FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Using Monte Carlo Tree Search to play chess
Student:	Jakub Král
Supervisor:	Ing. Daniel Vašata, Ph.D.
Study program:	Informatics
Branch / specialization:	Artificial Intelligence 2021
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2024/2025

Instructions

Creating a chess bot that can run on a desktop computer and reach a reasonable playing level is a non-trivial task. Traditional models that play using the minimax algorithm are underperforming because of the vast number of positions that need to be explored.
One way to deal with this problem is to use the Monte Carlo Tree Search (MCTS) algorithm to select suitable moves efficiently. Combined with a neural network used to approximate the value function, this is a very successful approach.

Specific assignment points:
1) Introduce the MCTS algorithm and its potential applications in reinforcement learning. Focus on the approaches of its use for playing chess.
2) In a suitable language, implement a model based on MCTS that will primarily learn by playing with itself (self-play).
3) Investigate whether it is possible to improve training by exploiting existing game databases, e.g., in the early stages of training.
4) Assess and discuss the quality of the learned model and, if relevant, test the model against some existing game model (e.g., on a suitable online platform) to verify the game power.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 14 February 2024 in Prague.

Bachelor's thesis

USING MONTE CARLO TREE SEARCH TO PLAY CHESS

Jakub Král

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Ing. Daniel Vašata, Ph.D.
May 16, 2024

Citation of this thesis: Král Jakub. *Using Monte Carlo Tree Search to play chess.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of code listings

I would mainly like to thank my thesis supervisor, Ing. Daniel Vašata, Ph.D, for guidance during the writing of this thesis and useful insights into the practical solution.

Declaration

In Prague on May 16, 2024

Abstract

This thesis deals with the use of the Monte Carlo tree search algorithm and its combination with neural networks and deep reinforcement learning to play chess. The theoretical part of this thesis acquaints the reader with the methods and algorithms of reinforcement learning. In the practical part a model was created such that would train and then play on a standard personal computer. This is solved by using convolutional neural networks, initial supervised learning and then reinforcement learning via self-play. A model that fulfills these requirements was created and runs, but the model plays on a level much lower than was aimed for at the beginning of this work.

Keywords chess, Julia, Monte Carlo tree search, reinforcement learning, deep reinforcement learning, temporal difference learning

Abstrakt

Tato práce se zabývá využitím algoritmu Monte Carlo tree search a jeho kombinace s neuronovými sítěmi a hlubokým posilovaným učením pro hraní šachů. Teoretická část této práce přiblíží čtenáři posilované učení a jeho algoritmy a metodami. V praktické části byl vytvořen model, který se učí a hraje na běžném stolním počítači. Toto je řešeno pomocí konvolučních neuronových sítí, počátečním supervizovaným učením a poté učením pomocí self-play a posilovaného učení. Model, který by tyto cíle splňoval, se podařilo vytvořit, hraje však na úrovni podstatně nižší, než jakou jsem si před začátkem práce představoval.

Klíčová slova šachy, Julia, Monte Carlo tree search, reinforcement learning, deep reinforcement learning, temporal difference learning

List of abbreviations

CNN	Convolutional neural network
DNN	Deep neural network
MCTS	Monte-Carlo Tree Search
MDP	Markov decision process
MSE	Mean squared error
NN	Neural network
RL	Reinforcement learning
TD	Temporal difference
UCB	Upper confidence bound

Introduction

Chess, sometimes also known as the immortal game, is a board game known to everyone and has held its position as the number one played board game in the world for centuries now. Due to its complexity and virtually infinite number of possible different games, chess has also been the point of study for a lot of computer scientists and AI (artificial intelligence) researchers. Today, computers are much stronger chess players than any human, but this strength comes at a great cost.

Engines that play using the minimax algorithm need someone with immense knowledge about the game to help create their evaluation function. Aside from that, the algorithm needs to be very efficient, otherwise the algorithm does not perform very well. In my own experience, for someone like me, it is almost impossible to write a chess engine that is better than humans due to both of these factors.

On the other hand, engines that play using the MCTS (Monte-Carlo tree search) algorithm do not need a chess expert nor do they need to be written very effectively because of the lower number of positions these engines need to look at. The problem here comes with the computational power needed for training models that guide this search.

As I do not have these computational resources, I decided to try and write an algorithm that can play decently even with very limited resources. One of the things that pushed me into this topic was that I tried writing an engine using the minimax algorithm before, but I never got it to a level where it could defeat me in the majority of games. Another inspiration for this attempt is Deepmind's AlphaZero, which uses MCTS, and is one of the best chess engines in the world today.

This thesis is for anyone who wants to create their own chess engine and run it on their computer, does not have thousands of dollars to buy machines for neural net training, or is not both a master at chess and a computer science wizard who can write an extremely effective search.

Goals

The main goal of this thesis is to create a chess engine that can run on a standard computer. The engine should incorporate MCTS and its use in reinforcement learning (RL). The model should learn mostly from self-play, but also explore other techniques and methods, such as learning from a preexisting database at the beginning of its training. After training, the model should be tested against an existing engine like Stockfish to evaluate its playing strength.

Chapter 1
Chess

Everyone is familiar with the game of chess. It has been the most played board game in the world for centuries. Its popularity even rose after the release of Netflix's show The Queen's Gambit in 2020. It is played on a board with 64 squares and 32 pieces of two colors, one for each player, and the objective of the game is to checkmate your opponent.

1.1 History

Although most of us have played a game of chess at some point in our lives and are somewhat familiar with the rules, modern chess as we know it has been developing for centuries to take its current form.

1.1.1 Early history and predecessors

There is no clear origin to the game of chess, but most sources agree that the Indian game chaturanga, which emerged around the sixth century, is the earliest predecessor. It is assumed to be played on an eight by eight board with five different pieces – the Mantri, similar to the modern queen, Ratha, the rook counterpart, Gaja, representing the bishop, Padati, the soldiers that are similar to the pawns, and the Raja, the predecessor of today's king. Except for Gaja, all the other pieces moved the same way as their modern counterparts. Quite similarly to checkmating the king in today's chess, to win the game of chaturanga, one had to capture the Raja.[1, 2]

The game then evolved into the Persian Chatrang (or Shatranj), which added a new piece, the firzān. To win the game, one had to capture the opponent's king or capture all other pieces.

Chatrang then spread all across Asia, taking on many different forms. The most popular version, Chinese chess, had nine files, ten rows, and a river in the middle of the board, slowing the game down relative to its western twin. [3]

Muslims adopted Chatrang as well, so when Islam spread more to Europe, the board game came with it. The main paths through which Chatrang spread to Europe were through Spain, Italy, and Turkey. The game then spread further west to Germany and north to England. Some time around this spreading the game started being called chess. [1, 2]

1.1.2 Evolution of rules

The rules as we know them today are very different from those of the game that spread through Europe a thousand years ago. For example, the ability of the pawns to move two squares on their first move came around the year 1300 but was not widely accepted for another 300 years. Two of the biggest changes to the rules became popular after the year 1475. Until then, there was no queen, but instead a counselor. The counselor could only move diagonally one square. Because a pawn could only be promoted to a counselor after reaching the eighth rank, pawn promotion was not very important. But then the counselor became the queen with its movement as we know it today, becoming the strongest piece on the board. Also, the elephant from chaturanga, which could only move a maximum of two squares, became the bishop with its increased range.

The last changes in the rules, en passant and castling, were known since the fifteenth century but gained general acceptance no sooner than in the eighteenth century. Other minor changes were introduced to the game as late as of nineteenth century until which a pawn could be only promoted to a queen if the queen had been previously captured (in certain parts of Europe). [3]

1.2 Rules

Before every game, the board is set up in the same way. The board is oriented in a way such that the left-bottom corner is black (or dark). The second rank (horizontal row) is filled with pawns and the first rank is filled as follows – rooks in the corners, next to them the knights and next to them the bishops. The queen is placed on the square that matches her color and the king is placed on the last remaining square. The black pieces are set up in the same way on the seventh and eighth ranks. The whole board configuration can be seen in Figure 1.1. [4]

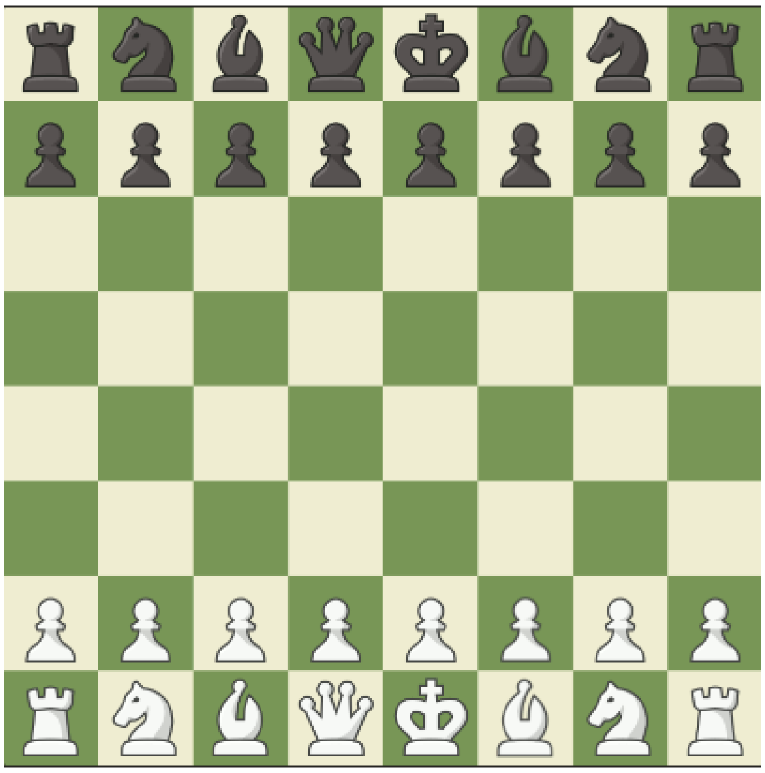

Figure 1.1 Board setup

Source: https://www.chess.com/learn-how-to-play-chess

1.2.1 Piece movement

Each piece has its own unique movement pattern. None of the pieces may move to a square occupied by a piece of the same colour. Pieces can move to a square occupied by a piece of the other color, capturing it and removing it from play. Kings cannot be captured but only given check or be checkmated. Check is when the king is on a square to which a piece of the opposite color can move (the square is attacked by said piece). Checkmate is when there are no legal moves that would remove the king from the check (either by moving the king or by blocking or capturing the attacking piece).

- The bishops can move diagonally any number of squares but cannot jump over other pieces.
- The rooks can move horizontally or vertically any number of squares but cannot not jump over other pieces.
- The queens can move as either a bishop or a rook.
- The knights move two squares horizontally or vertically and then one square perpendicularly to the original direction, but unlike the pieces mentioned above can jump over other pieces.
- The kings can can move to any square adjacent to their own.
- The pawns can move one square forward if there is no piece on that square, or two squared forward if there are no pieces on either of those squares and the pawn has not been moved yet, or one square diagonally forward if there is an opponent's piece.

The pieces can only move in such a way that check to their king does not arise from their movement. [4]

Aside from these basic moves, there are three special moves in chess. The first one is castling. *"When castling, the king moves two squares in the direction of the rook, the rook jumps over the king and lands on the square next to it. You cannot castle:*

- *if the King is in check,*
- *if there is a piece between the Rook and the King,*
- *if the King is in check after castling,*
- *if the square through which the King passes is under attack,*
- *if the King or the Rook has already been moved in the game"* [5].

The two other special moves, promotion and en passant, are done by the pawns. Promotion happens when a pawn reaches the opposite end of the board (eighth or first rank). The pawn must then be exchanged for a different piece of the same colour – a queen, a rook, a knight or a bishop, but not a king.

En passant is a move that can be performed only when an opponent's pawn moves two squares forward (from its original square). If a pawn attacks the square that the opponent's pawn "skipped over", it may move to that square and capture the opponent's pawn as if it moved to that square instead. This move is only available right after the opponent's pawn has moved two squares. [4]

1.2.2 End of game

To win a game of chess, a player must give a checkmate to their opponent. As mentioned above, this happens when the king is in check and has no way to get out of it. The only ways to get out of check are moving the king to another square (not attacked by the opponent), blocking the check with another piece (in case of checks by a rook, bishop, or a queen), or capturing the attacking piece. Another way to win a game of chess is if one of the players resigns or if he runs out of time (in games with time constraints).

Alternatively, the game can end in a draw. *"There are 5 reasons why a chess game may end in a draw:*

- *The position reaches a stalemate where it is one player's turn to move, but his king is NOT in check and yet he does not have any legal move*
- *The players may simply agree to a draw and stop playing*
- *There are not enough pieces on the board to force a checkmate (example: a king and a bishop vs. a king)*
- *A player declares a draw if the same exact position is repeated three times (though not necessarily three times in a row)*
- *Fifty consecutive moves have been played where neither player has moved a pawn or captured a piece"* [6].

1.3 Computers in chess

Today, chess computers are far stronger than any grandmaster has ever been. But this has not been the case for long. The first computer engine was designed by Claude Shannon and Alan Turing around the year 1950, after their collaboration during World War II. During the 1960s and 1970s, algorithms for chess computers were significantly improved, led by John Von Neumann's invention of the minimax algorithm. The other thing allowing for big improvements of chess engines was the improvement of hardware which allowed for deeper and faster minimax searches. [7]

However, computers were not on good enough to compete with chess masters and grandmasters. Chess computers kept becoming stronger, but the biggest moment of change in the history of computer chess came with the DeepBlue chess engine.

1.3.1 Deep Blue

The development of artificial intelligence was in full progress in the 1990s, but there was an unbeaten milestone – defeating a human world champion in chess. This was the goal of the Deep Blue team. Their engine was first tested against the reigning world champion Gary Kasparov in 1996. *"Deep Blue won the first game, which marked the first victory by a computer against a reigning world champion under regular time controls. But Kasparov recovered and won the match 4–2."* [8]

After the engine's loss the team hired more grandmasters to help with the engine, improved endgame databases and created a better evaluation function. After these upgrades, Deep Blue played against Gary Kasparov again in May 1997. This time Kasparov won the first game, but the second game was won by Deep Blue and the next three games were all drawn. Deep Blue won the last game in the end, achieving a 3.5-2.5 victory, becoming the first engine to defeat the world champion. [8]

1.3.2 Stockfish

Stockfish, currently the strongest chess computer in the world, was first created by Tord Romstad, Joona Kiiski, and Marco Costalba from Norway and launched in November 2008. It is a free, open-source chess engine written in C++ that is accessible on almost all operating systems.

Probably the most important part of a chess engine is its evaluation function. The evaluation function of Stockfish has been developed in cooperation with chess grandmasters and has a lot of criteria. Here are some of the most notable criteria contributing to how high the value of a position is:

- *Material advantage gives the player an upper hand over their opponent.*
- *Controlling the center area of the board is a good thing.*
- *Defending and protecting the important pieces give you more weight.*
- *Attacking the opponent's powerful pieces increases your dominance.*
- *Having pawns controlling the center of the board is advantageous.*
- *Having pawns isolated is a bad thing.*
- *If one or both bishops control the diagonal line on the board, you're in a good situation.*
- *Encircling the king with your other pieces is favorable." [9]*

Aside from the evaluation function function Stockfish uses Alpha-Beta search with pruning, which is a variant of the minimax algorithm. Other features contributing to its strength are iterative deepening and bitboards which allow faster and deeper searches than most other engines. Stockfish can run on as many as 512 CPU cores using multithreading and has a transposition table (table of positions reached by a different move order which then does not have to be evaluated again) with a maximum size of 32 terabytes. Aside from the evaluation function, Stockfish uses opening books and endgame databases for improved gameplay. [10]

As the best chess engine in the world for some time, Stockfish has accomplished more than any other chess engine. It has won 14 TCEC (Top Chess Engine Championships) since 2013 and has 7 second places. It also won the vast majority of TCEC Main league seasons in the years 2018-2023 finishing ahead of Leela Chess Zero, Komodo and other engines. [11]

"Stockfish also dominates the TCEC Fischer Random tournament, having won five out of six tournaments so far (as of June 2023)." [11] In Fischer Random pieces in the first/last row are placed randomly at the start of the game, the same for both players while the pawns remain on the second/seventh rank.

1.3.3 Alpha Zero

DeepMind's AlphaZero is not the first but is the best-known chess engine using purely neural nets to play the game. The engine is a successor of AlphaGo Zero, an algorithm that achieved a superhuman level of play in the game of Go, learning purely by self-play. AlphaZero uses the same, but generic algorithm without any domain knowledge except for the rules of the game, making the same algorithm able to play chess, shogi and go as well.

For a long time, engines that played using minimax and a handcrafted evaluation function had been the best at chess with no close competition. There are engines like DeepChess, Giraffe or NeuroChess that use neural nets trained using self-play to evaluate positions in combination with the minimax algorithm. AlphaZero uses a different approach. Instead of minimax, it uses the MCTS algorithm.

Unlike other chess engines, Alpha Zero does not use an evaluation function created in cooperation with professional chess players, but rather uses a deep neural network f_θ with parameters

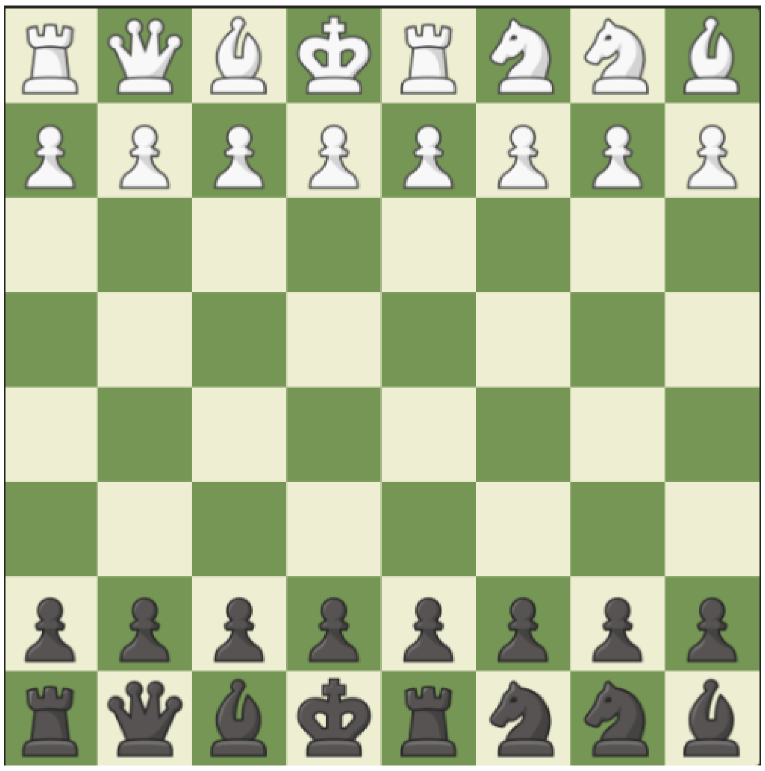

Figure 1.2 Possible Fischer random starting position

Source: https://www.chess.com/terms/chess960

θ to estimate the position values and policies. The value is a simple scalar value and the policy is a probability distribution over all the possible moves (illegal ones as well). This network was trained purely by self-play and its policy and value estimates are then used to guide the MCTS.

Unlike the minimax algorithm used by Stockfish and other engines, MCTS is a general-purpose algorithm that does not have any domain-specific knowledge except for knowing the legal moves and when a game is over. *"Each search consists of a series of simulated games of self-play that traverse a tree from root s_{root} to leaf. Each simulation proceeds by selecting in each state s a move a with low visit count, high move probability and high value (averaged over the leaf states of simulations that selected a from s) according to the current neural network f. The search returns a vector π representing a probability distribution over moves."* [12]

The neural net's parameters θ are trained purely by self-play and reinforcement learning (RL). The moves were chosen by the MCTS algorithm for both black and white players and the game value was determined at the end of the game – either -1, 0, or 1 for a loss, a draw and a win respectively. *"The neural network parameters θ are updated to minimize the error between the predicted outcome v_t and the game outcome z, and to maximize the similarity of the policy vector p_t to the search probabilities π_t. Specifically, the parameters θ are adjusted by gradient descent on a loss function l that sums over mean-squared error and cross-entropy losses respectively,*

$$(p, v) = f_\theta(s), \qquad l = (zv)^2 \pi^T \log p + c||\theta||^2 \tag{1.1}$$

where c is a parameter controlling the level of L2 weight regularisation. The updated parameters are used in subsequent games of self-play." [12]

The input of the NN is a tensor with 119 8×8 planes (2-dimensional vectors). The tensor includes 12 planes for specific piece types and color combinations and two planes for repetitions which are repeated 8 times – for the last 8 reached positions. The other planes are for castling, colour to move, move count and no-progress moves (moves where a pawn is not moved nor is any piece captured) count.

AlphaZero trained on 44 million games split into smaller batches for training. In each position, the MCTS ran for 800 iterations. The training was done on thousands of TPUs (tensor processing units) and took around 9 hours.

The strength of the engine was then evaluated against Stockfish 8, the 2016 TCEC champion, on 100 games with tournament controls of 1 minute per move. AlphaZero won convincingly, winning 25 and drawing 25 games with white while winning 3 and drawing 47 games with black and losing 0 games in total. AlphaZero also searched around 100 times fewer positions per second than Stockfish but concentrated more heavily on promising moves which is an approach more closely resembling human players. [12]

Chapter 2

Algorithms

Every chess engine needs an algorithm to play the game of chess. For decades, the vast majority of top chess engines used the minimax algorithm with handcrafted evaluation functions and its most notable modification – alpha-beta pruning. More recently, the Monte Carlo tree search algorithm in combination with NNs has gotten attention, mainly after the publication of the AlphaZero paper and a few of the top chess engines, AlphaZero included, are using this algorithm. While the algorithms have certain similarities, most notably using tree-like structure for search, their approach to the search is very different.

2.1 Minimax

The minimax algorithm is a popular tree search algorithm used in two-player zero-sum turn-based games with perfect information. The algorithm recursively traverses the nodes of the search tree from the root into the leaf nodes and then backs up the minimax value of the leaves up the tree. Each node represents one state of the game and the edges represent moves. In theory, the leaf nodes are terminal states of the game, but in most games, the state space is too big and such a tree would be infeasible to search. In such cases, some nodes, based on depth, are assigned as terminal and their value is determined by a static evaluation function.

The goal of the algorithm is to find the best move while assuming that the opponent can play perfectly. The two players are called MAX and MIN, where MAX tries to maximize the minimax value and MIN tries to minimize it. First, the algorithm traverses the nodes until leaf nodes are reached and these nodes receive their value based on the evaluation function. These values are then backed up the tree. At each node the player that is to move in that state chooses which move is the best – MAX chooses the move leading to the child node with the highest minimax value and MIN chooses the move leading to the child node with the lowest value – and assigns the value to the node. This propagation is done recursively until the root node is reached, where the move leading to the best value is chosen. [13, 14]

This can be seen in Figure 2.1, where white is the MAX player and black is the MIN player. Focusing on the left-most branch, you can see that values are assigned to the leaf nodes and at the level above MAX chooses the maximum of these values – maximum of -1 and 3 in this case, which is 3 – and assigns the value to the node. In the parent node, the MIN is the one choosing so he chooses the lowest possible value out of 5 and 3 – 3. In the root node, it is MAX's turn, so MAX chooses the move leading to 3, as it is the highest-valued child.

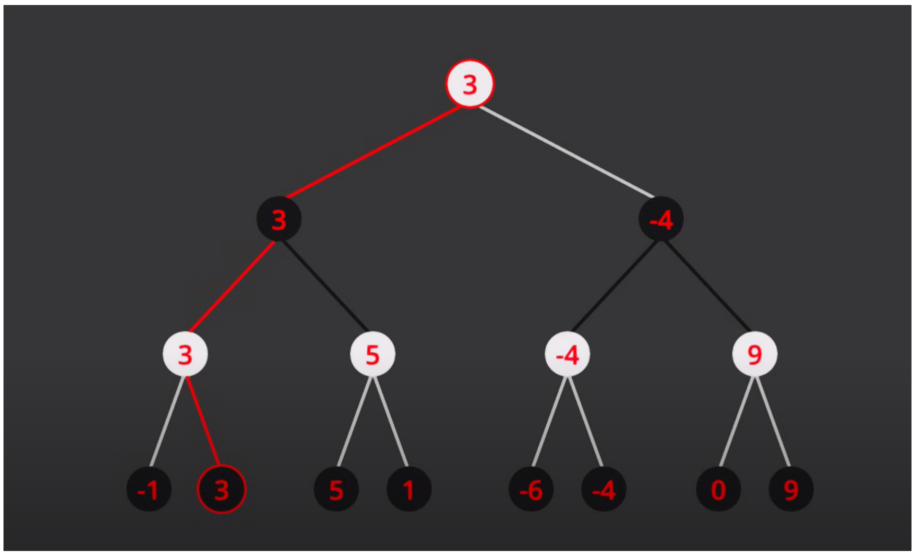

Figure 2.1 Minimax algorithm

Source: https://youtu.be/l-hh51ncgDI?si=UyryiaGS-j70WS4g

```
function minimax(node, depth, maximizingPlayer):
    if depth = 0 or node is a terminal node:
        return the heuristic value of the node
    if maximizingPlayer:
        bestValue = -infinity
        for each child node of node:
            v = minimax(child, depth - 1, FALSE)
            bestValue = max(bestValue, v)
        return bestValue
    else:
        bestValue = +infinity
        for each child node of node:
            v = minimax(child, depth - 1, TRUE)
            bestValue = min(bestValue, v)
        return bestValue
```

Code listing 1 Minimax pseudocode [15]

2.1.1 Alpha-Beta pruning

Alpha-beta pruning is an optimization technique for the minimax algorithm. The algorithm skips over nodes that cannot change the minimax value of the root node. This way the algorithm returns the same result as minimax, but can significantly speed up the process.

There are two additional parameters in the alpha-beta pruning algorithm compared to minimax – alpha (α) and beta (β). α is the highest value that MAX has found so far on the path leading to the node, while β is the lowest value the MIN player has found. The value of α can only be updated in MAX nodes – when value v of a searched child is higher than that of α, α is increased to value v. β is only updated in MIN nodes – when a value v lower than β is return from searching one of the children, β is lowered to v.

When a MAX node is searched and a value v higher than β is returned from one of its children, the rest of the subtree can be pruned and v can be returned immediately. This is because no matter the results from the other children, the value of the node will be at least v and when this value is propagated higher up the tree, it can never be chosen since the parent node is that of MIN, thus the lowest-value move is chosen there.

Pruning in MIN nodes works pretty much the same. When a value v lower than α is received from a child node, the whole subtree can be pruned and v can be returned. This is possible because the value of the searched node will always be lower or equal to v, which is lower than α, so it can never be chosen in the parent MAX node. [13, 16, 17]

```
function alpha_beta_pruning(node, alpha, beta, depth, is_maximiser):
    if the node is a terminal or depth == 0:
        return utility of node

    if is_maximiser == True:
        max_utility = -INF
        for each child_node of node:
            utility = alpha_beta_pruning(child_node, alpha, beta, depth + 1,
            FALSE)
            max_utility = max(utility, max_utility)
            alpha = max(utility, alpha)
            if alpha >= beta:
                break
        return max_utility
    else:
        min_utility = INF
        for each child_node of node:
            utility = alpha_beta_pruning(child_node, alpha, beta, depth + 1,
            TRUE)
            min_utility = min(utility, min_utility)
            beta = min(utility, beta)
            if alpha >= beta:
                break
        return min_utility
```

Code listing 2 Alpha-Beta pruning pseudocode [18]

An example of pruning is shown in Figure 2.2. The biggest pruning can be seen on the right branch from the root node. In this node, the α value is 3 since it is the highest value the root node has received so far (from its left child). The value returned from the left child of this node is -4 which is lower than α. For this reason, the other children (one child in this case) can be pruned.

2.1.2 Optimizations

One of the most effective optimizations for alpha-beta or even minimax algorithms is the use of transposition tables. These tables store positions that have already been evaluated so they don't have to be evaluated again if the same position is reached via a different move order. [13]

The effectiveness of alpha-beta pruning is dependent on the order of moves in which the algorithm searches. Achieving the perfect order of the searched moves is impossible, but a

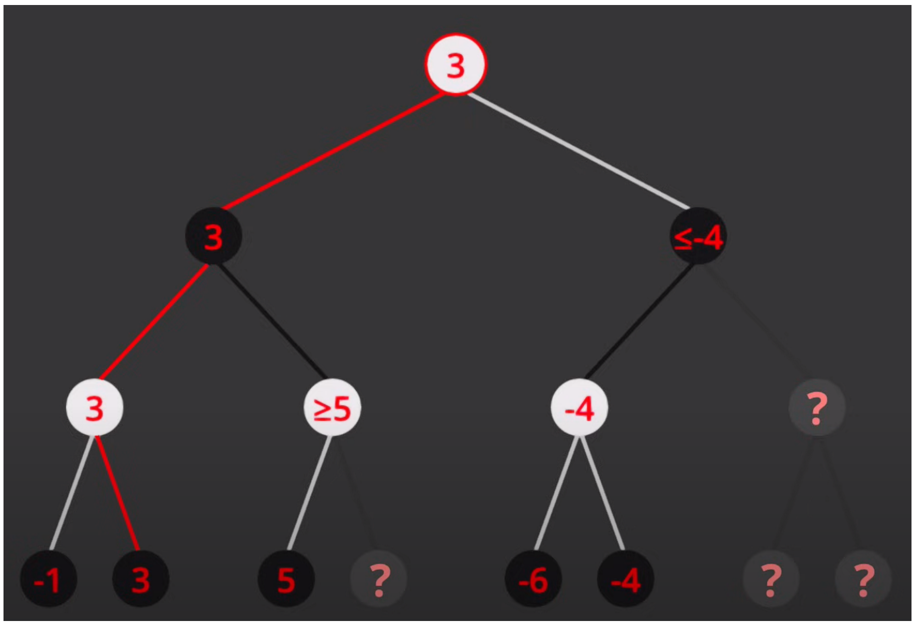

Figure 2.2 Alpha Beta pruning

Source: https://youtu.be/l-hh51ncgDI?si=UyryiaGS-j70WS4g

function that orders based on captures, threats, forward moves, and backward moves in this order, can be very effective. Another way to determine the order in which to search the moves is iterative deepening. *"First, search one ply deep and record the ranking of moves based on their evaluations. Then search one ply deeper, using the previous ranking to inform move ordering; and so on. The increased search time from iterative deepening can be more than made up by better move ordering. The best moves are known as killer moves, and to try them first is called the killer move heuristic."* [13]

2.2 Monte Carlo Tree Search

MCTS is an iterative, best-first search algorithm that builds a tree of all possible future game states. Unlike minimax, MCTS does not require an evaluation function but rather relies on random rollouts and average returns to determine the best move.

The search tree starts with only the root node and then in each iteration expands the tree by adding one or more nodes. Each node represents a game state and has to store at least two other attributes – the number of visits and total reward received after visiting the state. The search of the space state is fully random at first, but with each iteration, the algorithm can more accurately estimate the values of the states and thus choose the more promising ones to search. In each iteration, the algorithm goes through 4 phases – selection, expansion, simulation and backpropagation. [19, 20]

Selection

The first phase begins in the root node. The algorithm chooses one of the node's children based on a selection strategy at every level of the tree until a leaf node (or a node with unexpanded children) is reached. The selection strategy is called a tree policy. There are different strategies, but probably the most popular one is the Upper confidence bound (UCB), with which the

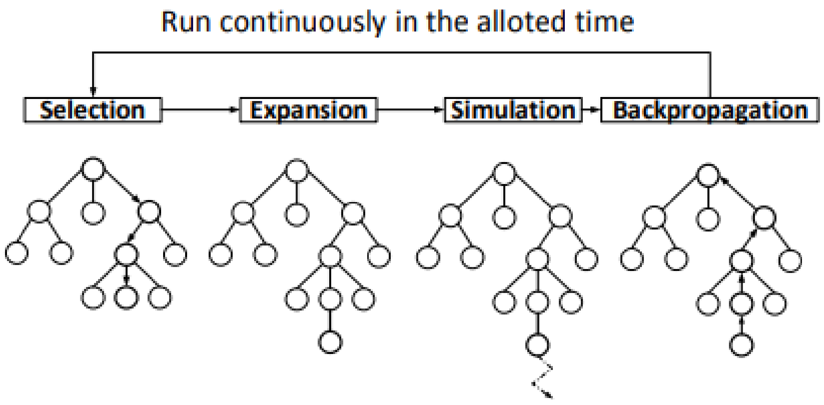

Figure 2.3 Monte Carlo Tree Search

Source: https://link.springer.com/article/10.1007/s10462-022-10228-y

resulting algorithm is called Upper Confidence Bound Applied to Trees (UCT):

$$\mathrm{UCB}(s_i) = v_i + C\sqrt{\frac{\ln N}{n_i}}, \tag{2.1}$$

where v_i is the value (average score) of the child node, N is the number of visits of the parent node, n_i is the number of visits of the child node and C is a constant, typically $\sqrt{2}$ or $\frac{\sqrt{2}}{2}$.

The UCT algorithm balances exploration and exploitation. The child's value v_i promotes exploitation since actions that lead to better results will have higher values, while the rest of the formula promotes exploration by assigning a higher value to the states that have been visited fewer times compared to the total visits of the parent node. Another component of the UCT algorithm is that all children are visited at least once before any child can be visited more times.

Expansion

When a node with unexpanded potential children is reached via the selection process and the node's state is not terminal, the node is expanded. In its most simple form, the expansion means adding one node (of the potential children) to the tree as a child of the expanded node. In some other cases, all the potential children are added.

Simulation

In the third step, random actions (moves) are selected by both players until a terminal state is reached. The terminal state is then evaluated, in chess giving either 1 for a white win, 0 for a draw and -1 for a black win.

Backpropagation

In the last step, each node traversed during the search is updated – the score of the game is added to its total reward and its number of visits is increased by one.

After the maximum number of iterations is reached or when the assigned time runs out, the best move is chosen. The best move is typically chosen as the move leading to the root's child with the highest visit count. [20, 21, 19, 22]

2.2.1 Considerations

The algorithm has multiple advantages compared to the minimax algorithm, the first of them being the absence of a need for any domain-specific knowledge except for the rules of the game since the algorithm does not use any function to evaluate positions but rather a statistical approach to the evaluation by random simulations. Another advantage of the algorithm is its ability to be stopped at any point while still giving the best current estimate. Another, and probably the biggest, advantage of the MCTS algorithm is its smaller tree compared to algorithms like minimax. For a game like chess the number of possible game continuations after just two moves ranges from hundreds to low thousands and increases to millions after just two more moves. MCTS limits this issue since it visits way fewer nodes but focuses on the more promising ones and is able to discard bad moves fairly quickly.

The algorithm has some disadvantages as well. While the number of searched nodes is a lot smaller than the one of minimax, the need to simulate games for each iteration adds up to a long time when the number of iterations is high. Also, the memory might be an issue due to the rapid growth of the game tree after a few iterations. Another big disadvantage of the MCTS algorithm is its unreliability in certain game states. In some states, the game might have a way to a certain victory, but the algorithm might not discover it with a limited number of iterations. The same goes for a certain loss with perfect play of the opponent while their not perfect play leads to a victory. [23, 21]

2.2.2 Alpha-MCTS

As previously mentioned in Section 1.3.3, in DeepMind's AlphaZero chess engine, a kind of MCTS is used to play instead of the traditional minimax algorithm, but with a few important adjustments.

Same as the basic MCTS algorithm it goes through the selection, expansion, and backpropagation, but the simulation step is omitted. Instead of simulating the whole game until the end a DNN is used to evaluate the position of the expanded node and then this value is backpropagated. The NN is also used during selection and expansion.

In expansion, the NN outputs not only the value of the position but a policy as well. This policy is then used in the selection phase, which uses a slightly adjusted UCB formula:

$$\mathrm{UCB}(s_i) = v_i + C\sqrt{\frac{\ln N}{n_i}} \cdot p_i. \tag{2.2}$$

This formula accounts not only for the value v_i of each node s_i but also for the policy value p_i assigned to the move leading to the node's position. [12, 24]

2.2.3 Action reduction

Even with its better search, focusing more on promising moves, MCTS is still very computationally intensive and needs a lot of resources. For that reason, there are numerous modifications to the algorithm that can enhance its performance. One such optimization is action reduction.

In the expansion phase, the MCTS algorithm adds new nodes into the tree for states resulting after performing an action in the game. For many problems, the number of possible actions can be too high. In such a case, the tree is growing sideways, with limited chances for the in-depth inspection of promising branches. The aim of the action reduction methods is to limit this effect by eliminating some of the actions. [21]

"Move Pruning describes the process by which a number of branches of a game tree are removed from consideration (hard pruning) or are de-prioritized from consideration, but may be searched later (soft pruning)." [25] An already mentioned pruning technique is Alpha-Beta pruning used in the minimax algorithm.

Another technique that limits the width growth of the search tree is progressive unpruning. Progressive unpruning reduces the branching factor in the selection phase and then progressively increases it with more searches. When a node is visited T times, most of its children are pruned. The children that are not pruned are the ones with the highest heuristic values. With an increasing number of node visits, the children are progressively unpruned and can be searched. [26]

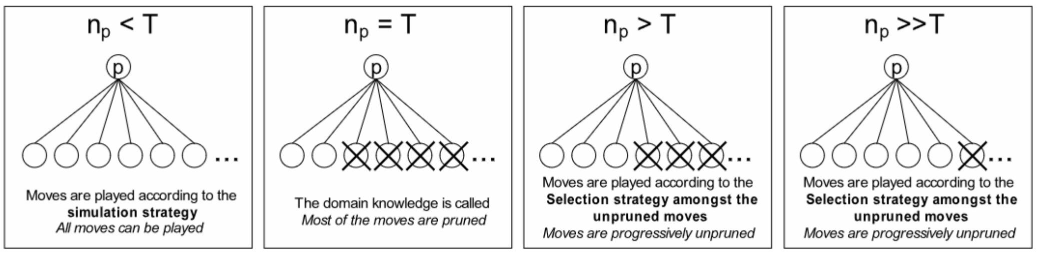
$n_p < T$
Moves are played according to the **simulation strategy** *All moves can be played*
$n_p = T$
The domain knowledge is called *Most of the moves are pruned*
$n_p > T$
Moves are played according to the **Selection strategy amongst the unpruned moves** *Moves are progressively unpruned*
$n_p >> T$
Moves are played according to the **Selection strategy amongst the unpruned moves** *Moves are progressively unpruned*

Figure 2.4 Progressive unpruning

Source: [26]

2.2.4 Virtual expansions

Another modification aiming to speed the algorithm up is Virtual expansions. It aims to spend more time on harder states and less time on easier states which is an ability that human players tend to have, but the MCTS algorithm lacks this ability. The modification introduces virtual expansion that estimates final visitations based on the current partial tree. The second part of this modification is a termination rule that decides when to terminate the search. [27]

The rule terminates simulations based on the tree statistics. If it shows that recent simulations do not change the root node visitations much then the simulations can be stopped. The policy $\pi_k(s)$ of the root node is defined by visitations of each child node at iteration k. *"Let $\Delta s(i, j)$ be the L_1 difference of $\pi_i(s), \pi_j(s)$, namely $\Delta_s(i, j) = ||\pi_i(s) - \pi_j(s)||_1$."* [27] The search loop can then be terminated if the algorithm has searched for at least rN iterations where N is the maximum number of iterations and $r \in (0, 1)$ is a hyper-parameter for determining the ratio of iterations that have to be searched before allowing termination and the condition of the difference $\Delta_s(k, k/2) < \epsilon$ where ϵ is a tolerance hyper-parameter has been fulfilled. Once this gap is small enough it is unnecessary to do more searches.

The problem with this approach is that π_i and π_j cannot be directly compared since the tree is expanded using UCT. This means that when the number of iterations increases the UCT rises as well thus the search focuses more on promising paths. This means that earlier distributions are more exploratory while the latter ones are more exploitative.

To solve this problem, virtual expansions are introduced. At expansion k, normal expansion requires $N - k$ evaluations by the NN for more accurate estimation $Q(s, a)$ for each child at the root node. The virtual expansion also expands $N - k$ times, but rather than the NN assumes that each child's $Q(s, a)$ does not change. After these virtual expansions, we get virtual policy $\hat{\pi}_i$. By doing virtual expansions on both π_i and π_j we get two distributions $\hat{\pi}_i$ and $\hat{\pi}_j$ that can be compared. The rule then becomes $\hat{\Delta}_s(k, k/2) = ||\hat{\pi}_i(s) - \hat{\pi}_j(s)||$ and when it is satisfied the algorithm can stop. [27]

Chapter 3

Reinforcement learning

While the base MCTS algorithm is purely probabilistic and has no heuristic evaluation functions, AlphaZero and other successful chess engines use Convolutional neural networks (CNNs) in combination with the algorithm which is also the approach that is used in the practical part of this thesis. This approach is encouraged by the board state being easily represented as a multi-dimensional tensor which resembles image representation used in NN applications.
For these NNs to work well, training functions need to be established. While supervised learning is quite straightforward and does not need any introduction here, these NNs can also learn via reinforcement learning (RL).

3.1 Overview

Reinforcement learning (RL) as a technique is quite similar to how humans or animals learn. From early development, we learn by interacting with our environment that we can observe via our senses and we try to achieve some goals. We usually do not get a set of actions needed to take to achieve said goal, but rather have to learn by trying different things, discarding those actions that lead nowhere and pursuing those that lead to our goals.

Similarly, RL is a machine learning method where an agent that is being trained is placed inside an environment which he can observe and interact with. The goal of the agent is to maximize the reward acquired in every state of the environment. The agent learns by interacting with the environment and getting reinforcers after every action – a positive reward or a negative penalization. These reinforcers can be both immediate or delayed based on the task.

Unlike in supervised learning, the agent does not know which actions to take but has to try different ones and visit different states using trial-and-error. The agent pursues actions that lead to states with higher rewards. To acquire the maximum reward possible, the agent tries to learn the optimal policy (actions taken are decided based on the policy the agent uses).

While the agent pursues strategies that lead to high rewards (exploits known good strategies), he must also explore new strategies that could lead to even better rewards later on. This is called the exploration-exploitation trade-off and is not present in other fields of machine learning. [28, 29]

3.2 History of reinforcement learning

The field of reinforcement learning was produced in the late 1980s, but before that, its parts were studied separately. The history of reinforcement learning has three main branches which were all pursued separately for some time before being combined into RL. The first thread is learning through trial and error, starting in the psychology of animal learning. The second thread concerns itself with the problem of optimal control and how to solve it using value functions and dynamic programming. The last thread is temporal difference methods.

The idea of trial-and-error learning goes back as far as the 1850s, or more explicitly to the 1894 use of the term by C. L. Morgan to describe his observations of animal behaviour. Probably one of the first to properly describe the essence of trial-and-error learning was Edward Thorndike, who wrote *"Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond."* [30] Thorndike called this the "Law of Effect" and it is the basis of many learning theories of other psychologists.

"Reinforcement" as a term came into the study of animal behaviour later, first appearing in the 1927 English translation of Pavlov's monograph on conditioned reflexes. Reinforcement is the strengthening of a behavioural pattern as a result of an animal receiving a stimulus – reinforcer – in relationship with a response or another stimulus. This was extended to the weakening of a pattern in behaviour, applying when termination of an event changes behaviour. These changes in behaviour must persist even after the stimulus is withdrawn, otherwise, it is not a reinforcer.

The first to apply this concept to computer learning was Alan Turing in 1948 with a "pleasure-pain system" that worked similarly to the Law of Effect: *"When a configuration is reached for which the action is undetermined, a random choice for the missing data is made and the appropriate entry is made in the description, tentatively, and is applied. When a pain stimulus occurs all tentative entries are cancelled, and when a pleasure stimulus occurs they are all made permanent."* [31]

Later works in trial and error learning were for example Claude Shannon's maze-running mouse which used trial and error to reach its goal location inside the maze while the maze itself remembered the successful directions via magnets and relays under the floor. An influential work was that of Donald Michie and his system MENACE developed in the early 1960s to learn to play tic-tac-toe using trial and error. It contained a matchbox for each possible position with a number of beads of different colors for each of the moves possible in a given position. At every turn, a bead was drawn randomly from the corresponding box to determine MENACE's move. After a game, beads were added to or removed from the boxes to reward or punish taken moves based on the result of the game.

Progress on trial-and-error learning was slowed down partly by confusion among researchers who often used supervised learning instead unknowingly. Revival of this technique and its correspondence to reinforcement learning can be credited mainly to Harry Klopf who recognized that main aspects of adaptive behaviour were being lost due to the focus on supervised learning.

As for the second thread, *"The term "optimal control" came into use in the late 1950s to describe the problem of designing a controller to minimize a measure of a dynamical system's behavior over time."* [29] An approach to solve this problem, developed by Richard Bellman, uses the concept of a dynamical system's state and a value function to define a functional equation, now known as the Bellman equation. The class of methods used for solving these problems by solving the equation are dynamic programming. Bellman also introduced Markovian decision processes (MDP) and Ronald Howard devised the iteration method for MDPs which are crucial to modern RL theory.

Dynamic programming suffers "the curse of dimensionality" which means that its computational resources grow exponentially with an increasing number of state variables. It is still better for solving stochastic optimal control problems than other general methods. It has been developed since the 1950s, including extensions to partially observable MDPs, approximation methods, and asynchronous methods.

The connections between dynamic programming and learning were not recognized for a long time, but probably the first to recognize this connection was Paul Werbos in 1977, who proposed an approximate approach to dynamic programming which he called "heuristic dynamic programming". Full integration of dynamic programming with online learning first occurred in the works of Chris Watkins in 1989. His treatment of RL using the MDP formalism has been since widely adopted and developed by many researchers, leading to "neurodynamic programming" – a combination of dynamic programming and neural networks – and "approximative dynamic programming".

The third thread, temporal-difference (TD) learning, is less distinct than the other two but has played a crucial role in the development of RL. The origins of temporal-difference methods lie partly in animal learning psychology, particularly in secondary reinforcers. *"A secondary reinforcer is a stimulus that has been paired with a primary reinforcer such as food or pain and, as a result, has come to take on similar reinforcing properties. "* [29]

The first to implement TD methods as a part of learning was Arthur Samuel in his checkers-playing program, though he did not refer to its connection to animal learning. In 1972, Klopf combined trial-and-error learning with an important component of TD learning. He came up with "generalized reinforcement", which meant that every component views all of its inputs as reinforcement terms – excitatory inputs and rewards and inhibitory inputs as punishments. This greatly differs from what we know as TD learning today, but he linked the idea of TD learning with trial-and-error learning and related it to the empirical database of animal learning psychology on which many researchers built later.

As mentioned at the beginning of this section, all three of these threads were brought together in 1989 by Chris Watkins and his development of Q-learning. There have been many other contributions to the field of RL in more recent years, but they will not be mentioned in this section. [29]

3.3 Markov decision processes

Before diving deeper into RL, it is important to know what MDPs are. MDP is a *"sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards."* [32] MDPs are defined as a tuple $\langle S, A, P, r, \gamma \rangle$, where S is the set of states, A is the set of actions, $P : S \times A \times S \to [0, 1]$ is the state transition probability kernel, where $P(s, a, s')$ is the probability of reaching state s' from state s by performing action a, r is the immediate reward function and γ is a discount factor. [33, 34]

For a problem to be an MDP, all the states must have the Markovian property – the probability of reaching state s' depends only on state s and not on the history of previous states.

To solve an MDP, a simple fixed action sequence would not suffice since the process is stochastic, therefore the need for a policy arises. A policy π specifies which actions to take in each state. Policy $\pi(s)$ specifies action taken in state s. This policy can be both deterministic or probabilistic.

Due to the stochastic nature of the environment, the execution of the same policy from an initial state s may yield a different state history (sequence of reached states) every time. This means that the quality of a policy is measured by the expected utility (sum of rewards) of histories created by said policy. An optimal policy π^* is a policy that yields the highest utility. *"For any MDP there exists an optimal policy π^* that is better than or equal to all other policies."* [33][32]

3.4 Core concepts

Same as MDPs, the reinforcement learning model consists of a few key components:

- a set of environment states S,

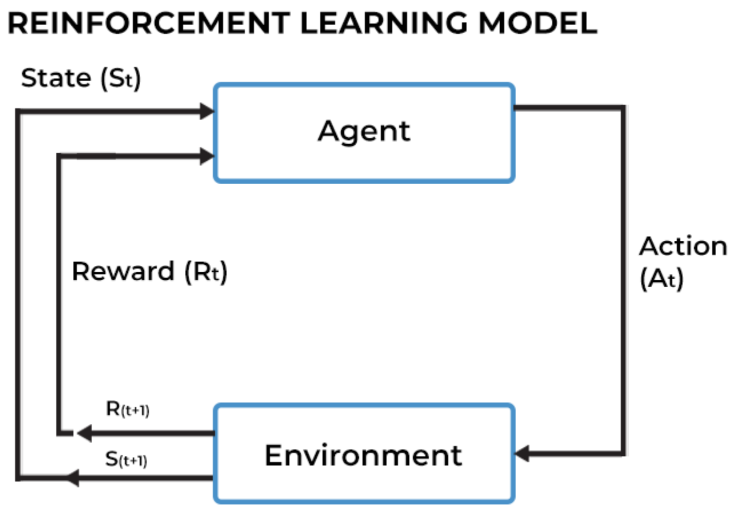

Figure 3.1 Reinforcement learning model

Source:
https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-reinforcement-learning/

- a set of actions A, and
- a set of reinforcement signals r. [35]

Unlike with classical MDPs, in RL, the agent is not given the MDP problem to solve but is placed into the MDP. It does not know the transition model or the reward function but has to learn these by trial and error. In the example of playing chess, the model might get a reward of 1 after checkmating the opponent, the reward of 0 after being checkmated or reward of $\frac{1}{2}$ after drawing the game. Rewards of this type – reward after the whole game is finished – are called sparse rewards. For most tasks immediate rewards are possible to implement – in the example of chess, it could be the capture of a piece. These immediate rewards make learning much easier. [36]

Same as in MDPs, the agent tries to find as good a policy as possible to maximize its long-term cumulative rewards. Another important function is the value function $v(s)$. It is the agent's mapping of states to their estimated long-term reward to be expected after visiting the state s and following the policy π. Value, as a secondary reinforcer, is propagated from states with immediate rewards (like the end of a game) to neutral states, discounting the value using the parameter γ. In combination, the policy is a rule for choosing an action based on the values of the immediate states that the possible actions lead to.

A well-known strategy for this choice is the greedy policy, similar to greedy strategies in other algorithms in that it chooses the action which leads to the state with the greatest value. A problem of this policy is that it does not allow for exploration of the other actions which could potentially lead to better rewards later on. [37]

3.4.1 Exploitation vs Exploration

Probably the most well-known and simplest illustration of this dilemma is the multi-armed bandit problem. In the problem, an agent is in a room with n slot machines with levers (one machine is called a ”one-armed bandit”), each machine having its own distribution of rewards unknown to the agent, and pulls a lever at each step t, to receive the reward from the chosen machine and tries to maximize the cumulative reward.

Each bandit process can be described by a pair of random sequences $(\{X(0), X(1), ...\}, \{R(X(0)), R(X(1)), ...\})$ where $X(n)$ is the state of the machine after it has been operated n times, and $R(X(n))$ is the reward obtained when the machine is operated for the nth time. The reward can be any positive real number, but the simplest and most well known is the Bernoulli bandit. In this case, each arm M_i always produces a reward of either 0 or 1 with some unknown probability μ_i. At each step t, the lever of only one of the machines is pulled, and reward from this machine is added. All the other machines stay in the state they had been in before this step and do not generate any reward.

The agent has to decide which lever to pull at each step – the one that has paid off the most so far, or another one that has not been tested much yet? If the agent always pulls the lever that has performed the best so far, it can easily stick to a suboptimal strategy. On the other hand, if it pulls a different lever and it is worse than the previously best one, it might be wasting time and decreasing the total reward collected. This is the essence of the exploitation vs exploration tradeoff. [38, 32, 39]

As previously discussed, a greedy strategy might not find the optimal policy and can get stuck in a local optimum, leading to suboptimal cumulative reward. There are multiple solutions to the exploration-exploitation dilemma, some of the most notable being the ϵ-greedy strategy, upper confidence bound (UCB), and Thompson sampling.

The ϵ-greedy strategy is quite popular because it is very simple. At each step t, the agent selects the currently best perceived arm with the probability $1 - \epsilon$ and selects a random arm with probability ϵ. The value of ϵ can be fixed the whole time or it can be decreased over time. [40]

The upper confidence bound (UCB) has already been introduced in the section on MCTS. In this problem, the algorithm estimates a confidence interval for each arm in which the value of the arm lies with high confidence, and it then chooses the arm with the highest upper bound of this confidence interval. The confidence interval is based both on the mean value of the arm as well as the number of times the arm has been visited. The second term is proportional to $\sqrt{1/N_i}$ where N_i is the number of times arm i has been chosen, which means that the more times the lever has been pulled, the smaller the exploratory bonus.

The exploratory value in the algorithm is usually multiplied by some function $g(N)$, giving the UCB formula as

$$\mathrm{UCB}(M_i) = \hat{\mu}_i + \frac{g(N)}{\sqrt{N_i}}, \tag{3.1}$$

where $\hat{\mu}_i$ is the mean value estimate and $g(N)$ is some function corresponding to the number of total actions performed. An example function $g(N)$ is $\sqrt{2\log(N)}$, giving the UCB formula

$$\mathrm{UCB}(M_i) = \hat{\mu}_i + \sqrt{\frac{2\log(N)}{N_i}}. \tag{3.2}$$

The algorithm then chooses the arm with the highest value of this function.

Another algorithm is Thompson sampling. This algorithm chooses the arm to pull randomly based on the probability that the arm is, in fact, the optimal one based on the data collected so far. Both UCB and Thompson sampling have far better results than the ϵ-greedy strategy. [39, 32]

3.5 Reinforcement learning algorithms

All reinforcement learning solutions can be split into two main categories – model-free and model-based methods. The two main types of model-free methods are value-based methods and policy gradient methods, the most notable of value-based methods being Q-learning. Model-free and model-based methods can and have been combined to great success.

3.5.1 Value-based methods

Value-based methods in RL aim to create a value function to predict the cumulative reward r of a state s rather than creating a policy π or predicting the best action a. The policy p can be then constructed based on this value function. [28, 41]

One of the simplest and most popular value-based methods is Q-learning. Q-learning can be viewed as not only a model-free RL method but also as a method of asynchronous dynamic programming. In its basic form, Q-learning keeps a table of all state-action combinations and their values $Q(s, a)$. This function estimates the reward r received from performing action a in state s. While the goal is the same – creating the best possible Q function to make decisions leading to the maximum cumulative rewards – there are two ways to achieve this. [28, 42, 41]

The first of them is the use of Bellman's equation for the Q-value function to find the optimal Q-value function $Q^*(s, a)$: *"*

$$Q^*(s,a) = (BQ^*)(s,a), \tag{3.3}$$

where B is the Bellman operator mapping any function $K : S \times A \to \mathbb{R}$ into another function $S \times A \to \mathbb{R}$ and is defined as follows:

$$(BK)(s,a) = \sum_{s' \in S} T(s,a,s')(R(s,a,s') + \gamma \max_{a' \in A} K(s',a')).\text{"} \tag{3.4}$$

[28]

This method is not always sufficient due to high-dimensional (or even continuous) state-space, in which fitted Q-learning is a better solution. In fitted Q-learning, the algorithm starts with a randomly initialized parameters θ. If these parameters θ are those of a DNN the algorithm is then referred to as deep Q-learning.

In this case, instead of trying to solve the Bellman equation, the algorithm learns using some loss function, typically mean squared error (MSE). With the parameters θ, the Q-values are $Q(s, a, \theta)$ and $R(s, a)$ is the reward function for state s and action a obtained from experience. The Q-value function is then trained by trying to minimize the value of MSE L_Q using gradient descent:

$$L_Q = (Q(s,a,\theta) - R(s,a))^2. \tag{3.5}$$

[28, 41]

3.5.2 Policy gradient methods

Policy gradient methods are a subclass of policy-based methods, which include evolution strategies and others. They optimize the policy to maximize the cumulative reward directly. This is done by gradient ascent with respect to the policy parameters. The policy may be stochastic or deterministic.

A stochastic policy $\pi(s)$ is the probability of taking action a in state s. Taking actions according to this policy generates a trajectory that has its cumulative reward. This reward then functions as a replacement for a Q function. Based on these rewards, the policy can be evaluated, giving the evaluation estimates Q^π_ω, and then improved using gradient ascent with respect to the value function estimation. This improvement increases the probability of actions with higher estimated rewards.

To prevent the policy from becoming deterministic, an entropy regularizer is often added to the gradient.

In practice, the $Q^\pi_\omega(s, a)$ is often estimated using Monte Carlo rollouts following policy π_ω. While this technique is unbiased, the main drawback is the need for multiple rollouts for a decent estimate of the return. A more efficient approach would be to use an estimate given by a value-based approach, as in the actor-critic method.

"The deterministic policy gradient is the expected gradient of the action-value function." [43] Unlike the stochastic gradient policy, deterministic policy gradient only integrates over the state space and can therefore be more efficiently estimated. In a discrete action space, the policy $\pi(s)$ can be built using a very direct approach:

$$\pi_{k+1}(s) = \underset{a \in A}{\text{argmax}} Q^{\pi k}(s, a). \tag{3.6}$$

This approach does not work well in continuous action spaces since every step would require a global maximisation. In such cases, the Deep Deterministic Policy Gradient or the actor-critic methods are more suitable. [28]

3.5.3 Actor-critic

In both deterministic and stochastic gradient methods, an estimate of a value function is needed to update the policy via gradient ascent. One way to achieve such estimation and its integration into the process is the actor-critic architecture. It consists of two parts – an actor that refers to the policy, and a critic that estimates the value function. In deep reinforcement learning, both the actor and the critic can be represented by a neural network. *"The actor uses gradients derived from the policy gradient theorem and adjusts the policy parameters w. The critic, parameterized by θ, estimates the approximate value function for the current policy $\pi : Q(s, a; \theta) \approx Q_\pi(s, a)$."* [28]

The critic's role is to estimate the state-value function $V(s, \theta)$ or action-value function $Q(s, a, \theta)$. To achieve good estimations of the value function, the critic usually learns via the TD(0) algorithm where at every iteration the value $Q(s, a; \theta)$ is updated towards a target value:

$$Q(s, a; \theta) = Q(s, a; \theta) + \alpha(r + \gamma Q(s', \pi(s'); \theta) - Q(s, a; \theta)), \tag{3.7}$$

where α is the learning rate, r is the reward and γ is the discount rate. This approach is simple and straightforward, but its drawback is its computational inefficiency and instability because of its use of a pure bootstrapping technique that has a slow backpropagation of rewards.

An example of a technique that tries to mitigate these drawbacks is the Retrace(λ) algorithm. This algorithm uses both on-policy and off-policy data, which leads to policy evaluation with lower bias and higher sample efficiency.

The actor selects actions based on its policy influenced by the critic's evaluations. To leverage off-policy data, the actor adjusts its policy not only based on direct interactions with its environment but also based on feedback from the critic. The updates are done using gradient ascent, trying to improve towards better rewards as estimated by the critic. [28]

3.5.4 Monte Carlo methods

Another family of model-free methods is Monte Carlo (MC) methods. Even though MC methods are usually linked to randomness, in RL they refer to methods based on averaging sample returns. These methods cannot be defined for continuous tasks, but only for episodic ones (all episodes terminate in finite time). The update of policies and value estimates can only be done after an episode completion.

The first use of MC methods is to learn the state-value function for policy π. To estimate the value of a state s, MC methods average out discounted cumulative rewards received after visiting the state s following policy π. With increasing number of episodes, this estimate comes closer to the true value of the state.

Since it is often possible to visit the same state multiple times during one episode, there are two different ways to compute the averages – first-visit and every-visit MC. In first-visit MC, only the reward received from the first visit of state s in the given episode is accounted for in the average. In every-visit MC, rewards from all visits of state s following policy π are considered in the average calculation.

MC methods can be used for estimating action-values as well. This is particularly useful if a model of the environment is not available to the agent since the agent would not know which action to perform to reach highly valued states. The method works essentially the same as for state-values but estimates the return after action a is taken from state s. Both first-visit and every-visit MC can be used here as well. [29]

To approximate the optimal policy π_*, MC methods iteratively improve the action-value function to be closer to the real value of the actions and then change the policy greedily with respect to the action-values – the greedy policy is then deterministic, choosing the action a with the highest estimated return in every state s.

3.5.5 Model-based methods

In model-based methods, the agent relies on a transition model of the environment. The model does not have to be known from the beginning, in which case, the agent has to learn it from experience, or the agent can know it, like in chess where it knows what moves are legal. The agent then plans its actions based on the reward signals from the environment.

The agent often learns a value function, same as in value-based methods, and this function can be replaced by an approximation, which helps in high-dimensional state spaces. *"When a model of the environment is available, planning consists in interacting with the model to recommend an action. In the case of discrete actions, lookahead search is usually done by generating potential trajectories. In the case of a continuous action space, trajectory optimization with a variety of controllers can be used."* [28] [36]

In lookahead search, a decision tree is iteratively built with its root being the current state. Its nodes store received rewards, and it focuses attention on promising trajectories. A difficulty with this approach is balancing exploration and exploitation. To face this challenge, Monte Carlo tree search, as described in Section 2.2, is often implemented in lookahead search.

For continuous action spaces, lookahead search is replaced by trajectory optimization. If the transition model is differentiable, the agent can directly compute an analytic policy gradient by backpropagation of rewards along trajectories. Gaussian processes can be used to learn a probabilistic model of the transitions. It can then use the uncertainty for planning and policy evaluation. This approach does, however, not scale well into high-dimensional spaces. An approach to scaling is the use of DNNs and their generalization abilities. [36, 28]

3.5.6 Combining model-free and model-based approaches

The choice of a model-free or a model-based method is often based primarily on the availability and accuracy of models of the environment and the task structure. While model-based methods may be more accurate than a model-free one in environments with an accurate state representation, they require more computational time because of their need for a search algorithm, which can be a problem in some scenarios where speed is critical or computational resources are limited.

To achieve better results and use the strengths of both of these techniques, it is often feasible to combine the model-free and model-based methods together. An example of this was already mentioned in Section 1.3.3 about AlphaZero. In DeepMind's approach, the model-free part is a deep neural network that estimates the immediate board value as well as the policy without modelling of future states. The model-based part is the adjusted MCTS, which simulates future states which is a kind of model-based planning. [28, 12]

3.6 Temporal difference learning

Temporal difference (TD) learning is one of the core concepts of reinforcement learning. As in other methods, the goal of TD learning is to learn the value function V(s_t). It combines ideas from both Monte Carlo methods and dynamic programming. Like Monte Carlo methods, TD methods learn from experience without the need for the environment's model. Unlike Monte Carlo methods and similarly to dynamic programming methods, TD updates its estimates based on other estimates and does not need to wait for the episode to end. [29]

When the agent transitions from state s_t to state s_{t+1}, it receives a reward r_{t+1}. Based on this reward and the prediction of the new value function $V(s_{t+1})$, the agent can update its estimate of value function $V(s_t)$ according to this formula:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]. \tag{3.8}$$

In the formula, α is the learning rate, which determines how fast the estimations should change and how fast the algorithm will converge – with higher values of α, the convergence is faster, but it has the possibility to fluctuate and not converge at all. γ is the discount factor. It determines how much are future estimates valuable to the prediction – as long as γ is not 1, the states farther down in time will have lower impact on the value estimate $V(s_t)$ that more immediate rewards.

The error for state $V(s_t)$ is defined as:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t). \tag{3.9}$$

If the value function $V(s_t)$ is updated immediately after one step, the algorithm is called TD(0), and it is the simplest TD method. [29, 44]

3.6.1 Control problem

In control problems, the agent learns action-value function $Q(s, a)$ instead of a state-value function $V(s)$. This is done in a very similar way to learning the state-value function, the difference being the addition of actions to the algorithm:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, A_t)\right]. \tag{3.10}$$

"If S_{t+1} is terminal, then Q(S_{t+1}, A_{t+1}) is defined as zero. This rule uses every element of the quintuple of events, (S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}), that make up a transition from one state–action pair to the next. This quintuple gives rise to the name Sarsa for the algorithm." [29]

In an on-policy algorithm based on the Sarsa prediction method, in every state an action a_t is taken based on policy π leading to state s_{t+1}, receiving reward r, and action a_{t+1} is chosen based on the policy. The action-value estimation $Q(s_t, a_t)$ is then updated based on the Formula 3.10 above. To visit every possible state, an ϵ-greedy or ϵ-soft policies are typically used. [29]

An alternative is an off-policy TD control method – Q-learning. The formula is very similar with one difference – the action is not chosen based on the policy but based on which action from state s_{t+1} has the highest estimated value:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, A_t)\right]. \tag{3.11}$$

"In this case, the learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed." [29]

All the mentioned TD methods can be extended from one-step (methods that use only one-step backup) to n-step TD methods. In this generalization, the algorithm accounts not only for rewards and estimations from the immediate next state, but n states ahead.

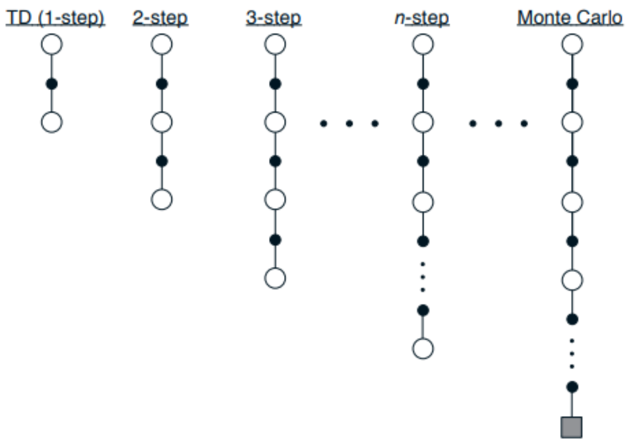

Figure 3.2 n-step backup TD

Source: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf

While the target for TD(0) is

$$r_{t+1} + \gamma V(s_{t+1}),$$

the target for a two-step backup would be

$$r_{t+1} + \gamma r_{t+2} + \gamma^2 V(s_{t+2}),$$

and the target for an n-step backup is

$$r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V(S_{t+n}). \tag{3.12}$$

If these methods go until the end of the episode, they essentially become Monte Carlo methods. [29]

Chapter 4

Practical solution

This chapter describes the practical solution of this thesis. The solution is built around concepts described earlier in the thesis, mainly MCTS, neural networks inspired by AlphaZero architecture, and reinforcement learning and TD learning. For a part of the solution, supervised learning was used as well to speed up the learning process.

4.1 Language and libraries

Even though the most commonly used programming language for machine learning is Python, I chose Julia for the implementation after some initial testing. The decision was made because Python was way too slow for the MCTS algorithm to run on my computer with reasonable results.

Julia is a language created for high performance, as can be seen on the benchmark in Figure 4.1. It is open-source under the MIT license and has over 1000 contributors and over 10,000 open-source packages.

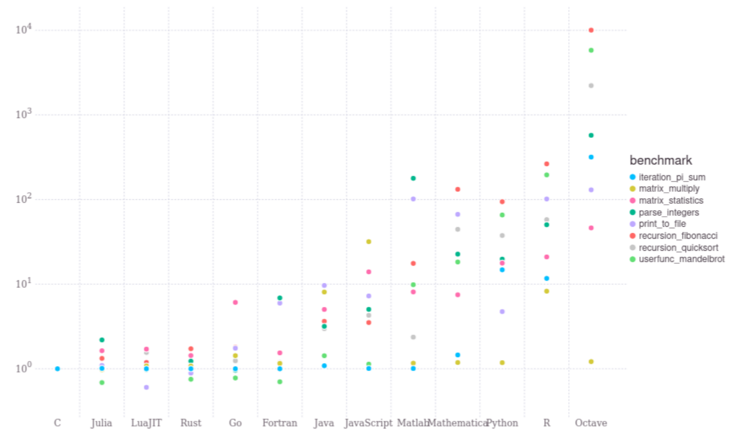

Figure 4.1 Julia speed benchmark

Source: https://julialang.org/benchmarks/

While its most popular use might be numerical and scientific computing, Julia is a general-purpose language with the capabilities to create whole applications. It has packages like GTK.jl for UI applications, Dash.jl for web applications, and many others. It has libraries like Plots.jl and PyPlot.jl for data visualizations. There are libraries like DataFrames.jl and CSV.jl for working with datasets and CSV files.

An important part of Julia's ecosystem is libraries for scientific computing. *"This can be seen in the abundance of scientific tooling written in Julia, such as the state-of-the-art differential equations ecosystem (DifferentialEquations.jl), optimization tools (JuMP.jl and Optim.jl), iterative linear solvers (IterativeSolvers.jl), Fast Fourier transforms (AbstractFFTs.jl), and much more. General purpose simulation frameworks are available for Scientific Machine Learning, Quantum computing and much more."* [45] There is a lot of libraries for machine learning, like MLJ.jl, which provides interface to common ML algorithms like decision trees, clustering, etc. [45]

The most important library for this thesis is the Flux.jl library. Flux is a pure-Julia stack for deep learning. It has GPU support via CUDA.jl package. Its ecosystem includes models for computer vision, graph neural networks, natural language processing and many other domains. Other Julia libraries can be easily incorporated directly into Flux models. [46]

4.2 MCTS implementation

The main component of MCTS is a Node structure. All nodes have their board, parent, and children, which are just an empty vector when the nodes are created. They also have the move leading to them, their probability of being chosen as estimated by the neural network, the visit count, and sum of values from their visits and q_value for faster calculations. The pruned children argument is only used in an adjusted search that uses adaptive action reduction or progressive unpruning. The q_value is set to be very high at the beginning so that the nodes that have not been visited even once are visited before other nodes are visited more times.

```
mutable struct Node
    game::SimpleGame
    args::Dict
    parent::Union{Node, Nothing}
    action_taken::Union{Int, Nothing}
    prior::Float64
    children::Vector{Node}
    visit_count::Integer
    value_sum::Float32
    pruned_children::Vector{Node}
    q_value::Float64

    Node(game, args, parent=nothing, action_taken=nothing, prior=0.0,
    pruned_children=[], q_value=0.0) =
    new(game, args, parent, action_taken, prior, [], 0, 0.0, [], 10 + prior)
end
```

Code listing 3 Node structure

As already mentioned, the four phases of MCTS are selection, expansion, simulation, and backpropagation. Since I used a neural net to estimate values of positions, the simulation step was omitted. My implementation of the search function first determines which player's turn it is so that it can later reverse values for the black player (black wants the result to be -1). Then the search itself runs until it reaches the maximum time or the maximum number of iterations. The maximum time and number of searches differ between training and testing since more search time should generate better results, but it takes too much time for training, so the number of searches is reduced to 200, while in testing the number of searches is 500.

```julia
function search(tree::MCTS)
    root = Node(tree.game, tree.args)
    time0 = time()
    color = sidetomove(tree.game.board)
    searches = 0
    while time() - time0 < tree.args["search_time"]
        && searches < tree.args["num_searches"]
        node = root
        value = 0.0
        while is_fully_expanded(node)
            node = select(node)
        end
        if !(isterminal(node.game))
            policy, value = tree.model.model(board_to_tensor(node.game.board))
            valid_moves = get_valid_moves(node.game.board)
            policy = vec(policy)
            policy = policy .* valid_moves
            policy = policy ./ sum(policy)
            expand(node, policy)
            value = only(value)
        else
            value = game_result(node.game)
        end
        if color == BLACK
            value = -value
        end
        backpropagate(node, value)
        searches += 1
    end
    action_probs = spzeros(Float64, 4096)
    for child in root.children
        action_probs[child.action_taken] = child.visit_count
    end
    action_probs = action_probs ./ sum(action_probs)
    print(searches, " ", time() - time0, " ")
    return action_probs, root.value_sum / root.visit_count
end
```

Code listing 4 MCTS code

In the selection part, the algorithm iterates over all the children, and selects the one with the highest UCB. The UCB formula is similar to the one described in Section 2.2.

The expansion phase is done with respect to the neural net's policy prediction for the expanded node. Since the policy is distributed over all 4096 moves (64×64 starting and target positions), it is not strictly necessary but very useful to zero out all the illegal moves. Another thing the expansion has to take into consideration is promotions. For simplicity, I make the algorithm always promote to a queen since the other options are very rarely beneficial. The values for each legal move in policy are then assigned to their respective children as the *prior* property and are used in UCB calculation as seen in Code listing 5.

```
function get_ucb(child::Node, node::Node)
    return child.q_value + node.args["C"] * sqrt(2*log(node.visit_count + 1)/
    (child.visit_count + 1)) * child.prior
end
```

Code listing 5 UCB calculation code

```
function expand(node::Node, policy)
    for move_idx in eachindex(policy)
        prob = policy[move_idx]
        if prob > 0
            move = int_to_move(move_idx)
            child_state = deepcopy(node.game)
            if ptype(pieceon(node.game.board, from(move))) == PAWN
                if Chess.rank(to(move)) == 8 || Chess.rank(to(move)) == 1
                    move = Move(move.from, move.to, QUEEN)
                end
            end
            domove!(child_state, move)
            child = Node(child_state, node.args, node, move_idx, prob)
            push!(node.children, child)
        end
    end
end
```

Code listing 6 Expansion code

Backpropagation is done in much the same way it is described in the theoretical part – the estimated value of the expanded node is propagated up the tree, and each node's number of visits increases by one. The one difference is that I update the q_value property during backpropagation since it is faster than calculating it in every selection phase.

The search function returns the visit distributions and the value of the root node.

4.3 Neural network

Another very important part of the practical solution is the neural net architecture. The architecture was heavily inspired by the AlphaZero architecture – a deep convolutional neural network with a policy head and a value head.

The core of the neural net are convolutional layers with an increasing number of filters. The final numbers of filters were chosen based on training and test results of different network architectures. In the final architecture, there are five convolutional layers, starting with 32 filters of size 3×3 and ending with a layer with 256 filters of size 5×5. Following each convolutional layer is a batch normalization layer to help with the network training stability and speed. The second, third, and fourth layers have a dropout layer after them as well to increase the network's generalization ability.

After the convolutional layers, a Flatten layer transforms the data into a form which the Dense layers can work with. After this layer, the network splits into the value head and the policy head. The policy head is just one layer that outputs move probabilities distributed over 4096 possible moves (illegal moves as well, the MCTS algorithm and other functions handle that). The value head consists of four additional layers, each lowering the number of neurons and finally giving an output using just one neuron and the tanh activation function to give a prediction between −1 and 1, with the higher the number, the higher the estimated probability of white victory.

```
function ChessNet()
    layers = []

    push!(layers, Conv((3, 3), 18=>32, pad=(1,1), stride=(1,1)))
    push!(layers, relu)

    push!(layers, Conv((3, 3), 32=>64, pad=(1,1), stride=(1,1)))
    push!(layers, BatchNorm(64))
    push!(layers, relu)
    push!(layers, Dropout(0.1))

    push!(layers, Conv((3, 3), 64=>128, pad=(1,1), stride=(1,1)))
    push!(layers, BatchNorm(128))
    push!(layers, relu)
    push!(layers, Dropout(0.1))

    push!(layers, Conv((3, 3), 128=>128, pad=(1,1), stride=(1,1)))
    push!(layers, BatchNorm(128))
    push!(layers, relu)
    push!(layers, Dropout(0.1))

    push!(layers, Conv((5, 5), 128=>256, pad=(0,0), stride=(1,1)))
    push!(layers, BatchNorm(256))
    push!(layers, relu)

    push!(layers, Flux.flatten)

    policy_head = Chain(Dense(4096, 4096), softmax)

    value_head = Chain(Dense(4096, 256, relu), Dense(256, 128, relu),
        Dense(128, 64, relu), Dense(64, 1, tanh))

    combined_heads = CombinedHeads(policy_head, value_head)

    model = Chain(layers..., combined_heads)

    return ChessNet(model)
end
```

Code listing 7 NN architecture

Other model architectures were considered. I tried an architecture with fewer convolutional layers, but the results were very poor. Another idea was the incorporation of dilation into the convolutional layers. In that architecture, the last layer still had 3×3 kernel, but the last two layers had a dilation of 2 (see Code listing 13). These dilations were replaced by a bigger kernel in the final model. The earlier architectures also had padding up until the last convolutional layer, but a pooling layer was added after for faster training. The padding was removed in the final model as well as the pooling layer. This increased the training speed significantly (by around 30 %) and increased the model's perceived performance as well. After the final model was trained, another architecture was tested which had a 5×5 kernel in the last layer as well, but no pooling layer. This NN had an additional dense layer instead of the pooling layer (see Code listing 12). This model did not perform better than the previous one because it had to be trained less (training took over twice as long on the same data), so the final model stayed the same.

As you can see from Code listing 7, the neural net takes in 18-plane tensors as input. The first 12 planes are for piece positions – each plane is for a different piece type and color combination – that are created as one-hot-encoded vectors of size 8×8 where ones are on the tiles the pieces occupy. The next 4 planes are for castling, and the last two planes are for en passant and the side which is to move. You can see the function for board-to-tensor conversion in Code listing 8.

4.4 Learning

The last big part of creating the engine is its training. I have split the training into three parts – supervised learning, RL via self-play without the MCTS algorithm, and RL via self-play using the MCTS algorithm.

Supervised learning was used in the initial stages of learning. The Stockfish engine was used to evaluate positions, find the best move in those positions, and create a small database of around 1,000,000 unique positions. The model then trained on these evaluated positions for around 10 epochs (some worse models for less, more promising models for more). For this, as well as for other training methods, a custom loss function (see Code listing 9) was needed since the neural net outputs a value as well as a policy. This function combines both cross-entropy for policy and MSE for value. The value loss is then multiplied by 40 since it reaches a lot lower values and would be insignificant to the loss otherwise.

The other way of training was self-play without using the MCTS algorithm. In this approach, the model played against itself and stored its predictions of policies and values. The moves to play were chosen based on their estimated probabilities from the policy generated by the model. The values were then adjusted using temporal difference. The policies were adjusted in accordance with basic RL techniques where the moves that led to victory were rewarded and the moves that led to losses were punished (probability of the chosen move was either increased or decreased respectively) as seen in Code listing 10. After these adjustments were made, the model learned on these new values using the loss function in Code listing 9 mentioned in supervised learning.

Training via self-play using MCTS, which was originally to be the main learning method, amounted to almost nothing. The reason for this is that the games were played too slowly, thus making it impossible to generate data fast enough to learn from them. To compare with supervised learning, there the model was able to train on 1,000,000 unique data points in around 2 hours. To generate this amount of data using self-play with MCTS the program would have to run for over 300 hours which makes this method unbearable with the available hardware.

In this case, the model chose moves with the ϵ-greedy strategy where ϵ was 0.1, so the moves were chosen randomly with the a probability 0.1, and with a probability 0.9, the best move according to the tree search was chosen. The values estimated were then adjusted using temporal difference methods similar to the training without MCTS and the target policy was taken as the visit distribution from the root node of the search tree.

4.5 Evaluation

The final model was evaluated by playing against the author, against other trained models, and against other bots online. Sadly, the model did not win any games except for the ones against other models created in this thesis.

```
function create_input_tensors(board::Board)
    tensor = zeros(UInt8, 18, 8, 8)

    # Mapping of pieces to tensor planes
    piece_map = Dict(
        PIECE_WK => 1, PIECE_WQ => 2, PIECE_WR => 3,
        PIECE_WB => 4, PIECE_WN => 5, PIECE_WP => 6,
        PIECE_BK => 7, PIECE_BQ => 8, PIECE_BR => 9,
        PIECE_BB => 10, PIECE_BN => 11, PIECE_BP => 12
    )

    # Fill the tensor for pieces
    for tile = 1:64
        piece = pieceon(board, Square(tile))
        if piece !== EMPTY
        tensor[piece_map[piece], div(tile - 1, 8) + 1,
            (tile - 1) % 8 + 1] = 1
        end
    end

    # Castling rights
    tensor[13, :, :] .= cancastlekingside(board, WHITE) ? 1 : 0
    tensor[14, :, :] .= cancastlequeenside(board, WHITE) ? 1 : 0
    tensor[15, :, :] .= cancastlekingside(board, BLACK) ? 1 : 0
    tensor[16, :, :] .= cancastlequeenside(board, BLACK) ? 1 : 0

    # Player's turn
    tensor[17, :, :] .= sidetomove(board) == WHITE ? 1 : 0

    # En passant square
    if epsquare(board) != SQ_NONE
        tile = square_to_int(tostring(epsquare(board)))
        tensor[18, div(tile - 1, 8) + 1, (tile - 1) % 8 + 1] = 1
    end

    return tensor
end
```

■ **Code listing 8** NN input creation

```
function loss(x, y_moves, y_value)
    y_pred_moves, y_pred_value = model.model(x)
    move_loss = Flux.crossentropy(y_pred_moves, y_moves)
    value_loss = Flux.mse(y_pred_value, y_value)
    return move_loss + 40 * value_loss
end
```

■ **Code listing 9** Custom loss function

```
function change_arrays(states, moves, policies, values, result)
    new_policies = Vector{Vector{Float32}}()
    new_values = copy(values)
    l = length(values)
    gamma = 0.9
    new_values[l] = result
    for i in 1:(l - 1)
        diff = new_values[l - i + 1] - new_values[l - i]
        new_values[l - i] += gamma * diff
    end
    turn = 1
    for i in 1:length(moves)
        policy = Vector(policies[i])
        adj = 1
        if turn == result
            adjustment = 1.02
        elseif turn == -result
            adjustment = 0.98
        else
            adjustment = 0.99
        end

        policy[moves[i]] *= adjustment
        policy /= sum(policy)
        push!(new_policies, policy)

        if turn == 1
            turn = -1
        else
            turn = 1
        end
    end
    new_policies = hcat(new_policies...)
    return new_policies, new_values
end
```

Code listing 10 Temporal difference and policy reinforcement

A game of the final model against the biggest model can be seen under this paragraph. The game was set up until the fourth move so that the models did not play from the initial position. In the game, both the engines made big mistakes, but the final model won at the end. As can be seen from the game (and all the others), the model is quite decent in the early stages of the game, because it has seen most of those positions during its training, but the later the game goes the worse mistakes the model makes.

A game of the final model (white) against the biggest trained model (black):

1. e4 d6 2. d4 c5 3. Nf3 cxd4 4. Nxd4 Nf6 5. Nc3 g6 6. f4 Bg7 7. Be3 O-O 8. Qd2 Nc6 9. Nxc6 bxc6 10. O-O-O Qb6 11. e5 Ng4 12. exd6 Bf5 13. Kb1 exd6 14. h4 Nf6 15. g4 Bd7 16. g5 Qa5 17. h5 Nxh5 18. Na4 Qc7 19. Qa5 Kh8 20. Bg2 Rfb8 21. Rd2 Be8 22. Bc5 Rb7 23. b3 Bf6 24. Bb6 Qe7 25. Bf3 axb6 26. Qxb6 Rab8 27. Qd8 Qd7 28. Nc5 Qc8 29. Ne4 Qd7 30. Nxf6 Nxf6 31. Bh5 h6 32. Bxg6 fxg6 33. gxf6 Kg8 34. f5 gxf5 35. Rd5 Kf7 36. Rc5 Rb4 37. c3 d5 38. Rh3 R8b7 39. Qc8 Qc7 40. Qd8 Qd7 41. Qc8 Qc7 42. Qd8 Bd7 43. Qc8 Be6 44. Rh6 Qd6

```
function training_self_game(model::ChessNet, starting_position::String,
    args::Dict{String, Float64})
    if starting_position == ""
        board = startboard()
    else
        board = fromfen(starting_position)
    end
    game = SimpleGame(board)
    arr = Vector{Tuple{String, SparseVector{Float64}, Float64}}()
    pos_arr = Vector{String}()
    num_moves = 0
    while !(isterminal(game))
        if fen(game.board) in pos_arr
            break
        elseif rand() < 0.1
            probs, _, value = tree_move(model, game, args)
            push!(arr, (fen(game.board), probs, only(value)))
            push!(pos_arr, fen(game.board))
            move = rand(moves(game.board))
            domove!(game, move)
            continue
        end
        probs, move, value = tree_move(model, game, args)
        move = int_to_move(Int(only(move)))
        push!(arr, (fen(game.board), probs, only(value)))
        push!(pos_arr, fen(game.board))
        domove!(game, move)
        if num_moves >= 100
            break
        end
    end
    result = game_result(game)
    println(result)
    return arr, result
end
```

Code listing 11 Self-play game with MCTS

45. Rh3 Qd7 46. Qd8 d4 47. Rd5 d3 48. Rc5 Bc4 49. Qc8 Ra4 50. Re5 Qc7 51. Qd8 Be6 52. Re1 Qe5 53. Rxd3 f4 54. Rg1 c5 55. Rg7#

Another notable game is against the author. The game can be seen below. The author played white and the bot played black. The bot played almost perfectly up until the fourteenth move (see position in Figure 4.2), after which it started making mistakes with basically every move. Other games of the final model can be seen in Section A.1.

1. e4 d5 2. exd5 Nf6 3. Nc3 Nxd5 4. Nxd5 Qxd5 5. Nf3 Qd8 6. d4 g6 7. Bc4 Bg7 8. O-O O-O 9. Be3 c5 10. dxc5 Nc6 11. Rb1 Bg4 12. h3 Bf5 13. Qxd8 Rfxd8 14. a3 Rac8 15. b4 b5 16. Bxb5 Nxb4 17. axb4 a6 18. Bxa6 Rxc5 19. Rfd1 Rc6 20. Rxd8+ Bf8 21. g4 Rc4 22. Bxc4 Bxg4 23. hxg4 Kg7 24. Rxf8 Kxf8 25. Bd4 h5 26. gxh5 gxh5 27. Ra1 f6 28. Ra8+ Kg7 29. b5 Kg6 30. b6 Kh6 31. b7 Kg7 32. b8=Q Kg6 33. Qg8+ Kh6 34. Be3#

The model has also been tested against Stockfish. Here, the model played a hundred games, 50 as white and 50 as black, from different positions. To no surprise, the model lost all the games.

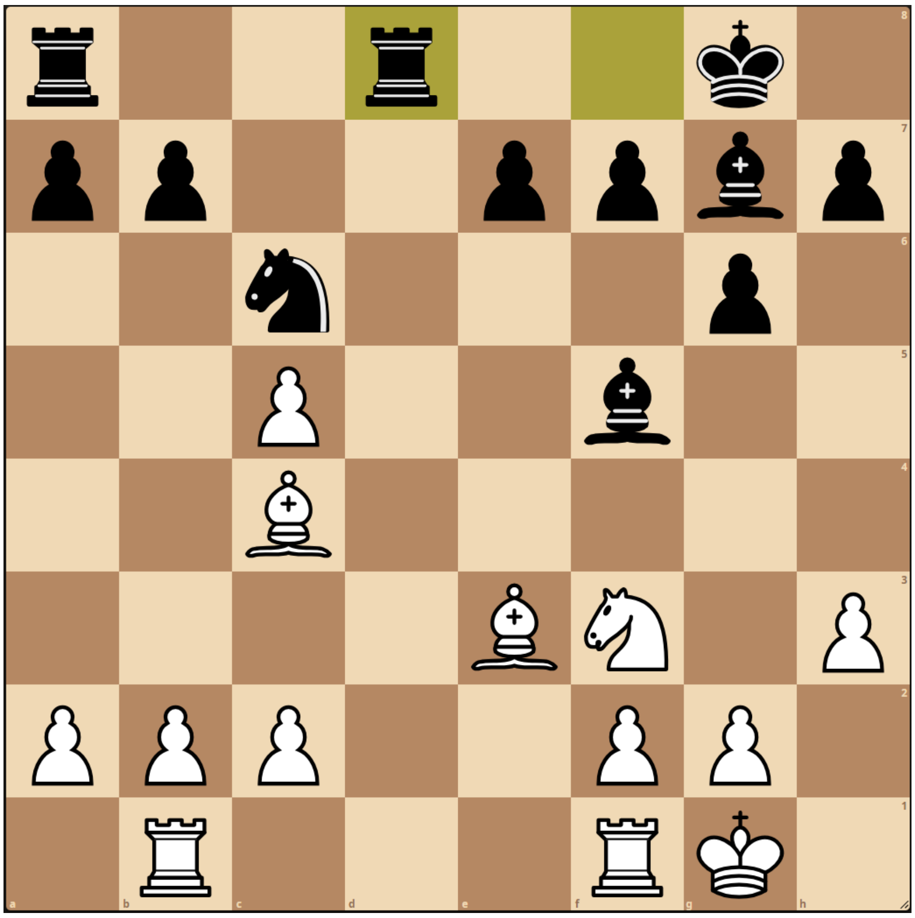

Figure 4.2 Position after 14 moves

Source: Author's game, played on `lichess.org`

A few testing games were also played against an online bot with 250 elo rating on `chess.com`. The created engine was able to draw one of the games but lost the others.

4.6 Discussion

All the best-performing models, including the final model, were trained not only using self-play but also supervised learning. Models that trained purely via self-play were not capable of the same level of play as the ones trained on already preexisting data. This was probably caused by not high enough quality of the training data generated by self-play. This gap in performance could get smaller with longer training time because the games played would become better with time as well as the engine's evaluation, which improves based on the experienced games. This was not possible with the available hardware, though, as the time needed for training this way would be unbearably long, so the inclusion of preexisting database and evaluated positions brought great value to the engine and its playing strength.

As seen from the results of the evaluation, the engine plays at a very low level. There are a few most likely reasons for this – the model architecture and training. Aside from these, a big difference compared to AlphaZero, by which this thesis was inspired, is that the model was trained on 4 Intel i3 CPUs while AlphaZero used thousands of TPUs, which is a huge difference.

[12]

A better architecture would not be difficult to create. First simple upgrade would be increasing the number of layers, probably mainly the convolutional ones, but increasing the number of dense layers for better value estimation might work as well. The number of neurons could be increased as well. This upgrade would slow down the training process significantly and would be inefficient with the author's hardware.

The second upgrade that could be done is removing the loss of information. As seen from Code listing 7, the last convolutional layer has a kernel of size 5×5, but no padding. While the bigger filter captures a bigger area of the board and thus allows the model to see interactions at longer distances, the omitted padding also means that some information around the edges of the board might be lost. A simple solution to this is adding padding of 2 to this layer, which would ensure that all the information is kept, but that comes at a great computational cost. Since the padding would increase the output size of the last layer, there would be two options for how the model would handle it. One is adding a pooling layer, but this approach has the same issue – there is information loss. The other approach, which was tested in the thesis, is adding a dense layer that would take this input and produce a smaller output, but this approach increases the time needed to make a prediction significantly. Due to the much slower training the model saw way fewer positions and thus its playing strength was not better than that of the final model.

Another way to increase the playing strength of the engine is better training. This would probably mean either training for longer or acquiring better hardware so the training time would not have to increase, but the amount of training done would increase significantly. Better training could also be achieved by making better training functions, whether that would mean incorporating regularization, different parameters for TD, or adjustment of policies or other techniques. Another way to increase the quality of the training would be an increased emphasis on critical positions – mainly those where a very beneficial capture is possible, or where a piece is threatened and has to retreat.

Another way to make the engine better would be a more efficient search. The inclusion of more MCTS modifications or better-written code could potentially speed the search up as well, which could lead to deeper searches and thus a better performance.

Chapter 5

Conclusion

The main goal of this thesis was to create a chess engine for playing chess using Monte Carlo tree search that could run and learn on a standard personal computer, which was accomplished. The model was evaluated against other existing chess engines, and the advantage of using a preexisting database of games was tested.

The resulting chess engine was written in the Julia programming language and its Flux library, which encapsulates everything needed for working with artificial neural networks. The chess engine can be run in the command line and can be played against by the user or trained further by additional self-play or via supervised learning.

This work has delved into the algorithms for playing chess, discussing their differences and similarities. It then focused on reinforcement learning and its use in solving the practical goals of this thesis.

The main finding of this work is that, even with many optimizations, the hardware for creating chess engines might be as important as the software. Even with the use of a very fast language like Julia instead of the industry standard Python, the speed of data preparation and training the engine was not sufficient to achieve a considerable playing strength.

Appendix A
Appendix

Some games that did not fit into the main part of the thesis are mentioned here, as well as two of the more successful NN architectures.

A.1 Games

In this game, the final model (black) played against the biggest model (white) and the big model won, but both models started making mistakes as soon as move 4, where both of them missed a possibility to capture the white queen for free.

1. e4 e5 2. d4 exd4 3. Qxd4 Nc6 4. Nc3 Nf6 5. Qd5 Nb4 6. Bg5 Be7 7. Nf3 h6 8. Be3 O-O 9. O-O-O Nc6 10. e5 Ne4 11. Nxe4 d6 12. exd6 cxd6 13. Nf6+ Bxf6 14. g4 Be7 15. Bxa7 Bh4 16. Ba6 b6 17. Nd4 Bxf2 18. Ne6 Nxa7 19. Nxf8 Kxf8 20. Qa5 Qc7 21. g5 Be3+ 22. Kb1 h5 23. Bb5 Bb7 24. Rd4 Bc8 25. Rdd1 Bb7 26. Rd4 Bc8 27. Bf1 Bd7 28. g6 Ke8 29. Rh4 f5 30. Ka1 Bb5 31. Be2 Bd7 32. Bf1 Bb5 33. Be2 Bd7 34. Rh3 b5 35. a3 Rc8 36. a4 Rb8 37. Bd3 Qb7 38. Qc7 Bf4 39. Bxf5 Qf3 40. Qxd6 Be6 41. c4 Bd7 42. Be4 Be6 43. cxb5 Bd7 44. Qb6 Bf5 45. Qb7 Bd7 46. Qxa7 Rc8 47. b6 Qxh3 48. Qa8 Qxh2 49. Rf1 Qf2 50. Rd1 Qh2 51. Rf1 Qf2 52. Rd1 Qh2 53. b4 Qf2 54. Rg1 Qh2 55. Rf1 Qf2 56. Rd1 Kd8 57. b7 Qh2 58. Rd2 Qf2 59. b8=Q Bxd2 60. Qaa7 Be3 61. Qaa8 Bd4+ 62. Kb1 Qb2# *

Game between the created engine (black) and an online bot with 250 elo rating (white) The game was not finished, but the online bot had a big advantage at the end and the game shows the kinds of mistakes the bot makes.

1. d4 d5 2. c4 dxc4 3. Nf3 Nf6 4. e3 e6 5. Bxc4 Bd6 6. Bxe6 fxe6 7. O-O Nbd7 8. g4 b6 9. Kh1 Bb7 10. d5 Nxd5 11. e4 c6 12. g5 N5f6 13. Qxd6 Nxe4 14. Qxe6+ Qe7 15. Qxe7+ Kxe7 16. Nc3 Nxc3 17. bxc3 Nf6 18. gxf6+ Kxf6 19. Rg1 Rad8 20. Bg5+ Kg6 21. Be3+ Kf7 22. Ne5+ Kg8 23. Kg2 Rf8 24. Rh1 c5+ 25. f3 Rd8 26. Kg1 Rd2 27. Bxd2 Bxf3 28. Nxf3 h6 29. Bf4 *

Game of the biggest model (white) against the final model (black). Both models played decently until the twentieth move.

1. e4 d5 2. exd5 Nf6 3. Nf3 Nxd5 4. d4 e6 5. c4 Nf6 6. Nc3 c5 7. d5 exd5 8. cxd5 Be7 9. Bd3 O-O 10. O-O Bg4 11. h3 Bh5 12. g4 Bg6 13. Re1 Nbd7 14. d6 Bxd6 15. Ng5 Qb6 16. Bc4 a6 17. Nf3 h6 18. Nh4 Kh8 19. Bd2 Qxb2 20. Rb1 Qb6 21. Nf3 Qb2 22. Bxh6 gxh6 23. Qd5 Qb4 24. Qxd6 Qxc3 25. Qb6 Qb4 26. Bxa6 Nd5 27. Rbc1 Nc3 28. Rxc3 bxa6 29. Rcc1 Rfc8 30. Kh1 Nb8 31. h4 Qa4 32. Qc6 Qxa2 33. Qd5 Nc6 34. g5 h5 35. Qd6 Kg8 36. Re6 Bc2 37. Ree1 Bd1 38. Rexd1 a5 39. Qxc5 Qc4 40. Qb6 Rab8 41. Re1 Ne7 42. Qh6 Rb2 43. Ra1 Nd5 44. Rxa5 Qc3 45. Rf1 Qc2 46. Kh2 Re8 47. Kh1 Rc8 48. Kh2 Rc5 49. Kg3 Rc4 50. g6 fxg6 51. Ra4 Rxa4 52. Nd2 Rab4 53. Qxh5 Kf7 54. Qh7+ Ke8 55. Qh8+ Kd7 56. h5 Kc6 57. f4

Kc7 58. Qh7+ Kb6 59. Qh8 Kc5 60. Qh7 Kb5 61. Qb7+ Ka5 62. Qa8+ Kb6 63. Qh8 Kc5 64. Qh7 Kb5 65. Qb7+ Ka5 66. Qa8+ Kb6 67. Qg8 Kc5 68. Qxg6 Rb1 69. h6 Nf6 70. Qg8 Nh5+ 71. Kh3 Rd1 72. Qg6 Re1 73. Rf3 Qc1 74. Rd3 Qb2 75. Kh2 Re5 76. Rf3 Qa2 77. Rh3 Rf5 78. Kh1 Qc2 79. Nf3 Rc4 80. h7 Qc1+ 81. Kh2 Qf1 82. Nd4 Ng3 83. Ne6+ Kc6 84. h8=Q Qf2 #

A.2 Previous NN architectures

Here are the architectures of the biggest NN created, and then the NN with dilations in the last convolutional layer.

```
function ChessNet()
    layers = []

    push!(layers, Conv((3, 3), 18=>32, pad=(1,1), stride=(1,1)))
    push!(layers, relu)

    push!(layers, Conv((3, 3), 32=>64, pad=(1,1), stride=(1,1)))
    push!(layers, BatchNorm(64))
    push!(layers, relu)

    push!(layers, Conv((3, 3), 64=>128, pad=(1,1), stride=(1,1)))
    push!(layers, BatchNorm(128))
    push!(layers, relu)

    push!(layers, Conv((5, 5), 128=>128, pad=(2,2), stride=(1,1)))
    push!(layers, BatchNorm(128))
    push!(layers, relu)

    push!(layers, Flux.flatten)

    push!(layers, Dense(8192, 4096, relu))


    policy_head = Chain(Dense(4096, 4096), softmax)

    value_head = Chain(Dense(4096, 256, relu), Dense(256, 128, relu),
        Dense(128, 64, relu), Dense(64, 1, tanh))

    combined_heads = CombinedHeads(policy_head, value_head)

    model = Chain(layers..., combined_heads)

    return ChessNet(model)
end
```

Code listing 12 Big NN architecture

```julia
function ChessNet()
    layers = []

    push!(layers, Conv((3, 3), 18=>32, pad=(1,1), stride=(1,1)))
    push!(layers, relu)

    push!(layers, Conv((3, 3), 32=>64, pad=(1,1), stride=(1,1)))
    push!(layers, BatchNorm(64))
    push!(layers, relu)
    push!(layers, Dropout(0.1))

    push!(layers, Conv((3, 3), 64=>128, pad=(1,1), stride=(1,1)))
    push!(layers, BatchNorm(128))
    push!(layers, relu)
    push!(layers, Dropout(0.1))

    push!(layers, Conv((3, 3), 128=>128, pad=(2,2),
        stride=(1,1), dilation=(2,2)))
    push!(layers, BatchNorm(128))
    push!(layers, relu)

    push!(layers, Conv((3, 3), 128=>256, pad=(2,2),
        stride=(1,1), dilation=(2,2)))
    push!(layers, BatchNorm(256))
    push!(layers, relu)

    push!(layers, Flux.flatten)

    policy_head = Chain(Dense(4096, 4096), softmax)

    value_head = Chain(Dense(4096, 256, relu), Dense(256, 128, relu),
        Dense(128, 64, relu), Dense(64, 1, tanh))

    combined_heads = CombinedHeads(policy_head, value_head)

    model = Chain(layers..., combined_heads)

    return ChessNet(model)
end
```

Code listing 13 NN architecture with dilations

Bibliography

1. ALFARSI, Haroun. *History and Origins of Chess: From India to Persia and Europe - Profolus — profolus.com* [https://www.profolus.com/topics/history-origins-of-chess-game/]. [N.d.]. [Accessed 04-03-2024].
2. MURRAY, H. J. R. *A History Of Chess*. Clarendon Press, Oxford, 1913.
3. *Chess - History — britannica.com* [https://www.britannica.com/topic/chess/History]. [N.d.]. [Accessed 04-03-2024].
4. *fide.com* [https://www.fide.com/FIDE/handbook/LawsOfChess.pdf]. [N.d.]. [Accessed 29-02-2024].
5. *The Rules of Chess — sakkpalota.hu* [https://www.sakkpalota.hu/index.php/en/chess/rules]. [N.d.]. [Accessed 02-03-2024].
6. CHESS.COM TEAM. *How to Play Chess: Learn the Rules & 7 Steps To Get You Started — chess.com* [https://www.chess.com/learn-how-to-play-chess]. [N.d.]. [Accessed 29-02-2024].
7. MISTREAVER. *History Of Chess Computer Engines - Chessentials — chessentials.com* [https://chessentials.com/history-of-chess-computer-engines/]. 2019. [Accessed 15-03-2024].
8. *Deep Blue | IBM — ibm.com* [https://www.ibm.com/history/deep-blue]. [N.d.]. [Accessed 15-03-2024].
9. HERCULES, Andrew. *How Does Stockfish Work? The Engine's Blueprint Simply Explained - Maroon Chess — maroonchess.com* [https://maroonchess.com/how-does-stockfish-work/]. [N.d.]. [Accessed 16-03-2024].
10. EDITORIAL STAFF. *Stockfish Chess Engine: The Ultimate Guide — chessjournal.com* [https://www.chessjournal.com/stockfish/]. [N.d.]. [Accessed 16-03-2024].
11. *Stockfish - Chess Engines — chess.com* [https://www.chess.com/terms/stockfish-chess-engine]. [N.d.]. [Accessed 16-03-2024].
12. SILVER, David; HUBERT, Thomas; SCHRITTWIESER, Julian; ANTONOGLOU, Ioannis; LAI, Matthew; GUEZ, Arthur; LANCTOT, Marc; SIFRE, Laurent; KUMARAN, Dharshan; GRAEPEL, Thore; LILLICRAP, Timothy; SIMONYAN, Karen; HASSABIS, Demis. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. Available from arXiv: 1712.01815 [cs.AI].
13. NORVIG, Peter; RUSSEL, Stuart. *Artificial intelligence: a modern appraoch, fourth edition*. Pearson Education, 2020.

14. CAMPBELL, Murray S.; MARSLAND, T.A. A comparison of minimax tree search algorithms. *Artificial Intelligence*. 1983, vol. 20, no. 4, pp. 347–367. ISSN 0004-3702. Available from DOI: `https://doi.org/10.1016/0004-3702(83)90001-2`.

15. BANSAL, Vasu. *Min-Max Algorithm in Artificial Intelligence - Scaler Topics — scaler.com* [`https://www.scaler.com/topics/artificial-intelligence-tutorial/min-max-algorithm/`]. 2023. [Accessed 12-05-2024].

16. PEARL, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Publishing Company, 1984. Addison-Wesley series in artificial intelligence. ISBN 9780201055948. Available also from: `https://books.google.cz/books?id=0XtQAAAAMAAJ`.

17. GEORGE T. HEINEMAN Gary Pollice, Stanley Selkow. *Algorithms in a Nutshell*. O'Reilly Media, Inc., 2008.

18. CHAKRABARTI, Purnadip. *Alpha Beta pruning - Scaler Topics — scaler.com* [`https://www.scaler.com/topics/artificial-intelligence-tutorial/alpha-beta-pruning/`]. 2023. [Accessed 12-05-2024].

19. CHASLOT, Guillaume; WINANDS, Mark; HERIK, H.; UITERWIJK, Jos; BOUZY, Bruno. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*. 2008, vol. 04, pp. 343–357. Available from DOI: `10.1142/S1793005708001094`.

20. BROWNE, Cameron B.; POWLEY, Edward; WHITEHOUSE, Daniel; LUCAS, Simon M.; COWLING, Peter I.; ROHLFSHAGEN, Philipp; TAVENER, Stephen; PEREZ, Diego; SAMOTHRAKIS, Spyridon; COLTON, Simon. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*. 2012, vol. 4, no. 1, pp. 1–43. Available from DOI: `10.1109/TCIAIG.2012.2186810`.

21. ŚWIECHOWSKI, Maciej; GODLEWSKI, Konrad; SAWICKI, Bartosz; MAŃDZIUK, Jacek. Monte Carlo Tree Search: a review of recent modifications and applications. *Artificial Intelligence Review*. 2022, vol. 56, no. 3, pp. 2497–2562. ISSN 1573-7462. Available from DOI: `10.1007/s10462-022-10228-y`.

22. KEMMERLING, Marco; LÜTTICKE, Daniel; SCHMITT, Robert H. Beyond games: a systematic review of neural Monte Carlo tree search applications. *Applied Intelligence*. 2023, vol. 54, no. 1, pp. 1020–1046. ISSN 1573-7497. Available from DOI: `10.1007/s10489-023-05240-w`.

23. RADKE, Parag. *Monte Carlo Tree Search: A Guide — builtin.com* [`https://builtin.com/machine-learning/monte-carlo-tree-search`]. 2023. [Accessed 02-04-2024].

24. VARTY, Josh. *Alpha Zero And Monte Carlo Tree Search — joshvarty.github.io* [`https://joshvarty.github.io/AlphaZero/`]. [N.d.]. [Accessed 06-04-2024].

25. SEPHTON, Nick; COWLING, Peter I.; POWLEY, Edward; SLAVEN, Nicholas H. Heuristic move pruning in Monte Carlo Tree Search for the strategic card game Lords of War. In: *2014 IEEE Conference on Computational Intelligence and Games*. 2014, pp. 1–7. Available from DOI: `10.1109/CIG.2014.6932892`.

26. CHASLOT, Guillaume; WINANDS, Mark; HERIK, H.; UITERWIJK, Jos; BOUZY, Bruno. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*. 2008, vol. 04, pp. 343–357. Available from DOI: `10.1142/S1793005708001094`.

27. YE, Weirui; ABBEEL, Pieter; GAO, Yang. *Spending Thinking Time Wisely: Accelerating MCTS with Virtual Expansions*. 2022. Available from arXiv: `2210.12628 [cs.AI]`.

28. FRANÇOIS-LAVET, Vincent; HENDERSON, Peter; ISLAM, Riashat; BELLEMARE, Marc G.; PINEAU, Joelle. An Introduction to Deep Reinforcement Learning. *Foundations and Trends® in Machine Learning*. 2018, vol. 11, no. 3–4, pp. 219–354. ISSN 1935-8245. Available from DOI: `10.1561/2200000071`.

29. RICHARD S. SUTTON, Andrew G. Barto. *Reinforcement learning: an introduction.* The MIT Press, 2015.

30. THORNDIKE, Edward. *Animal intelligence: experimental studies.* Macmillan Press, 1911.

31. TURING, Alan. *Intelligent machinery.* 1948. Tech. rep.

32. PETER NORVIG, Stuart Russel. *Artificial intelligence: a modern appraoch, fourth edition.* Pearson Education, 2020.

33. AHILAN, Sanjeevan. *A Succinct Summary of Reinforcement Learning.* 2023. Available from arXiv: `2301.01379 [cs.AI]`.

34. BUFFET, Olivier; PIETQUIN, Olivier; WENG, Paul. *Reinforcement Learning.* 2020. Available from arXiv: `2005.14419 [cs.LG]`.

35. LESLIEPACKKAELBLING MichaelL.Littman, AndrewW.Moore. Reinforcement Learning: A Survey. *JournalofArti cialIntelligenceResearch 4 (1996).* 1996.

36. NORVIG, Peter; RUSSEL, Stuart. *Artificial intelligence: a modern appraoch, fourth edition.* Pearson Education, 2020.

37. GALLISTEL, C. R. Reinforcement Learning. *Journal of Cognitive Neuroscience.* 1999.

38. MAHAJAN, Aditya; TENEKETZIS, Demosthenis. Multi-Armed Bandit Problems. In: 2007, pp. 121–151. ISBN 978-0-387-27892-6. Available from DOI: `10.1007/978-0-387-49819-5_6`.

39. LI, Yuxi. *Deep Reinforcement Learning.* 2018. Available from arXiv: `1810.06339 [cs.LG]`.

40. KULESHOV, Volodymyr; PRECUP, Doina. *Algorithms for multi-armed bandit problems.* 2014. Available from arXiv: `1402.6028 [cs.AI]`.

41. JAEGER, Bernhard; GEIGER, Andreas. *An Invitation to Deep Reinforcement Learning.* 2023. Available from arXiv: `2312.08365 [cs.LG]`.

42. WATKINS, Christopher; DAYAN, Peter. Technical Note: Q-Learning. *Machine Learning.* 1992, vol. 8, pp. 279–292. Available from DOI: `10.1007/BF00992698`.

43. LI, Yuxi. *Deep Reinforcement Learning: An Overview.* 2018. Available from arXiv: `1701.07274 [cs.LG]`.

44. KONEN, Wolfgang. *Reinforcement Learning for Board Games: The Temporal Difference Algorithm.* 2015. Tech. rep. Cologne University of Applied Sciences.

45. BEZANSON, Jeff; KARPINSKI, Stefan; ET AL. *The Julia Programming Language — julialang.org* [`https://julialang.org/`]. [N.d.]. [Accessed 29-04-2024].

46. PAL, Avik; SHIDHAR, Ayush; ET AL. *Flux.jl — fluxml.ai* [`https://fluxml.ai/`]. [N.d.]. [Accessed 29-04-2024].

List of appendices

`README.md`	instructions on how to run everything
`models`	
`self_play_model.jld2`	model trained purely via self-play
`final_model.jld2`	the best-performing model
`dilation_model.jld2`	model with dilations instead of bigger kernel
`big_model.jld2`	big model without loss of information
`src`	
`board_class.jl`	implementation of function working with he board
`mcts.jl`	MCTS implementation
`moves.jl`	implementation of moves functions
`self_play.jl`	implementation of self-play training
`test.jl`	implementation of testing functions
`data_reader.jl`	file with functions for manipulating lichess database
`model.jl`	NN implementation
`supervised_training.jl`	implementation of supervised learning
`data`	
`common_games.txt`	file with most common positions after 4 moves
`evaluated_positions.bin`	positions evaluated by Stockfish