BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

Integration of Neural Networks and Minimax Strategies in Chess Engine Development

Supervisor
Asist. univ. dr. Florentin Bota

Author
Cadar Eduard

2023

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENŢĂ

Integrarea Reţelelor Neuronale şi a Strategiilor Minimax în Dezvoltarea unui Motor de Şah

Conducător ştiinţific
Asist. univ. dr. Florentin Bota

Absolvent
Cadar Eduard

2023

ABSTRACT

The usual way of searching for a move from a chess position is through tactics or strategy. Tactics are series of moves that would bring an immediate advantage, while strategy refers to a general 'sense' of advantage on the board, without too much calculation (having better developed pieces, better pawn structure etc.) [10].

Chess engines are usually built using a minimax algorithm, along with an evaluation function that takes into consideration the positions of the pieces on the board. This approach is able to find the moves that would bring material gain (capturing pieces) in a given depth, but cannot find moves that would slowly improve the position. Another approach would be training a neural network on professional chess games, which would be able to find good positional moves, but may be weak at finding tactical moves.

An algorithm combining these two approaches should yield better results than each of them on their own.

Contents

List of Figures

Chapter 1

Introduction

Despite being a centuries-old game, chess still remains unsolved, and is one of the most extensively researched areas of artificial intelligence. Due to the complexity of the game and the large number of possible positions, conventional methods of computing and searching for the best move are ineffective and produce unsatisfactory results, necessitating the use of heuristics in search, evaluation, and move selection.

Chess needs so much creativity and profound reasoning that it was originally considered that computers would never be able to perform it. And it remained that way for a long time, until IBM's Deep Blue [1] defeated reigning World Chess Champion Garry Kasparov in 1997, winning 2 matches, drawing 3, and losing 1. Since then, the best chess engines have improved to the point that no human is able to win a single match against them.

There are several properties of the chess game that make it ideal for computers:

- Deterministic - all potential games result in either a win for white, a win for black, or a draw
- Finite - games cannot continue indefinitely (there are two restrictions to avoid this: a draw occurs after three repetitions of the same position, and 50 moves without a capture or a pawn move is also a draw)
- Complete information - there is no concealed information or ambiguity as there is in a card game, and the game is played sequentially; both players have access to the same information
- Zero-sum - the goals of the competitors are opposite: a win for one player is a loss for his opponent, and a draw results in a game sum of zero. If a position is worth +10 to one player, then the opponent's score is -10

In the next chapter some of the techniques and algorithms used in programming chess engines are described, as well as some of the optimization methods that modern chess engines utilize. In the third chapter an introduction is given to the state of

the art in chess engines and how an engine's strength is determined. In the fourth chapter I will describe the methods and algorithms I used in building my chess engine, and in the fifth chapter I will describe the tools and technologies with which I built the game and the engine. The sixth chapter presents an evaluation of the engine, and in the seventh chapter the conclusions of the work are highlighted.

Chapter 2

Techniques and algorithms used

In his 1950 paper, "Programming a Computer for Playing Chess" [18], Claude E. Shannon describes how a chess engine might be implemented. Although not clearly stated, from his work we can deduce the three main parts a machine would need to do to be able to play chess, which are still valid and used in modern engines:

- Generating the legal moves of a position
- Searching for legally reachable positions from a position
- Evaluating the positions

2.1 Move generation

2.1.1 Legality

Regarding legality of the moves, move generation can be pseudo-legal or legal. In pseudo-legal move generation, the moves generated follow the normal rules of each piece's movement, but they are not checked to see if the king is left or moved into check. In legal move generation only legal moves are generated, checking beforehand if the king would be left in an illegal position. Pins, along with en passant moves, are particularly difficult to check. That is why the first approach is generally used: generating the pseudo-legal moves first, checking each move to see if it leaves the board in a valid state (the king of the side that moved is not in check), and removing the invalid moves.

2.1.2 Bitboards

Chess engines often use bitboards to represent the board in a piece centric manner. They are essentially 64-bit integers, where each bit corresponds to a square on the

chessboard. This approach reduces used space and makes move generation and position evaluation more efficient. There are multiple types of bitboards, some of the most common ones being:

- Occupancy bitboards: they represent the positions of the pieces, a bit representing if the square is occupied by a piece of a certain type or not. There are separate bitboards for each piece type and for each player
- Attack bitboards: these bitboards indicate the attacking squares for each type. For example, the attack bitboard for a knight shows all the squares a knight on a specific square can move to
- Check bitboards: these bitboards represent squares under attack by the opponent's pieces. They help determine if a move puts the king in check

2.2 Search

2.2.1 Minimax

The minimax algorithm represents the search as a tree, where the root node is the current position, edges to its children are the possible moves and the children are the positions reached by making each respective move. Then, for each child node (position) the process continues: edges for the legal moves, child nodes for reached positions. If enough computing power was available, generating the nodes would stop when a position which can be given a perfect evaluation would be reached - that is, an end position, which would be evaluated with win for white, draw, or win for black. Since from most positions the tree would be far too big to generate, the tree is usually generated to some depth (the evaluations would become more accurate the deeper the tree is generated), and the positions reached are then evaluated using a numerical heuristic function. The evaluations are then propagated back to the root node in the following way: for each node that has all of its children evaluated, if it is white's turn, the node is given the maximum value of its child nodes, and if it is black's turn, the node is given the minimum value of its child nodes instead.[10]

The complexity of the minimax algorithm in chess is dependent on the depth of the search tree and the number of legal moves on each level.

Assuming an average branching factor of around 35 moves per position, the number of positions to be evaluated by the minimax algorithm can grow exponentially with the depth of the search tree. For example, at a depth of 4 ply (i.e., 4 moves ahead), there are roughly $35^4 = 1,500,625$ positions to be evaluated.

In general, the time complexity of the minimax algorithm is $O(b^d)$, where b is the branching factor and d is the depth of the search tree. This can make it computa-

tionally infeasible to search deeply in the game tree, since there are a large number of possible moves and the branching factor is high.

To address this issue, techniques like alpha-beta pruning, transposition tables, and iterative deepening have been developed to reduce the number of positions that need to be evaluated by the minimax algorithm. Additionally, more advanced algorithms like Monte Carlo Tree Search and neural networks can be used to guide the search and reduce the number of positions that need to be evaluated, further improving the efficiency of the algorithm.

2.2.2 Alpha-beta pruning

Alpha-beta pruning is an optimization technique used in the minimax algorithm that can reduce the number of nodes that need to be evaluated in the game tree, making the search more efficient. Minimax searches through all leaf nodes to find the minimax value, while alpha-beta prunes leaves that have no influence on the outcome.

The algorithm works by maintaining two values: alpha and beta. Alpha represents the best score that the maximizing player has found so far, while beta represents the best score that the minimizing player has found so far.

As the algorithm searches through the game tree, it compares the scores of each possible move to alpha and beta. If a score is found that is worse than alpha (for the maximizing player) or better than beta (for the minimizing player), then that branch of the game tree can be pruned, since it will not lead to a better outcome [2].

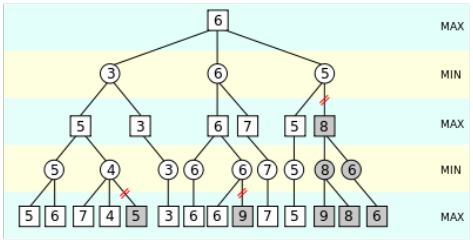

Figure 2.1: Alpha-beta pruning

The time complexity of alpha-beta pruning in chess is also dependent on the depth of the search tree and the branching factor of the game. In the worst case scenario, where alpha-beta pruning has no effect, the time complexity of alpha-beta pruning is still $O(b^d)$, the same as for the minimax algorithm. But alpha-beta pruning, in the best case, has a time complexity of $O(b^{\frac{d}{2}})$, which is significantly less than that of the classic minimax algorithm.

The effectiveness of alpha-beta pruning depends on the ordering of the moves in the search tree. If the moves are ordered in such a way that the best moves are

considered first, then alpha-beta pruning can quickly identify the best move and avoid searching deeper than necessary.

2.2.3 Quiescence search

When searching for positions, if the search stops after a fixed depth, some of the positions might be in a situation where on the next move the material evaluation could be totally reverted (a big piece about to be captured, a pawn about to be promoted, etc.). In those cases, the evaluation would not be so reliable, and the situation might be reversed in the next move.

To deal with these situations, quiescence search is an improvement that continues the search in non-quiescent positions until a quiescent position is found. A position is considered quiescent if there are no forcing moves (checks, threats) or moves that have a big impact on material (captures, promotions).

2.2.4 Iterative deepening

When limiting the search by time, performing the minimax to a fixed depth has some disadvantages: if the time has elapsed and the algorithm didn't finish, the results are probably bad, but, on the other hand, if the algorithm finished there might still be some time left, which could be used to improve the result [2].

Adrian De Groot, dutch psychologist and chess master, first mentioned the idea of iterative deepening in "Thought and Choice in Chess" [9]. With this improvement, the search starts at depth one, and then the depth is incremented and the search is done again, until the given time elapses.

Along with the ability of the algorithm to work well with time limits, iterative deepening introduces an additional advantage: the moves can be ordered at each level using the results given by the previous search, and this has shown to bring improvements to the number of branches cut by alpha-beta pruning.

2.2.5 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a search algorithm particularly effective in situations where the state space is large and difficult to exhaustively explore, such as in chess. It is based on the concept of random sampling or simulation. It builds and explores a search tree by repeatedly performing a four-step process:

- Selection: starting from the root node of the search tree (the current position), MCTS navigates down the tree by selecting child nodes according to a selection policy, which tipically balances exploration (visiting unexplored moves) and exploitation (favoring moves already visited that have a high win rate)

- Expansion: once a leaf node is reached, the algorithm expands the tree by generating other child nodes representing possible moves from that position
- Simulation: after expanding the tree, MCTS performs simulations from the newly added child nodes, which involves playing a game from the current state to the end using a simple policy (even random moves) until a terminal state is reached
- Backpropagation: the result of the simulation is propagated up the tree, updating the statistics of nodes (for example visit count, win count); this information is used to estimate the evaluation of each node and guide future selection

By repeating these steps for a certain number of iterations or until a time limit is reached, MCTS gradually builds up knowledge about the current position.

2.3 Evaluation

As mentioned earlier, if unlimited computing power was available, any position could be labeled as winning for white, draw, or winning for black. Because this is not the case, heuristics are used to assign an approximated evaluation to a position.

2.3.1 Centipawns

The main part of an evaluation function is the material advantage, that is, the number and types of pieces on each side. Pieces are usually assigned a base value, as can be seen in fig. 2.2. The king is either assigned no value, because the objective of chess is to checkmate the opponent's king, not capture it, therefore its value is considered irrelevant, or is assigned a very high value (bigger than all the other pieces combined), so the engine prioritizes the safety of the king above all else. In this case, the engine would attempt to avoid any moves that might put its own king in danger, even if it means giving up material or positional advantage.

pawn	knight	bishop	rook	queen
1	3	3	5	9

Figure 2.2: Basic piece values

But there are other aspects of a position that need to be taken into consideration by an evaluation function, for example development of the pieces (a piece that is not developed and doesn't attack/defend any squares is not of so much use). Some features of the position are not as important as to add a full point to the evaluation, so usually centipawns are used to evaluate a position.

For example, some positional advantage a player has might be worth about three quarters of a pawn. A centipawn is a score unit corresponding to a hundredth of a pawn, so three quarters of a pawn would be 75 centipawns. That means a pawn will be worth 100 centipawns, a knight will be worth 300 centipawns, and so on.

2.3.2 Handcrafted functions

Most engines use a handcrafted function to evaluate positions, measuring features of the board that are considered to give an advantage. Some basic evaluation features are:

- Material - the number and types of pieces
- Pieces positions - where the pieces are placed (in the center, at the edge, in the corner of the board)
- Pawn structure - connected pawns (allied pawns on adjacent files) are considered good, isolated pawns (with no allied pawns on adjacent files) are considered weak, passed pawns (pawns that do not have any opponent pawns on the same file or adjacent files in front of them) are strong, because they cannot be traded by opponent pawns, etc.
- Mobility - the number of legal moves a player has in the position
- Trapped pieces - pieces that don't have any safe squares to move to are considered weak
- King safety - existence of allied pawns in front of the king, distance of opponent pieces to the king, etc.

Most of the times, handcrafted evaluation functions also take into consideration the phase of the game - the development of the pieces is the main concern in the opening, followed by attacking and defending in the middle game, and pawn promotion in the endgame.

2.3.3 Neural network

Another way of building an evaluation function is through training a neural network. Neural networks are a class of machine learning models inspired by the structure and functioning of the human brain. They are designed to learn and make predictions or decisions based on input data. Neural networks excel at tasks involving pattern recognition, classification, regression, and other complex data analysis tasks.

Neural networks are composed of interconnected nodes called neurons. They are organized into layers: an input layer, one or more hidden layers, and an output layer. Each neuron receives some values as input, applies a mathematical operation, and produces an output that is passed on to the next layer.

One approach to training the neural network for evaluating chess positions is to use supervised learning, by giving it the position, sometimes along hand-crafted features as input (the ones previously mentioned: material, mobility etc.), and the corresponding evaluations, such as the scores given by human experts or strong chess engines.

Another approach is to use reinforcement learning, where the network is trained by playing against itself and learning from the outcomes of the games. This technique has the advantage of generating its own training data, as well as the ability to find patterns that are stronger and more complex than those that people can think of, if sufficiently trained.

Chapter 3

State of the art chess engines

3.1 Top Chess Engine Championship

The Top Chess Engine Championship (TCEC) is widely regarded as one of the most prestigious and competitive computer chess tournaments in the world.

TCEC is an online tournament that features the strongest chess engines competing in a round-robin format (each engine plays in turn against every other engine), followed by knockout stages to determine the champion. The engines compete on powerful hardware, and each match consists of multiple games with different time controls and openings to ensure a diverse range of positions and scenarios.

TCEC has been held several times a year since 2010, and 24 seasons have concluded as of May 2023. Stockfish holds the record of being a 14-time winner of the TCEC. Stockfish and Leela Chess Zero have dominated the competition in the past years, with the final having been played between the two engines in 9 of the last 11 seasons.

3.2 Elo rating system

The Elo rating system is a method used to determine the comparative levels of skill among players in games with zero-sum outcomes. FIDE ("Fédération Internationale des Échecs", the World Chess Federation), adopted the Elo rating system in 1970, and is still using it to this day.

A player's Elo rating is represented by a numerical value that might increase or decrease based on the outcomes of the rated games they play.

Elo's rating system operates on the premise that every player possesses an unknown current playing ability, which is approximated by a rating. When players with (unknown) strengths R_A and R_B engage in a game, the projected score for player A is assumed to be

$$E = \frac{1}{1 + 10^{-(R_A - R_B)/400}},$$

where the score corresponds to 1 in the event of player A's victory, $1/2$ in case of a draw, and 0 if player A loses [8].

Following each match, the victorious player subtracts rating points from their defeated opponent, with the total points gained or lost being determined by the disparity in their ratings. If the player with a higher rating wins, only a small number of rating points are deducted from the lower-rated player. However, if the lower-rated player wins unexpectedly, a substantial number of rating points are exchanged.

When a game results in a draw, the lower-rated player gains a small number of points from the higher-rated player. Therefore, this rating system possesses a self-correcting mechanism. Players whose ratings are either too low or too high will eventually outperform or underperform, respectively, compared to the expectations set by the rating system. Consequently, these players will gain or lose rating points until their ratings align with their true playing ability.

Elo ratings are solely comparative and hold validity only within the specific rating pool in which they were computed, rather than serving as an absolute measure of a player's proficiency. The highest Elo rating attained by a human was 2882, accomplished by Magnus Carlsen in May 2014 on the FIDE ratings list [7]. It is estimated that top chess engines possess over 3400 Elo points.

3.3 Stockfish

The strongest chess engine is currently Stockfish, an open-source project. It uses a minimax algorithm to search for the most promising positions, along with alpha-beta pruning, iterative deepening, and other improvements to reduce the search-space [13]. For evaluating the positions, Stockfish uses a hand-crafted evaluation function that consists of multiple features: material, position of the pieces on the board, pawn structure, mobility of pieces, king safety etc. [11]

Recent versions of Stockfish have been improved with NNUE (Efficiently Updatable Neural Networks). NNUE is a neural network-based evaluation function. Unlike traditional evaluation functions that are based on handcrafted features and heuristics, NNUE evaluates a chess position by analyzing it directly with a neural network. Because the neural network can learn and improve its evaluation of positions over time through training, the evaluation function becomes more flexible and adaptable.

Stockfish uses a multi-layer neural network architecture that takes as input a rep-

resentation of the current board state and outputs a score that reflects the expected outcome of the game. The neural network is trained using a dataset of high-quality chess games, where the inputs are positions from the games and the outputs are the eventual outcomes of the games.

NNUE has been a significant development in computer chess, as it has led to significant improvements in the playing strength of the Stockfish engine. In particular, NNUE has improved Stockfish's ability to evaluate endgame positions, which were previously challenging for traditional evaluation functions to handle.

3.4 AlphaZero

AlphaZero is an engine first published in 2017 by Google division Deepmind. It mastered the games of chess, shogi and go, being given only the rules of the games, with no additional domain-specific human knowledge or data.

AlphaZero uses Monte Carlo Tree Search instead of the minimax approach. This algorithm does not need an evaluation function, since it can compute meaningful evaluations based on random playouts that reach terminal game states, where the win/draw/loss outcome can be applied. Still, combining it with a strong evaluation function helps improve it.

DeepMind, the team behind the engine, built an evaluation function through deep reinforcement learning, training it with the matches the engine played itself. This brings the advantage that there is no need for hand-crafted feature selection and no data, since it generates the data by itself, and, with enough training, the model can learn its own useful features.

Another advantage of the Monte Carlo Tree Search algorithm is that the performance is expected to gradually grow with the computation time and the number of iterations. Iterative deepening in minimax can provide some resemblance to this behavior, but it is typically less "smooth" because the method takes far longer each time the depth is raised than it did for the previous depth. If it runs out of time, it must abandon the current search at the current depth limit; the last search will be useless, and the results from the previous search will be used instead.

AlphaZero was trained for 9 hours in chess, and reached Stockfish's level in the first 4 hours of training. AlphaZero was tested by the DeepMind team against Stockfish 8. In 1000 games, it won 155, lost only 6, and drew the other 839. AlphaZero played some additional matches starting from common human openings, and it managed to defeat Stockfish in each of them, "suggesting that AlphaZero has mastered a wide spectrum of chess play" [19].

Stockfish 8 is estimated to have around 3400 Elo points, while AlphaZero's playing strength would correspond to an Elo of over 3400 (fig. 3.1).

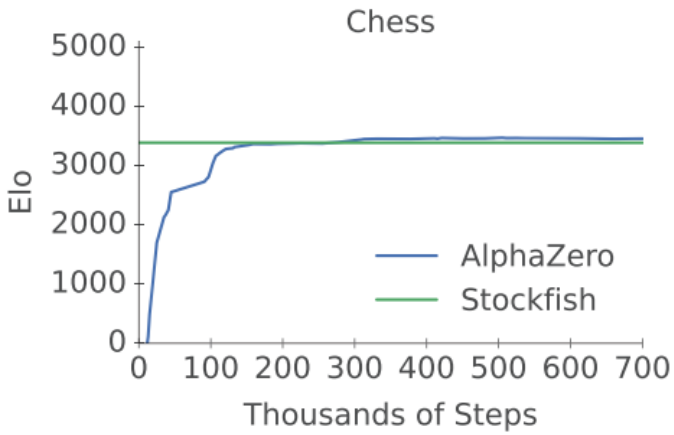

Figure 3.1: AlphaZero Elo evolution over training steps [19]

3.5 Leela Chess Zero

Leela Chess Zero (LCZero) is a computer chess engine based on the principles of DeepMind's AlphaZero. Since AlphaZero is not open-source, and there was no way to test the results stated in DeepMind's paper, LCZero was created by a group of developers led by Gary Linscott, who started the project in 2018. The engine was largely driven by the community of chess enthusiasts and programmers, who contributed to the project by providing computing resources.

LCZero developers also brought some key improvements to the AlphaZero approach:

- Efficient implementation: LCZero is designed to run efficiently on standard hardware, such as CPUs and GPUs commonly available to consumers. This makes it more accessible than AlphaZero, which was trained on specialized hardware

- Improved neural network architecture: LCZero's neural network architecture is optimized for the game of chess, with modifications that allow it to better represent the game state and learn from self-play. This has led to better performance in practice. The developers also added Squeeze and Excitation (SE) layers to the neural network, which improve the representation of CNNs (Convolutional Neural Networks), by enabling them to selectively attend to the most important features of the input data

- Dynamic evaluation: LCZero uses a dynamic evaluation function during self-play, which allows it to better evaluate the value of positions and guide its search. Unlike AlphaZero, which uses a fixed evaluation function throughout the self-play process, LCZero updates its evaluation function after each game played, based on the results and the positions encountered. This allows it to

adapt to changes in the playing style and strategy of its opponents

In April 2018, LCZero became the first engine to join the TCEC that used a deep neural network. At first, it didn't perform well, but by September 2018, it achieved a level of competitive performance comparable to the strongest chess engines in the world, and in 2019 and 2020 it won the competition. Since joining the TCEC, LCZero has achieved significant success, having won the competition twice, secured the second place on seven occasions, and achieved third place once.

On its official website [12], LCZero is currently rated over 3500 Elo points (fig. 3.2).

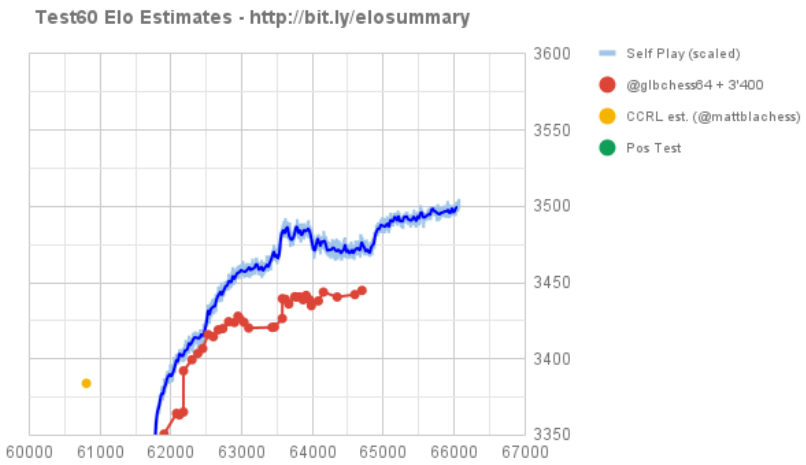

Figure 3.2: Leela Chess Zero Elo evolution by different metrics [12]

Chapter 4

Methodology

In this chapter I will describe the implementation of the three main steps taken by the engine to find the best move: generating legal moves for a position, searching for legally reachable positions from the current position, and then evaluating each one and determining which one is the best [14].

4.1 Move generation

The engine computes the pseudo-legal moves of the pieces of the player that is moving, then it does each move, checks the state of the board, adds the move to the list of legal moves if the board is in a valid state, and then undoes the move.

To check the state of the board I used a threat map (fig. 4.1), which keeps track of the squares that are under attack or are defended by at least one piece (including empty squares, squares with allied pieces, and squares with opponent pieces). After each move, the threat map is updated in the following way: for each white piece on the board, it sets ”IsAttackedByWhite” to true for all the squares it attacks, and for each black piece on the board, it sets ”IsAttackedByBlack” to true for all the squares it attacks.

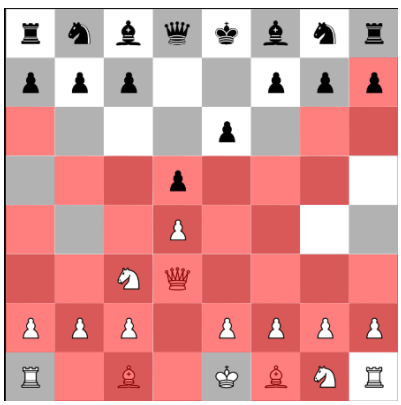

Figure 4.1: White threat map

To test the move generation, I implemented a Perft [17] (performance test) function. Perft is a tool to measure the efficiency and corectness of a chess engine's move generator. It involves counting the number of legal positions that can be reached from a given position within a specified depth. Perft results are used to check that the chess engine is correctly implemented, and Perft is also used as a benchmark to compare the speed of different chess engines or move generation algorithms.

4.2 Search

The simplest way of searching in the game tree is through a fixed depth minimax algorithm, which would be mathematically accurate, but would not yield good results and would be very slow. However, in combination with some optimizations and heuristics, it can compute faster and find better moves.

4.2.1 Alpha-beta pruning

The minimax algorithm was improved with Alpha-beta pruning, which preserves the accuracy of the algorithm and significantly reduces the number of nodes visited if combined with a good move ordering heuristic [20].

A simple pseudocode for Alpha-beta pruning is shown in fig. 4.2:

```
function alphabeta(node, depth, alpha, beta, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node

    if maximizingPlayer
        v := -infinity
        for each child of node
            v := max(v, alphabeta(child, depth - 1, alpha, beta, FALSE))
            alpha := max(alpha, v)
            if beta ≤ alpha
                break (* beta cut-off *)
        return v
    else
        v := +infinity
        for each child of node
            v := min(v, alphabeta(child, depth - 1, alpha, beta, TRUE))
            beta := min(beta, v)
            if beta ≤ alpha
                break (* alpha cut-off *)
        return v
```

Figure 4.2: Alpha-beta pruning pseudocode

The heuristic I used in combination with Alpha-beta pruning is MVV-LVA (Most Valuable Victim - Least Valuable Agressor): the moves that capture pieces with high

values are first, and if multiple moves capture a piece of the same value, the moves with the least valuable agressor are prioritized [15]. The heuristic is based on the observation that in chess, it is often advantageous to capture pieces of higher value than the capturing piece, while avoiding exchanges where the capturing piece is more valuable than the captured piece. It is also relatively easy to implement, and does not require much computational overhead. Other heuristics use the evaluations of previous searches or the results from other branches, and require maintaining and updating additional data structures to keep track of the move history and their performance.

4.2.2 Quiescence search

When the minimax search reaches the maximum depth in a branch, it evaluates the position and propagates it back. With the quiescence search optimization, the position is first checked for potential moves that could disrupt the evaluation in the following move.

I considered captures and promotions to be disruptive moves, since they have a big impact on material. If there are no such moves available, the algorithm continues as normal. But if there are, the current static evaluation is kept as a ”stand-pat” score (the term is used in poker, to describe a situation where a player chooses to keep their hand as it is without drawing any additional cards), to establish a lower bound on the evaluation.

The ”disruptive” moves are played, and, from the positions reached, quiescence search is applied again, until a certain depth is reached. Each new position is evaluated using the static function, and if the evaluation is better than alpha (the lower bound, which is initialized with the stand-pat score), then the evaluation score is assigned to alpha. At the end of the quiescence search, the alpha score is returned. I used 4 as the maximum quiescence search depth, to allow the engine to detect certain series of exchanges, but not search too deep and use up too much time.

The stand-pat score is used based on the Null Move Observation, which states that there is almost always a move that improves the current position - it assumes the player to move is not in Zugzwang (a position in which it is disadvantageous to move, as every move inevitably results in a worse position). The stand-pat assumes that even if all the disruptive moves are searched, and none of them increase alpha, one of the non-disruptive moves can most likely raise alpha. This is not valid if every move is searched.

In fig. 4.3 is shown an example of a simple position where quiescence search improves the engine’s play. If it is white to move and the engine searches with a depth of 1 with no quiescence search, the best move found is queen takes knight,

but that move gives up the queen, because black can recapture with his queen. If the engine searches with the quiescence search improvement, it will generate the queen recapture as a non-quiescent move and will assign a low score to the queen takes knight move.

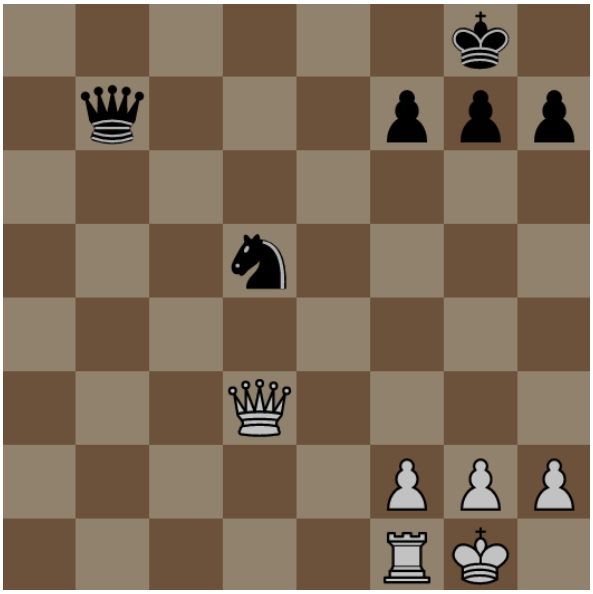

Figure 4.3: Quiescence search example

4.2.3 Iterative deepening

The iterative deepening starts computing the evaluations for the minimax at depth 1, and then increments the depth up to the given maximum depth. After the time given as an input elapses, the search is stopped and the move with the best evaluation found on the previous depth search is returned. I set the maximum time to 50 seconds, but this can be easily changed from a parameter, depending on the length of the match and accuracy of moves the user of the engine desires.

4.3 Evaluation

I used two methods of evaluating a chess board position. The first one is a simple evaluation function based on the position of the pieces on the board, and the second one is a neural network model, trained on matches played at grandmaster level.

4.3.1 Board pieces

This evaluation function assigns a value to the board configuration based on the position of the pieces and the state of the game (opening/middlegame/endgame).

I considered the state to be the opening if there were less than 13 moves made, middlegame if there are at least a queen or 3 minor pieces (rook/bishop/knight) on the board, and the endgame otherwise.

```
PAWN   =   100
KNIGHT =   320
BISHOP =   330
ROOK   =   500
QUEEN  =   900
KING   = 20000
```

Figure 4.4: Base values for the pieces

```
-50,-40,-30,-30,-30,-30,-40,-50,
-40,-20,  0,  0,  0,  0,-20,-40,
-30,  0, 10, 15, 15, 10,  0,-30,
-30,  5, 15, 20, 20, 15,  5,-30,
-30,  0, 15, 20, 20, 15,  0,-30,
-30,  5, 10, 15, 15, 10,  5,-30,
-40,-20,  0,  5,  5,  0,-20,-40,
-50,-40,-30,-30,-30,-30,-40,-50,
```

Figure 4.5: Knight value table

Figure 4.6: King value tables

```
-30,-40,-40,-50,-50,-40,-40,-30,
-30,-40,-40,-50,-50,-40,-40,-30,
-30,-40,-40,-50,-50,-40,-40,-30,
-30,-40,-40,-50,-50,-40,-40,-30,
-20,-30,-30,-40,-40,-30,-30,-20,
-10,-20,-20,-20,-20,-20,-20,-10,
 20, 20,  0,  0,  0,  0, 20, 20,
 20, 30, 10,  0,  0, 10, 30, 20
```

(a) King middlegame value table

```
-50,-40,-30,-20,-20,-30,-40,-50,
-30,-20,-10,  0,  0,-10,-20,-30,
-30,-10, 20, 30, 30, 20,-10,-30,
-30,-10, 30, 40, 40, 30,-10,-30,
-30,-10, 30, 40, 40, 30,-10,-30,
-30,-10, 20, 30, 30, 20,-10,-30,
-30,-30,  0,  0,  0,  0,-30,-30,
-50,-30,-30,-30,-30,-30,-30,-50
```

(b) King endgame value table

Each piece has a base value (fig. 4.4), to which another value is added based on the square the piece occupies. Each piece has an 8x8 table, with a value for each square. The tables contain values as to encourage development of the pieces (example - fig. 4.5 shows the table for the white knights). The tables for black pieces are horizontally simetric to the ones for the white pieces. The king has an additional table for the endgame, because in the opening and middlegame it is encouraged to castle and discouraged to move far from the first rank (fig. 4.6a), while in the endgame it is encouraged to go towards the center and play an active part in the game (fig. 4.6b) [5].

4.3.2 Neural network

For the second evaluation function, I trained a neural network on grandmaster chess matches. The neural network evaluation alone would be even weaker than the previous one, since it would not consider material imbalance. The material evaluation is still computed, and the neural network evaluation is added to the material value. The material evaluation outputs values usually between -10 and 10, and the neu-

ral network outputs values between -1 and 1, so it is only relevant when deciding between moves of close material evaluations.

Dataset

The dataset consists of chess matches played at grandmaster level downloaded in PGN format from the website chessabc.com [3].

A PGN (portable game notation) file contains multiple chess matches. Each match has several headers (name of player with white pieces, name of player with black pieces, event it was played at, date it was played on, result, elo of players etc.), an empty line, and then the ”movetext section”. The matches in a PGN file are separated by two empty lines. Following is a sample PGN match:

```
[Event "Carlsen - Anand World Championship Match"]
[Site "Sochi RUS"]
[Date "2014.11.09"]
[Round "2"]
[Result "1-0"]
[White "Magnus Carlsen"]
[Black "Viswanathan Anand"]

1.e4 e5 2.Nf3 Nc6 3.Bb5 Nf6 4.d3 Bc5 5.O-O d6 6.Re1 O-O 7.Bxc6
bxc6 8.h3 Re8 9.Nbd2 Nd7 10.Nc4 Bb6 11.a4 a5 12.Nxb6 cxb6
13.d4 Qc7 14.Ra3 Nf8 15.dxe5 dxe5 16.Nh4 Rd8 17.Qh5 f6 18.Nf5
Be6 19.Rg3 Ng6 20.h4 Bxf5 21.exf5 Nf4 22.Bxf4 exf4 23.Rc3 c5
24.Re6 Rab8 25.Rc4 Qd7 26.Kh2 Rf8 27.Rce4 Rb7 28.Qe2 b5 29.b3
bxa4 30.bxa4 Rb4 31.Re7 Qd6 32.Qf3 Rxe4 33.Qxe4 f3+ 34.g3 h5
35.Qb7 1-0
```

The movetext section contains the chess moves, move numbers, optional annotations and a concluding game termination marker (the result of the match). The result can be ”1-0” (white won), ”0-1” (black won) or ”1/2-1/2” (draw). The moves are written in SAN (standard algebraic notation). SAN identifies each square on the board as a combination of file (a-h) and rank (1-8) (see fig. 4.7).

Each piece can be identified by a letter (K - king, Q - queen, R - rook, B - bishop, N - knight, P - pawn). Simple SAN moves contain only the destination square, and the letter of piece that moves, if the piece is not a pawn (for example, pawn to c6 is written simply as c6, while knight to c6 is written as Nc6). There are additional rules for moves [4]:

- Moves that capture a piece contain the ’x’ character right before the destination square (ex: Bxb4)

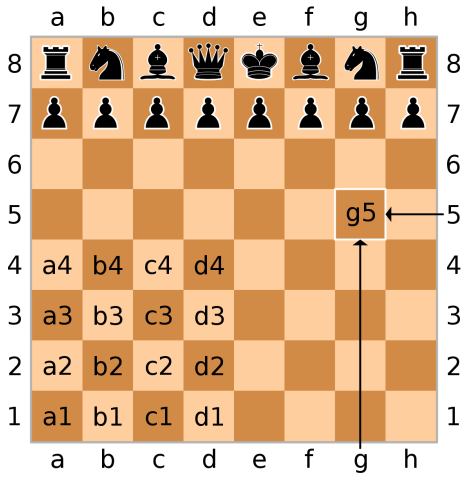

Figure 4.7: Board squares

- Castling kingside is written as 'O-O', while queenside castling is written as 'O-O-O'
- Moves of pawns to the last rank (promotion moves) contain the equal sign ('=') right after the destination square, followed by the letter of the piece that the pawn promotes to (ex: e8=Q)
- If there are multiple pieces of the same type that can move to the destination square, the file of the originating square is added right after the piece (ex: Nbc6); if this does not solve the ambiguation, the rank of the originating square is added instead (ex: N8c6), and if neither of these works, both the file and the rank are added (ex: Nb8c6)
- If the move checks the opponent's king, a plus sign ('+') is added at the end of the move (ex: Qe7+); if the move checkmates the opponent's king, the octothorpe sign ('#') is added instead (ex: Qe7#)

I computed the positions that occured in the matches in the dataset, and the value I assigned to each position is the mean of the results of the matches in which the position appeared. For example: if a certain position appeared in 10 matches, and 4 of the matches ended in a win for white, 2 in a win for black, and the other 4 were draws, the value assigned to the position is:

$$\frac{(4*1)+(2*-1)+(4*0)}{10} = 0.2$$

(a win for white is assigned 1, a win for black is assigned -1, and a draw is 0).

There are a number of 8.5 million positions in the dataset, and 5.5 million of those positions are distinct.

Model

The model receives as input a position and outputs a value between -1 (winning for black) and 1 (winning for white). The position is encoded as 70 values. Following is the encoding for the starting position:

$$4, 1, 0, 0, 0, 0, 7, 10, 2, 1, 0, 0, 0, 0, 7, 8, 3, 1, 0, 0, 0, 0, 7, 9, 5, 1, 0, 0, 0, 0, 7, 11, 6, 1, 0,$$
$$0, 0, 0, 7, 12, 3, 1, 0, 0, 0, 0, 7, 9, 2, 1, 0, 0, 0, 0, 7, 8, 4, 1, 0, 0, 0, 0, 7, 10, 1, 1, 1, 1, -1, 0$$

The input layer contains 70 nodes. The first 64 values describe the positions of the pieces (first 8 values are the pieces on a1, a2, a3, ..., a8, then b1, b2, b3, ..., b8, etc.) - each square is assigned a value based on the piece it contains (see fig. 4.8).

```
NO PIECE     = 0
WHITE PAWN   = 1   BLACK PAWN   = 7
WHITE KNIGHT = 2   BLACK KNIGHT = 8
WHITE BISHOP = 3   BLACK BISHOP = 9
WHITE ROOK   = 4   BLACK ROOK   = 10
WHITE QUEEN  = 5   BLACK QUEEN  = 11
WHITE KING   = 6   BLACK KING   = 12
```

Figure 4.8: Squares encoded values

The next 4 values are boolean (0 - false, 1 - true) and encode the castling rights (white can castle kingside, white can castle queenside, black can castle kingside, and black can castle queenside). The next value is the en passant file (the file of the moved pawn, encoded to a value from 0 to 7, if the last move pushed a pawn two squares, -1 otherwise). The last value is the player to move in the current position (0 for white, 1 for black).

There is only one hidden layer, with 32 nodes, with a Rectified Linear Unit (ReLU) activation function.

The output layer contains one node, the evaluation of the position determined by the neural network, with a hyperbolic tangent (Tanh) activation function, so the output is between -1 and 1.

Training

The model was trained for 100 epochs, on a batch-size of 64, using Stochastic Gradient Descent (SGD) as the optimizer and Mean Squared Error (MSE) as the loss function.

Chapter 5

Technologies and implementation

5.1 Chess engine

5.1.1 Tools

C#

The chess engine implementation was written in C#, as it offers a convenient way of modelling the chess entities through its object-oriented behaviour.

The most preffered alternative for chess engines is C++, because of its execution speed. While C++ might be a better choice in terms of performance [16], it is a programming language that is more difficult to develop and maintain due to its complex syntax and low-level memory manipulation. C# also has robust access to helpful libraries, further simplifying the implementation process.

Another option is Python, which offers greater ease of use, but is generally slower due to the fact that it uses dynamic typing and is an interpreted language, not a compiled one.

Keras

The neural network was trained using Keras, an open-source deep learning framework written in Python. Keras provides a simple and intuitive API that allows users to quickly prototype and build deep learning models. It can run on top of different deep learning libraries, such as TensorFlow, Theano, or Microsoft Cognitive Toolkit (CNKT).

Keras is built with a modular architecture, allowing users to create neural networks by stacking layers. It includes a variety of pre-defined layers such as Dense (fully connected), Convolutional, Recurrent, which can be easily combined to construct complex architectures. In fig. 5.1 is shown an example of building a neural

network with an input layer of 70 nodes, a hidden layer with 32 nodes, and an output layer with one node.

```
Sequential model = new Sequential();
model.Add(layer: new Dense(units: 32, activation: "relu", input_shape: new Shape(70)));
model.Add(layer: new Dense(units: 1, activation: "tanh"));
model.Compile(optimizer: "sgd", loss: "mean_squared_error");
```

Figure 5.1: Keras neural network

5.1.2 Model

The implementation of the chess logic was done using several classes and enumeration types (shown in figures below). The enumeration types are *GameState* (fig. 5.2), *MoveType* (fig. 5.3), *PieceColor* (fig. 5.2) and *PieceType* (fig. 5.5).

```
public enum GameState
{
    INPROGRESS,
    DRAW,
    STALEMATE,
    CHECKMATE
}
```

Figure 5.2: Game state enum

```
public enum MoveType
{
    NORMAL,
    CASTLE,
    PROMOTION,
    ENPASSANT
}
```

Figure 5.3: Move type enum

```
public enum PieceColor
{
    NONE,
    WHITE,
    BLACK
}
```

Figure 5.4: Piece color enum

```
public enum PieceType
{
    NONE,
    PAWN,
    KNIGHT,
    BISHOP,
    ROOK,
    QUEEN,
    KING
}
```

Figure 5.5: Piece type enum

SquareCoords

The *SquareCoords* class encapsulates the coordinates of a square: it contains the number of the file and the number of the rank. It has a simple constructor with the two fields as parameters, and also a constructor that receives a string parameter, checks if it is a valid square string (has two characters, first one denoting the file - a

letter from a to h, and the second one denoting the rank - a number from 1 to 8), and assigns the corresponding file and rank.

Piece

A $Piece$ class is composed of two enumeration fields: a $PieceType$ and a $PieceColor$. I have chosen not to make a class for each piece and have it inherit a basic $Piece$ class, since the behavior of each piece differs significantly, and inheritance would complicate the implementation.

Square

A $Square$ class includes the square's coordinates on the board, the piece that occupies it (which might be null if there is no piece on the square), and two boolean values, $IsAttackedByWhite$ and $IsAttackedByBlack$, which are used for the threat maps.

SimpleMove

The $SimpleMove$ class contains minimal information for identifying a move: the $SquareCoords$ of the origin square, the $SquareCoords$ of the destination square, the $PieceType$ of the promotion choice (in case it is a promotion move), and an additional field with the $PieceType$ of the captured piece, used for sorting moves during the engine's search.

MoveInfo

The $MoveInfo$ class contains the fields that the $SimpleMove$ class contains, plus some more fields that help during the undoing of the move: a $MoveType$ enumeration, the $Piece$ that moves, the $SquareCoords$ of the en-passant square before the move, if there was one (null otherwise), an integer denoting the previous half-move clock (how many consecutive half-moves without capturing a piece or moving a pawn were made until this move), and four boolean values denoting the previous castling rights: $PrevWhiteCanCastleKing$, $PrevWhiteCanCastleQueen$, $PrevBlackCanCastleKing$, and $PrevBlackCanCastleQueen$.

Board

The $Board$ class holds the configuration of the chess board: it contains the squares of the board (which in turn contain the pieces). The squares are memorised in a two-dimensional array (fig. 5.6), simplifying the process of obtaining a square on the board by file and rank.

```
public Square[,] Squares = new Square[Utils.FILES_COUNT, Utils.RANKS_COUNT];
```

Figure 5.6: The $Squares$ field of the $Board$ class

Game

The $Game$ class is the largest and it contains most of the functionality needed for a chess match. It contains a $Board$ field, which denotes the coordinates of the pieces in the current game position, and other information regarding the evolution of the game:

- A $PieceColor$ field denoting the player to move
- A $SquareCoords$ field which contains the "en-passant square", if the last move pushed a pawn two squares (the en-passant square being the square where a pawn can capture by making an en-passant move)
- A $GameState$ enum denoting the current state of the game
- An integer denoting the number of moves since the game started
- An integer denoting the number of consecutive half-moves without captures or pawn moves
- Four boolean values denoting the castling rights
- A list of $MoveInfo$ objects containing the moves played in the game
- A list of $SimpleMove$ objects containing the legal moves in the position of the game's board
- A dictionary containing the positions that appeared in the game, with the key being the FEN (Forsyth-Edwards Notation) [6] description of the position and the value being an integer denoting the number of times the position appeared; this is used for the threefold repetition rule

The $Game$ class has a method for setting the position from a FEN string, which contains all the necessary information for identifying a position. It is a single line of characters, made up of six fields, separated by a space.

The first field contains the piece placement, rank by rank, separated by a '/' (slash) symbol. Each piece is represented by a single character, with uppercase letters for white pieces (K for king, Q for queen, R for rook, B for bishop, N for knight, and P for pawn), and lowercase letters for black pieces. A decimal digit counts consecutive empty squares on a rank. The second field specifies the player to move ('w'

for white, 'b' for black). The third field indicates the castling rights - 'K' for white kingside castling, 'Q' for white queenside castling, 'k' for black kingside castling, and 'q' for black queenside castling. If no castling is possible for any side, a hyphen (-) is used. If there is a potential en passant capture available, the fourth field specifies the target square. If there is no en passant target square, a hyphen (-) is used. Otherwise, the target square is represented using algebraic notation (e.g., 'e3'). The fifth field represents the halfmove clock. The sixth and last field indicates the current move number. It starts at 1 and is incremented after each black move.

Following is the FEN string representation of the starting position for a standard chess game:

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
```

Figure 5.7: FEN string for starting position

The most important and complex method of the $Game$ class is $Move$. It receives as parameters two $SquareCoords$, the coordinates of the origin and destination square, and optionally a $PieceType$ parameter, which specifies the promotion piece, if it is a promotion move. Besides making the move on the board, the method checks the move to determine whether it affects any castling rights (if it is a rook move, a king move, or a rook is captured), if it changes the en passant square, checks the fifty moves rule, the threefold repetition rule, and then it updates the threat maps and the list of legal moves.

In fig. 5.8 the dependency diagram between the classes is shown.

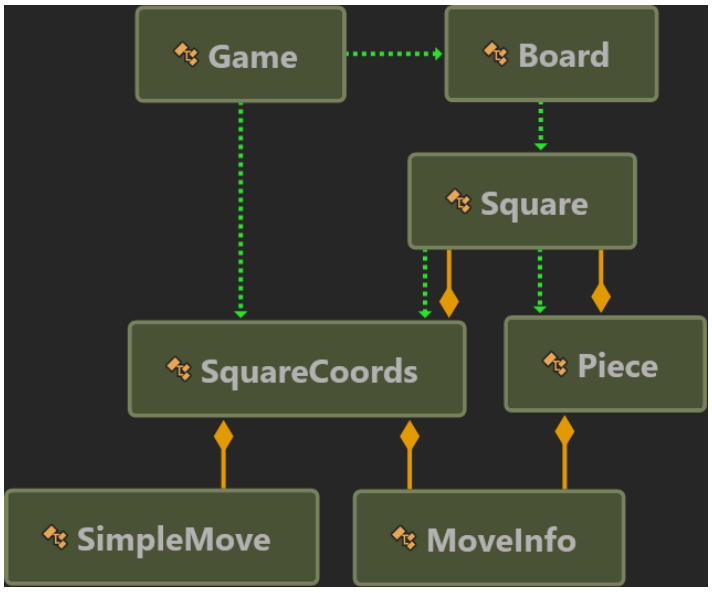

Figure 5.8: Dependency diagram for the chess game project

5.2 Chess game interface

I created the chess interface in Unity, a game development engine that provides a visual editor and C# scripting capabilities. I chose Unity because of its user-friendly and intuitive interface, as well as its ability to easily integrate external C# libraries, such as the chess engine I built.

On the game interface I added inputs that allow the player to change the engine parameters: the minimax depth, the usage of iterative deepening and quiescence search, and also the time limit for the iterative deepening and the maximum depth for the quiescence search.

Chapter 6

Results and evaluation

To test the engine's performance, I used the precision at k (P@K) metric. Precision at k is an evaluation metric commonly used in recommendation systems to measure the accuracy of the top-k ranked items. It assesses the proportion of relevant items within the top-k items recommended.

As "relevant items" I chose the top 5 moves given by Stockfish, and I compared them to the top 5 moves found by my engine in several positions. For each position, I calculated the number of moves found by the engine that are also in the top 5 moves found by Stockfish and I divided it by 5. At the end, I averaged all of the results.

Although this way of testing doesn't provide a comprehensive assessment of the overall strength or playing performance of the chess engine, it is useful in comparing between the two approaches available in the engine I built: the static material evaluation, and the evaluation that also uses the neural network.

The evaluations were computed for 45 tactical positions, with the same algorithm parameters for both approaches - depth of 3, alpha beta pruning, no iterative deepening (meaning no time limit), and quiescence search with a maximum quiescence depth of 3. The averages of the results are shown in the following table:

Evaluation method	**Precision at 5**
Material	0.322727
Neural network	0.336364

From the results, the neural network approach seems to slightly improve the evaluation.

The engine is able to win against the first two levels of Stockfish, but fails against level 3, when Stockfish starts to make more sensible moves. Following are the matches against level 2 and level 3 of Stockfish, in PGN format.

Against level 2, Stockfish playing with white:

1. e4 Nc6 2. d4 Nf6 3. Bd3 Nxd4 4. c3 Ne6 5. Nd2 Nf4 6. Nb3 Nxg2+ 7. Kd2 Nf4 8. Bc2 d5 9. e5 Nd7 10. Ke1 Ng6 11. f4 Ndxe5 12. Qd4 Nxf4 13. Nc5 Ng2+ 14. Kf2 Nc6 15. Qa4 Bd7 16. Ne2 Ne5 17. Qb3 Bh3 18. Rd1 Ng4+ 19. Kg3 Nxh2 20. Nxb7 Qd7 21. Qxd5 Qxd5 22. Kxh2 Qxb7 23. Nd4 Bg4 24. Rd2 Nf4 25. Ba4+ Bd7 26. Bb3 f6 27. Rb1 Qe4 28. Nc2 Bf5 29. Kg1 Rb8 30. Kf2 Rb7 31. Kg3 Qe5 32. Bc4 Qe4 33. Ba6 Rb6 34. Rh2 Rxa6 35. Bxf4 Rxa2 36. Rf1 Rxb2 37. Nb4 Rxh2 38. c4 Qg2# 0-1

Stockfish makes many mistakes, which lead to my engine gaining a big material advantage and eventually delivering a checkmate.

Against level 3, Stockfish playing with white:

1. e4 Nc6 2. Nf3 Nf6 3. Nc3 Ng4 4. h3 Nge5 5. d3 Nxf3+ 6. gxf3 Rb8 7. d4 e5 8. dxe5 Nxe5 9. f4 Nc6 10. Bc4 f5 11. Qd5 Qf6 12. exf5 Qe7+ 13. Kd2 Nb4 14. f6 Nxd5 15. fxe7 Nxe7 16. Nd5 Nxd5 17. Ke2 Nb4 18. Bb3 Ke7 19. Be3 Nc6 20. Rhg1 Kd6 21. f5 Ke5 22. Rgd1 Kxf5 23. f3 Ne5 24. Rg1 Bd6 25. Rxg7 Ra8 26. Bd5 Rb8 27. a3 Ra8 28. Rf1 Rb8 29. f4 Bxa3 30. Bd4 Nf3 31. Rxf3 Re8+ 32. Kd3 Be7 33. Rxh7 Ra8 34. c3 Bd6 35. Rh6 Re6 36. Rh5+ Kg6 37. Rg5+ Kh7 38. Rfg3 Bxf4 39. Rh5+ Bh6 40. Rg7+ Kh8 41. Rg6+ Rf6 42. Rhxh6# 1-0

Although Stockfish still gives up a piece at one point, my engine makes bad positional moves and Stockfish manages to find a checkmate.

Chapter 7

Conclusions

Based on the results, the neural network appears to be able to improve the evaluation function slightly, even with a simple network architecture, brief training, and a relatively small dataset, compared to the vast number of positions that can appear in chess.

There are several parts of the engine which could be improved. It is currently very slow, a problem which could be partially solved by implementing a faster move generation algorithm and by representing the chess board as bitboards. Spending less time on generating moves would leave more time for the engine to search to a greater depth, allowing it to provide more accurate evaluations.

Another way of improving the engine's speed is through more aggressive pruning, based on some heuristics. This would create the risk of overlooking some moves, but it would help the engine focus on more promising variations.

For a reliable evaluation of the engine's strength, it should implement the UCI (Universal Chess Interface) protocol [21] and be uploaded on a site where it receives an ELO rating based on the results of the matches against other engines.

Bibliography

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

[2] Jeroen WT Carolus. Alpha-beta with sibling prediction pruning in chess. *Amsterdam: University of Amsterdam*, 2006.

[3] Chess ABC, 2023. URL: `https://www.chessabc.com/en`.

[4] Steven J Edwards et al. Standard portable game notation specification and implementation guide, 1994. URL: `https://ia902908.us.archive.org/26/items/pgn-standard-1994-03-12/PGN_standard_1994-03-12.txt`.

[5] Simplified evaluation function, 2018. URL: `https://www.chessprogramming.org/index.php?title=Simplified_Evaluation_Function`.

[6] Forsyth-Edwards Notation, 2023. URL: `https://www.chessprogramming.org/Forsyth-Edwards_Notation`.

[7] FIDE Top 100 Players May 2014, 2014. URL: `https://ratings.fide.com/toparc.phtml?cod=305`.

[8] Mark E Glickman and Albyn C Jones. Rating the chess rating system. *CHANCE-BERLIN THEN NEW YORK-*, 12:21–28, 1999.

[9] A.D. Groot. *Thought and choice in chess*. 07 2014.

[10] Dominik Klein. Neural networks for chess. *arXiv preprint arXiv:2209.01506*, 2022.

[11] Matthew Lai. Giraffe: Using deep reinforcement learning to play chess. *arXiv preprint arXiv:1509.01549*, 2015.

[12] Leela Chess Zero training, 2023. URL: `https://training.lczero.org/`.

[13] Shiva Maharaj, Nick Polson, and Alex Turk. Chess ai: competing paradigms for machine intelligence. *Entropy*, 24(4):550, 2022.

[14] T Anthony Marsland. A review of game-tree pruning. *ICGA journal*, 9(1):3–19, 1986.

[15] Most Valuable Victim - Least Valuable Agressor, 2019. URL: `https://www.chessprogramming.org/index.php?title=MVV-LVA&oldid=15328`.

[16] Justin Onyarin Ogala and Deborah V Ojie. Comparative analysis of c, c++, c# and java programming languages. *GSJ*, 8(5):1899–1913, 2020.

[17] Perft (performance test), 2023. URL: `https://www.chessprogramming.org/Perft`.

[18] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[19] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[20] Eric Thé. *An Analysis of Move Ordering on the Efficiency of Alpha-beta Search*. PhD thesis, McGill University Libraries, 1992.

[21] Universal Chess Interface (UCI) protocol, 2023. URL: `https://www.wbec-ridderkerk.nl/html/UCIProtocol.html`.