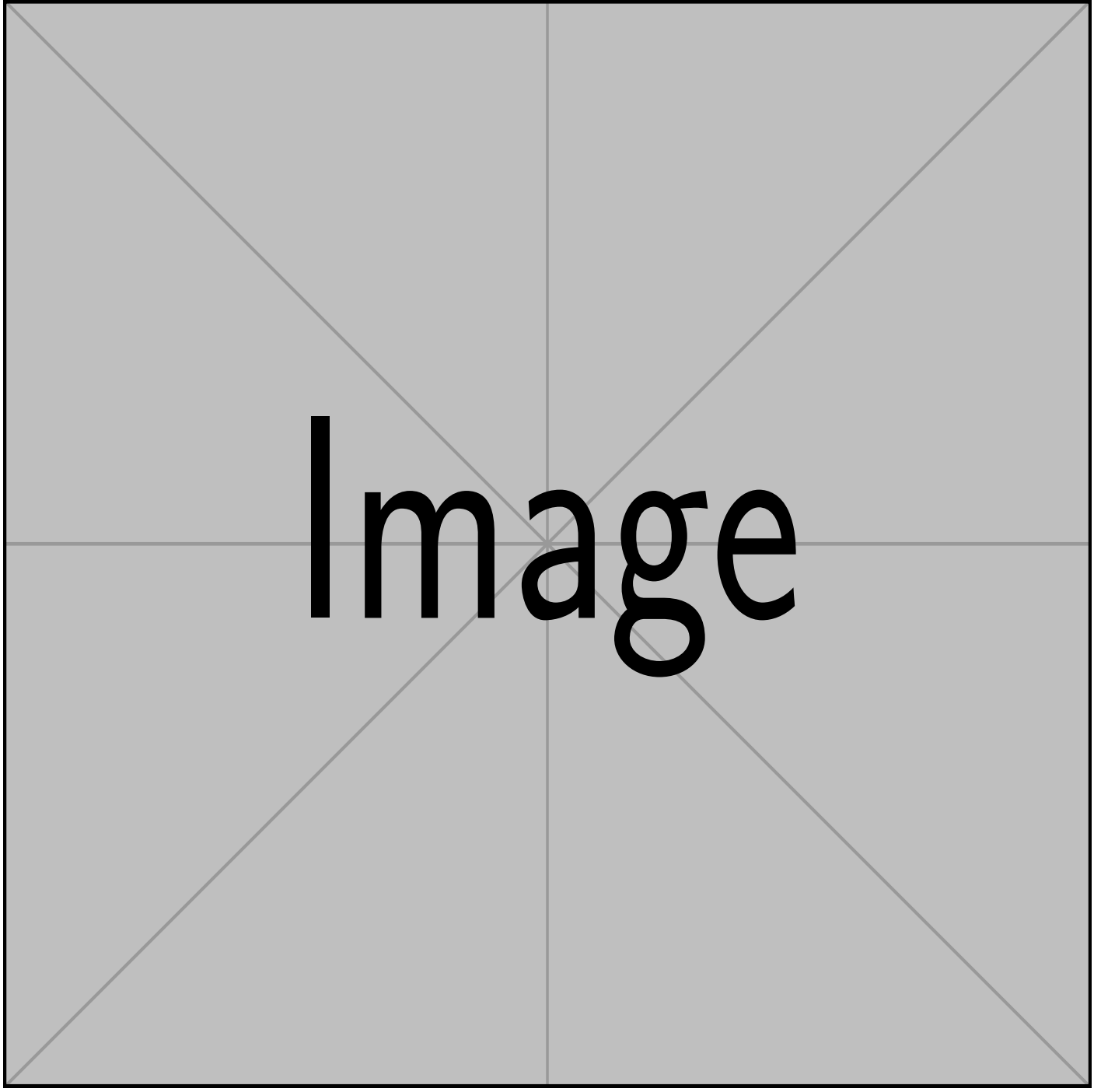

HOW TO CREATE A CHESS ENGINE USING DEEP REINFORCEMENT LEARNING

A CRITICAL LOOK AT DEEPMIND'S ALPHAZERO

INTERNAL PROMOTOR: WOUTER GEVAERT

EXTERNAL PROMOTOR: <NAME HERE>

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 to program the chess engine and I will reflect on the results of the project. To do this, I will contact multiple people 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 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, 1st June 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. With this comes a section on advice for companies that wish to implement similar algorithms.

Contents

List of figures

List of abbreviations

Glossary

1 Introduction

Chess is not only one of the 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 [1], 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 [2], which uses the minimax algorithm with alpha-beta pruning to calculate the best move.

More recently, researchers at Google DeepMind have developed a new algorithm called AlphaZero [3]. 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. This was done as part of a research project between November 2021 and January 2022. The code was written with lots of trial and error, as DeepMind released very little information about the detailed workings of the algorithm.

2 Research

2.1 What is a chess engine?

According to Wikipedia [4], 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 [2], use a variant of the minimax algorithm that employs alpha-beta pruning.

2.2.1 The minimax algorithm

The minimax algorithm [5] 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 [6], 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:

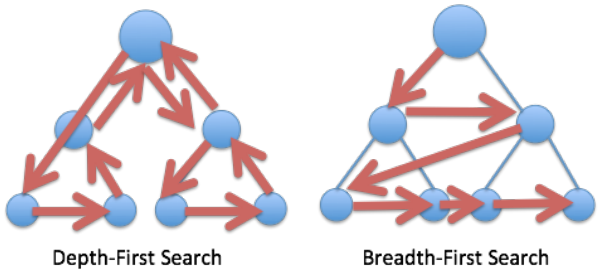

Figure 1: Depth-First search vs Breadth-First search [6]

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 [7] 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. [5]

```
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 necessary amount of nodes to get a good estimation of the strength of a position is so high, the algorithm needs to be optimized. Alpha-beta pruning [8] 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 always 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.

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 [9].

Monte Carlo Tree Search (MCTS) [10] 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

To create this search tree for a certain position, MCTS will run the following algorithm hundreds of times. Each of these runs is called an MCTS **simulation**. One simulation consists of four steps:

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) [11]
   - 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.

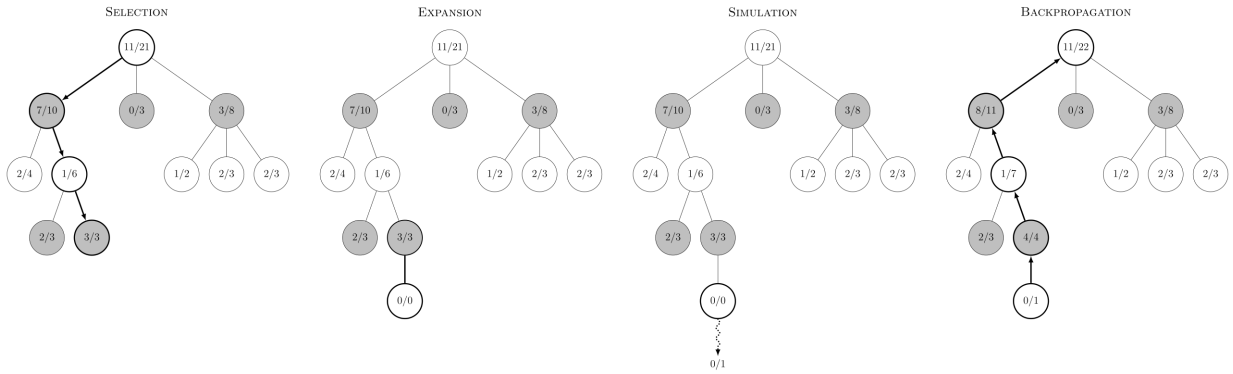

Figure 2: The 4 steps of the MCTS algorithm [10]

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 [12]. It is played on a rectangular grid of (usually) 19 by 19 lines. The rules are relatively simple, but due to its

extremely complex possibilities, 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 3: Go board [12]

2.4.1 AlphaGo

In 2014, DeepMind Technologies [13], a subsidiary of Google, started developing a new algorithm called AlphaGo to play Go [14]. Previously, the strongest Go engines were still only good enough to win against amateur Go players [15]. The algorithm used a combination of the MCTS algorithm and a deep neural network to evaluate positions.

AlphaGo was built [15], [16] 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 recording the 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 [17]. AlphaGo Zero uses a different kind of network than AlphaGo. Instead of using two separate networks, it will combine the two outputs into a policy head and a value head. It's also using different layers: residual layers instead of convolutional layers [18].

2.5 AlphaZero

AlphaZero is a generalized version of AlphaGo Zero, created to master the games of chess, shogi ("Japanese chess"), and Go [3], [19]. 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 (white and black can both 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 119 8x8 boards of boolean values.

2.5.2 Neural network layers

DeepMind tested multiple neural network architectures for AlphaGo Zero [20]. 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 [18]:

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 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 dataset. This is done by letting the engine play against itself for a high amount of matches. Every move, data is collected and stored in a 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) [21]. A TPU is an application-specific integrated circuit (ASIC [22]) 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 [23]. Lc0 was adapted from Leela Zero [24], a Go computer that attempted to replicate the results AlphaGo Zero [17].

It is written in C++ [25], 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 [26].

3.2 Class structure

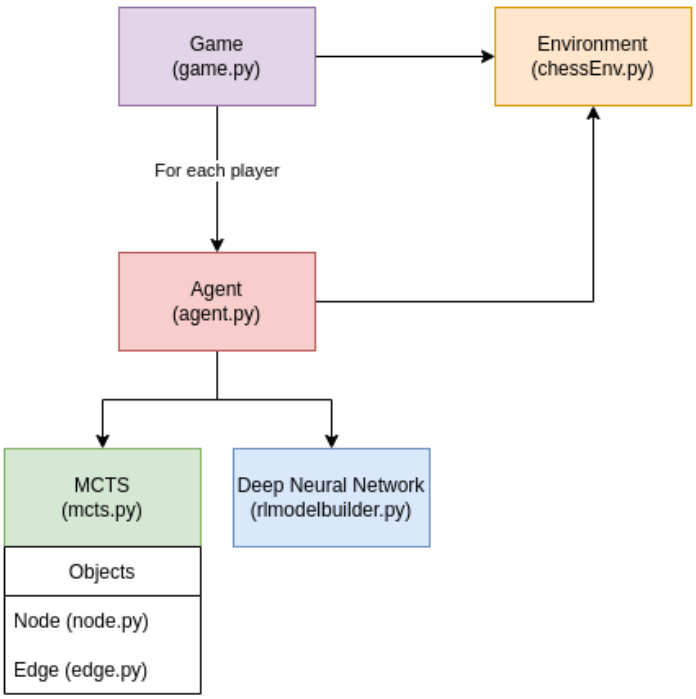

Figure 4: Basic class structure for the code responsible for playing a game

3.2.1 Making one move

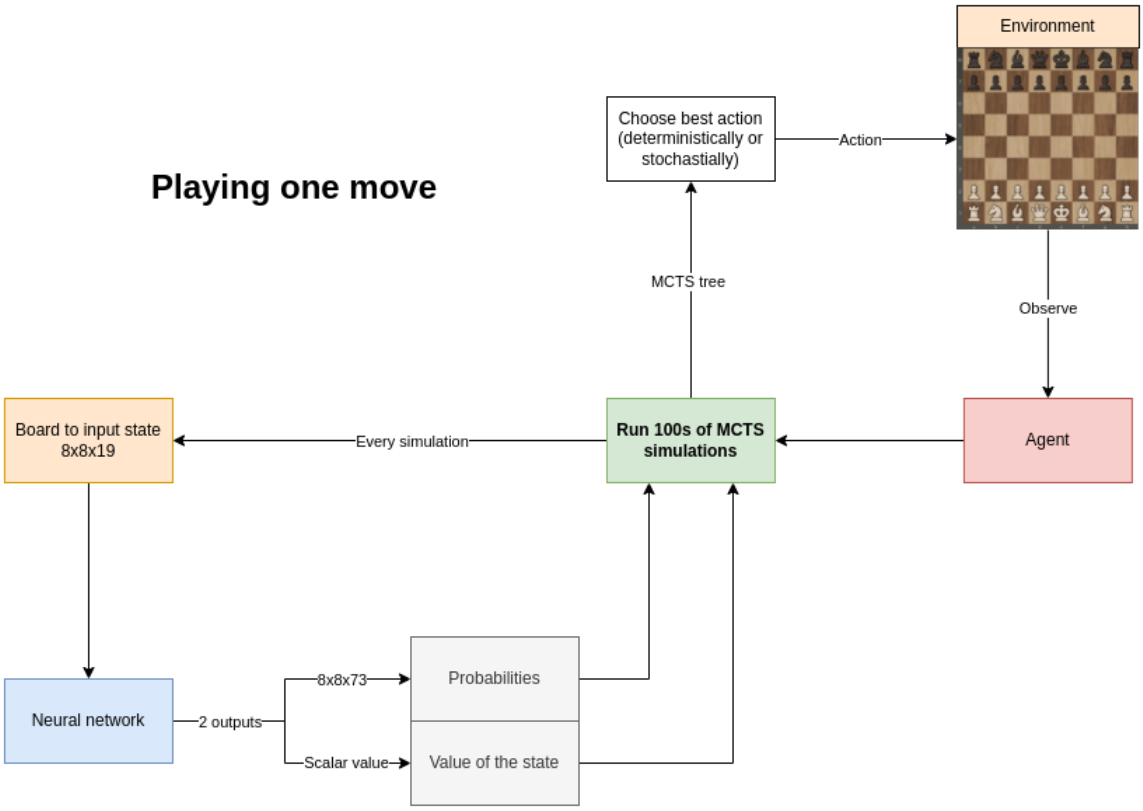

Figure 5: Pipeline to make one move

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.

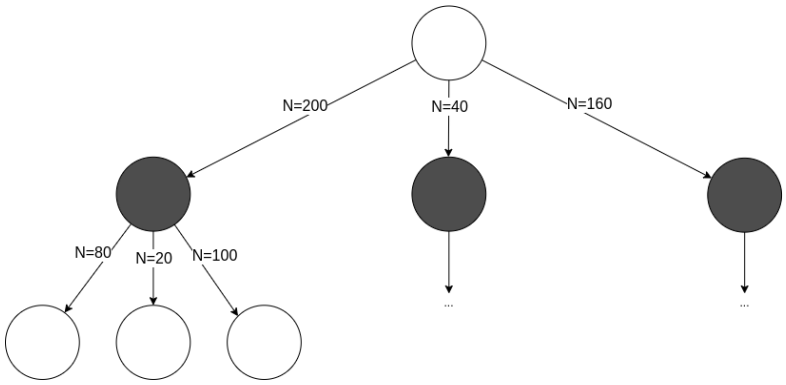

Figure 6: Example tree after 400 simulations. N = amount of times the selection step selects that edge

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) [27]
- 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 [28]. A recursive function was written to create the tree and output it to an SVG file.

3.5 The MCTS algorithm

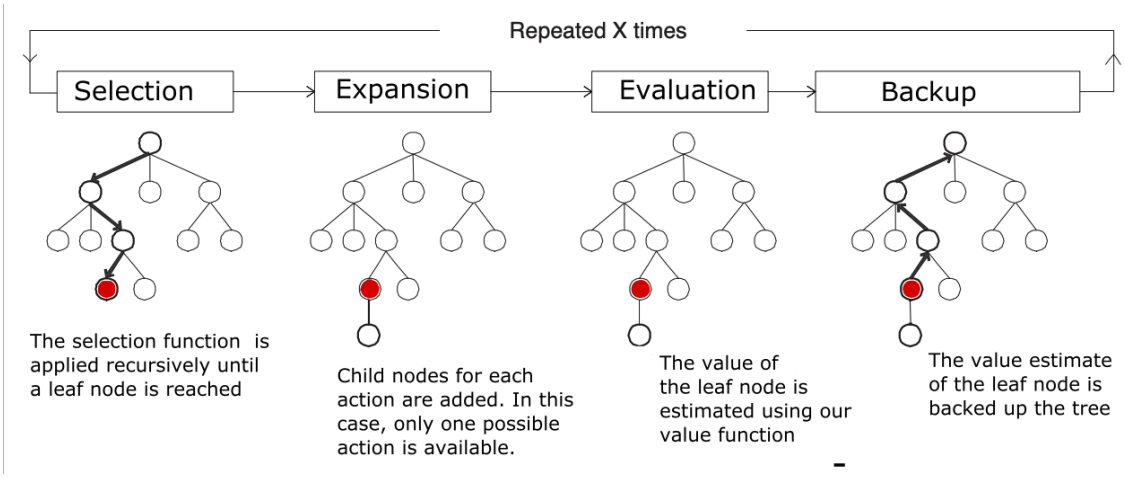

Figure 7: The four steps in AlphaZero's MCTS algorithm [29]

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 [19] 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. 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. This policy is 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 8: A subset of the 73 8x8 planes from the policy output. The brighter the pixel, the better the move. Grey padding was added to make a better visual presentation of the output planes.

Figure 9: 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:

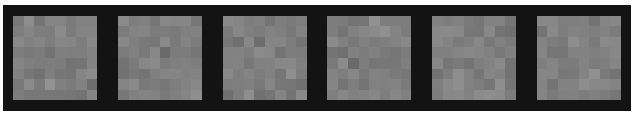

Figure 10: Some output planes from an untrained model. Black padding was used instead of gray to make it clearer.

This shows it is clear the trained model has some understanding of which moves are legal, without giving it any knowledge of the rules of chess.

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

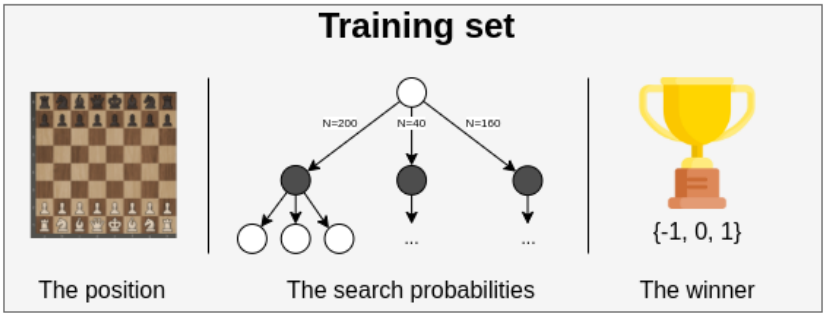

Figure 11: The training set consists of: the input state, the move probabilities, and the eventual winner

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 added 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. The idea is to let the agent play from a certain given position (the puzzle), instead of from the start of the game, and see if the agent can find the correct solution. This would in theory give the model a better understanding of common chess tactics, without having to play whole games from start to finish.

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 [30]. 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

When creating the dataset, mate-in-1 and mate-in-2 puzzles were also added to the dataset. 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.

3.7 Training the neural network

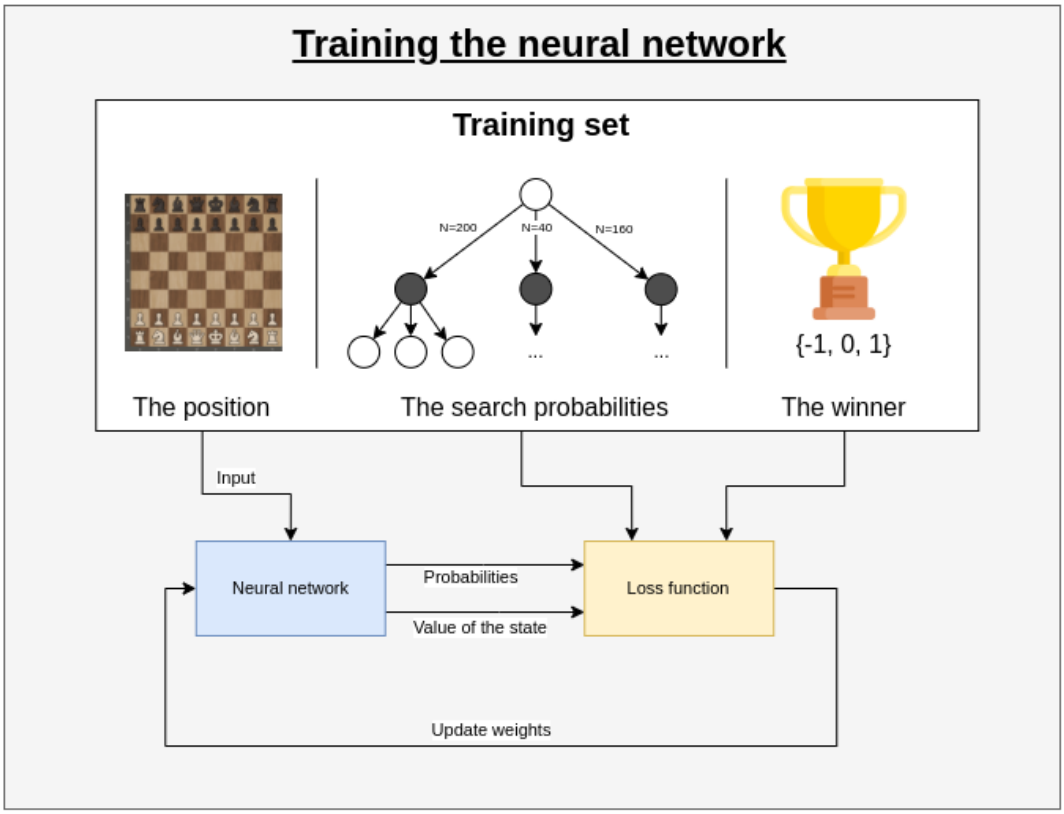

Figure 12: The training pipeline

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

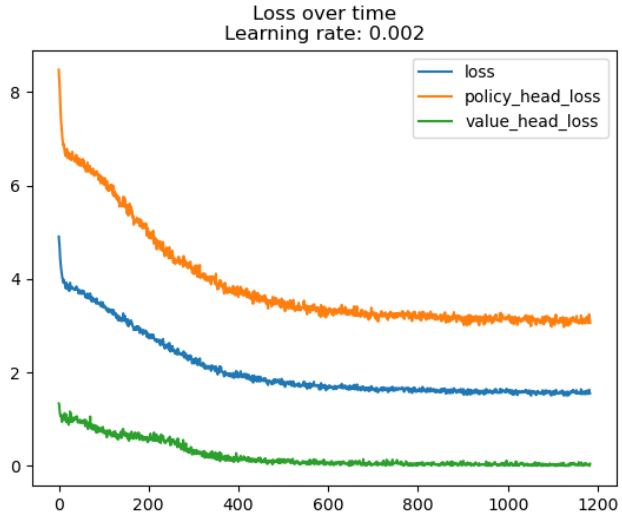

Figure 13: First training session

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, and a batch size of 64. The model uses the Adam optimizer.

After training, the model was saved to disk.

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. This time, due to time constraints, the dataset only had 50,000 positions.

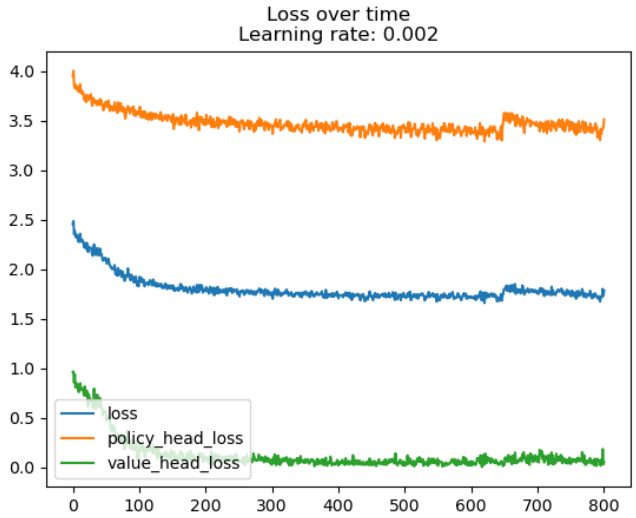

Figure 14: Second training session

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.

3.8.1 Without multiprocessing

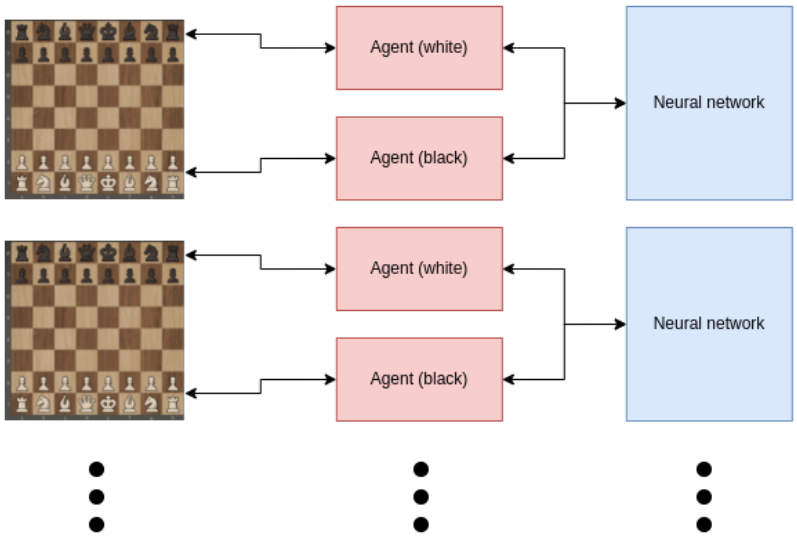

Figure 15: Self-play without multiprocessing

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

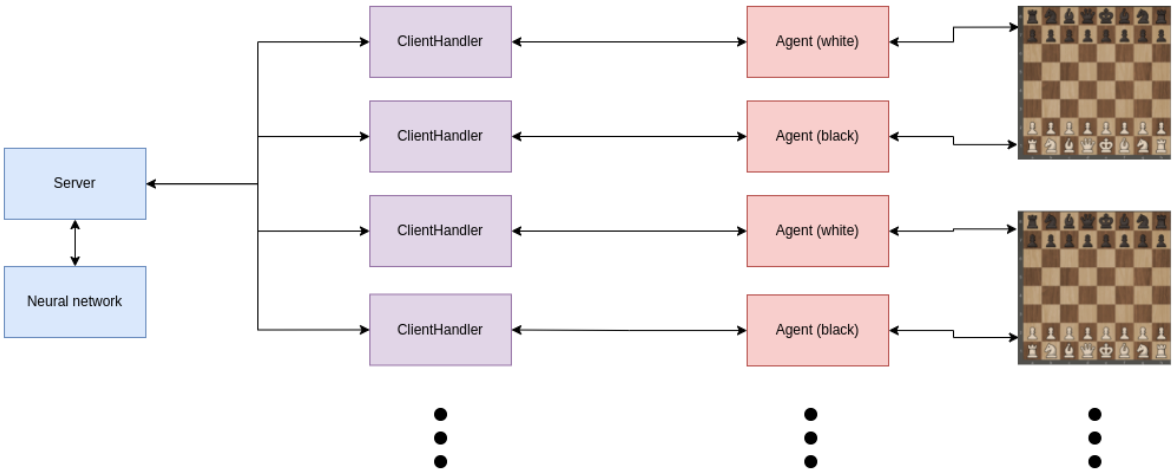

Figure 16: Self-play with multiprocessing

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 [31], [32]. 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 [33]. 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 [34].

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 17: 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 18: 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.

With a docker-compose file, the server and client images can be easily started. The number of clients to run in parallel can be easily configured in that file.

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 project [35].

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.

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 [36]. 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 [37]. 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

5 Advice

6 Conclusion

7 Bibliography

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

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

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

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

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

[6] 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).

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

[8] "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).

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

[10] "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).

[11] *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).

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

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

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

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

[16] *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).

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

[18] *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).

[19] 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).

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

[21] "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).

[22] "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).

[23] "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).

[24] "Leela Zero," *Wikipedia*, Oct. 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Leela_Zero&oldid=1049955001 (visited on 04/08/2022).

[25] *Lc0*, LCZero, Apr. 2022. [Online]. Available: https://github.com/LeelaChessZero/lc0 (visited on 04/08/2022).
[26] *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).
[27] "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).
[28] *Graphviz*. [Online]. Available: https://graphviz.org/ (visited on 04/09/2022).
[29] S. Bodenstein, *AlphaZero |*, Sep. 2019. [Online]. Available: https://sebastianbodenstein.net/post/alphazero/ (visited on 04/09/2022).
[30] *Lichess.org open database*. [Online]. Available: https://database.lichess.org/ (visited on 04/09/2022).
[31] A. Adefokun, *Chess-board ahira-justice*, Feb. 2022. [Online]. Available: https://github.com/ahira-justice/chess-board (visited on 04/10/2022).
[32] *PyGame*. [Online]. Available: https://www.pygame.org/news (visited on 04/10/2022).
[33] zjeffer, *Chess-board zjeffer*, Jan. 2022. [Online]. Available: https://github.com/zjeffer/chess-board (visited on 04/10/2022).
[34] A. Adefokun, *Chess-board pul request*, Feb. 2022. [Online]. Available: https://github.com/ahira-justice/chess-board/pull/5 (visited on 04/10/2022).
[35] zjeffer, *Chess engine with Deep Reinforcement learning*, Feb. 2022. [Online]. Available: https://github.com/zjeffer/chess-deep-rl (visited on 04/09/2022).
[36] "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).
[37] zjeffer, *Chess-deep-rl-cpp*, Apr. 2022. [Online]. Available: https://github.com/zjeffer/chess-deep-rl-cpp (visited on 04/10/2022).

8 Appendix