43]. Only the parameters of Eq. 6.7 are tuned. The parameters of Eqs. 6.4 and 6.6 are chosen by using a combination of the parameters selected for Eq. 3.4 (see Appendix B and Eq. 6.7. To give an insight into the confidence intervals, all visualizations concerning the parameter tuning of a UCT function in this thesis (including the Appendix) are included with a 95% confidence interval.

7.3.1 Number of Threads

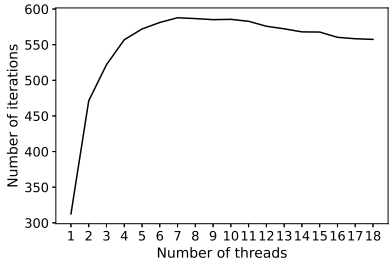
Both $\text{MCTS}_{\text{bench}}$, MCTS_{NIM}, and MCTS_{POC} are using multiple threads to increase the number of iterations. However, because of the synchronization (to prevent race conditions) and the parallelization overhead, adding more threads does not result in more iterations indefinitely. Because of this, the number of threads needs to be tuned to optimize the number of iterations. This is achieved by performing a single search (of 1 second) 100 times on the initial board position of Breakthrough (as seen in Figure 2.1a) without reusing the tree. Figure 7.2 shows the average iterations/second achieved by the search algorithms over these 100 searches.



(a) $\text{MCTS}_{\text{bench}}$



(b) MCTS_{NIM}



(c) MCTS_{POC}

Figure 7.2: Average iterations/second on the initial board positions when using a different number of threads

The $\text{MCTS}_{\text{bench}}$ algorithm shows a near-linear increase in iterations due to its cheap play-outs, as seen in Figure 7.2a. The number of iterations continues to grow until it reaches the hardware limitations at 16 threads, with approximately 113,000 iterations per second, since the Java Virtual Machine also requires some threads.

In contrast to $\text{MCTS}_{\text{bench}}$, MCTS_{NIM} has significantly fewer iterations per second with a less noticeable improvement (and even decrease) as the number of threads increases, as shown in Figure 7.2b. This performance issue is due to the two GPUs used by the NN, which impose limitations on the iterations per second. Exceeding the optimal number of threads results in a decrease in the number of iterations per second, as most threads have to wait for the GPU to complete computations. Nevertheless, Figure 7.2b suggests that using 2 threads per GPU (4 in total) yields the best performance with approximately 570 iterations per second.

Based on Figure 7.2c, the inclusion of a play-out with heuristic evaluations, as seen in MCTS_{POC}, results in an increase in the number of iterations, while experiencing less severe decreases in iterations as the number of threads increases beyond the optimal amount compared to MCTS_{NIM}. This is because the threads do not need to wait as long for the GPU to complete computations, as some threads perform play-outs while others evaluate the children. This allows the architecture to use twice as many threads per GPU as MCTS_{NIM}, resulting in a total of 8 threads with approximately 585 iterations per second.

In summary, based on Figure 7.2, 16 threads are selected for $\text{MCTS}_{\text{bench}}$, 4 threads are selected for MCTS_{NIM} and 8 threads are selected for MCTS_{POC}.

7.3.2 Initial Influence of Estimated Value

After the number of threads is determined, the parameters for Eq. 6.7 used by MCTS_{NIM} can be tuned. First, the initial influence of the estimated value is tuned, using $C = \sqrt{2}$, $s = 0$, and $\alpha_{\text{min}} = 0$ as starting points. The initial influence is selected by testing different α_{init} values as shown in Figure 7.3. Based on the figure, $\alpha_{\text{init}} = 0.6$ results in the best performance (92.0%), which is therefore selected.

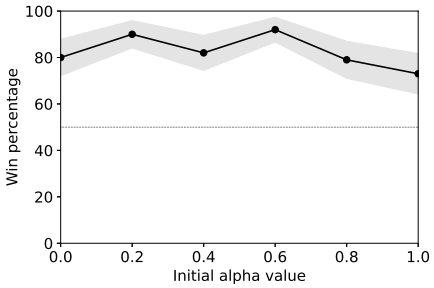


Figure 7.3: Win percentage of MCTS_{NIM} over different α_{init} values against $\alpha\beta_{\text{bench}}$ for 100 games

7.3.3 Exploration

Secondly, the exploration value is tuned, using the found $\alpha_{\text{init}} = 0.6$, and $s = 0$ and $\alpha_{\text{min}} = 0$ as starting points. Also here, the exploration constant is selected by testing different C values, as shown in Figure 7.4. The figure shows that $C = 2$ results in the best performance (92.0%).

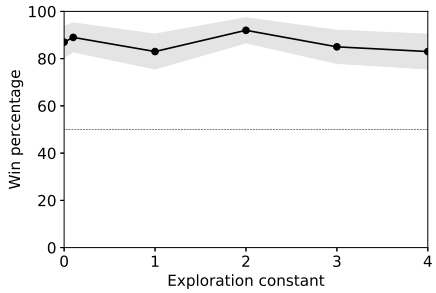


Figure 7.4: Win percentage of MCTS_{NIM} over different C values against $\alpha\beta_{\text{bench}}$ for 100 games

7.3.4 Slope

After that, the slope value is tuned, using the found $\alpha_{\text{init}} = 0.6$ and $C = 2$, and $\alpha_{\text{min}} = 0$ as starting point. Similar to previous parameters, different s values are tested, as shown in Figure 7.5. It appears that $s = 0.1$ resulted in the best performance (95.0%).

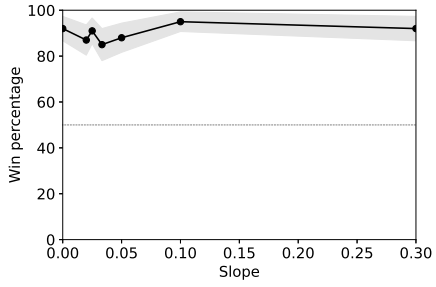


Figure 7.5: Win percentage of MCTS_{NIM} over different s values against $\alpha\beta_{\text{bench}}$ for 100 games

7.3.5 Minimum Influence of Estimated Value

Lastly, different α_{min} are tested using the found parameters of $\alpha_{\text{init}} = 0.6$, $C = 2$ and $s = 0.1$. According to Figure 7.6, it turns out that $\alpha_{\text{min}} = 0.3$ results in the best-performing search algorithm (96.0%).

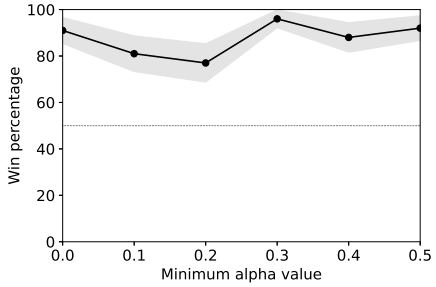


Figure 7.6: Win percentage of MCTS_{NIM} over different $\alpha_{\min}$ values against $\alpha\beta_{\text{bench}}$ for 100 games

7.4 Results

In order to determine the best enhanced implicit UCT function (see Section 6.1) to use in combination with MCTS_{NIM}, a comparison is conducted between each UCT function and the two benchmark models in Subsection 7.4.1. To better compare these win rates among the UCT functions, games are also played between completed UBFM and the benchmark models in Subsection 7.4.2. Lastly, games are played between MCTS_{NIM} and completed UBFM in Subsection 7.4.3 to assess their respective performances. Due to time constraints, all these experiments are carried out on Breakthrough, while Lines of Action is only used as verification in Section 7.8.

7.4.1 Comparing the UCT functions of MCTS_{NIM} against Benchmark Models

The three different suggested UCT functions (Eqs. 6.4, 6.6 and 6.7) are compared to the original implicit UCT function (Eq. 3.4) to see whether and which

Model	**UCT function**	**Parameters**
$\text{MCTS}_{\text{base}}$	$(1-\alpha)\frac{w_i}{n_i} + \alpha v_i^\tau + C \times \sqrt{\frac{ln(n_p)}{n_i}}$	$\alpha = 0.8$, and $C = 0.0001$
$\text{MCTS}_{\text{alpha}}$	$(1-\alpha_i)\frac{w_i}{n_i} + \alpha_i v_i^\tau + C \times \sqrt{\frac{ln(n_p)}{n_i}}$	$\alpha_{\text{init}} = 0.6$, $\alpha_{\min} = 0.3$, $s = 0.1$, and $C = 0.0001$
$\text{MCTS}_{\text{exploration}}$	$(1-\alpha)\frac{w_i}{n_i} + \alpha v_i^\tau + P(v^\tau, \frac{C}{n_p})_i \times \sqrt{\frac{ln(n_p)}{n_c}}$	$\alpha = 0.8$, and $C = 2$
$\text{MCTS}_{\text{combined}}$	$((1-\alpha_i)\frac{w_i}{n_i} + \alpha_i v_i^\tau + P(v^\tau, \frac{C}{n_p})_i \times \sqrt{\frac{ln(n_p)}{n_c}}$	$\alpha_{\text{init}} = 0.6$, $\alpha_{\min} = 0.3$, $s = 0.1$, and $C = 2$

Table 7.2: MCTS_{NIM} models using different UCT functions

enhancement performs best. This is accomplished by creating four different MCTS_{NIM} variants using the four different implicit UCT functions. The notation of these variants and the UCT function used are described in Table 7.2. These algorithms' other phases are as described in Chapter 6.

As seen in Section 7.3, the parameters of the UCT function from $\text{MCTS}_{\text{combined}}$ (Eq. 6.7) are tuned. However, as a different UCT function is used, its parameters must be tuned again. The parameter values used by $\text{MCTS}_{\text{base}}$ are optimized similarly to $\text{MCTS}_{\text{combined}}$, as detailed in Appendix B. resulting in $\alpha = 0.8$ and $C = 0.0001$. Due to time constraints, the parameter values for $\text{MCTS}_{\text{alpha}}$ and $\text{MCTS}_{\text{exploration}}$ are a combination of these two tuned parameter values, based on which enhancement is used (see Table 7.2).

All these search algorithms played 300 games (100 as player one, and 100 as player two, with 1 second per move) against the benchmark models. See Table 7.3 for the results together with the confidence intervals of 95% [29].

	Bot 2	
Bot 1	$\alpha\beta_{\textbf{bench}}$	$\textbf{MCTS}_{\textbf{bench}}$
$\textbf{MCTS}_{\textbf{base}}$	73.0 ±5.0	84.3 ±4.1
$\textbf{MCTS}_{\textbf{alpha}}$	92.3 ±3.0	96.0 ±2.2
$\textbf{MCTS}_{\textbf{exploration}}$	83.3 ±4.2	88.0 ±3.7
$\textbf{MCTS}_{\textbf{combined}}$	94.7 ±2.5	91.0 ±3.2

Table 7.3: Win percentage of MCTS_{NIM} using different implicit UCT functions against the benchmark models with 1 second per move for 300 games

To prove that the enhancements perform significantly better than $\text{MCTS}_{\text{base}}$, a two-tailed test is performed between the win percentage of $\text{MCTS}_{\text{base}}$ and the win percentages $\text{MCTS}_{\text{alpha}}$, $\text{MCTS}_{\text{exploration}}$ and $\text{MCTS}_{\text{combined}}$. See Table 7.4 for the calculated p-values [56].

	Bot 2	
Bot 1	$\alpha\beta_{\textbf{bench}}$	$\textbf{MCTS}_{\textbf{bench}}$
$\textbf{MCTS}_{\textbf{alpha}}$	0.0000	0.0000
$\textbf{MCTS}_{\textbf{exploration}}$	0.0022	0.1934
$\textbf{MCTS}_{\textbf{combined}}$	0.0000	0.0130

Table 7.4: P-values of the proposed UCT functions used by MCTS_{NIM} against $\text{MCTS}_{\text{base}}$

A significant difference can be found for all three UCT functions when playing games against $\alpha\beta_{\text{bench}}$, while only $\text{MCTS}_{\text{alpha}}$ and $\text{MCTS}_{\text{combined}}$ have a significant difference compared to $\text{MCTS}_{\text{base}}$ when playing games against $\text{MCTS}_{\text{bench}}$. The less performing $\text{MCTS}_{\text{base}}$ and $\text{MCTS}_{\text{exploration}}$ are excluded from further experiments due to time constraints. Since $\text{MCTS}_{\text{alpha}}$ and $\text{MCTS}_{\text{combined}}$ have similar results, 300 games are played between them to determine which UCT function performs best (see Table 7.5).

	Bot 2
Bot 1	**$\text{MCTS}_{\text{combined}}$**
$\text{MCTS}_{\text{alpha}}$	62.7 ±5.5

Table 7.5: Win percentage of $\text{MCTS}_{\text{alpha}}$ against $\text{MCTS}_{\text{alpha}}$ with 1 second per move for 300 games

Tables 7.3 and 7.5 show that $\text{MCTS}_{\text{alpha}}$ performs better than $\text{MCTS}_{\text{combined}}$, meaning that based on these experiments Eq. 6.6 is preferred over Eq. 6.7 for MCTS_{NIM}.

7.4.2 Completed UBFM against the Benchmark Models

To better understand the performance of the MCTS_{NIM} variations, 300 games are also played between completed UBFM and the benchmark models (with 1 second per move). See Table 7.6 for the results together with the confidence bounds of 95%.

	Bot 2	
Bot 1	**$\alpha\beta_{\text{bench}}$**	**$\text{MCTS}_{\text{bench}}$**
Completed UBFM	85.0 ±4.0	87.7 ±3.7

Table 7.6: Win percentage of completed UBFM against the benchmark models with 1 second per move for 300 games

The table shows that the win rate of completed UBFM is lower than the win rates of $\text{MCTS}_{\text{alpha}}$ and $\text{MCTS}_{\text{combined}}$ (see Tables 7.3). Therefore, for these two algorithms, a two-tailed test has been performed with respect to completed UBFM. See Table 7.7 for the p-values.

	Bot 2	
Bot 1	**$\alpha\beta_{\text{bench}}$**	**$\text{MCTS}_{\text{bench}}$**
$\text{MCTS}_{\text{alpha}}$	0.0046	0.0002
$\text{MCTS}_{\text{combined}}$	0.0001	0.1860

Table 7.7: P-values of MCTS_{NIM} against completed UBFM with respect to the benchmark models

$\text{MCTS}_{\text{alpha}}$ performs significantly better against both benchmark models, while $\text{MCTS}_{\text{combined}}$ only performs significantly better against $\alpha\beta_{\text{bench}}$. Which again shows the preference of Eq. 6.6 over Eq. 6.7 for MCTS_{NIM}.

7.4.3 MCTS_{NIM} vs completed UBFM

Since both $\text{MCTS}_{\text{alpha}}$ and $\text{MCTS}_{\text{combined}}$ perform better than completed UBFM against the benchmark models, $\text{MCTS}_{\text{alpha}}$ and $\text{MCTS}_{\text{combined}}$ are also

compared against the completed UBFM algorithm itself, by playing 500 games (250 as player one, 250 as player two, with 1 second per move). The results, together with a confidence bound of 95%, can be seen in Table 7.8.

Bot 1	**Bot 2** **Completed UBFM**
$\mathbf{MCTS_{alpha}}$	62.0 ±4.3
$\mathbf{MCTS_{combined}}$	60.6 ±4.3

Table 7.8: Win percentage of MCTS_{NIM} against completed UBFM with 1 second per move for 500 games

According to the table, the lower bound of the confidence interval for both algorithms is still above 50%, indicating that both algorithms perform better than completed UBFM when the algorithms play against each other. Again, a (slight) preference of Eq. 6.6 over Eq. 6.7 is shown, meaning that Eq. 6.6 is the best performing tested implicit UCT function for MCTS_{NIM} in Breakthrough.

7.5 Additional Variants

Many different selection and play-out strategies have been explored to create the MCTS_{NIM} architecture described in Chapter 6. The variants that have been explored are:

- **Adding Progressive Bias:**
  Addition of Progressive Bias to Eq. 3.4. This resulted in the UCT function as shown in Eq. 7.1.

$$UCT^{IM+PB} = (1-\alpha)\frac{w_i}{n_i} + \alpha v_i^\tau + C \times \sqrt{\frac{ln(n_p)}{n_i}} + \frac{v_i^\tau}{l_i+1}. \tag{7.1}$$

- **Adding GRAVE:**
  Like Progressive Bias, GRAVE has also been added to Eq. 3.4. This resulted in the UCT function as shown in Eq. 7.2.

$$\begin{aligned} UCT^{IM+GRAVE} =& (1-\beta) * ((1-\alpha)\frac{w_i}{n_i} + \alpha v_i^\tau) + \\ & \beta * AMAF_i + C \times \sqrt{\frac{ln(n_p)}{n_i}}. \end{aligned} \tag{7.2}$$

- **Rescaling the estimated values:**
  Since the estimated values by the trained network are unknown, the range of values from the children could be either too narrow or too wide. Therefore, different approaches have been tried to rescale the estimated values before using them in Eq. 3.4. The following rescaling strategies have been tried:

  - Multiply the difference with respect to the mean of v^τ.
  - Use a fixed difference between the minimum and maximum value with respect to the mean
  - Use fixed bounds and scale the values relatively the same to the mean.

- **Addition of Transposition Tables:**
  MCTS cannot recognize transpositions (see Subsection 3.2.1). This means that when using a NN, many (expensive) evaluations must occur multiple times. Therefore, a transposition table has been added to keep track of transpositions in the leaf evaluator, such that Eq. 3.4 could be calculated faster.
- **Different initial MCTS score values:**
  The initial MCTS score (also known as first player urgency) can significantly impact the search algorithm's performance. It decides whether all children should be expanded first, or whether to focus only on the best child, and everything in between. Besides using the MCTS score of the parent, there is also tested with a draw score and infinity for the initial MCTS scores in Eq. 3.4.
- **Different virtual loss values:**
  Like the initial MCTS score, the virtual loss can also significantly impact the search algorithm's performance. It indicates that a node should not be selected too much when many threads already use it. However, when using NN, the estimated value gets backpropagated instead. Therefore, using a virtual loss of 1 could be too harsh since the NN will (almost) never return a 1. This resulted in using a virtual loss of 0 in Eq. 3.4 instead.
- **Different exploration formulas:**
  Both AlphaGo Zero and Polygames changed the exploration part of their UCT function (as seen in Eq. 6.1 and 6.2) [14, 52]. Thus, the following exploration formulas have also been tried in combination with UCT^{IM} (Eq. 3.4).

$$P(v^\tau, \frac{C}{\mathrm{T}})_i, \text{the softmax only}, and \tag{7.3}$$

$$P(v^\tau, \frac{C}{\mathrm{T}})_i \times \sqrt{\frac{n_p}{n_c}}, \text{inspired by Eq. 6.1}, and \tag{7.4}$$

$$P(v^\tau, \frac{C}{\mathrm{T}})_i \times \sqrt{\frac{1}{n_c}}, \text{inspired by Eq. 6.2}, \tag{7.5}$$

  where $P(v^\tau, \frac{C}{\mathrm{T}})_i$ is the softmax value with temperature $\frac{C}{\mathrm{T}}$ of child i based on v_τ.

- **Removing the implicit minimax value:**
  To test that the implicit minimax value did indeed increase the performance of the search algorithm described in Chapter 6, an implementation where no implicit minimax back-ups took place has also been made.
- **Increasing α over time:**
  Instead of decreasing the α value, there is also tested with increasing the α value linearly over time, meaning that the search algorithm focussed more on the implicit minimax values when the number of visits increased (in Eq. 6.7).
- **Use different exploration constant for top K children:**
  The experiments showed that a lower exploration constant C resulted in better results when using Eq. 3.4 in combination with a NN. However, this means that there is almost no exploration. To fix this, a low exploration constant was used for all children, while for the top K children, a new UCT value was calculated with a second exploration constant.
- **Sample uniform from top K children:**
  Instead of using a different exploration constant for the top K children, it is also possible to sample them uniformly. Meaning that there is still some exploration.
- **Multiply with a random constant:**
  By adding randomness to all children, the exploration can be increased even more. This is done by multiplying the UCT values with a random constant.

The following play-out strategies have been tried on $\text{MCTS}_{\text{implicit}}$ algorithm with Eq. 3.4 for selection:

- **Random play-out:**
  As a benchmark, the default play-out is used. Instead of evaluating the moves with a NN, the moves are performed randomly.
- **Random play-out with fixed-depth early termination:**
  Instead of fully completing a game, it is also possible to terminate the play-out at a fixed depth. The backpropagated value is the estimated value by the NN of the game position in which the play-out is terminated.
- **Greedy play-out using a NN:**
  As a second benchmark, a full play-out is performed using the NN's evaluations.
- **Greedy play-out using a NN with fixed-depth early termination:**
  Also, for the second benchmark, the play-out can be terminated at a fixed depth. The backpropagated value is the estimated value by the NN of the game position in which the play-out is terminated.
- **MAST:**
  Since a NN is computationally expensive, it is also possible to perform the play-outs with MAST (see Section 3.3) while using a NN for the selection.

All the mentioned variants did not increase or even decrease the performance of the search algorithm. Since only 1400 games of Breakthrough can be run per day (when both algorithms get 1 second per move), the algorithms have only been run against completed UBFM for 100 games. See appendix C for the performance of the search algorithms. Please note that the parameters for the shown variants have not been tuned.

7.6 Training with $\text{MCTS}_{\text{implicit}}$

In an effort to potentially improve the training process and improve the performance of MCTS_{NIM} and completed UBFM, four new variations of $\text{MCTS}_{\text{implicit}}$ were introduced as an alternative to completed descent (refer to Section 6.5 for further information) to be used during training. These variations were combined with the same setup, as detailed in Section 7.1, and trained for 12 hours, using the tuned parameters from Appendix B and Section 7.3 for Eq. 3.4 and Eq. 6.7, respectively. The performances of MCTS_{NIM} and completed UBFM using these new NNs were evaluated by playing 100 games (with 1 second per move) against the benchmark models and are presented in Table 7.9. For a baseline comparison, the network trained by completed descent (after 12 hours of training) was also included in the table.

Training search algorithm	**Bot 1**	**Bot 2** $\alpha\beta_{\textbf{bench}}$	**Bot 2** $\textbf{MCTS}_{\textbf{bench}}$
Completed descent	MCTS_{NIM}	74.0 ±8.6	97.0
	Completed UBFM	71.0 ±8.9	96.0
Descent based with Eq. 3.4 for selection	MCTS_{NIM}	65.0 ±9.3	86.0 ±6.8
	Completed UBFM	69.0 ±9.1	89.0 ±6.1
UBFM based with Eq. 3.4 for selection	MCTS_{NIM}	66.0 ±9.3	57.0 ±9.7
	Completed UBFM	75.0 ±8.5	78.0 ±8.1
Descent based with Eq. 6.7 for selection	MCTS_{NIM}	1.0	0.0
	Completed UBFM	9.0 ±5.6	1.0
UBFM based with Eq. 6.7 for selection	MCTS_{NIM}	0.0	0.0
	Completed UBFM	1.0	0.0

Table 7.9: Performance of MCTS_{NIM} and completed UBFM against benchmark models after training with different $\text{MCTS}_{\text{implicit}}$ variants

According to the results presented in Table 7.9, the network fails to converge after 12 hours of training when $\text{MCTS}_{\text{implicit}}$ with Eq. 6.7 are used during training. On the other hand, the table shows that when $\text{MCTS}_{\text{implicit}}$ variants with Eq. 3.4 are used during training, the network converges, but its performance is still slightly worse compared to completed descent. Using any of the suggested $\text{MCTS}_{\text{implicit}}$ variants does not improve the performance of either MCTS_{NIM} or completed UBFM.

7.7 Results Proof of Concept

To evaluate the performance of MCTS_{POC}, similar experiments to the ones in Section 7.4 are conducted. The parameters values of Equation 6.4 for MCTS_{POC} are determined through the process described in Appendix B, resulting in $\alpha = 0.8$ and $C = 0$. The win percentages of MCTS_{POC} playing 300 games against the benchmark models and 500 games against completed UBFM, with 1 second per move, can be found in Table 7.10.

	Bot 2		
Bot 1	$\alpha\beta_{\textbf{bench}}$	$\textbf{MCTS}_{\textbf{bench}}$	**Completed UBFM**
$\textbf{MCTS}_{\textbf{POC}}$	87.7 ±3.7	97.3 ±1.8	57.8 ±4.3

Table 7.10: Win percentage of MCTS_{POC} against the benchmark models and completed UBFM with 1 second per move

Similar to MCTS_{NIM}, the lower bound of the confidence interval for MCTS_{POC} is still (slightly) above 50%, indicating that MCTS_{POC} also performs better than completed UBFM when the two algorithms play against each other. As in Subsection 7.4.2, a two-tailed test is conducted to compare the performance of MCTS_{POC} against completed UBFM with respect to the benchmark models. The results are presented in Table 7.11.

	Bot 2	
Bot 1	$\alpha\beta_{\textbf{bench}}$	$\textbf{MCTS}_{\textbf{bench}}$
$\textbf{MCTS}_{\textbf{POC}}$	0.3417	0.0000

Table 7.11: P-values of MCTS_{POC} against completed UBFM with respect to the benchmark models

The table shows no significant difference in performance compared to completed UBFM when playing against $\alpha\beta_{\text{bench}}$. However, a significant difference can be observed when playing against $\text{MCTS}_{\text{bench}}$, indicating that MCTS_{POC} is significantly better than completed UBFM in some cases.

MCTS_{POC} and the two top-performing variants of MCTS_{NIM}, namely $\text{MCTS}_{\text{alpha}}$ and $\text{MCTS}_{\text{combined}}$, have similar performances. However, to obtain a more precise comparison between MCTS_{POC} and these two MCTS_{NIM} variants, a total of 300 games are played between them, whose results are shown in Table 7.12.

	Bot 2	
Bot 1	$\textbf{MCTS}_{\textbf{alpha}}$	$\textbf{MCTS}_{\textbf{combined}}$
$\textbf{MCTS}_{\textbf{POC}}$	46.0 ±5.6	53.3 ±5.6

Table 7.12: Win percentage of MCTS_{POC} against $\text{MCTS}_{\text{alpha}}$ and $\text{MCTS}_{\text{combined}}$ with 1 second per move for 300 games

The results of the three algorithms demonstrate comparable performances, indicating that each algorithm has its potential. Despite the (slightly) better performance of $\text{MCTS}_{\text{alpha}}$, it should not discourage the use of MCTS_{POC} with a policy network, as it still showed (slightly) better performance than $\text{MCTS}_{\text{combined}}$ even when using a heuristic evaluation function. Besides that, MCTS_{POC} achieved a slightly higher win percentage than $\text{MCTS}_{\text{alpha}}$ against $\text{MCTS}_{\text{bench}}$. Thus, MCTS_{POC} is still considered a viable option. Furthermore, the results show the preference for Eq. 6.6 over Eq. 6.7 in MCTS_{NIM} again.

7.8 Lines of Action

In order to validate the results from the experiments in Section 7.4, some experiments using the configurations outlined in Sections 7.1 to 7.3 are repeated on the game of Lines of Action. Lines of Action is selected due to its highly-tactical slow-progression nature, which exhibits many of the properties that pose challenges for MCTS, making it not trivial to make good performing MCTS algorithms [62]. Additionally, Lines of Action is known to prefer the $\alpha\beta$ search algorithm, making it an interesting game to compare the proposed MCTS_{NIM} algorithm against.

The only change made in the setup of the experiments concerns the benchmark models. $\alpha\beta_{\text{bench}}$ uses the Ludii evaluation function for Lines of Action instead (see Subsection 3.2.1), while $\text{MCTS}_{\text{bench}}$ is replaced with $\text{MCTS}_{\text{default}}$ (using 16 threads), as $\text{MCTS}_{\text{bench}}$ includes the Maarten Schadd's Breakthrough leaf evaluation function, which is also the reason MCTS_{POC} is not tested for this game. It is worth mentioning that training on the game of Lines of Action results in 500 games per day, compared to Breakthrough's 1000 games per day, since games take longer to complete. This means that the network was updated fewer times but with more data each iteration. Table 7.13 shows the results from $\text{MCTS}_{\text{alpha}}$, $\text{MCTS}_{\text{combined}}$ and completed UBFM against the benchmark models after playing 300 games.

	Bot 2	
Bot 1	$\alpha\beta_{\textbf{bench}}$	$\textbf{MCTS}_{\textbf{default}}$
$\textbf{MCTS}_{\textbf{alpha}}$	14.5 ±4.0	100
$\textbf{MCTS}_{\textbf{combined}}$	7.5 ±3.0	100
Completed UBFM	7.0 ±2.9	100

Table 7.13: Win percentage of $\text{MCTS}_{\text{alpha}}$, $\text{MCTS}_{\text{combined}}$ and completed UBFM against the benchmark models with 1 second per move for 300 games of Lines of Action

As shown in Table 7.13, it is not surprising to see that the $\alpha\beta$ search algorithm is still the preferred choice over the MCTS_{NIM} variants or completed UBFM in this highly-tactical slow-progression game. To assess whether the performance of $\text{MCTS}_{\text{default}}$ was improved by the MCTS_{NIM} variants and the com-

pleted UBFM algorithms, a total of 300 games were played between $\text{MCTS}_{\text{default}}$ and $\alpha\beta_{\text{bench}}$. The results showed that $\text{MCTS}_{\text{default}}$ lost all 300 games, with a win rate of 0%. This suggests that the MCTS_{NIM} variants and completed UBFM improve the performance concerning $\text{MCTS}_{\text{default}}$. As in Sections 7.4 and 7.7, a two-tailed test is conducted to compare the performance of $\text{MCTS}_{\text{alpha}}$ and $\text{MCTS}_{\text{combined}}$ against completed UBFM with respect to $\alpha\beta_{\text{bench}}$ ($\text{MCTS}_{\text{default}}$ is excluded from this test since the win percentages of all tested algorithms in Table 7.13 are similar). The results are presented in Table 7.14.

	Bot 2
Bot 1	$\alpha\beta_{\textbf{bench}}$
$\textbf{MCTS}_{\textbf{alpha}}$	0.0030
$\textbf{MCTS}_{\textbf{combined}}$	0.8133

Table 7.14: P-values of $\text{MCTS}_{\text{alpha}}$ and $\text{MCTS}_{\text{combined}}$ against completed UBFM with respect to the $\alpha\beta_{\text{bench}}$ in Lines of Action

Table 7.14 shows that only the two-tailed test of $\text{MCTS}_{\text{alpha}}$ against completed UBFM results in a significant difference, highlighting the preference of Eq. 6.6 over Eq. 6.7 in this game as well. Similar to previous experiments, the MCTS_{NIM} variants and completed UBFM are also played against each other to verify these results. Table 7.15 shows the results from the $\text{MCTS}_{\text{alpha}}$ and $\text{MCTS}_{\text{combined}}$ against completed UBFM after playing 500 games.

	Bot 2
Bot 1	**Completed UBFM**
$\textbf{MCTS}_{\textbf{alpha}}$	64.1 ±4.2
$\textbf{MCTS}_{\textbf{combined}}$	54.8 ±4.4

Table 7.15: Win percentage of $\text{MCTS}_{\text{alpha}}$ and $\text{MCTS}_{\text{combined}}$ against completed UBFM with 1 second per move for 500 games of Lines of Action

The table demonstrates that $\text{MCTS}_{\text{alpha}}$ and $\text{MCTS}_{\text{combined}}$ perform better than completed UBFM in Lines of Action, even when no game-specific parameter tuning is applied. Moreover, the lower confidence interval bound of $\text{MCTS}_{\text{alpha}}$ is notably higher than 50%, indicating that this algorithm outperforms the completed UBFM by a significant margin. In addition, $\text{MCTS}_{\text{alpha}}$ even performs better against completed UBFM in Lines of Action than in Breakthrough, confirming the validity of the algorithm's effectiveness in other games.

The results in Table 7.13 and Table 7.15 indicate that $\text{MCTS}_{\text{alpha}}$ also performs better than $\text{MCTS}_{\text{combined}}$, suggesting that Eq. 6.6 is also preferred over Eq. 6.7 in this game. To further demonstrate this preference, 300 games of Lines of Action are played with 1 second per move between them. The results of these games are shown in Table 7.16.

Bot 1	Bot 2 $\mathbf{MCTS_{combined}}$
$\mathbf{MCTS_{alpha}}$	70.7 ±5.2

Table 7.16: Win percentage of $\mathrm{MCTS_{alpha}}$ against $\mathrm{MCTS_{combined}}$ with 1 second per move for 300 games of Lines of Action

The results of $\mathrm{MCTS_{alpha}}$ against $\mathrm{MCTS_{combined}}$ in Table 7.16 show a strong preference for Eq. 6.6 over Eq. 6.7 again. Meaning that also for Lines of Action, Eq. 6.6 is the best performing tested implicit UCT function for $\mathrm{MCTS_{NIM}}$.

Chapter 8

Conclusions and Future Research

This chapter serves as a conclusion to the research presented in this thesis. The problem statement and research questions, introduced in Chapter 1, are revisited and evaluated in Sections 8.1 and 8.2, taking into account the findings presented throughout the thesis. Additionally, Section 8.3 provides directions for potential future research in the form of new research ideas and improvements to the current study.

8.1 Research Questions

This section answers the four proposed research questions based on the results of this thesis.

1. *Can Deep Reinforcement Learning and Monte Carlo Tree Search using implicit Minimax backups be combined?*

The proposed Monte Carlo Tree Search using Network-based Implicit Minimax algorithm combines Monte Carlo Tree Search and Deep Reinforcement Learning. The algorithm incorporates modifications similar to state-of-the-art algorithms like AlphaGo and Polygames, with changes made to the phases of the MCTS architecture. More precisely, the play-out phase was removed, and the estimated values were backpropagated instead, improving the performance. The most significant improvement, however, came from an enhancement to the UCT selection function. Out of the three proposed UCT functions, the UCT function that uses implicit minimax values with a linear decreasing α value performed best. The function prioritizes implicit minimax values during the initial stages and gradually shifts focus towards backpropagated win rates as the number of visits to a node increases. Furthermore, the architecture from the proof of concept demonstrated that combining play-outs guided by a heuristic evaluation can also result in an effective search algorithm.

2. *On which perfect-information games does the model perform well?*

The results of the proposed Monte Carlo Tree Search using Network-based Implicit Minimax algorithms and the proof of concept have shown their effectiveness in the game of Breakthrough, as both have outperformed the tested benchmark models.

To further verify the generalizability of these algorithms, their performance was also tested in the game Lines of Action, which also showed positive results. The successful performance of the proposed algorithms in multiple games highlights their potential for application in other games as well.

3. *Can the enhanced MCTS algorithm improve the descent framework?*

The performed experiments on Breakthrough showed that using the enhanced implicit UCT function (Eq. 6.7) in the descent framework resulted in neural networks that did not converge, whereas using the original implicit UCT algorithm (Eq. 3.4) led to convergence almost as quickly as the networks trained using completed descent.

However, none of the suggested $\text{MCTS}_{\text{implicit}}$ algorithms for training resulted in better-performing networks than the one trained using completed descent after 12 hours of training. This indicates that, with the current parameter configuration, the use of MCTS_{NIM} and $\text{MCTS}_{\text{implicit}}$ did not improve the performance of the descent framework. Therefore, completed descent should still be used.

4. *How does the enhanced MCTS algorithm perform against state-of-the-art algorithms?*

While both playing against the benchmark models as against each other, the proposed Monte Carlo Tree Search using Network-based Implicit Minimax algorithm and the proof of concept performed better than the state-of-the-art algorithm, completed UBFM. The proposed search algorithms achieved win rates of 62.0% and 57.8% in Breakthrough, respectively, making them both promising search algorithms. This has also been validated by playing games in Lines of Action, in which the proposed Monte Carlo Tree Search using Network-based Implicit Minimax algorithm achieved a win rate of 64.1%.

8.2 Problem Statement

After exploring the research questions and finding answers to them, the problem statement can be addressed. The problem statement was formulated as follows:

> *How to improve Monte Carlo Tree Search by using Deep Reinforcement Learning*

The experiments carried out in this thesis have confirmed that combining Monte Carlo Tree Search with Deep Reinforcement Learning is possible by using the enhancements discussed in response to previous research questions. The

results show that these modifications improved the algorithm's performance, as it outperformed benchmark models and, most significantly, a state-of-the-art algorithm in the games of Breakthrough and Lines of Action.

8.3 Future Research

Although the MCTS algorithm using Network-based Implicit Minimax outperformed the completed UBFM algorithm, there is still room for improvement. One area of focus for improvement is the UCT selection function, as changes to this have already led to significant performance gains.

So far, the parameters used by the enhanced UCT function have been optimized using only a simple technique and a limited range of values. Tuning the parameters can significantly improve the performance of the enhanced MCTS search algorithm, making it a priority for further improvement. However, other enhancements to the UCT function could potentially also improve the performance, for example, by decreasing the α value non-linear instead.

In addition to the UCT function, modifications can also be made to other phases of the MCTS algorithm, for example, by backpropagating the implicit minimax values instead. On top of that, the proof of concept has already shown that changes to the play-out phase of the algorithm could also lead to a significant improvement in performance. It would be interesting to see if adding a fast policy network (as used by AlphaGo) would increase its performance since better play-outs could be performed.

Improvements can also be made to the training of the neural networks. Currently, the search algorithms are trained with 1 second per move, meaning that the network aims to predict the value found after 1 second of searching. It would be interesting to see if increasing the time per move results in better-performing search algorithms since the network will learn to make better estimates. Additionally, the neural network could be trained for a more extended period of time, such as the 30 days used by Cohen-Solal.

As attempted, changing the search algorithm during training can also improve the performance of the NN. Due to time constraints, only a limited number of variations were attempted in the current experiments on the game of Breakthrough. However, the experiments did show the importance of the UCT function used during training. By testing the other suggested UCT functions, creating more variations, trying different games, optimizing the weight tuning for both training and usage of the neural networks, and training for a longer time, it is potentially possible to develop a NN that outperforms the one trained by the completed descent method, since (almost) similar results have already been found.

Finally, the conducted experiments have been limited to a small number of games. Further testing is necessary to determine if the results also hold for other games.

Bibliography

[1] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-Time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2):235–256, 2002.

[2] Yngvi Bjornsson and Hilmar Finnsson. Cadiaplayer: A Simulation-based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4–15, 2009.

[3] Stevo Bozinovski. A Self-Learning System using Secondary Reinforcement. *Cybernetics and Systems Research*, pages 397–402, 1982.

[4] Dennis M. Breuker. *Memory versus Search in Games*. PhD thesis, Maastricht University, 1998.

[5] John S. Bridle. Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition. In *Neurocomputing*, pages 227–236. Springer, 1990.

[6] Cameron B. Browne. *Automatic Generation and Evaluation of Recombination Games*. PhD thesis, Queensland University of Technology, 2008.

[7] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo Tree Search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[8] Cameron B. Browne, Dennis J.N.J. Soemers, Éric Piette, Matthew Stephenson, and Walter Crist. Ludii Language Reference. `https://ludii.games/downloads/LudiiLanguageReference.pdf`, 2022.

[9] Cameron B. Browne, Matthew Stephenson, Éric Piette, and Dennis J.N.J. Soemers. A Practical Introduction to the Ludii General Game System. In *Advances in Computer Games*, pages 167–179. Springer, 2019.

[10] Bernd Brügmann. Monte Carlo Go. Technical report, Physics Department, Syracuse University, Syracuse, NY, 1993.

[11] Jay Burmeister and Janet Wiles. The challenge of Go as a domain for ai research: a comparison between Go and chess. In *Proceedings of Third Australian and New Zealand Conference on Intelligent Information Systems. ANZIIS-95*, pages 181–186. IEEE, 1995.

[12] Murray Campbell, A. Joseph Hoane Jr, and Feng-hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[13] Tristan Cazenave. Generalized rapid action value estimation. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[14] Tristan Cazenave, Yen-Chi Chen, Guan-Wei Chen, Shi-Yu Chen, Xian-Dong Chiu, Julien Dehos, Maria Elsa, Qucheng Gong, Hengyuan Hu, Vasil Khalidov, Cheng-Ling Li, Hsin-I Lin, Yu-Jin Lin, Xavier Martinet, Vegard Mella, Jeremy Rapin, Baptiste Roziere, Gabriel Synnaeve, Fabien Teytaud, Olivier Teytaud, Shi-Cheng Ye, Yi-Jun Ye, Shi-Jim Yen, and Sergey Zagoruyko. Polygames: Improved Zero Learning. *ICGA Journal*, 42(4):244–256, 2020.

[15] Guillaume M. J.B. Chaslot, Mark H.M. Winands, H. Jaap van den Herik, Jos W.H.M. Uiterwijk, and Bruno Bouzy. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

[16] Guillaume M.J.B. Chaslot, Mark H.M. Winands, and H. Jaap van den Herik. Parallel monte-carlo tree search. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H.M. Winands, editors, *Computers and Games*, pages 60–71. Springer Berlin Heidelberg, 2008.

[17] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-Column Deep Neural Networks for Image Classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649. IEEE, 2012.

[18] Quentin Cohen-Solal. Learning to play Two-Player Perfect-information Games without Knowledge. *arXiv preprint arXiv:2008.01188*, 2020.

[19] Quentin Cohen-Solal. Completeness of Unbounded Best-First Game Algorithms. *arXiv preprint arXiv:2109.09468*, 2021.

[20] Quentin Cohen-Solal and Tristan Cazenave. Minimax Strikes Back. *arXiv preprint arXiv:2012.10700*, 2020.

[21] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *International Conference on Computers and Games*, pages 72–83. Springer, 2006.

[22] Adriaan D. De Groot. Thought and Choice in Chess. De Gruyter Mouton, 1960.

[23] M. Dienstknecht. Enhancing Monte Carlo Tree Search by Using Deep Learning Techniques in Video Games. Master's thesis, Maastricht University, 2018.

[24] Kunihiko Fukushima and Sei Miyake. Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Visual Pattern Recognition. In *Competition and Cooperation in Neural Nets*, pages 267–285. Springer, 1982.

[25] Sylvain Gelly and David Silver. Combining Online and Offline Knowledge in UCT. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280, 2007.

[26] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, Inc., 2022.

[27] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep Sparse Rectifier Neural Networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

[28] Kerry Handscomb. 8x8 Game Design Competition. *Abstract Games*, (7), 2001.

[29] Ernst A. Heinz. Self-play Experiments in Computer Chess Revisited. In H.J. van den Herik and B. Monien, editors, *Advances in Computer Games 9*, pages 73–92. Maastricht University, 2001.

[30] Donald E. Knuth and Ronald W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[31] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo Planning. In *European Conference on Machine Learning*, pages 282–293. Springer, 2006.

[32] Richard E. Korf and David Maxwell Chickering. Best-First Minimax Search. *Artificial intelligence*, 84(1-2):299–337, 1996.

[33] Marc Lanctot, Mark H.M. Winands, Tom Pepels, and Nathan R. Sturtevant. Monte Carlo Tree Search with Heuristic Evaluations using Implicit Minimax Backups. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[34] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient Backprop. In *Neural Networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[35] Long-Ji Lin. *Reinforcement Learning for Robots using Neural Networks*. Carnegie Mellon University, 1992.

[36] Richard J. Lorentz. Amazons discover Monte-Carlo. In *International Conference on Computers and Games*, pages 13–24. Springer, 2008.

[37] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The Expressive Power of Neural Networks: A View from the Width. *Advances in Neural Information Processing Systems*, 30, 2017.

[38] T Anthony Marsland. A Review of Game-Tree Pruning. *ICCA Journal*, 9(1):3–19, 1986.

[39] Pim Nijssen and Mark H.M. Winands. Playout Search for Monte-Carlo Tree Search in Multi-Player Games. In *Advances in Computer Games*, pages 72–83. Springer, 2011.

[40] Keiron O'Shea and Ryan Nash. An Introduction to Convolutional Neural Networks. *arXiv preprint arXiv:1511.08458*, 2015.

[41] Tom Pepels, Mandy J.W. Tak, Marc Lanctot, and Mark H.M. Winands. Quality-based Rewards for Monte-Carlo Tree Search Simulations. In *ECAI 2014*, pages 705–710. IOS Press, 2014.

[42] Éric Piette, Dennis J.N.J. Soemers, Matthew Stephenson, Chiara F. Sironi, Mark H.M. Winands, and Cameron B. Browne. Ludii–The Ludemic General Game System. In *ECAI 2020*, pages 411–418. IOS Press, 2020.

[43] Michael J.D. Powell. An Efficient Method for finding the Minimum of a Function of Several Variables without calculating Derivatives. *The Computer Journal*, 7(2):155–162, 1964.

[44] Frank Rosenblatt. The Perceptron: a Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, 65(6):386, 1958.

[45] Sid Sackson. *A Gamut of Games*. Random House, New York, NY, USA., 1969.

[46] Arthur L. Samuel. Some Studies in Machine Learning using the Game of Checkers. *IBM Journal of Research and Development*, 11(6):601–617, 1967.

[47] Maarten P.D. Schadd. *Selective Search in Games of Different Complexity.* PhD thesis, Maastricht University, 2011.

[48] Jonathan Schaeffer. The History Heuristic. *ICGA Journal*, 6(3):16–19, 1983.

[49] Robert J Schalkoff. *Artificial Neural Networks*. McGraw-Hill Higher Education, 1997.

[50] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thomas Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.

[51] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A General Reinforcement Learning Algorithm that masters Chess, Shogi, and Go through Self-play. *Science*, 362(6419):1140–1144, 2018.

[52] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Van Den Driessche, Thore Graepel, and Demis Hassabis. Mastering the Game of Go without Human Knowledge. *Nature*, 550(7676):354–359, 2017.

[53] Chiara F. Sironi and Mark H.M. Winands. Comparison of rapid action value estimation variants for general game playing. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2016.

[54] David J. Slate and Lawrence R. Atkin. Chess 4.5—the Northwestern University Chess Program. In *Chess skill in Man and Machine*, pages 82–118. Springer, 1983.

[55] Nathan Sturtevant. An Analysis of UCT in Multi-Player Games. *ICGA Journal*, 31(4):195–208, 2008.

[56] Michael Sullivan. *Statistics. Informed Decisions Using Data.* Pearson, fifth edition, 2017.

[57] J. von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[58] Mark H.M. Winands. Analysis and Implementation of Lines of Action. Master's thesis, Maastricht University, 2000.

[59] Mark H.M. Winands. Monte Carlo Tree Search in Board Games. In Ryohei Nakatsu, Matthias Rauterberg, and Paolo Ciancarini, editors, *Handbook of Digital Games and Entertainment Technologies*, chapter 3, pages 47–76. Springer Singapore, 2017.

[60] Mark H.M. Winands and Yngvi Björnsson. $\alpha\beta$-based Play-Outs in Monte-Carlo Tree Search. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 110–117. IEEE, 2011.

[61] Mark H.M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-Carlo Tree Search Solver. In *International Conference on Computers and Games*, pages 25–36. Springer, 2008.

[62] Mark H.M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte carlo tree search in lines of action. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):239–250, 2010.

[63] Patrick Henry Winston. *Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., 1984.

[64] Albert L. Zobrist. A New Hashing Method with Application for Game Playing. *ICGA Journal*, 13(2):69–73, 1990.

Appendices

Appendix A

Number of Games for Experience Replay

The original descent framework presented in [18] handled experience replay differently as described in Section 5.5. As a result, the number of games had to be tuned, and three values were tested: 1 (no experience replay), 3, and 5. Due to time constraints, the neural network was trained for only 12 hours. The completed UBFM then used this network to play 100 games in Breakthrough against benchmark models with 1 second per move. Even though the confidence intervals are large when only using 100 games, Figure A.1 shows that a network trained on three games resulted in the quickest convergence among the tested values. The same setup has been used, as described in Section 7.1.

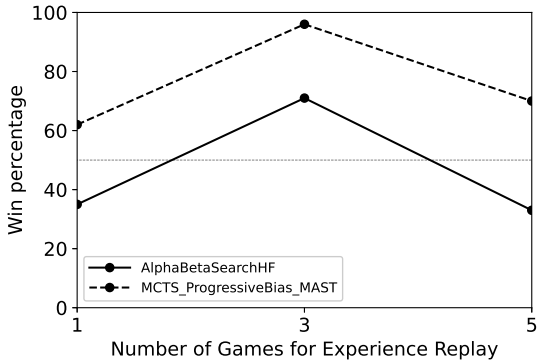


Figure A.1: Performance of Neural Network after training with different number of games in experience replay

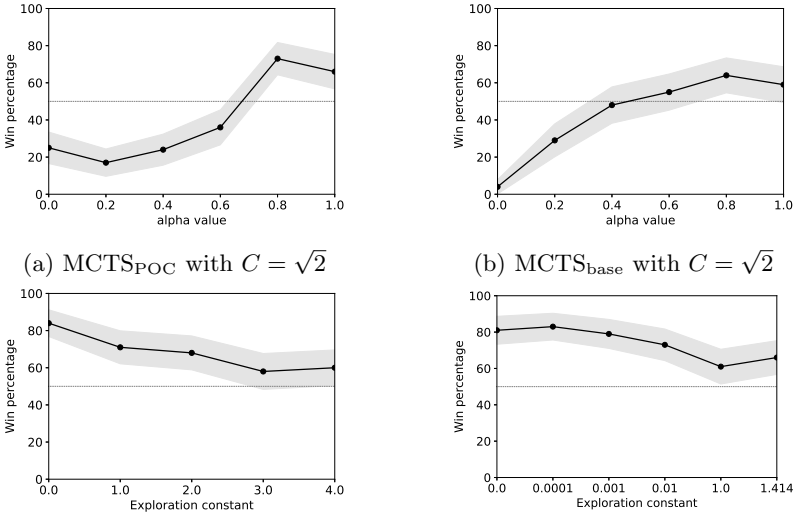
As mentioned in Section 7.6, it is worth mentioning that the performance of the 12 hour version using completed descent performs slightly better on $\text{MCTS}_{\text{bench}}$ than the 36 hour version (as seen in Section 7.4.2). This does

not mean the network is "better". The network performs significantly worse on $\alpha\beta_{\text{bench}}$, showing that the NN did not learn enough game positions to perform well at $\alpha\beta_{\text{bench}}$ yet. On top of that, the experiments have only been tested for a limited amount of games (with large confidence intervals).

Appendix B

Parameter Selection of $\text{MCTS}_{\text{base}}$ and MCTS_{POC}

Both MCTS_{POC} and $\text{MCTS}_{\text{base}}$ use a different UCT formula, as seen in Sections 6.4 and 7.4, respectively. This means that the parameters used need to be tuned separately. Both algorithms only require α and C to be tuned (see Eq. 3.4 and 6.4). The approach used in Section 7.3 has also been applied for selecting these parameters. A total of 100 games has been run against $\alpha\beta_{\text{bench}}$ with 1 second per move. See Figure B.1 for the results.



(a) MCTS_{POC} with $C = \sqrt{2}$

(b) $\text{MCTS}_{\text{base}}$ with $C = \sqrt{2}$

(c) MCTS_{POC} with $\alpha = 0.8$

(d) $\text{MCTS}_{\text{base}}$ with $\alpha = 0.8$

Figure B.1: Trying different α and C values to tune the parameters of MCTS_{POC} and $\text{MCTS}_{\text{base}}$

First, for both MCTS_{POC} and $\text{MCTS}_{\text{base}}$, different α values have been tested with $C = \sqrt{2}$ (see Figures B.1a and B.1b). This resulted in $\alpha = 0.8$ as the best-performing tested value for both algorithms, having a win percentage of 73.0% and 64.0% for MCTS_{POC} and $\text{MCTS}_{\text{base}}$, respectively. By using this α value, different C have been tested as well (see B.1c and B.1d). Figure B.1c shows that for MCTS_{POC} the best exploration constant is $C = 0$ with a win percentage of 84.0%, while Figure B.1d shows that for $\text{MCTS}_{\text{bench}}$ the best exploration constant is $C = 0.0001$ with a win percentage of 83.0%.

It is important to note that the exploration constants for MCTS_{POC} and $\text{MCTS}_{\text{base}}$ may appear to be much lower than that of $\text{MCTS}_{\text{combined}}$, which uses $C = 2$ as seen in Section 7.3. However, this is not a fair comparison as $\text{MCTS}_{\text{combined}}$ uses a softmax function that also scales down the exploration values, resulting in smaller values as well. Thus, when considering the exploration constants of these algorithms, they are more similar than they may appear at first.

Appendix C

Additional Variants

The suggested variants from Section 7.5 have been tested for 100 games (50 as player one, 50 as player two, with 1 second per move) against completed UBFM (see Table C.1). If the requirement of more than four wins is met [29], the confidence bound of 95% is also calculated.

Bot 1	Average	As player 1	As player 2
$\text{MCTS}_{\text{base}}$ with GRAVE	0.0	0.0	0.0
$\text{MCTS}_{\text{base}}$ with with Progressive Bias	11.0 ±6.1	2.0	20.0 ±11.1
$\text{MCTS}_{\text{base}}$ with multiplied difference	24.0 ±8.4	28.0 ±12.4	20.0 ±11.1
$\text{MCTS}_{\text{base}}$ with fixed difference	16.0 ±7.2	20.0 ±11.1	12.0 ±9.0
$\text{MCTS}_{\text{base}}$ with fixed bounds	19.0 ±7.7	22.0 ±11.5	16.0 ±10.2
$\text{MCTS}_{\text{base}}$ with TT	27.0 ±8.7	34.0 ±13.1	20.0 ±11.1
$\text{MCTS}_{\text{base}}$ with $Q_{\text{init}} = 0$	27.0 ±8.7	32.0 ±12.9	22.0 ±11.5
$\text{MCTS}_{\text{base}}$ with $Q_{\text{init}} = \infty$	16.0 ±7.2	20.0 ±11.1	12.0 ±9.0
$\text{MCTS}_{\text{base}}$ with no virtual loss	31.0 ±9.1	40.0 ±13.6	22.0 ±11.5
$\text{MCTS}_{\text{base}}$ with Eq. 7.3	25.0 ±8.5	30.0 ±12.7	20.0 ±11.1
$\text{MCTS}_{\text{base}}$ with Eq. 7.4	33.0 ±9.2	36.0 ±13.3	30.0 ±12.7
$\text{MCTS}_{\text{base}}$ with Eq. 7.5	26.0 ±8.6	30.0 ±12.7	22.0 ±11.5
$\text{MCTS}_{\text{base}}$ without implicit minimax	17.0 ±7.4	26.0 ±12.2	8.0
$\text{MCTS}_{\text{alpha}}$ with an increasing alpha	23.0 ±8.2	30.0 ±12.7	16.0 ±10.2
$\text{MCTS}_{\text{base}}$ with two exploration constants	0.0	0.0	0.0
$\text{MCTS}_{\text{base}}$ with random sample from top K	8.0 ±5.3	8.0	8.0
$\text{MCTS}_{\text{base}}$ with random multiplier	36.0 ±9.4	42.0 ±13.7	30.0 ±12.7
$\text{MCTS}_{\text{base}}$	42.0 ±9.7	58.0 ±13.7	26.0 ±12.2
$\text{MCTS}_{\text{base}}$ with random play-out	10.0 ±5.9	8.0	12.0 ±9.0
$\text{MCTS}_{\text{base}}$ with random play-out and fixed-depth early termination	12.0 ±6.4	10.0 ±8.3	14.0 ±9.6
$\text{MCTS}_{\text{base}}$ with greedy play-out	0.0	0.0	0.0
$\text{MCTS}_{\text{base}}$ with greedy play-out and fixed-depth early termination	6.0 ±4.7	0.0	12.0 ±9.0
$\text{MCTS}_{\text{base}}$ with MAST	7.0 ±5.0	6.0	8.0
MCTS_{POC} with Eq. 6.7	18.0 ±7.5	24.0 ±11.8	12.0 ±9.0

Table C.1: Win percentage of other variations against completed UBFM for 100 games (50 as player one, 50 as player two).

It is worth noting that the parameters of these MCTS_{NIM} variants are not tuned, indicating that it is still possible to achieve higher win percentages. However, it should be mentioned that all these variants in their current configurations did not improve or even decreased the performance of MCTS with Network-based Implicit Minimax.