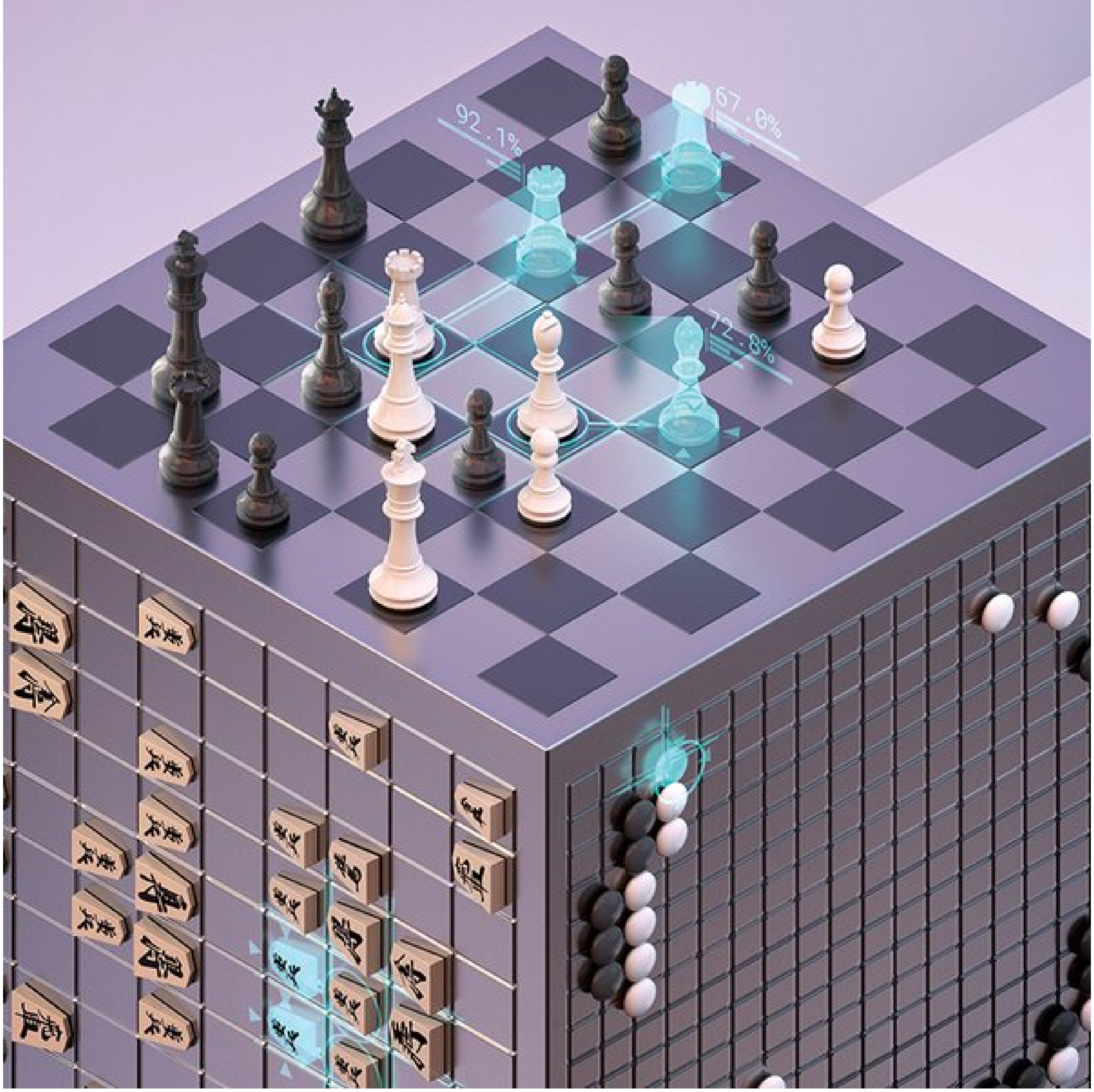

Figure 1: Image source: Rosenbloom, Lois [1]

HOW TO CREATE A CHESS ENGINE WITH DEEP REINFORCEMENT LEARNING

A CRITICAL LOOK AT DEEPMIND'S ALPHAZERO

INTERNAL PROMOTOR: WOUTER GEVAERT

EXTERNAL PROMOTOR: TOM VANDECAVEYE

RESEARCH CONDUCTED BY

TUUR VANHOUTTE

FOR OBTAINING A BACHELOR'S DEGREE IN

MULTIMEDIA & CREATIVE TECHNOLOGIES

HOWEST | 2021-2022

Preface

This bachelor thesis is the conclusion to the bachelor program Multimedia & Creative Technologies at Howest college West Flanders in Kortrijk, Belgium. The program teaches students a wide range of skills in the field of computer science, with a focus on creativity and Internet of Things. From the second year on, students can choose between four different modules:

1. **AI Engineer**
2. **Smart XR Developer**
3. **Next Web Developer**
4. **IoT Infrastructure Engineer**

This bachelor thesis was made under the **AI Engineer** module. The subject of the thesis is a critical look at the result of my research project in the previous semester. The goal of the project was to create a chess engine in Python with deep reinforcement learning based on DeepMind's AlphaZero algorithm.

I will explain the research I needed to create it, the technical details on how I programmed the chess engine and I will reflect on the results of the project. To do this, I will contact multiple people and communities familiar with the field of reinforcement learning to get a better understanding of the impact of this research on society. Based on this, I will give advice to people and companies who wish to implement similar algorithms.

I would like to show gratitude to Wouter Gevaert for his enthusiastic support in the creation of my research project and this thesis. I also want to thank the other teachers at Howest Kortrijk, who shared their knowledge and expertise in programming and AI in very interesting classes.

Furthermore, I'm grateful to my external promotor, Tom Vandecaveye, who has agreed to read and grade this thesis as an unbiased party. Tom is an employee of dotOcean, the company I did my internship in at Howest.

Finally, I would like to thank my parents for giving me the chance to have a good education, and the motivation to get the best I can out of my studies.

Tuur Vanhoutte, 30th May 2022

Abstract

This bachelor thesis answers the question: “How to create a chess engine using deep reinforcement learning?”. It explains the difference between normal chess engines and chess engines that use deep reinforcement learning, and specifically tries to recreate the results of AlphaZero, the chess engine by DeepMind, in Python on consumer hardware.

The technical research shows what is needed to create my implementation using Python and TensorFlow. It shows how to program the chess engine, how to build the neural network, and how to train and evaluate the network. During the creation of this chess engine, it was crucial to create a huge amount of data through self-play.

The thesis contains a reflection on the results of my research project, which proposes a solution to the problem of creating a high amount of games through self-play. It also reflects on the impact of this research on society, and the viability of this type of artificial intelligence in the future. It follows with advice to people and companies who wish to implement similar algorithms.

Finally, the conclusion offers a definitive answer to the research question, based on the previous sections. It explains what went wrong and how to fix it.

Contents

List of figures

List of tables

List of abbreviations

Glossary

1 Introduction

Chess is not only one of the oldest and most popular board games in the world, it is also a breeding ground for complex algorithms and more recently, machine learning. Chess is theoretically a deterministic game: no information is hidden from either player and every position has a calculable set of possible moves. Because the branching factor of chess is about 35-38 moves [2], meaning that in every position an average of 35-38 moves are possible, calculating if a position is winning or losing requires an enormous amount of calculations.

Throughout the entire history of computer science, researchers have continuously tried to find better ways to calculate if a position is winning or losing. The most famous example is the StockFish engine [3], which uses the minimax algorithm with alpha-beta pruning to calculate the best move.

Recently, researchers at Google DeepMind have developed a new algorithm called AlphaZero [4]. This thesis explores the concept of AlphaZero, how to create a chess engine based on it, and the impact of the algorithm on both the world of chess and the rest of society.

Research has been conducted by investigating what is needed to recreate the results of AlphaZero by programming a simple implementation using Python and TensorFlow Keras. This was done as part of a research project between November 2021 and February 2022. The code was written with lots of trial and error, as DeepMind released very little information about the detailed workings of the algorithm. It also only released a simple version of the algorithm in pseudocode, so it isn't possible to directly compare AlphaZero with other chess engines.

As I have been playing chess for most of my life, I was naturally very interested in the workings of AlphaZero and why it worked so well against the StockFish engine. Because AlphaZero was trained on supercomputers, I wanted to investigate where its flaws lie when implementing it in Python on consumer hardware.

2 Research

2.1 What is a chess engine?

According to Wikipedia [5], a chess engine is a computer program that analyzes chess or chess variant positions, and generates a move or list of moves that it regards as strongest. Given any chess position, the engine will estimate the winner of that position based on the strength of the possible future moves up to a certain depth. The strength of a chess engine is determined by the amount of moves, both in depth and breadth, that the engine can calculate.

2.2 How do traditional chess engines work?

Contemporary chess engines, like StockFish [3], use a variant of the minimax algorithm that employs alpha-beta pruning.

2.2.1 The minimax algorithm

The minimax algorithm [6] is a general algorithm usable in many applications, ranging from artificial intelligence to decision theory and game theory. The algorithm tries to minimize the maximum amount of loss. In chess, this means that the engine tries to minimize the possibility for the worst-case scenario: the opponent checkmating the player. For games where the player needs to maximize a score, the algorithm is called maximin: maximizing the minimum gain.

Minimax recursively creates a search tree [7], with chess positions as nodes and chess moves as edges between the nodes. Each node has a value that represents the strength of the position for the current player. At the start of the algorithm, the tree only consists of a root node that represents the current position. It then explores the tree in a depth-first manner by continuously choosing random legal moves, creating nodes and edges in the process.

This means that it will traverse the tree vertically until a certain depth is reached:

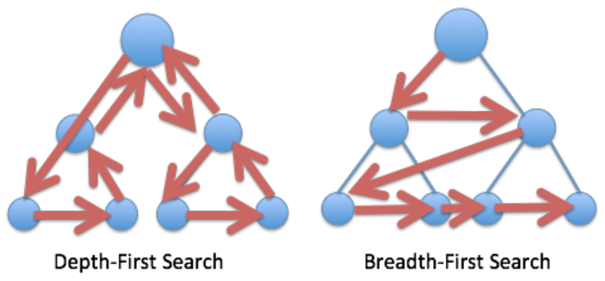

Figure 2: Depth-First search vs Breadth-First search [7]

When that happens, that leaf node's position is evaluated and its value is returned upwards to the parent node. The parent node looks at all of its children's values, and receives the maximum value when playing white, and the minimum value when playing black.

This repeats until the root node receives a value: the strength of the current position.

2.2.2 The evaluation function

The value estimation of leaf nodes is done by an evaluation function [8] written specifically for the game. This function can differ from engine to engine, and is usually written with help from chess grandmasters.

2.2.3 Pseudocode

The algorithm is recursive; it calls itself with different arguments, depending on which player's turn it is. In chess, white wants to maximize the score, and black wants to minimize it [6].

```
function  minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := - inf
        for each child of node do
            value := max(value, minimax(child, depth - 1, FALSE))
        return value
    else (* minimizing player *)
        value := + inf
        for each child of node do
            value := min(value, minimax(child, depth - 1, TRUE))
        return value
```

Calling the function:

```
// origin = node to start
// depth = depth limit
// maximizingPlayer = TRUE if white, FALSE if black
minimax(origin, depth, TRUE)
```

2.2.4 Alpha-beta pruning

Because the amount of nodes necessary to get a good estimation of the strength of a position is so high, the algorithm needs to be optimized. Alpha-beta pruning [9] aims to reduce the amount of nodes that need to be explored by minimax. It does this by cutting off branches in the search tree that lead to worse outcomes.

Say you're playing the white pieces. You want to minimize your maximum loss, which means you want to make sure that black's score is as low as possible. Minimax assumes that the opponent will play the best possible move. If one of white's possible moves leads to a position where black gets a big advantage, it will eliminate that branch of the search tree. As a result, the amount of nodes to explore is greatly reduced, while retaining a good estimation of the strength of the position.

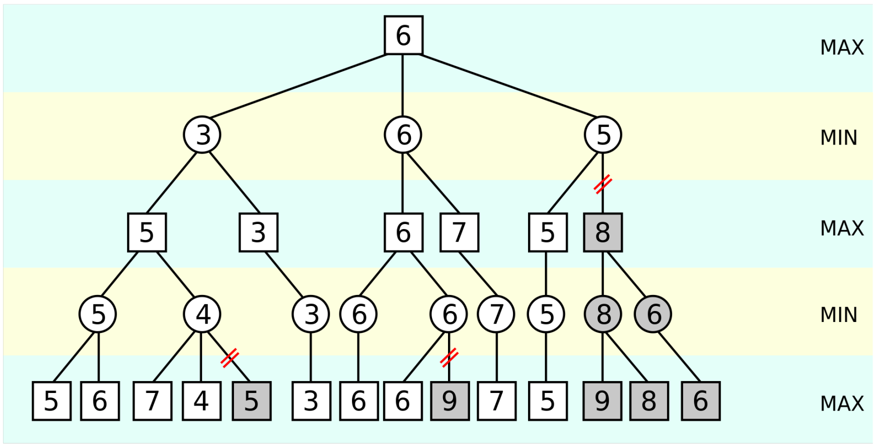

Figure 3: Example of alpha-beta pruning in minimax

In the above example: white has a value of 6 in the root node. If white plays the move on the right that leads to a position with a value of 5, the next move black wants to play will be the one that leads to the position with the lowest possible value. Therefore, it obviously will never play the move that leads to the position with a value of 8, as that would be a winning position for white. This means that whole branch can be pruned.

2.3 Monte Carlo Tree Search

The biggest problem with minimax algorithms that use a depth limit is the dependency on the evaluation function. If the evaluation function makes incorrect or suboptimal estimations, the algorithm will suggest bad moves. Developers of contemporary chess engines like StockFish continuously try to improve this function. Since 2020, StockFish has been using a sparse and shallow neural network as its evaluation function. This neural network is still trained using supervised learning, not (deep) reinforcement learning.

Using alpha-beta pruning can also bring about some problems. Say the player can sacrifice a piece to get a huge advantage later in the game. The algorithm might cut off the branch and never explore that winning line, because it considers the sacrifice a losing position [10].

Monte Carlo Tree Search (MCTS) [11] is a search algorithm that can be used to mitigate these problems. MCTS approximates the value of a position by creating a search tree using random exploration of the most promising moves.

2.3.1 The algorithm

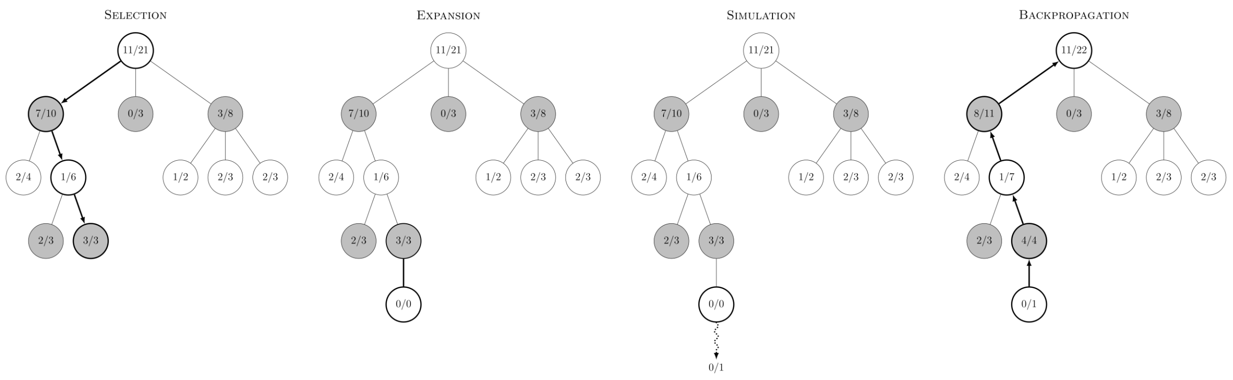

Figure 4: The 4 steps of the MCTS algorithm [11]

To create this search tree for a certain position, MCTS will run the following algorithm hundreds of times, consisting of four steps. Every execution of the algorithm is called an MCTS **simulation**. Do not confuse this with the third step of the algorithm, which is also called *simulation*.

1. **Selection**:
    - Starting from the root node, select a child node based on a formula of your choice.
    - Most implementations of MCTS use some variant of Upper Confidence Bound (UCB) [12]
    - Keep selecting nodes until a node has been reached that has not been visited (= “expanded”) before. We call this a leaf node.
    - If the root node is a leaf node, we immediately proceed to the next step.
2. **Expansion**:
    - If the selected leaf node is a terminal node (the game ends), proceed to the backpropagation step.
    - If it doesn’t, create a child node for every possible action that can be taken from the selected node.

3. **Simulation**:
    - Choose a random child node that was expanded in the previous step.
    - By only choosing random moves, simulate the rest of the game from that child node's position.
4. **Backpropagation**:
    - Return the simulation's result up the tree.
    - Every node tracks the number of times it has been visited, and the number of times it has lead to a win.

For chess, this algorithm is very inefficient, because of its necessity to simulate an entire game of chess in the third step of every simulation. Just to calculate the value of one position would need hundreds of these simulations to get a good estimation. This is why the selection formula needs to be chosen carefully; it's important to select nodes in a way that balances exploration and exploitation.

2.4 Go

Go is a Chinese two-player strategy board game that uses white and black stones as playing pieces [13]. It is played on a rectangular grid of (usually) 19 by 19 lines. The rules are relatively simple, but due to its extremely large amount of possible actions, Go has been a very popular playground for AI research similarly to chess. Go's much larger branching factor compared to chess makes it very difficult to evaluate a position using traditional methods like minimax with alpha-beta pruning.

Figure 5: Go board [13]

2.4.1 AlphaGo

In 2014, DeepMind Technologies [14], a subsidiary of Google, started developing a new algorithm called AlphaGo to play Go [15]. Previously, the strongest Go engines were only good enough to win against amateur Go players [16]. The algorithm used a combination of the MCTS algorithm and a deep neural network to evaluate positions.

AlphaGo was built [16], [17] by first training a neural network with supervised learning by using data from human games. The weights of that network were then copied to a new reinforcement learning network. That network was used to create a training set through self-play. By playing against itself and every move recording the current board state, the moves the network considered, and the eventual winner of the game, a training set was created. That training set was then used to train the reinforcement learning network. A separate network (the value network) was used to estimate the value of a position.

2.4.2 AlphaGo Zero

Because AlphaGo still used some amateur games to learn from, the next step was creating a version of AlphaGo that learns completely from scratch. That's why DeepMind developed AlphaGo Zero [18]. AlphaGo Zero uses a different kind of network than AlphaGo. Instead of using two separate networks, it will combine the two networks into one with two outputs: a policy output and a value output. It's also using different layers: residual layers instead of convolutional layers [19].

2.5 AlphaZero

AlphaZero is a generalized version of AlphaGo Zero, created to master the games of chess, shogi ("Japanese chess"), and Go [4], [20]. For chess, AlphaZero was evaluated against StockFish version 8 by playing 1000 games with 3 hours per player, plus 15 seconds per move. It won 155 times, lost 6 times and the remaining games were drawn. AlphaZero uses a single neural network with two outputs, just like AlphaGo Zero.

2.5.1 Neural network input

The input to the network represents the state of the game. It has the following shape: $N \cdot N \cdot (M \cdot T + L)$:

- N is the board size
  - $N = 8$ in chess.
- M is the number of different pieces on the board,
  - Two players with six types of pieces each
  - Every piece is represented by its own 8x8 board of boolean values
  - For every square: 1 if the piece is on that square, 0 if it isn't
  - $M = 12$ in chess
- T is the amount of previous moves that are used as input, including the current move.
  - AlphaZero used $T = 8$ for both chess, shogi, and Go.
  - This gives the network a certain history to learn from
- L represents a set of rules specific to the game
  - $L = 7$ in chess
  - 1 plane to indicate whose turn it is
  - 1 for the total amount of moves played so far
  - 4 for castling legality (both players can castle kingside or queenside under certain conditions)
  - 1 to represent a repetition count (in chess, 3 repetitions results in a draw).
- $\Rightarrow 8 \cdot 8 \cdot (12 \cdot 8 + 7)$.

This means that the input to the neural network is composed of 119 8x8 boards of values. The M planes that encode the pieces are repeated T times in the input, resulting in 112 boards. For the planes representing integers, every square in that 8x8 plane will be assigned the integer value. Other planes are one-hot encoded boolean boards.

2.5.2 Neural network layers

DeepMind tested multiple neural network architectures for AlphaGo Zero [21]. The following parts were used in these networks:

- Convolutional block
  - A convolution layer
  - Batch normalization layer
  - ReLu activation function
- Residual block
  - This consists of two convolutional blocks, and a skip-connection
  - The skip-connection will combine the input of the block to the output of the first two convolutional blocks

These networks were tested by DeepMind during development of AlphaGo Zero [19]:

1. **'dual-res':** a single tower of 20 residual blocks with combined policy and value heads. This is the architecture used in AlphaGo Zero.
2. **'sep-res':** two towers of 20 residual blocks each: one with the policy head and one with the value head.
3. **'dual-conv':** a single tower of 12 convolutional blocks with combined policy and value heads.
4. **'sep-conv':** two towers of 12 convolutional blocks each: one with the policy head and one with the value head. This is the network used in AlphaGo.

AlphaZero uses the same network architecture as AlphaGo Zero: dual-res.

2.5.3 Neural network output

The neural network has two outputs:

- A policy head, which represents a probability distribution over the possible actions.
- A value head, which represents the value of the current position.

While the value head simply outputs a single float value between -1 and 1, the policy head is quite a bit more complicated. It outputs a vector of probabilities, one for each possible action in the chosen game. For chess, 73 different types of actions are possible:

- 56 possible types of "queen-like" moves: 8 directions to move the piece a distance between 1 and 7 squares.
- 8 possible knight moves
- 9 special "underpromotion" moves:
  - If a pawn is promoted to a queen, it is counted as a queen-like move (see above)
  - If a pawn is promoted to a rook, bishop, or knight, it is seen as an underpromotion (3 pieces)
  - 3 ways to promote: pushing the pawn up to the final rank, or diagonally taking a piece and landing on the final rank
  - $\Rightarrow 3 \cdot 3 = 9$

These 73 actions are each represented by a plane of 8x8 float values. Say the first plane is a queen-like move to move a piece one square northwest, the second plane could be the same type of move, but a distance of two squares, and so on. The squares on these planes represent the square from which to pick up a piece.

The result is a $73 \cdot 8 \cdot 8$ vector of probabilities, so 4672 float values.

2.6 Training the network

To train this type of network, it's necessary to create a large dataset. This is done by letting the engine play against itself for a high amount of matches. For every action taken by the agent, data is collected and stored in the training set. For complex games like chess, shogi and Go, this training set needs to be huge because of the extremely large amount of possible situations.

2.6.1 Tensor Processing Units (TPU)

Because of the requirement to play a high amount of matches against itself, it was necessary to calculate MCTS simulations in parallel on as fast as possible hardware. To help with these calculations, DeepMind used Google's newly created Tensor Processing Units (TPU) [22]. A TPU is an application-specific integrated circuit (ASIC [23]) that is specifically built for machine learning with neural networks. Since 2018, these TPUs have been made publicly available to rent through Google's Cloud Platform. Smaller TPUs can be purchased from Google.

2.7 Leela Chess Zero

Leela Chess Zero (lc0) is a free, open-source project that attempts to replicate the results of AlphaZero [24]. Lc0 was adapted from Leela Zero [25], a Go computer that attempted to replicate the results AlphaGo Zero [18].

It is written in C++ [26], and it has managed to play at a level that is comparable to the current best version of StockFish. Because lc0 is a community driven project, volunteers can help create training games through self-play using their own computers. This made it possible to feed millions of chess games into the network.

3 Technical research

3.1 Introduction

Creating the chess engine required programming the following parts:

- The MCTS algorithm
- A tree data structure with nodes and edges
- The neural network
- The training pipeline
- The evaluation pipeline
- A way to store every move to a dataset
- A class to make the engine play against itself
- A GUI to play against the engine
- Docker containers for easily scaling and distributing the program

All code was written in Python. The neural network was made using the TensorFlow Keras library. Chess rules and helper functions were implemented using the open-source library python-chess [27].

3.2 Class structure

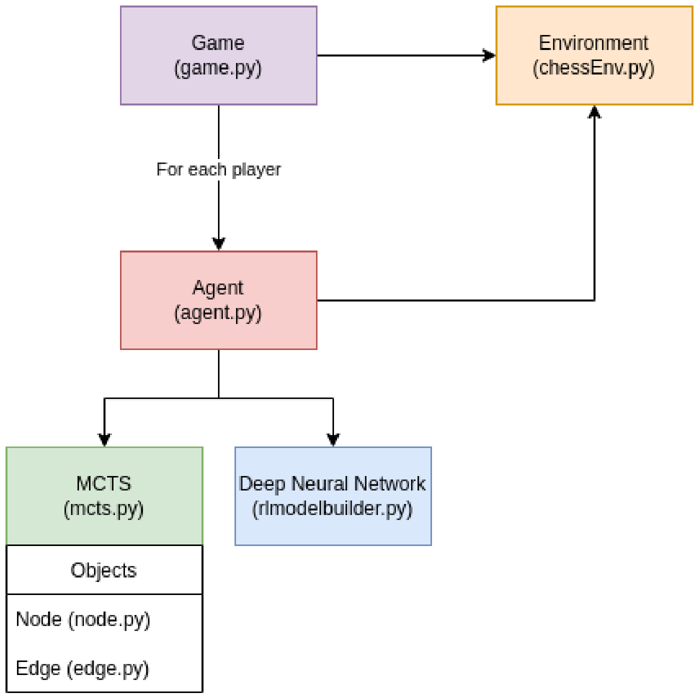

Figure 6: Basic class structure for the code responsible for playing a game

3.2.1 Making one move

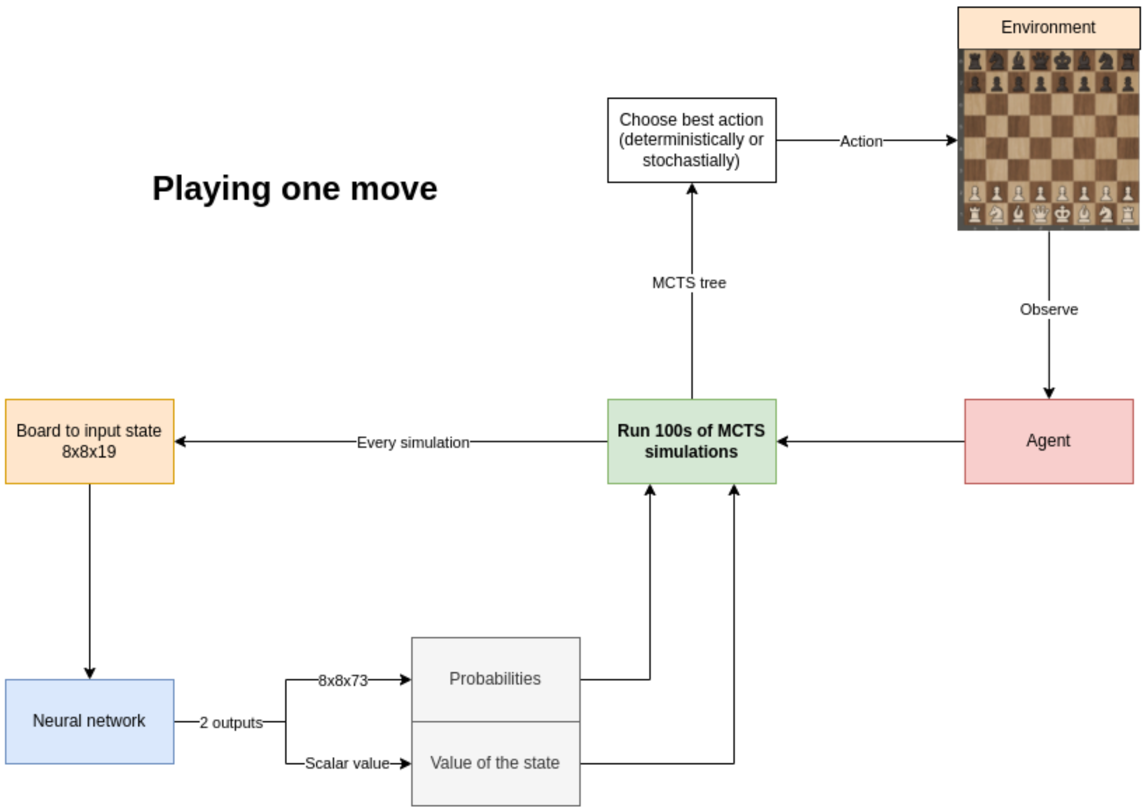

Figure 7: Pipeline to make one move

To make a move, an Agent (white or black) observes the environment: the chessboard. The agent calls upon the MCTS class to create a tree with as root the current state of the chessboard. The MCTS class will run the MCTS algorithm hundreds of times. This amount is configurable in the config file. Higher amounts result in a more accurate estimation of the position's value, but also in longer computation times.

Every MCTS simulation, the neural network will be called to evaluate a position. The two outputs, the policy and the value, will be used to update the tree.

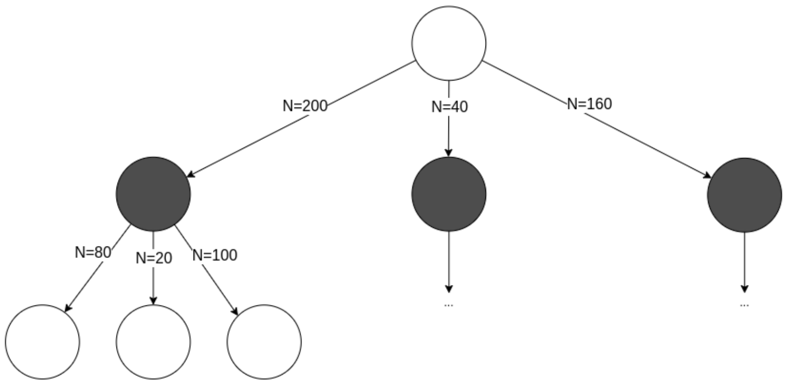

Figure 8: Example tree after 400 simulations. N = amount of times the selection step selects that edge

After the simulations are done, the agent will pick the best move from the tree. It can do this in two ways:

- Deterministically: choosing the most visited move
- Stochastically: creating a uniform distribution of the visit counts and picking a move from that distribution

Stochastic selection is better when creating a training set, as it will result in a more diverse dataset. Deterministic selection is better when evaluating with a previous network, or playing against the network competitively. In that case, picking the most visited move is the best choice.

3.3 The neural network

Initially, a prototype of the neural network was created with randomly initialized weights. The Python class to create the model was immediately made with customizability in mind: the input and output shapes can be given as arguments, and the sizes of the convolution filters can be changed using a configuration file.

The neural network architecture is the same as AlphaZero's (see the Research section).

3.4 A tree structure with nodes and edges

As mentioned before, the MCTS algorithm creates a tree structure to represent the possible future states after the current position.

Node and Edge classes were written. The Node class represents a position in the game, and holds the following data:

- The position: a string representation of the board using the Forsyth-Edwards Notation (FEN) [28]
- The current player to move (boolean)
- A list of edges connected to this node
- The visit count of this node, initialized to 0
- The value for this node, initialized to 0

The Edge class represents a move. It holds the following data:

- The input node (the position from which the move was made)
- The output node (the resulting position after taking the move)
- The move itself: an object of the Move class from the python-chess library, which holds:

  - The source square and the target square of the move
  - If the move was a promotion: the piece it was promoted to
- The prior probability of this move
- The visit count of this edge, initialized to 0
- The value for this action, initialized to 0

The tree can also be plotted using the Graphviz library [29]. A recursive function was written to create the tree and output it to an SVG file.

3.5 The MCTS algorithm

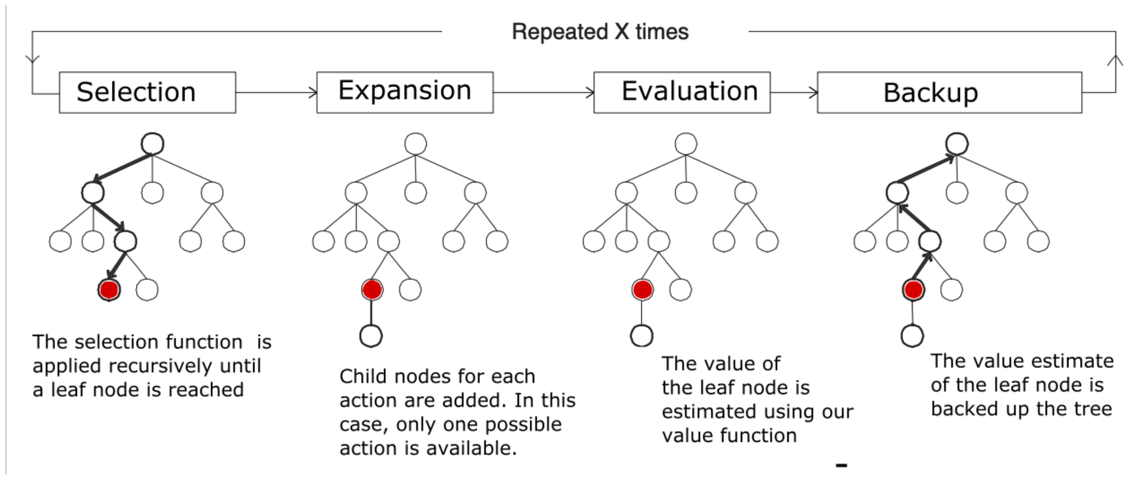
Repeated X times
Selection
Expansion
Evaluation
Backup
The selection function is applied recursively until a leaf node is reached
Child nodes for each action are added. In this case, only one possible action is available.
The value of the leaf node is estimated using our value function
The value estimate of the leaf node is backed up the tree

Figure 9: The four steps in AlphaZero's MCTS algorithm [30]

A class was written to hold an MCTS tree. It holds the agent who created the tree and the current position as the root node.

3.5.1 The selection step

For the selection step, the following UCB formula [20] was used to determine which edge to select:

$$UCB = \left(\log(\frac{(1 + N_{\text{parent}} + C_{\text{base}})}{C_{\text{base}}}) + C_{\text{init}} \right) \cdot P \cdot \frac{\sqrt{N_{\text{parent}}}}{(1+N)} \tag{1}$$

- C_{base} and C_{init} are constants that can be changed in the config file. The same values as AlphaZero were used.
- N_{parent} is the visit count of the input node
- N is the visit count of the edge
- P is the prior probability of the edge

The selection step combines this UCB formula with the edges action-value and visit count:

$$Q = \frac{W}{N+1} \tag{2}$$

$$V = \begin{cases} UCB + Q & \text{if white} \\ UCB - Q & \text{if black} \end{cases} \tag{3}$$

The edge with the highest value (V) is selected. After selection, the visit count for the edge's output node is incremented by one. We call this output node the 'leaf node'.

3.5.2 The expansion step

The leaf node is expanded by creating a new edge for each possible (legal) move. If there are no legal moves, the outcome (draw, win, loss) is checked and the leaf node is passed to the next step in the algorithm.

If there are legal moves, the leaf node is given as an input to the neural network.

While AlphaZero uses an input shape of 119 8x8 boolean boards, I opted for a much lighter input shape. The input used in this project only has 19 boards:

- 1 board to show which turn it is: 1 is white and 0 is black
- 4 boards to show if each player still has castling rights
- 1 board to show if a draw is possible after 50 moves without a capture or pawn move
- 12 boards to show the positions of the pieces on the board
- 1 board to show the en passant square: if a pawn can be taken en passant, this square is set to 1

For example:

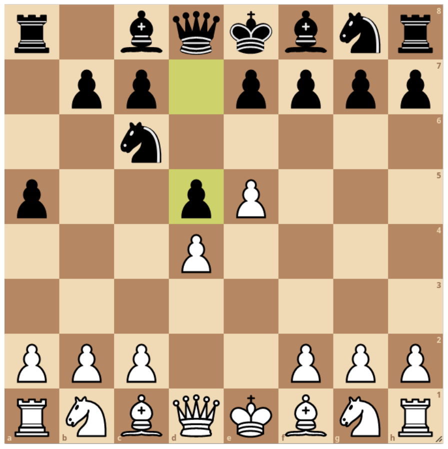

Figure 10: Board example

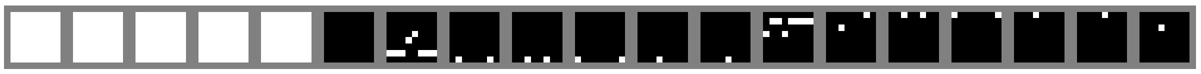

Figure 11: The input state converted from the above example board

Notice how the last square shows the square were en passant is possible. Grey padding was added to make a better visual presentation of the output planes.

The neural network will return the policy and the value of the position. As described in the research part of this thesis, the value is a float between -1 and 1, and the policy is a 73x8x8 tensor of floats. This policy is then mapped to a dictionary, where keys are moves and the values are the probabilities of each move.

The neural network gets no information about the rules of chess, so it is necessary to filter out the illegal moves.

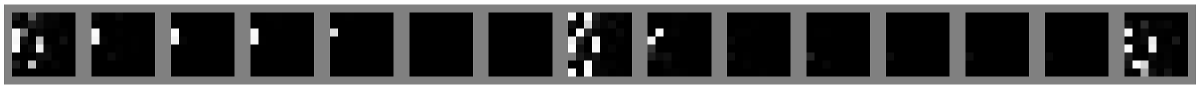

Figure 12: A subset of the 73 8x8 planes from the policy output. The brighter the pixel, the better the move.

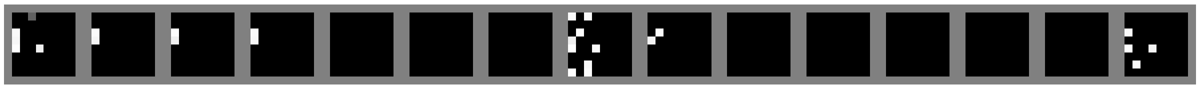

Figure 13: The same subset, but with the illegal moves filtered out

The above two images were made using a trained model. The policy output of an untrained model with random weights looks like this:

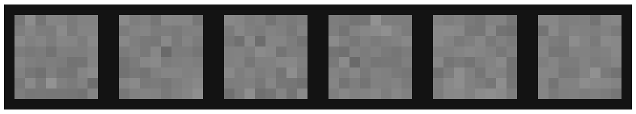

Figure 14: Some output planes from an untrained model.

(Black padding was used instead of gray to make it clearer.)

This clearly shows the trained model has at least some level of understanding of which moves are legal, without giving it any knowledge about the rules of chess. The model isn't even told we're playing chess: it learns the correct outputs completely on its own.

3.5.3 The evaluation step

The value received from the neural network is now assigned to the leaf node.

3.5.4 The backpropagation step

The value from the leaf node is now also added to every selected node in the path from the root to the leaf node. Concretely, for every selected edge in the path, the following values are changed:

- The edge's input node's visit count is incremented by 1
- The edge's visit count is incremented by 1
- The edge's value is incremented by the value of the leaf node

3.6 Creating the dataset

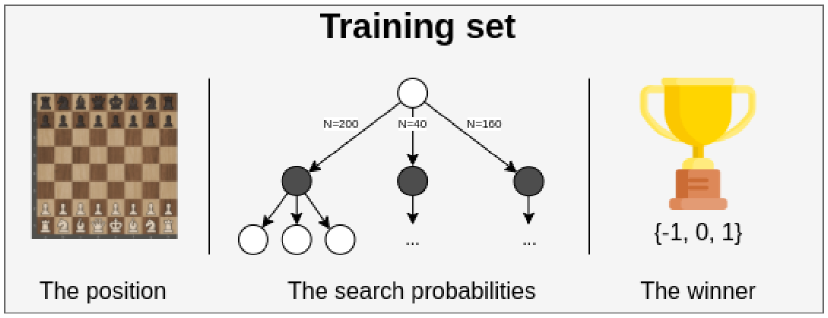

Figure 15: The training set consists of: the input state, the move probabilities, and the eventual winner

To improve the performance of the neural network, the algorithm needs to play against itself for a high amount of games. Every move is saved to memory in a simple Python list. Once the game is over, the winner is assigned to every move in memory. The whole game is then saved to a binary file in NumPy's .npy format. When a new game starts, the memory is cleared.

3.6.1 Creating a dataset from puzzles

Because creating the dataset is extremely time-consuming, I came up with the idea to also create data from chess puzzles. A chess puzzle is a position with one simple goal: find the best move or sequence of moves. Lichess.org, the open source chess website, has a huge database of over 2 million of these puzzles publicly available for free [31]. These are the most common puzzle categories:

- Mate-in-X: find the best moves to checkmate the opponent in a maximum of X moves
- Capturing one of the opponent's undefended pieces
- Getting a positional advantage

The idea is to let the agent play from a certain given position (the start of the puzzle), instead of from the start of the game, and see if the agent can find the correct solution. To stay true to the idea of creating a chess engine without any human intervention, the agent is never told if it has reached the end of the puzzle. It simply plays moves until a checkmate happens. A move limit is imposed to prevent the agent from taking too long, and if this limit is reached the puzzle is discarded.

This would in theory give the model a better understanding of common chess tactics, without having to play whole games from start to finish.

Apart from playing full games, the network was trained with mate-in-1 and mate-in-2 puzzles. This is because these kinds of puzzles end quickly (and are thus less time-consuming), and they have a clear winner: the player who can checkmate the opponent. The Lichess database has over 400,000 puzzles of these two types.

One problem I noticed with this approach, especially with mate-in-1 puzzles, is that the network will learn that whoever is playing the first move will always win. The value network will always assign a value close to 1 when it's white's turn, and a value close to -1 when it's black's turn. That's why it is crucial to maintain a good balance between real games and puzzles.

3.7 Training the neural network

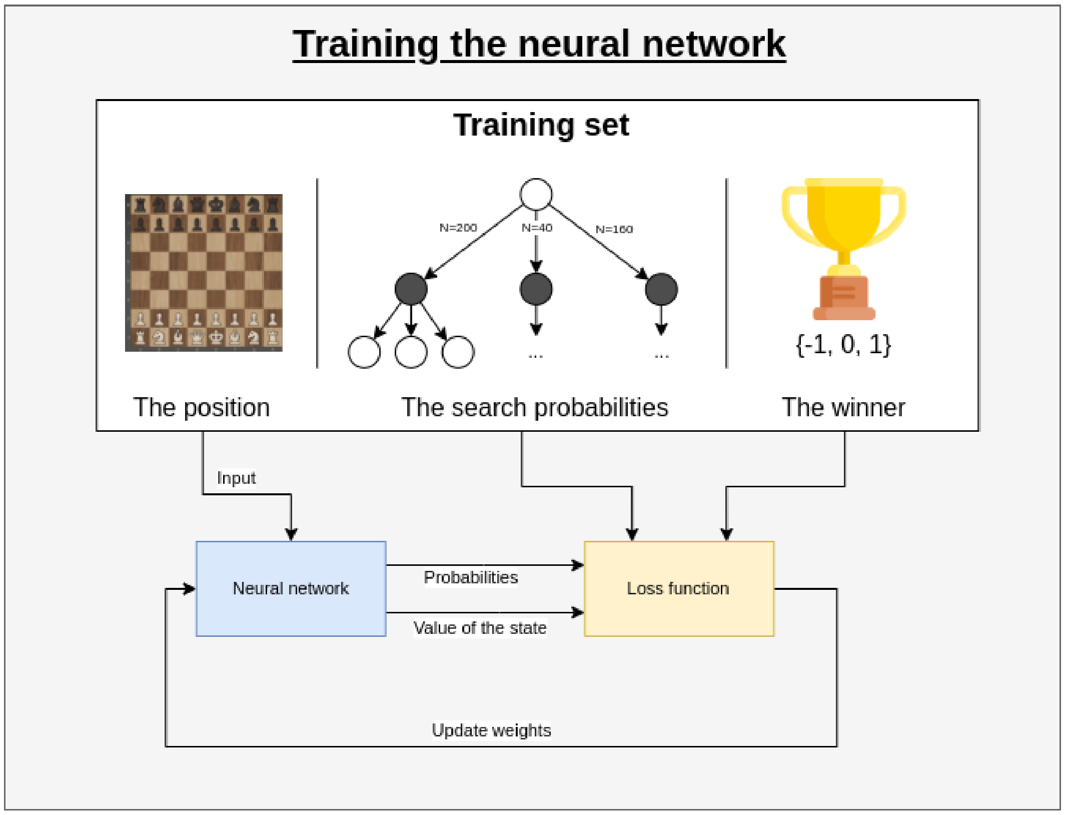

Figure 16: The training pipeline

To train the neural network, every saved game's binary file is loaded into memory. The memory is shuffled to avoid the neural network accidentally learning time dependent patterns. Random batches of the training set are then processed through the neural network.

The position is used as the input to the neural network. The network's outputs are then compared to the move probabilities and the winner from the dataset:

- The move probabilities are converted to a 73x8x8 tensor, which is compared with the output of the network
- The winner is compared with the value output of the network.

The training pipeline employs two separate loss functions. The policy head uses categorical cross-entropy and the value head uses mean squared error.

3.7.1 The first training session

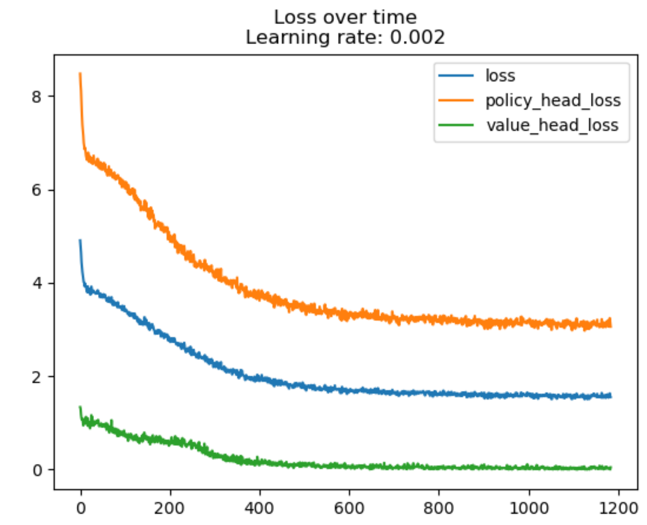

Figure 17: First training session

Naturally, the first training session started with a random model. The dataset was created by running self-play for many hours, resulting in a dataset of around 76,000 positions. Training was performed with a learning rate of 0.002 with the Adam optimizer, and a batch size of 64.

After training, the model was saved to disk, so it can be used to run self-play again.

3.7.2 The second training session

The previous model was used as the starting point for the second training session. A new dataset was made using only games created by that model. Due to time constraints, this dataset was only made up of 50,000 positions.

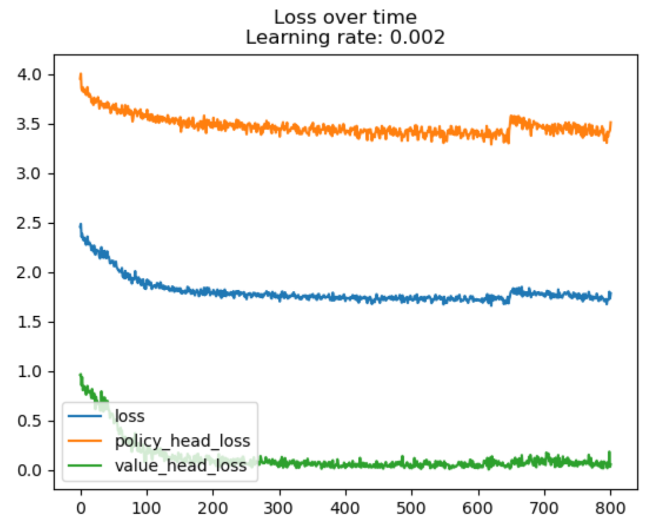

Figure 18: Second training session

It's clear the model manages to learn something at the start of the training session, but seems to plateau after 150 batches. Because the first training session started from a random model, it managed to learn a lot more before reaching a plateau after around 600 batches.

3.8 Multiprocessing

Because self-play is very time-consuming, there needed to be a way to play multiple games in parallel on the same system.

3.8.1 Without multiprocessing

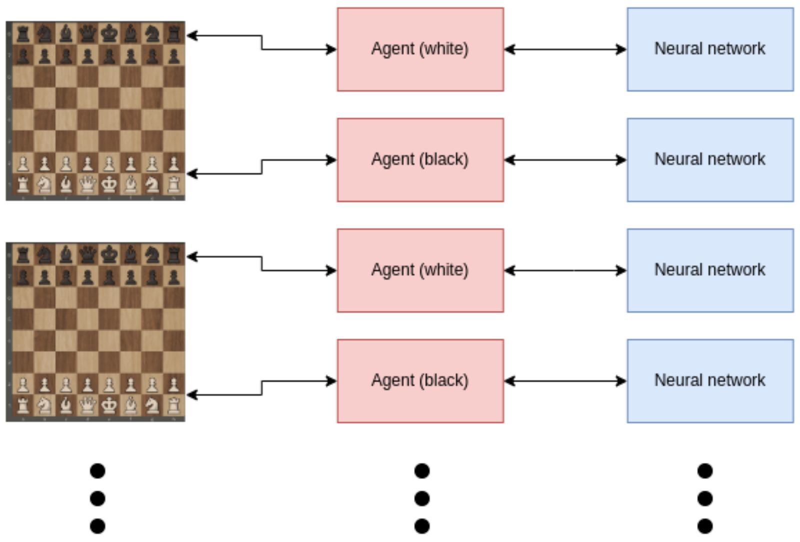

Figure 19: Self-play without multiprocessing

Previously, every game was played in its own process, but every agent needed to send predictions to a neural network. This resulted in every process creating its own copy of the neural network. This is extremely heavy for the GPU, and it's impossible to scale. Due to VRAM limits, only two games could be played in parallel on my system (RTX 3070).

3.8.2 With multiprocessing

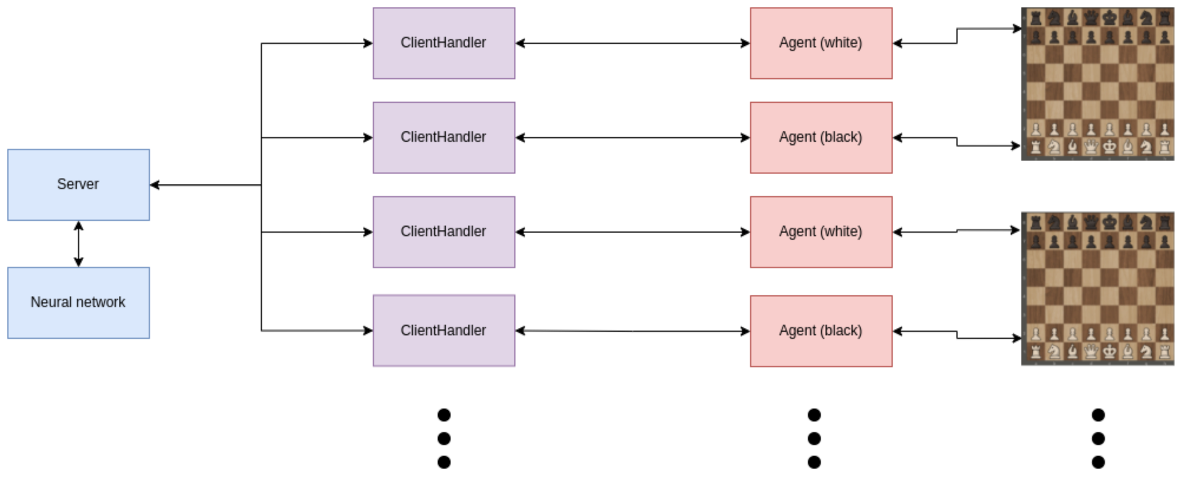

Figure 20: Self-play with multiprocessing

To solve the issue of parallel self-play, I created a client-server architecture with Python sockets. The server side has access to the neural network. For every client (the self-play process), the server creates a ClientHandler in a new thread.

Here's how one client works:

1. The MCTS algorithm runs 100s of simulations every move. Every simulation sends a chess position to the neural network.
2. The agent sends the chess position through a socket.
3. The ClientHandler receives the chess position and sends it to the server.
4. The server calls the network's predict function and returns the outputs to the client handler.
5. The ClientHandler sends the outputs back to the client.

This socket communication does have a small overhead in both time and CPU usage, but it's much faster than the previous method because it is much more scalable. Here's a comparison of the two methods, with the server running on the first system:

	No multiprocessing	**With multiprocessing (8 games)**
RTX 3070 + Ryzen 7 5800H	50 simulations/sec	30 simulations/sec per game
GTX 1050 + i7 7700HQ	30 simulations/sec	15 simulations/sec per game

Table 1: Comparison of multiprocessing and non-multiprocessing self-play

With 8 games in parallel on the first system, I managed to get an average speed of $30 \cdot 8 = 240$ simulations per second. The second system is much less powerful and needed to connect over Wi-Fi, but it still managed an average of 120 simulations per second.

3.9 A GUI to play against the engine

The GUI was based on a GitHub project that someone created for simply visualizing a chessboard with PyGame [32], [33]. I greatly improved and expanded that project to include a way for the player to interact with the board, play against an engine, play against yourself, and more [34]. I created a pull request to include my changes in the original author's GitHub project, but it was declined due to being due extensive and out-of-scope for the project [35].

Using this code, a GUI class was created for playing against the engine. The player can choose a color and play against the engine. Clicking on a piece will highlight its square, clicking on another square will move the piece to that square. Right-clicking will cancel the move.

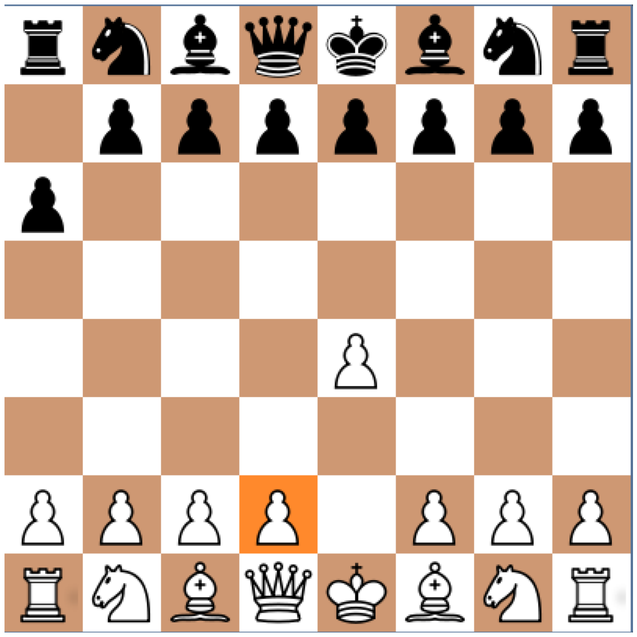

Figure 21: Chessboard example when playing against the engine

Promoting a pawn can be done by simply moving a pawn to the last rank. A menu will pop up to choose the piece to promote to.

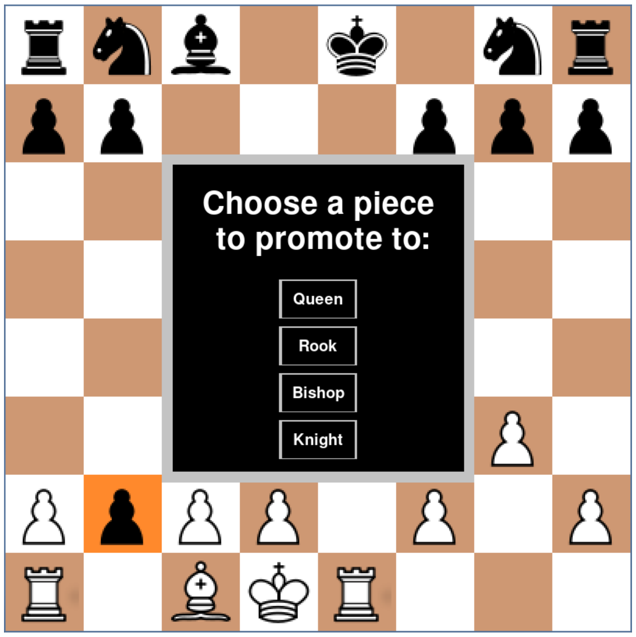

Figure 22: Promoting a pawn

3.10 Docker images

Because the amount of data needed to train the neural network is very large, two docker images were created:

- A server image that runs the neural network and listens for clients.
- A client image that runs self-play and sends chess positions to the server.

Using a docker-compose file, the server and client containers can be managed easily. The number of clients to run in parallel can be configured in that file, along with many other settings.

It's recommended to run the server on a fast GPU-equipped system, and the clients on a system with many high-performance CPU cores. The clients do not need a GPU to run self-play.

3.11 The final project

This section walks through all executable programs included in the final project [36].

3.11.1 Creating your own untrained AI model

```
$ python rlmodelbuilder.py --help
usage: rlmodelbuilder.py [-h] [--model-folder MODEL_FOLDER] [--model-name MODEL_NAME]

Create the neural network for chess

options:
  -h, --help            show this help message and exit
  --model-folder MODEL_FOLDER
                        Folder to save the model
  --model-name MODEL_NAME
                        Name of the model (without extension)
```

This will create a new model with the name <NAME> in the folder <FOLDER>. The model parameters (amount of hidden layers, input and output shapes if you want to use the network for a different game, the amount of convolution filters, etc.) can be changed by editing the config.py file.

3.11.2 Creating a training set through self-play

Creating the docker containers:

```
# in the repo's code/ folder:
docker-compose up --build
```

This will create one server and the amount of clients that is configured in the docker-compose.yml file. That file can also be used as a reference to deploy a Kubernetes cluster for parallel self-play with high scalability and reliability.

Using the created GUI, it is possible to visualize the self-play of all these boards in real-time. This can be done by settings the “SELFPLAY_SHOW_BOARD” environment variable in docker-compose.yml to “true”. For every replica of the client, a new PyGame window will open where the board will change in real-time.

You can also manually run self-play or create data using puzzles:

```
$ python selfplay.py --help
usage: selfplay.py [-h] [--type {selfplay,puzzles}] [--puzzle-file PUZZLE_FILE]
    [--puzzle-type PUZZLE_TYPE] [--local-predictions]

Run self-play or puzzle solver

options:
    -h, --help            show this help message and exit
    --type {selfplay,puzzles}
                          selfplay or puzzles
    --puzzle-file PUZZLE_FILE
                          File to load puzzles from (csv)
    --puzzle-type PUZZLE_TYPE
                          Type of puzzles to solve. Make sure to set a
                          puzzle move limit in config.py if necessary
    --local-predictions   Use local predictions instead of the server
```

3.11.3 Evaluating two models

To determine whether your new model is better than the previous best, you can use the evaluate.py script. It will simulate matches between the two models and record the wins, draws and losses.

In chess, white inherently has a slightly higher chance of winning because they can play the first move [37]. Therefore, to evaluate two models, each model will both play white and black an equal amount of times.

```
$ python evaluate.py --help
usage: evaluate.py [-h] model_1 model_2 nr_games

Evaluate two models

positional arguments:
  model_1     Path to model 1
  model_2     Path to model 2
  nr_games    Number of games to play (x2: every model plays both white and black)

```

```
options:
  -h, --help  show this help message and exit
```

3.11.4 Playing against the AI

```
$ python main.py --help
usage: main.py [-h] [--player {white,black}] [--local-predictions] [--model MODEL]

options:
  -h, --help            show this help message and exit
  --player {white,black}
                        Whether to play as white or black. No argument means random.
  --local-predictions   Use local predictions instead of the server
  --model MODEL         For local predictions: specify the path to the model to use.
```

This will start the GUI application to allow you to play against the engine.

3.12 Porting to C++

Optimization is extremely important for chess engines, especially when the engine needs to play against itself to create a dataset. That is why Python was not an ideal choice to implement this chess engine.

Currently, I'm writing a C++ version of the engine in my spare time [38]. Instead of TensorFlow Keras, I've opted to use PyTorch instead. So far, I've noticed that the C++ version with PyTorch manages to run the MCTS algorithm four times faster than the Python version: 200 simulations per second instead of 50.

4 Reflection

4.1 Introduction

This section is a reflection on the project. It will answer the following questions:

- What are the strengths and weaknesses of this research project?
- Is the result of the project usable for corporations?
- What are the possible obstacles for companies that wish to implement this?
- What is the added value for companies?
- Which alternatives are there?
- Is there a socio-economic impact present?
- Is there opportunity for further research?

To help answer these questions, I have consulted with Dr. Thomas Moerland, a postdoctoral researcher at the Reinforcement Learning Group at Leiden University, in The Netherlands [39]. With his experience in researching AI and specifically Reinforcement Learning, he was able to give me some insights into how AlphaZero could impact society in industries besides gaming.

I have also asked questions to the following online communities:

- The AI Stack Exchange community
- The Leela Chess Zero Discord community
- The TalkChess forum, a forum affiliated with The Computer Chess Club (CCC)

4.2 Strengths and weaknesses

This project successfully implements the MCTS algorithm as modified by DeepMind for AlphaZero and their preceding work, AlphaGo and AlphaGo Zero. The resulting chess engine can be played against using the GUI, and it can be used to create a dataset for further improving the neural network. The project serves as an example of how AlphaZero's MCTS algorithm can be implemented in a chess engine using Python and consumer-grade hardware.

The project offers a way to easily create your own neural network with the rlmodelbuilder.py script. The code is written with modularity and adaptability in mind: by changing parameters in the config.py file, you can easily create a different model for chess, or for similar applications with different input and output shapes. The MCTS algorithm can be used as a reference if you want to implement it other applications.

The ultimate goal of this research project was to create a chess engine that is at least capable of winning against amateur players of around 1000 ELO. This was not possible with my consumer-grade hardware: the chess engine still seems to play very randomly and has not found a good winning strategy. The biggest hurdle was the lack of computational resources to create a large dataset.

However, the project does offer a solution to create more data through self-play: with the docker containers, it is easy for other people to help create a dataset by deploying their own cluster of servers and clients. This way, a global dataset could be made by combining self-play data from volunteers around the world. It can then be used to train a new model.

This is exactly how Leela Chess Zero operates: the developers created a simple program for volunteers to run, which makes the latest model play against itself and uploads the data back to the developers. Once there is enough data, the developers can retrain the model. By continuously repeating this process, Leela Chess Zero slowly became a powerful chess engine on par with the best chess engines in the world [40].

4.3 Is the result of the project usable in the corporate world?

While AlphaZero and its variants have mostly been used in gaming applications, the project is not limited to this. For instance, AlphaGo has been used by Google to properly ventilate their data centers more efficiently. The algorithm helped lower data center cooling costs by 40% [41]. It does this by periodically pulling data from many sensors and feeding that data to a deep neural network to predict how different combinations of potential actions will affect future energy consumption [42].

Theoretically, this project can be used to solve virtually any problem that can be defined as an agent acting upon an environment with discrete state and action spaces [43].

Reinforcement learning in general has been used to solve a wide range of problems. Many websites that have recommendation systems have already used reinforcement learning to improve which products get recommended to users [44]. For example, Netflix uses contextual bandits to personalize artwork to the user [45], [46]. This avoids the need for waiting to collect data from users, training a model, waiting for an A/B test to conclude, to then finally recommend the best artwork. By then, too many users have ignored an item they would have otherwise clicked on.

In the field of robotics, (deep) reinforcement learning can be used to teach a robot through self-play to perform simple tasks that previously required manual programming.

4.4 What are the possible obstacles for companies that wish to implement this?

Depending on the application, the amount of data necessary to train a sufficiently powerful model can be extremely large. It's necessary for the company to have a good infrastructure to create this data. Companies can also invest in cloud solutions like the Google Cloud Platform to take advantage of their TPUs [47].

Because these types of algorithms need to be trained through self-play, applications should be written in a programming language that is capable of training and inferring neural networks, but should also be very performant and efficient. Fast, low-level programming languages like C++ often require much more development time to create the same application compared to high-level programming languages like Python. This should also be factored into the development process.

AI frameworks like TensorFlow and PyTorch are available in both Python and C++, but documentation and support is lacking in the C++ versions. This means companies who wish to implement this type of application will need to hire C++ developers who are also experienced with neural networks.

4.5 What is the added value for companies?

Options are always useful, and this project offers a way to solve problems using AI without having to manually collect data. The reinforcement learning agent will collect data for you by playing against itself. This means the algorithm will get better in a "fire-and-forget" manner: just launch a cluster of self-play agents and wait until it gathers enough data. You could create an automatic pipeline to retrain the model after a set amount of time or after a certain amount of games.

4.6 Which alternatives are there?

For chess specifically, there are many chess engines that can be used for both analysis and playing against. StockFish is the most popular, and as mentioned before, Leela Chess Zero is also open source and works the same way as AlphaZero.

For industries other than the gaming industry, many problems are solvable by using machine learning algorithms instead of reinforcement learning solutions. Supervised learning methods have a very wide range of applications, provided you have the data to train a model.

However, if you don't have a means to gather data, there are many reinforcement learning algorithms other than AlphaZero:

- Q-learning
- SARSA
- DQN
- Actor-Critic methods like A2C or A3C
- ...

Currently, sending inputs from the client to the server for predicting outputs with the neural network happens through Python sockets. A good alternative for that is gRPC. It would allow the clients to directly call a method on the server without having to resort to Python sockets, which might be slower. [48]

4.7 Is there a socio-economic impact present?

Sadly, algorithms like AlphaZero fail at solving even relatively simple problems like chess when the model does not have enough data to train on. [39]. Given enough time and data, though, these types of algorithms can be used to achieve superhuman performance in applications that were previously impossible to find optimal solutions for.

DeepMind's ultimate goal is to "[...] solve intelligence, developing more general and capable problem-solving systems, known as artificial general intelligence (AGI)." [49]. This is a slow and extremely difficult process, but DeepMind is confident AlphaZero and its variants are a big step towards that goal [50].

Others are more pessimistic about the progress DeepMind has made [51], [52], explaining that while AlphaGo (Zero) and AlphaZero have certainly been a huge breakthrough in their respective games, this does not mean their results a particularly big step towards AGI.

> I think there is a huge gap between the simple games we now solve and the 'artificial general intelligence' that DeepMind is trying to achieve. I believe that the most progress won't come from algorithms, but from calculations. For general AI, we need incredibly strong simulators where a huge amount of data can be passed through for a very long time. Just like the millions of years of evolution our brains have been through, our brains have been gathering data non-stop throughout our lives. [51]

4.8 Is there opportunity for further research?

While reinforcement learning has existed for a long time, deep reinforcement learning techniques have only recently been used. This is because of the lack of computational resources to train a well-performing model. With high performance computing devices and cloud solutions getting cheaper and more powerful, a lot of research opportunities open up. Deep Reinforcement learning for autonomous driving has recently become an active area of research in both academia and industry [53].

For this project specifically, there is still a lot of research opportunity to try to solve the problem of creating a large enough dataset to train a sufficiently powerful model.

5 Advice

5.1 Introduction

This section gives advice to people who wish to implement something similar to the project. It includes recommendations and tips for when people want to program an application that uses the algorithms and concepts researched in this thesis.

5.2 When should you use this technology?

This algorithm has a very specific purpose. It should only be used in situations where both the action space and state space are discretely defined. Both of these should also be relatively small, as bigger action spaces and state spaces will require much more data to successfully train a model. Therefore, it is important to keep the problem as simple as possible.

5.3 Recommendations

When you want to implement AlphaZero or similar algorithms, it is recommended to prioritize writing code in the most efficient way possible. Every step of the algorithm should be programmed with multiprocessing and multithreading in mind. Furthermore, the choice of programming language should be considered carefully. While most AI frameworks are primarily written for use in Python, and most documentation is written for Python, the language just isn't fast enough for something that needs this much data through self-play. C++ would be a better choice.

As was apparent from the results of the research project, generating training data takes a very long time. A good solution to this problem is to first pretrain the model on a small dataset. This will result in a model that is at least slightly familiar with the environment. Without a pretrained model, the algorithm will start from nothing and make completely random actions. With a pretrained model, the algorithm will have some idea on what decisions to make based on the data it was given in pretraining.

5.4 Tips when programming a similar application

This section will give some useful tips for programming an application that uses AlphaZero's MCTS algorithm.

5.4.1 Creating the tree data structure

The MCTS algorithm requires a tree to be constructed. As mentioned before, the nodes of the tree are the positions of the game, and the edges between those nodes are the actions that can be taken from those positions.

It is important to avoid memory management issues when creating the tree. The MCTS algorithm can create quite large trees, especially when using hundreds of simulations. Every node object should therefore only include the necessary data. For example, a very early version of my MCTS algorithm included all previous moves of the game in every node, creating a huge memory leak. This was fixed by only storing the current board state as a FEN string in each node.

5.4.2 Programming the MCTS algorithm

When programming the MCTS algorithm, it is crucial to make every step as efficient as possible. In the expansion step of the algorithm, the state of the leaf node should be converted to an object that can be used as an input in the neural network. The function that converts the leaf's state can be very slow if it's not programmed efficiently. As that function needs to be called for every simulation, every line of code should be optimized for speed.

Calling the neural network is definitely the slowest step of the algorithm. A good way to take advantage of this is to send batches of data to the network instead of one input at a time. This can be done by creating multiple threads to "crawl" through the tree at the same time during the selection step of the algorithm. Once every thread has finished by finding a leaf node, every leaf node can be sent to the network at once. For every output received by the neural network, a thread can again be started for the third and fourth steps. This will result in a significant speedup. The difficulty of programming this is mainly making sure the "crawlers" during the selection step don't all select the same nodes. That would result in a less explored MCTS tree.

5.4.3 Choosing the right input for the neural network

As mentioned in the technical research part of this thesis, I opted for a smaller input shape than the one defined in AlphaZero's paper. This is because I wanted to make the neural network as fast as possible.

You should carefully consider what data the neural network would need to make a decision, based on the problem you're trying to solve. For example: when Google used AlphaGo to make their data center's temperature control more efficient, they opted to use the sensors in the data center as inputs to the neural network. You should make sure to omit any unnecessary data from the input.

6 Conclusion

This thesis was created to critically evaluate my research project that tried to answer the question “How to create a chess engine using deep reinforcement learning”. The aim of the project was to create a chess engine based on DeepMind’s AlphaZero in Python with TensorFlow Keras. This was attempted by programming the MCTS algorithm and a neural network that can train on data collected from self-play. The ultimate goal of the project was not to create a chess engine that was capable of beating AlphaZero, but rather to create a simple version of AlphaZero that could beat amateur chess players of around 1000 ELO.

Sadly, the resulting chess engine has not managed to reach this goal. This is because of the lack of computational resources necessary to create a big enough dataset through self-play. However, this thesis does offer some solutions to this problem:

- Pretrain the model on a small dataset
- Use docker containers to set up a cluster of self-play agents

This thesis also serves as a warning to people who wish to implement similar algorithms without having the necessary access to powerful hardware. It recommends investigating certain alternatives to AlphaZero, such as other reinforcement techniques like Q-learning, SARSA, and DQN, but also supervised learning techniques if training data is available.

7 Bibliography

[1] L. Rosenbloom, *Top dog: AlphaZero / AlphaStar*, Mar. 2019. [Online]. Available: `https://medium.datadriveninvestor.com/top-dog-alphazero-alphastar-7b2730a6431a` (visited on 04/22/2022).

[2] *Branching Factor - Chessprogramming wiki*. [Online]. Available: `https://www.chessprogramming.org/Branching_Factor` (visited on 03/28/2022).

[3] "Stockfish (chess)," *Wikipedia*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Stockfish_(chess)&oldid=1079146589` (visited on 03/28/2022).

[4] "AlphaZero," *Wikipedia*, Jan. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=AlphaZero&oldid=1065791194` (visited on 02/01/2022).

[5] "Chess engine," *Wikipedia*, Apr. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Chess_engine&oldid=1080874516` (visited on 04/05/2022).

[6] "Minimax," *Wikipedia*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Minimax&oldid=1076761456` (visited on 04/05/2022).

[7] M. Eppes, *How a Computerized Chess Opponent "Thinks" — The Minimax Algorithm*, Oct. 2019. [Online]. Available: `https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1` (visited on 04/05/2022).

[8] "Evaluation function," *Wikipedia*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Evaluation_function&oldid=1079533564` (visited on 04/06/2022).

[9] "Alpha–beta pruning," *Wikipedia*, Jan. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=1068746141` (visited on 02/01/2022).

[10] *Minimax and Monte Carlo Tree Search - Philipp Muens*. [Online]. Available: `https://philippmuens.com/minimax-and-mcts` (visited on 04/06/2022).

[11] "Monte Carlo tree search," *Wikipedia*, Apr. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Monte_Carlo_tree_search&oldid=1081107255` (visited on 04/06/2022).

[12] *ML | Monte Carlo Tree Search (MCTS)*, Jan. 2019. [Online]. Available: `https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/` (visited on 04/07/2022).

[13] "Go (game)," *Wikipedia*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Go_(game)&oldid=1079941654` (visited on 04/06/2022).

[14] "DeepMind," *Wikipedia*, Feb. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=DeepMind&oldid=1072182749` (visited on 04/07/2022).

[15] "AlphaGo," *Wikipedia*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=AlphaGo&oldid=1077595428` (visited on 04/07/2022).

[16] *AlphaGo*. [Online]. Available: `https://www.deepmind.com/research/highlighted-research/alphago` (visited on 04/07/2022).

[17] *Mastering the game of Go with Deep Neural Networks & Tree Search*. [Online]. Available: `https://www.deepmind.com/publications/mastering-the-game-of-go-with-deep-neural-networks-tree-search` (visited on 04/07/2022).

[18] "AlphaGo Zero," *Wikipedia*, Feb. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=AlphaGo_Zero&oldid=1073216893` (visited on 04/08/2022).

[19] *Mastering the game of Go without Human Knowledge*. [Online]. Available: `https://www.deepmind.com/publications/mastering-the-game-of-go-without-human-knowledge` (visited on 04/07/2022).

[20] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *arXiv:1712.01815 [cs]*, Dec. 2017. arXiv: `1712.01815 [cs]`. [Online]. Available: `http://arxiv.org/abs/1712.01815` (visited on 02/01/2022).

[21] *Neural Networks - Chessprogramming wiki*. [Online]. Available: `https://www.chessprogramming.org/Neural_Networks#ANNs` (visited on 04/07/2022).

[22] "Tensor Processing Unit," *Wikipedia*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Tensor_Processing_Unit&oldid=1077688658` (visited on 04/07/2022).

[23] "Application-specific integrated circuit," *Wikipedia*, Feb. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Application-specific_integrated_circuit&oldid=1074023806` (visited on 04/07/2022).

[24] "Leela Chess Zero," *Wikipedia*, Apr. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Leela_Chess_Zero&oldid=1080962634` (visited on 04/08/2022).

[25] "Leela Zero," *Wikipedia*, Oct. 2021. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Leela_Zero&oldid=1049955001` (visited on 04/08/2022).
[26] *Lc0*, LCZero, Apr. 2022. [Online]. Available: `https://github.com/LeelaChessZero/lc0` (visited on 04/08/2022).
[27] *Python-chess: A chess library for Python — python-chess 1.9.0 documentation.* [Online]. Available: `https://python-chess.readthedocs.io/en/latest/` (visited on 04/08/2022).
[28] "Forsyth–Edwards Notation," *Wikipedia*, Feb. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Forsyth%E2%80%93Edwards_Notation&oldid=1069540048` (visited on 04/09/2022).
[29] *Graphviz.* [Online]. Available: `https://graphviz.org/` (visited on 04/09/2022).
[30] S. Bodenstein, *AlphaZero |*, Sep. 2019. [Online]. Available: `https://sebastianbodenstein.net/post/alphazero/` (visited on 04/09/2022).
[31] *Lichess.org open database.* [Online]. Available: `https://database.lichess.org/` (visited on 04/09/2022).
[32] A. Adefokun, *Chess-board ahira-justice*, Feb. 2022. [Online]. Available: `https://github.com/ahira-justice/chess-board` (visited on 04/10/2022).
[33] *PyGame.* [Online]. Available: `https://www.pygame.org/news` (visited on 04/10/2022).
[34] zjeffer, *Chess-board zjeffer*, Jan. 2022. [Online]. Available: `https://github.com/zjeffer/chess-board` (visited on 04/10/2022).
[35] A. Adefokun, *Chess-board pul request*, Feb. 2022. [Online]. Available: `https://github.com/ahira-justice/chess-board/pull/5` (visited on 04/10/2022).
[36] zjeffer, *Chess engine with Deep Reinforcement learning*, Feb. 2022. [Online]. Available: `https://github.com/zjeffer/chess-deep-rl` (visited on 04/09/2022).
[37] "First-move advantage in chess," *Wikipedia*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=First-move_advantage_in_chess&oldid=1080096428` (visited on 04/10/2022).
[38] zjeffer, *Chess-deep-rl-cpp*, Apr. 2022. [Online]. Available: `https://github.com/zjeffer/chess-deep-rl-cpp` (visited on 04/10/2022).
[39] *Thomas Moerland – Postdoc, Leiden University, The Netherlands.* [Online]. Available: `https://thomasmoerland.nl/` (visited on 04/13/2022).
[40] "Top Chess Engine Championship," *Wikipedia*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Top_Chess_Engine_Championship&oldid=1077985240` (visited on 04/12/2022).
[41] S. R. |. 2. December and 2018, *Has Google cracked the data center cooling problem with AI?* May 2020. [Online]. Available: `https://techwireasia.com/2020/05/has-google-cracked-the-data-centre-cooling-problem-with-ai/` (visited on 04/13/2022).
[42] *How AI helps better manage and run data centers*, Dec. 2018. [Online]. Available: `https://techwireasia.com/2018/12/how-ai-helps-better-manage-and-run-data-centers/` (visited on 04/14/2022).
[43] "Reinforcement learning," *Wikipedia*, Apr. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Reinforcement_learning&oldid=1081715019` (visited on 04/14/2022).
[44] nbro, *Answer to "Are there any applications of reinforcement learning other than games?"* Nov. 2020. [Online]. Available: `https://ai.stackexchange.com/a/24355/54037` (visited on 04/13/2022).
[45] N. T. Blog, *Artwork Personalization at Netflix*, Dec. 2017. [Online]. Available: `https://netflixtechblog.com/artwork-personalization-c589f074ad76` (visited on 04/13/2022).
[46] P. Surmenok, *Contextual Bandits and Reinforcement Learning*, Oct. 2017. [Online]. Available: `https://towardsdatascience.com/contextual-bandits-and-reinforcement-learning-6bdfeaece72a` (visited on 04/13/2022).
[47] *Cloud Computing Services | Google Cloud.* [Online]. Available: `https://cloud.google.com/` (visited on 04/13/2022).
[48] *Introduction to gRPC.* [Online]. Available: `https://grpc.io/docs/what-is-grpc/introduction/` (visited on 04/15/2022).
[49] *DeepMind | About.* [Online]. Available: `https://www.deepmind.com/about` (visited on 04/13/2022).
[50] *AlphaZero: Shedding new light on chess, shogi, and Go.* [Online]. Available: `https://www.deepmind.com/blog/alphazero-shedding-new-light-on-chess-shogi-and-go` (visited on 04/13/2022).
[51] *Email exchange between Tuur Vanhoutte and Dr. Thomas Moerland*, Apr. 22.

[52] DukeZhou, *Answer to "Is AlphaZero an example of an AGI?"* Nov. 2018. [Online]. Available: `https://ai.stackexchange.com/a/9170/54037` (visited on 04/13/2022).

[53] “Deep reinforcement learning,” *Wikipedia*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Deep_reinforcement_learning&oldid=1078792888` (visited on 04/14/2022).

8 Appendix

8.1 Report guest speaker: ML6

8.1.1 Introduction

On the 20th of January, I attended a very interesting talk by Matthias Feys, CTO of ML6. ML6 is an AI consultancy company based in Belgium, Germany, The Netherlands and Switzerland that serves customers across Europe. They have more than 90 machine learning & data engineering experts and are the largest and fastest growing AI company in Europe.

The talk was about how the company tries to increase the explainability of their AI solutions.

8.1.2 What is Explainable AI?

The inner workings of how an AI gets to a decision is often very difficult to explain. Say you have some inputs and a neural network that takes those inputs and produces one output: Yes or No. Without explainability, you can't tell the customer why the network makes the decision. The customer just has to take your word for it.

An improvement to the above example is instead of outputting a boolean value, a percentage could be used instead. This would allow the customer to understand how sure the network is of its decision.

While AI technology is improving extremely quickly, adoption of these technologies by other companies is often still a hurdle to overcome. Here are three scenarios in which Explainable AI can drive adoption. All of these scenarios have the goal of building trust in the AI.

1. AI is weaker than human
2. AI is on par with human
3. AI is stronger than human

8.1.3 AI is weaker than human

In this scenario, adoption is very difficult, which is why explainability will be primarily about improving the AI. By explaining how the AI makes a decision, the customer can understand how they can help improve it.

You can visualize the outputs of a model. For example: drawing lines and bounding boxes around the output of a neural network is a great way to show what the contribution of the model is.

You can improve the model by capturing lots of data, and either letting the customer label the data themselves, or by fixing incorrect labels. This is called Active Learning, and it focuses on the data where the model is weak. While waiting for a model to improve, you can manually create a simpler model that can solve the specific edge case that went wrong before.

8.1.4 AI is on par with human

When the AI is as good as the human counterpart, you want to build trust in the AI by discussing how it works. It's difficult to convince someone to consider using the AI over the human solution if they are equally valid options.

In this scenario, explainability can help humans to use the result of the AI to their advantage. You're basically letting the AI and the human work together to solve a problem.

8.1.5 AI is stronger than human

Stronger than human performance is great, but it's better if you can also explain why it's the better option. Explainability here is about explaining a complicated concept and informing humans.

The information you can learn from an AI is incredibly useful to companies in order to improve their machines, processes, and products.

8.1.6 Five generic design patterns

Here are 5 tips that are useful for explaining AI.

1. Problem reframing
2. Interpretable models
3. Feature attribution
4. Transparency & transferability
5. Intuitive visualizations

Problem reframing

It's often better to completely reframe the problem before you start to think about programming an AI to solve it. For example: when a customer asks to classify whether a car is damaged or not, you can reframe the problem as an object detection problem. An object detector can then be trained to detect specific locations of damage on the car. This still answers the customer's initial needs, but is much easier to explain.

Interpretable models

Some models are generally more interpretable than others. Linear models, for example, can be very easy to interpret, but these aren't very performant. LSTMs on the other hand, can be very performant, but they are hard to interpret. It's crucial to find a correct balance between both performance and interpretability. AI architectures can be combined to solve a problem in a way that is both performant and easy to understand.

Feature attribution

It's important to understand which features are contributing the most or the least to the output of the model. For example: if you have a model that recommends a product, you can show which features were had the most impact when recommending that specific product. This can be done with techniques like SHAP values, saliency maps or LIME.

Transparency & transferability

If the data changes drastically, for example due to a crisis like the coronavirus, the model might not be relevant anymore. If a model uses another model, it is important that the first model is explainable as well. Validation of data is also very important: if the data is filled with mistakes, the output can not be trusted. That is why it's necessary to run a standard set of checks on the data to uphold the quality of the dataset.

Intuitive visualizations

For example, when a model recommends a product, a visual representation can be shown of other products that are similar to the recommended product, from most similar to least similar. This gives the customer a good sense of how it ranks products to your preferences.

For computer vision, saliency maps can show the importance of each pixel in the image when making a decision.

8.1.7 Conclusion

The main conclusion of the talk was that explainable AI is not just about using SHAP to explain the features, but it is about much more than that. Explainability is not just part of developing a model, it is present in the whole pipeline from start to finish: it starts with the framing of the problem and ends with the user interface and user experience.

The goal is to increase trust and adoption.

8.1.8 Critical reflection

I thought it was a very interesting talk about a problem that isn't always given adequate attention in the AI community. It is also something I am personally guilty of: when developing an AI, explainability is one of the last things I think of, when it should be part of the developing process from the start. For example, my bachelor thesis about creating a chess engine using deep reinforcement learning is a project where almost no explainability is possible. This makes developing the AI very difficult, as it's not clear why the model makes its decisions.

I was particularly intrigued by the way saliency maps can be used to explain the importance of groups of pixels in an image [1]. This way, mistakes that a model makes can be easily visualized: if a group of pixels is deemed as important by the model, but it's a false positive, actions can be taken to improve the model based on that edge case.

8.1.9 Sources

[1] "Saliency map" Wikipedia, Oct. 2021. [Online]. Available: https://en.wikipedia.org/w/index. php?title=Saliency_map&o
(visited on 04/16/2022)

8.2 Installation manual

8.2.1 System requirements

- An Nvidia GPU with at least 3GB of VRAM
- A good CPU that can handle the MCTS algorithm
- At least 5GB of free RAM

8.2.2 Python packages (for local/non-dockerized use)

1. Install the latest version of Python (I used Python 3.10).
2. In a terminal, run the following commands:

```
python3 -m pip install pip --upgrade
python3 -m pip install -r requirements.client.text
python3 -m pip install -r requirements.server.text
```

8.2.3 Docker

The software comes with two Docker images and a docker-compose file. This is used to create a training set using self-play. To make sure the GPU can be used inside the server's Docker container, follow these instructions to install Docker and the Docker utility engine:

`https://docs.nvidia.com/ai-enterprise/deployment-guide/dg-docker.html`

Using these images, it is possible to deploy a whole cluster of servers and clients, to create a training set in parallel.

8.3 User manual

8.3.1 Introduction

I made a chess engine that uses deep reinforcement learning to play moves. It is possible to host the AI model on a GPU-equipped server, or host the model locally.

This software has multiple features:

- Create a training set using a specific AI model, by playing chess games against itself.
- Deploy a whole cluster of servers and clients to create a training set in parallel.
- Train the network using a training set created by self-play.
- Evaluate two different AI models by playing a given number of games against each other.
- Play against an AI

To create a training set, you first need to either create your own AI model, or use my pretrained model.

8.3.2 Create your own AI model

To create your own AI model, run the following command:

```
python rlmodelbuilder.py --model-folder <FOLDER> --model-name <NAME>

# For a detailed description of the parameters, run:
python rlmodelbuilder.py --help
```

If you want to change parameters like the number of hidden layers, the input and output shapes (if you want to use the AI for other games), or the amount of convolution filters, change values in the config.py file.

8.3.3 Use my pretrained model

You can find a link to my pretrained model in the README.md file in the repository. Copy the model.h5 file to the 'models/' folder.

8.3.4 Create a training set using the chosen model

There are two ways to create a training set:

- Play full chessgames with one model playing white, and a copy of the model playing black.
- Solve chess puzzles using the AI model.

The first method can easily be deployed using the docker-compose file:

```
# in repository's code/ folder:
docker-compose up --build
```

This will deploy one prediction server and 8 clients ⇒ 8 parallel games will play. The data for the training set will be stored in ./memory. The amount of clients can be changed in the docker-compose file. You can also use the two docker images in a cluster like Kubernetes, to reliably deploy many more servers and clients in parallel.

Using the created GUI, it is possible to visualize the self-play of all these boards in real-time. This can be done by settings the "SELFPLAY_SHOW_BOARD" environment variable in docker-compose.yml to "true". For every replica of the client, a new PyGame window will open where the board will change in real-time.

To manually run self-play or to solve puzzles, you can run the selfplay.py file with specific arguments. For a detailed description of the arguments, run:

```
python3 selfplay.py --help
```

8.3.5 Manual (non-dockerized) self-play with full games

```
# if you want to send predictions to a server, start the server first:
python3 server.py
```

```
# if you want to predict locally instead of on a server, add --local-predictions:
python3 selfplay.py --local-predictions
```

The games will be saved in the './memory' folder.

8.3.6 Solving puzzles (non-dockerized)

You can download the puzzles file here: `https://database.lichess.org/#puzzles`. This is a .csv file consisting of more than 2 million chess puzzles, with many different types ("mateIn1", "mateIn2", "endgame", "short", etc...).

Training with these puzzles is faster than training with full games, and the AI can more easily learn patterns like mating moves, or other patterns that are difficult to solve.

```
python3 selfplay.py --type puzzles \
    --puzzle-file <PATH-TO-CSV-FILE> \
    --puzle-type <TYPE>
```

Just like with self-play, you can add --local-predictions if you want to predict locally instead of on a server. If the puzzle ends in a checkmate within the move limit, the game will be saved to memory. That is why for now, only puzzles of type "mateInX" are supported. The puzzle move limit can be changed in the config.py file.

8.3.7 Training the AI using a created training set

```
python3 train.py \
    --model <PATH-TO-MODEL> \
    --data-folder <PATH-TO-TRAINING-SET-FOLDER>

# For a full description of the parameters, run:
python3 train.py --help
```

You can change parameters like batch size and learning rate in the config.py file.

8.3.8 Evaluating two models

```
python3 evaluate.py evaluate.py <model_1> <model_2> <nr_games>

# For example, to run 100 matches between two models, run:
python3 evaluate.py models/model_1.h5 models/model_2.h5 100

# For a full description of the parameters, run:
python3 evaluate.py --help
```

The white player in chess inherently has a slightly higher chance of winning, because they can play the first move. Therefore, to evaluate two models, each model has to play as white and black an equal amount of games.

Therefore, running n matches means the evaluation will consist of $2 \cdot n$ games, because each model will play both colors once per match.

The output of this command will be an overview of the results of the matches:

```
Evaluated these models for 100 matches:
    Model 1 = models/model_1.h5, Model 2 = models/model_2.h5
The results:
Model 1: X wins
Model 2: X wins
Draws: X
```

8.3.9 Playing against the AI

To play against the AI, you can run the following command:

```
# [brackets] indicate optional arguments
python3 main.py [--player <white|black>] \
    [--local-predictions] \
    [--model <PATH-TO-MODEL>]
```

- If you don't specify a player, a random side will be chosen for you.
- If you add --local-predictions, you also have to specify a model.

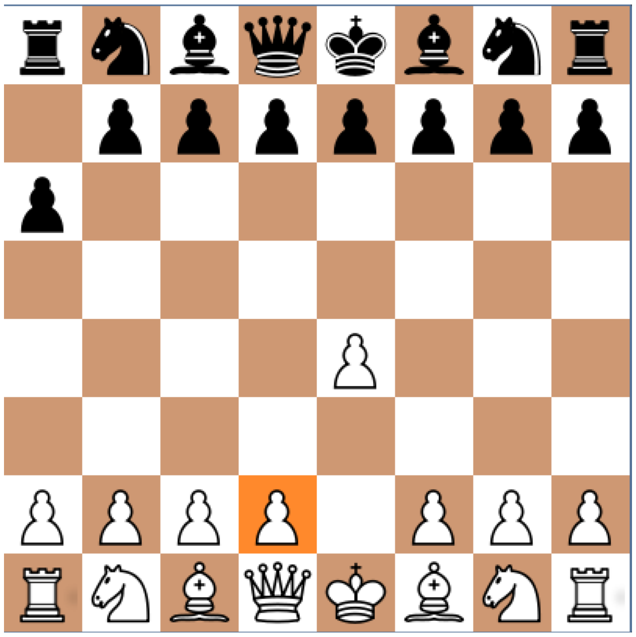

Figure 23: The chessboard GUI

- You can click on a piece to move it, then click a destination square
- When you click a piece, the square lights up to show it is selected

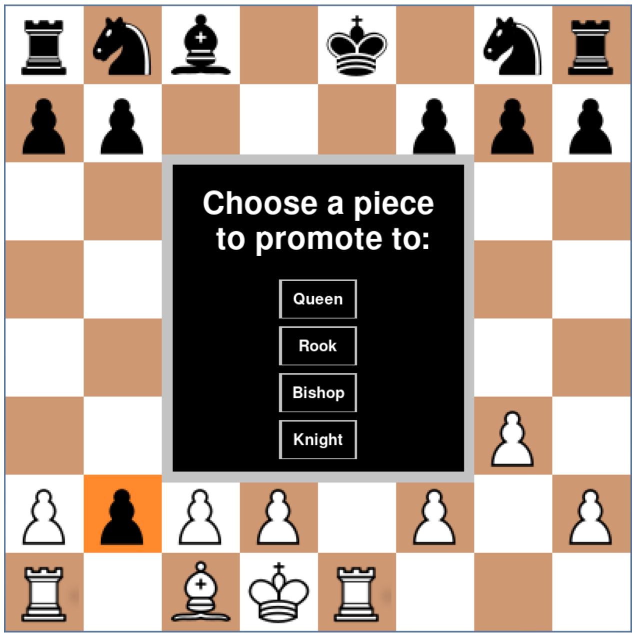

Figure 24: Promoting a pawn

8.3.10 Changing the AI difficulty

- You can change the difficulty of the AI by changing the amount of MCTS simulations per move in the config.py file.
- This will also change the amount of time it takes for the AI to make a move.