TDDE19 – Tjack Report

Group 1
Angus Lothian (anglo547)
Bence Nagy (benna038)
Erik Lundin (erilu186)
Justus Karlsson (juska933)
Marcus Nolkrantz (marno874)
Mattias Ljung (matju387)
Oskar Lundin (osklu414)

January 9, 2022

Contents

1 Introduction

Chess is a world renowned two-player board game that dates back to ancient history. The goal is to checkmate the opponent's king using your own pieces while avoiding the opponent from doing the same to yourself. The game is set up symmetrically on an 8x8 board where both players have 16 pieces of 6 types each, and the different pieces can all move in different ways. Chess was classified by Jon von Neumann from a game-theoretical standpoint as a zero-sum deterministic perfect-information game [1]. In chess the amount of moves that each player can perform varies depending on the position, but an average branching factor of 35 is often used. This high branching factor along with the average game length of 80 full-moves gives the game of chess an astronomically large estimated game-tree complexity of 10^{123} which makes it unfeasible to completely explore the game-tree and "solve chess" [2]. However, it is still necessary to think several moves ahead to capably reason about which moves to select. These factors combined with the discrete nature of the game makes chess an interesting setting for developing computer agents that play games. In fact, the field of computer chess has a rich history and is connected with the beginnings of the sciences of Artificial Intelligence and general computing.

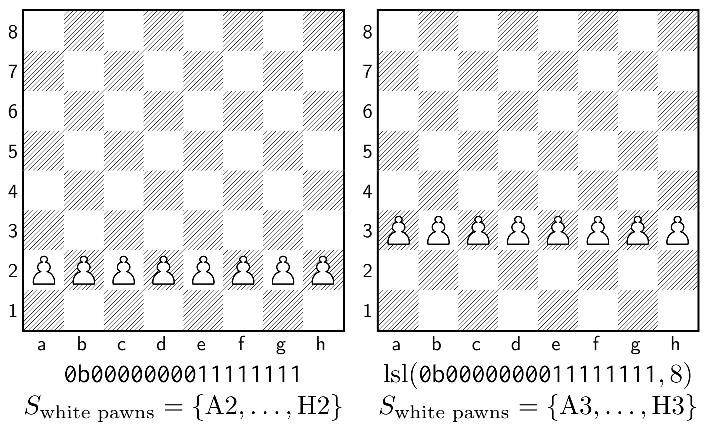
In computer chess, the main approach that is used is some form of searching algorithm that explores the game tree and thus reasons about the future positions that can be reached, in combination with some evaluation function that evaluates the positions and some logic that backpropagates this evaluation to the root state. Claude E. Shannon coined the concepts of Type A and Type B search strategies for the game of chess [3]. Type A strategies can be seen as a form of brute-force that searches all positions up to a certain depth, where the states are then evaluated in some way. Type B strategies however, use some intuitive understanding of the game to explore more promising lines deeper. Today, the most succesful chess engines, such as Stockfish and Leela Chess Zero [4], still use these approaches.

Stockfish[1] uses a Type A search strategy, namely a classical minimax search with alpha-beta pruning, in combination with a Neural Network evaluation function based on the Efficiently Updatable Neural Network (NNUE) that was introduced for Shogi[2], also known as Janapenese chess. The minmax algorithm searches as deep as possible as the engine can during the time it is given, evaluates the state with a neural network, and then, assuming that both players play optimally, recursively backpropogates the maximum or minimum child node value (depending on if it is our turn or the opponents turn at the specific depth) back all the way to the root state to select the best move.

Leela Chess Zero[3] on the other hand uses a Type B search strategy based on the Monte Carlo tree search approach by AlphaZero, where a reinforcement learning agent learns a neural network that guides the search tree to find good moves. AlphaZero learns through the reinforcement learning framework of trial and error via self-play, where it plays against itself over and over again to become better at the game.

In this project, we aim to implement both of these approaches, together with a third approach that uses alpha-beta search in combination with an handcrafted evaluation function. However, it is worth to note that both of the original implementations used large amounts of computational resources to be able to train their neural networks, and with us only having access to a few GPUs, do not aim to come close to the originals in terms of results. Nonetheless, the selected approaches offer a good variety of methods and valuable experience to us, the developers, as they require efficiently implemented code, large support systems such as a chess library, and are connected to state of the art methods in the field of Artificial Intelligence.

[1]https://stockfishchess.org/
[2]https://github.com/asdfjkl/nnue/blob/main/nnue_en.pdf
[3]https://lczero.org/



8 7 6 5 4 3 2 1
a b c d e f g h
0b0000000011111111
$S_{\text{white pawns}} = \{\text{A2}, \ldots, \text{H2}\}$
lsl(0b0000000011111111, 8)
$S_{\text{white pawns}} = \{\text{A3}, \ldots, \text{H3}\}$

Figure 1: To see how bitsets can be used to manipulate the board efficiently, we consider an example. In the left bitset for white pawns, bit 8-15 are set so there are white pawns on square A2-H2. If a logical left-shift is executed on this bitset's bitstring 8 times, the board to the right is obtained where all pawns have been pushed to rank 3.

2 Method

This chapter presents the methods used to establish the results.

2.1 Chess Library

Just as a human needs knowledge of the rules of chess to play it, so does a computer. In particular, given a certain board state, a chess playing program needs to be able to compute the set of legal moves and apply one of them to obtain the next board state. If the state space is to be searched, these two operations also have to be fast.

We developed our own chess library for this project, *libchess*, which implements the rules of chess and various utilities. The library uses bit arrays of size 64 to represent sets of squares on the chess board. Bit 0 corresponds to square A1, bit 1 to square A2, etc. If a bit is set to 1, it is in the set. The board consists of one bitset per piece type and color indicating which squares are occupied. Bitwise operations allow these sets are queried and manipulated in parallel. An example of how this is used for pawn pushes is presented in Figure 1.

2.2 Minimax search with alpha-beta pruning

The underlying principle for our first approach was the Minimax search algorithm with alpha-beta pruning, similar to the Stockfish chess engine previously mentioned. The Minimax algorithm works by creating a search tree in which the nodes are game states and the edges are actions. Each depth level in the tree corresponds to a certain player's turn, in our case alternating between black and white. The algorithm will find the best action in a certain node by recursively calling itself on each child node and taking the maximum or minimum value (depending on whose turn it is) of that. The base cases of the recursion are the terminal nodes, or leaves in the tree, which would be a checkmate or stalemate in our case. At the base cases the value of the node will be set to -1, 0, or 1 for a black win, stalemate and white win respectively. After that, the action with the best value is chosen.

Since the branching factor in chess is high (around 35 on average) the search tree in the Minimax algorithm would become extremely large. A pure Minimax search would because of this be impossible in practice. For this reason we used a heuristic to be able to evaluate nodes that are not terminal. This makes it possible to set a maximum search depth. The heuristic can be either handcrafted (e.g. counting the pieces currently on the board) or learning based (e.g. an artificial neural network).

To further speed up this algorithm alpha-beta pruning was implemented. Alpha-beta pruning works by ignoring nodes where at least one case has been found to prove its value to be worse than any of the previously explored nodes. This way, the number of nodes that are evaluated are decreased.

A problem with this algorithm arises when using a heuristics that only looks at which pieces are present on the board, and not their position. For example, if a queen can get captured in the next move a heuristic like this would not reflect this in the evaluation and would be very misleading. To mitigate this problem we extended the algorithm with quiescence search. Quiescence search works by defining nodes as either stable or unstable, and if a node is unstable the maximum search depth would be ignored and the algorithm keeps searching until there are no unstable children. In our case we defined a node as unstable if a capture or promotion is possible in the next move.

2.3 Hand-crafted evaluation function

The hand-crafted evaluation function is based on the "Simplified Evaluation Function" designed by Tomasz Mchniewsky, which is thoroughly described on chessprogramming.org[4]. The evaluation function is based on combining piece values which represents the material in play for a player, as well as positioning values that captures how it is better to have certain pieces on certain squares on the board compared to others.

The piece values reflects how valuable different pieces are on the value scale of centipawn, which basically details the value on a scale where a pawn is worth 100 points. The piece values are detailed in Table 2.3. The piece value for a player is calculated as the sum of the piece values for the pieces he has in play.

Piece type	Centipawn value
Pawn	100
Knight	320
Bishop	330
Rook	500
Queen	900
King	20000

Table 1: Centipawn values of the different piece types.

The positional values are specific for each piece type, and are represented by an $8x8$ matrix which stores an integer value for each square to show how valuable a piece of that type is at that square. The positional value matrix for pawns are shown in Table 2.3. In the positional value matrix it can be seen that there is an incentive to advance the central pawns to get control over the center squares, which is known to be a very powerful general strategy. In the value matrix it is also reflected that it is very valuable to advance pawns to promote them, which is reflected by the fact that the most valued positions for pawns are the position just before their promotion. The rest of the position value matrices are detailed on chessprogramming.org[5]. The position value of a player is then calculated by summing this positional values of all of his pieces.

The position values were then scaled to reflect the fact that material differences are more important than positional differences when less material is on board. This was done as shown below:

$$v_{pos\_scaled} = v_{piece}/V_{piece\_max} * v_{pos},$$

[4]https://www.chessprogramming.org/Simplified_Evaluation_Function

[5]See footnote 4

8	0	0	0	0	0	0	0	0
7	50	50	50	50	50	50	50	50
6	10	10	20	30	30	20	10	10
5	5	5	10	25	25	10	5	5
4	0	0	0	20	20	0	0	0
3	5	-5	-10	0	0	-10	-5	5
2	5	10	10	-20	-20	10	10	5
1	0	0	0	0	0	0	0	0
	a	b	c	d	e	f	g	h

Table 2: Positional values for pawns.

so as the piece values is decreased, the relative size of the positional value is decreased. Following this, a player's score is calculated by summing his piece values and positional values. Finally, the difference between the ego player's score and the opponent's score is calculated to get the heuristic score of the position from the ego player's perspective.

2.4 NNUE evaluation function

Evaluating chess positions is an important step in the Alpha-Beta pruning algorithm. A simple, yet effective method is to use a handcrafted heuristic based on specific features on the board, as described in Section 1. Even though this method works fine with easy-to-implement heuristics, making the evaluations reliable in all states of the game is difficult and requires a lot of expert knowledge and weight fine-tuning.

To avoid this issue, a more modern approach is to train a neural network to evaluate chess positions. This method has the advantage that the engine does not "inherit" chess knowledge from the developers (from how features are weighted in the handcrafted heuristic). Instead, it relies on the training data to tell whether a chess position is considered good or bad.

Unfortunately, during Alpha-Beta pruning, millions of positions need to be evaluated each second. This puts a time constraint on position inference that is not possible to achieve with standard forward passed through a complex enough network. Because of that neural networks have not been used for evaluation by chess engines until recently, when Stockfish decided to partially add an Efficiently Updatable Neural Network (NNUE) [5] to their engine. This turned out to be successful, in terms of performance, which makes the NNUE a promising evaluation approach to use in this work. The following sections present how such an NNUE model was implemented and added to the Alpha-Beta pruning algorithm.

2.4.1 Dataset Collection

To create a dataset that could be used to train the model over four million FEN strings (representation of chess positions) were downloaded from Stockfish open Github with positions for engine development[6]. The most recent Stockfish engine was then used to evaluate each position. The evaluation scores were stored in a text file together with the corresponding FEN strings. In order to evaluate each position for a sufficient amount of time, the code was optimized and multi-threaded.

2.4.2 Model Implementation

The NNUE training model was implemented in libtorch (PyTorch C++ API). The aspect that makes NNUE suitable for chess is the encoding of the chess position that is fed to the first layer of

[6]https://github.com/official-stockfish/books

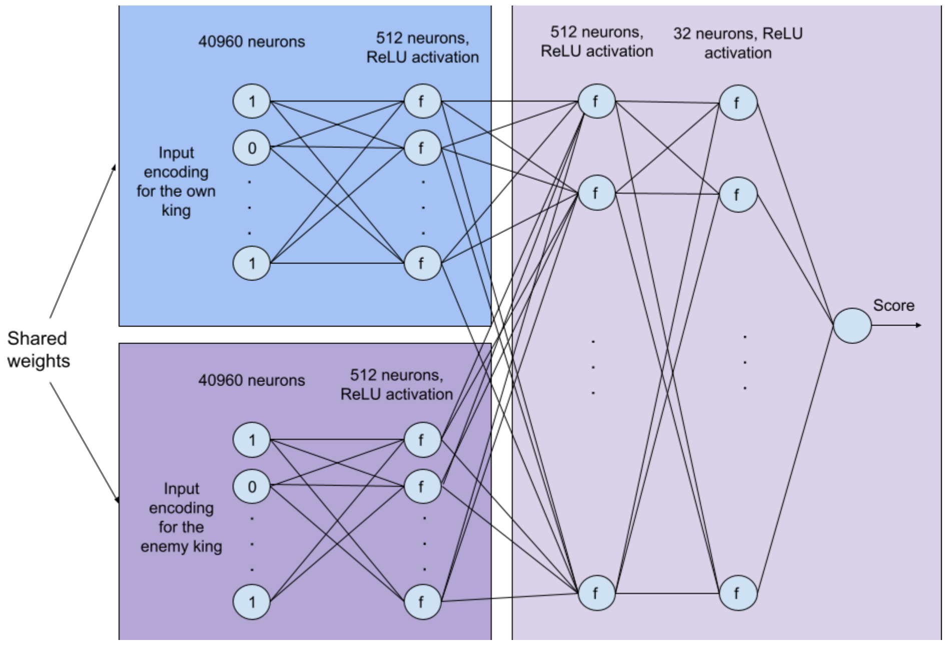

40960 neurons
512 neurons, ReLU activation
Input encoding for the own king
Shared weights
40960 neurons
512 neurons, ReLU activation
Input encoding for the enemy king
512 neurons, ReLU activation
32 neurons, ReLU activation
Score

Figure 2: NNUE architecture.

the network, which enables fast inference. This encoding is called HalfKP [5]. HalfKP transforms each position to a sparse binary vector, where the truth value of each entry is determined from the following equation:

$$\begin{aligned} encoded\_vector_i = 1 \text{ if } (king\_color,\ king\_square,\ piece\_color,\ piece\_type,\ piece\_square) \\ \forall king\_color \in \{own\_king,\ enemy\_king\}, \\ \forall king\_square \in \{A1,\ ...,\ H8\}, \\ \forall piece\_color \in \{own\_perspective,\ enemy\_perspective\}, \\ \forall piece\_type \in \{pawn,\ rook,\ knight,\ bishop,\ queen\} \\ \forall piece\_square \in \{A1,\ ...,\ H8\}, \end{aligned}$$

An important thing to note is that the piece colors are defined as own and enemy instead of black and white. This is a trick that makes it possible to use the same model weights independent of if it is black or white's turn to play. However, in order for this to work the board has to be mirrored vertically for all combinations where the king color is black.

When the input encoding logic was implemented the NNUE model could be created. The network architecture can be seen in Figure 2. The data was divided 90%/5%/5% into train/validation/test datasets. The loss function used during training was a squeezed version of the mean squared error, as defined below:

$$Loss_{mse} = \frac{1}{n}\sum_{i=1}^{n} \mid sigmoid(\frac{prediction[i]}{410}) - sigmoid(\frac{target[i]}{410}) \mid^2$$

During training different hyperparameters (optimizers, number of epochs, learning rate, and batch size) were tested. However, since the training took a long time and the computational power on the available hardware was limited, it was not possible to perform a grid search over different hyperparameter configurations. Instead, the final hyperparameters were decided by training a few epochs with different configurations and choosing the one that seemed most promising based on train and validation loss trends. The following hyperparameters were used in the final model:

Optimizer: Stochastic gradient descent

Number of epochs: 20

Batch size: 128

Learning rate: 0.1

2.4.3 Efficient inference

To evaluate positions during search special inference logic was added. This was done to avoid expensive encoding of each position and to reduce the weight multiplications between the input and the first hidden layer - which was possible due to the structure of the HalfKP encoding. Every time a piece moved (unless it was a king) a maximum of three entries could change values in the encoded vector. This was utilized by storing the output from the first hidden layer, for the own player and the enemy player's perspective. The weights of the moving pieces could then be added or subtracted to that output and the result could then be propagated through the rest of the network. This is more efficient since the encoding of the input is avoided and the multiplications are reduced. The indices of the moving pieces were calculated with the following formula (for each perspective):

$$idx = 640 * king\_square + 320 * side + 64 * piece\_type + piece\_square$$

Since the HalfKP encoding represents the relations between pieces and kings, every time a king moves the position has to be encoded and propagated through the complete network. This is expensive, but since it only happens rarely (in relation to other moves) the benefit of the HalfKP encodings, in terms of inference time, is still large.

2.5 SigmaZero: Self-Play and a General Reinforcement Learning Algorithm

The second approach used to create a chess-playing program is an implementation of the *AlphaZero* algorithm described by Silver et al. (2017) [6]. *AlphaZero* is a deep neural network that is combined with a tree search heuristic called Monte Carlo Tree Search (MCTS). In this section, we cover our implementation of the MCTS algorithm, neural network, reinforcement learning algorithm and the training of the model as well as some tricks we used for speeding up the model. We call our implementation *SigmaZero*.

2.5.1 Monte Carlo Tree Search

Our variation of the MCTS algorithm was implemented as a C++ library wrapped around our chess engine. In essence, the algorithm works by iteratively building up a tree of nodes that represent different board states. When a new game starts, a main node representing the current board state is initialized. For each real move that is to be made, the algorithm iteratively builds a tree of moves from the initial position by choosing the most interesting position in the tree to evaluate, making the most promising move given that position, and adding the state given by the new position to the tree. When the algorithm has run for a sufficient amount of iterations and the tree is sufficiently large (usually after a fixed number of iterations), the move that is the best according to the search is selected [7]. Note that the algorithm needs some kind of evaluator that evaluates possible moves and board states.

More formally, a main node N_{root} is initialized for the current board state S. A child node N_{child} is initialized for each possible move from N_{root}. For each tree search iteration, the current tree is traversed from N_{root}. At each depth of the tree search, a node to traverse from is selected based on the UCB1 score, which scores nodes based on their probability of leading to a win, how much our policy, how many times they have been visited before. When a leaf node is reached, it is expanded with child nodes if the reached state is not terminal. The evaluator's policy $\vec{\pi}$, i.e. probabilities of selecting the moves corresponding to the child nodes, are assigned to each child node, and the value of the traversed node is set according to its valuation V. Furthermore, the leaf nodes amount of visits N is increased by one. The value and amount of visits are then backpropagated through the tree, all the way back to N_{root}. This means that for each iteration, one leaf node in the tree is expanded (if it is not a terminal node). The variation of $UCB1$ used in our implementation is given by equation 1.

$$\begin{gathered} UCB1 = \gamma \cdot \pi + V, \\ \gamma = \log \frac{N_{parent} + C_1 + 1}{C_1} + C_2 \cdot \frac{\sqrt{N_{parent}}}{N_{self} + 1}, \\ C_1 = 19652, \\ C_2 = 1.25 \end{gathered} \tag{1}$$

When a total of I iterations have been made, the perceived best move to make in N_{root} is made by selecting the child node to N_{root} that has the highest amount of visits, N. In our case, the algorithm was implemented as a Node class that has a state, value, prior (move probability) and amount of visits stored. Each nodes children and parent node are stored as pointers.

2.5.2 Deep Neural Network as an Evaluator

The evaluator that was used is a shallower version of the network introduced in [6]. The network essentially consists of 4 parts - an input block, a residual stack and two heads, one value head and one policy head. The input of the network is a $8{*}8{*}((12{+}2){*}H{+}9)$ tensor. H is an unsigned integer and corresponds to the amount of previous board states that should be passed to the network. In our case, only the current state of the board was fed to the network, so $H = 1$. The other 9 layers correspond to other features. Table 2.5.2 describes what each layer in the input stack corresponds to.

Input stack layer	Contents
1-12	Each layer is a binary representation of positions for each piece type for both colors
13-14	Amount of repetitions of the position given by 1-12
15	layer of ones or zeros (white or black's turn)
16	amount of total moves
17-18	white's castling
19-20	black's castling
21	amount of uneventful turns

Table 3: Explanation of input feature stack to the network given the case $H = 1$. If $H > 1$, layers 1-14 are repeated for each previous position

The network architecture is described by table 2.5.2. The policy head outputs a tensor of logits for each possible move in the game of chess (8x8x73 in total). In each evaluation of a leaf node in the monte carlo tree search, the outputs of the network are used to estimate the value of that node and its children. The policy heads output is softmaxed, and mapped to the child nodes by a) setting the logits of moves that are non-valid to 0 and b) softmaxing the resulting logits. The value heads output which is a scalar between -1 and 1 is simply assigned as the value of the leaf node.

Module	**Contents**	**Connects to**
Input	Input layer (8x8x21)	
	2d Conv (in: 128, out: 128, kernel size: 3x3, padding: 1)	
	2d BatchNorm (128 features)	
	ReLU	Residual block
Residual block (x10)	Residual layer (split)	
	2d Conv (in: 128, out: 128, kernel size: 3x3, padding: 1)	
	2d BatchNorm (128 features)	
	ReLU	
	2d Conv (in: 128, out: 128, kernel size: 3x3, stride: 1, padding: 1)	
	2d BatchNorm (128 features)	
	Residual layer (join)	
	ReLU	Value head | Policy head
Value head	2d Conv (in: 128, out: 128, kernel size: 1x1)	
	2d BatchNorm (1 feature)	
	ReLU	
	Flatten	
	Linear (in: 64, out: 256)	
	ReLU	
	Flatten	
	Linear (in: 256, out: 1)	
	Tanh	-
Policy head	2d Conv (in: 128, out: 2, kernel size: 1x1)	
	2d BatchNorm (2 features)	
	ReLU	
	Flatten	
	Linear (in: 128, out: 8x8x73	-

Table 4: Neural network architecture

When training the neural network, we use the loss function

$$L = (z - v)^2 - p^T * \log \pi \tag{2}$$

where z is the output of the value head, v is the value of the node based on the MCTS, p is the policy head's output and π is a vector containing the child nodes posterior distribution, i.e. each child's total visits divided by the sum of the children's total visits.

2.5.3 Self-play

The parameters of the network are updated through self-play reinforcement learning. Games are continually played using MCTS guided by the neural network with the latest parameters. When a self-play game reaches a terminal state, its outcome is recorded and it added to a fixed-size first in, first out replay buffer. Board states, move probabilities and game outcomes are continually sampled from the replay buffer and used to update the network's parameters using the loss function and gradient descent.

2.5.4 Implementation

The AlphaZero algorithm is sample-inefficient and requires a lot of computing power to converge towards good play in reasonable time. For this reason we train the network on one GPU, and use multiple GPUs for self-play. The replay buffer stores the 16384 latest self-play game replay states. Each self-play agent receives updated network parameters after the network has been trained on 512 batches of size 256.

For self-play games, we run 600 Monte-Carlo iterations for each move. To speed up the game completion rate, we take inspiration from Wu (2020) [8] and reduce the number of iterations to 100 on 75% of the moves. Reducing the number of iterations has an impact on the quality of the moves, so these are not recorded and used for training. By spending less time on certain moves, we reach board states where checkmates are more likely quicker. Although certain moves are discarded, we found that this reduced the time needed to fill the replay buffer.

Despite this, we found that convergence towards good play was slow. This is likely due to the credit assignment problem [9] (see Section 4.2). As an alternative approach we tried using material score as value function, which is more immediate. Instead of labelling each move in a replay as winning (1), losing (-1) or drawing (0), we score each non-terminal move according the normalized material difference in relative piece value (1 for pawns, 3 for knights and bishops, 5 for rooks and 9 for queens). This should cause the algorithm to learn to make good piece exchanges.

2.6 Elo Estimation

The most common way of measuring the skill of a chess player is the Elo rating system [10]. It is a comparative rating that is initialized to some value and then updated incrementally with each played game. We use Elo rating as a way to evaluate our implementations. To simulate a pool of opposing players we use Stockfish. For each estimation, we intialize the rating to $R = 1500$ and tune Stockfish to play at that level. We calculate the expected outcome as $E = \frac{1}{2}$ and update the rating according to $R' = R + 40 \cdot (S - E)$ where S is the actual score (1 if the game was won, $\frac{1}{2}$ if the game was drawn and 0 if the game was lost). This process is repeated until a convergence criterion is met. The final Elo rating can not be used as an absolute measure of the playing skill of an implementation, but it can be used to compare the different implementations.

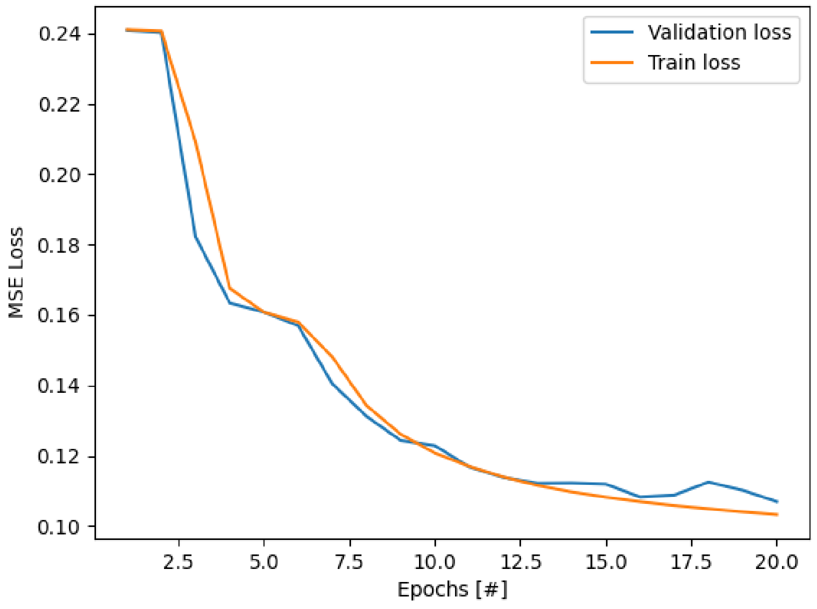


Figure 3: Training and validations loss for NNUE.

3 Results

This section presents the results of the implemented methods.

3.1 Alpha-Beta with hand-crafted evaluation function

The alpha-beta pruning algorithm with a handcrafted heuristic was evaluated using the Elo estimation method mentioned in section 2.5. The Elo was estimated to be around 1350, but since this is the lowest possible score from our estimator we believe that there is a high uncertainty to this result.

However, when playing against humans the engine managed to beat many of us, including all challengers that played against it during the presentation.

3.2 Alpha-Beta with NNUE evaluation function

The NNUE based evaluation engine was not able to win a single game against the lowest rated (1350 Elo) engine, possible to play against through the Elo estimation script (see section 2.6). Further, when played against, the moves made by the engine often seemed kind of random with a lack of intuition.

During the training of the evaluation network, the model seemed to learn some kind of relationship between positions and evaluation scores. Figure 3 shows how the training and validation loss evolved during training.

3.3 SigmaZero

As mentioned in the Section 2.5.4, we trained two different versions of SigmaZero. The first version trained using the traditional value function of the end-outcome of the game. The second version trained using difference in material as the value function, essentially encouraging the engine to make moves that gain material. A graph of the training losses for the two engines can be found in Figure 4 and 5. While the material-value version certainly felt a lot better than the end-outcome version when looking at the games it played, no such conclusion could be draw from only looking at the training loss of the two engines. Therefor we had to find another way to do a quantitative comparison.

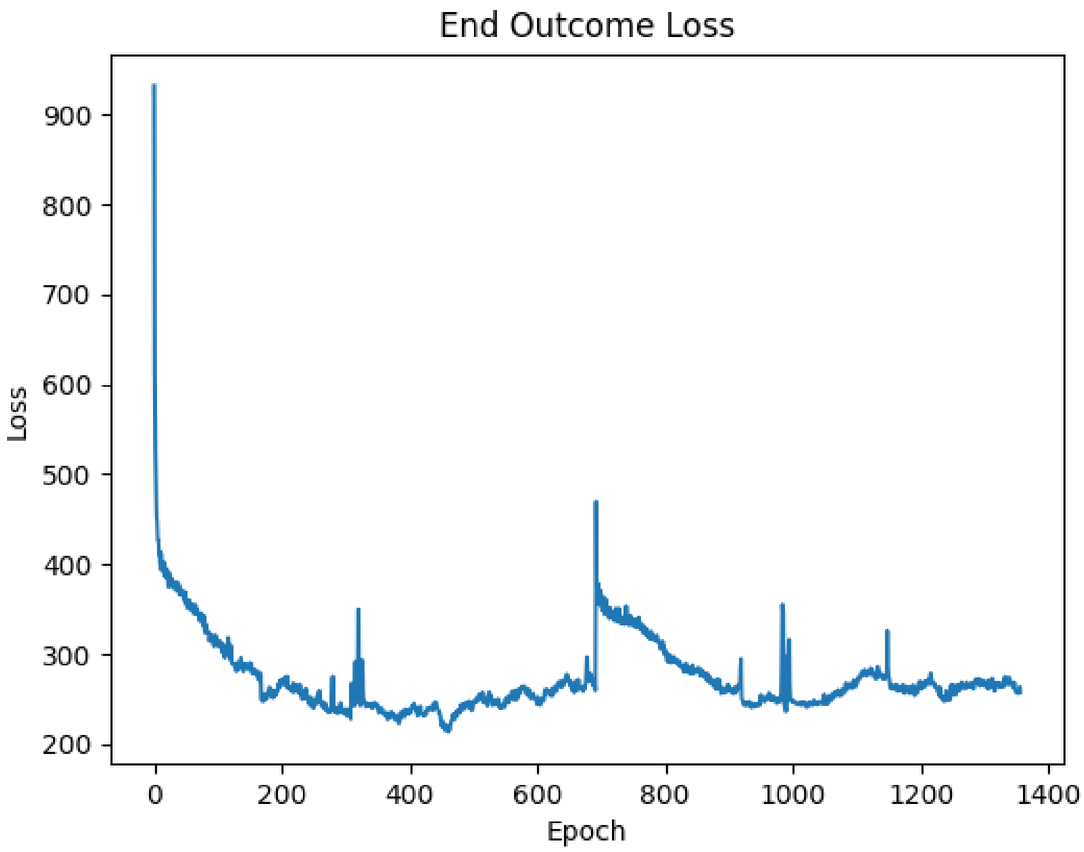


Figure 4: End-Outcome Training Loss

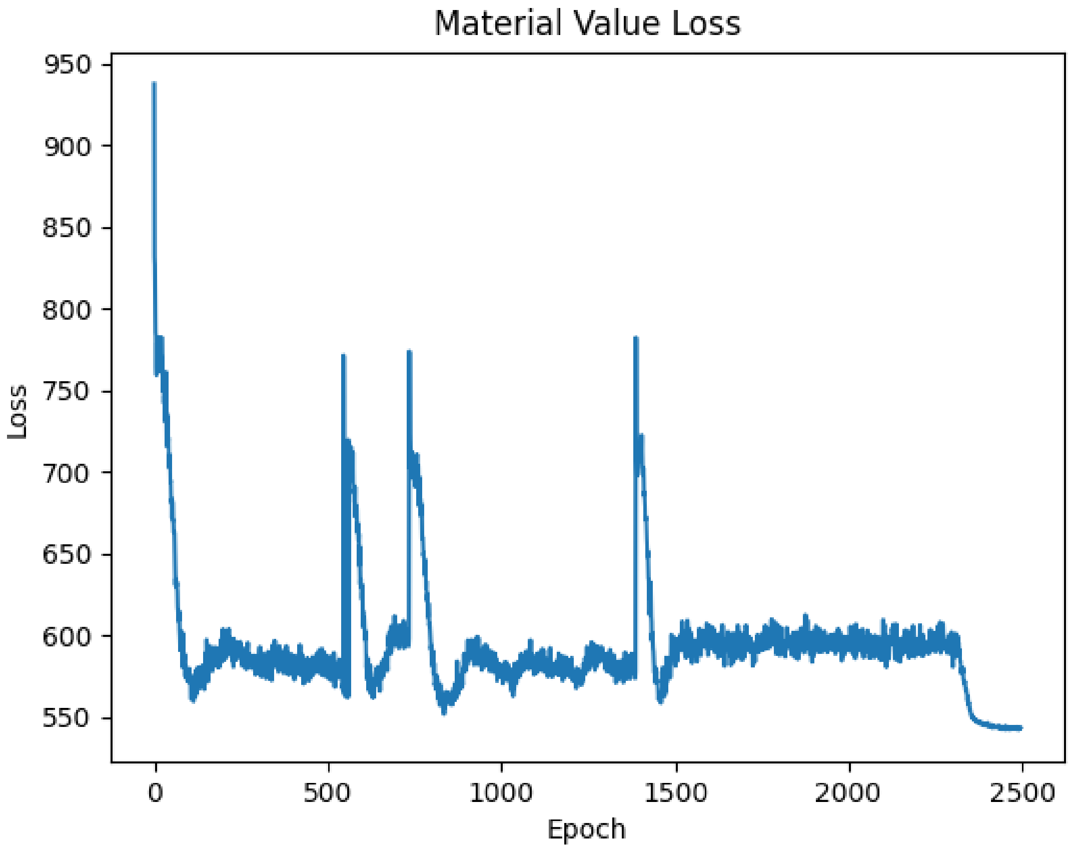


Figure 5: Material-Value Training Loss

There were difficulties when trying to do an Elo estimation of the two engines, because both engines lost every game. Thus we can conclude that neither engine was good enough to beat the Stockfish versions that was used to measure Elo (the worst of which had an Elo of 1350). Instead we had the group member Justus Karlsson play 3 games against each engine to do a quantitative comparison between the two engines. For reference, Justus has an Elo of about 800 on *chess.com*. The results were conclusive, the material-value version won its games by 3-0 and the end-outcome version lost by 0-3. The full games can be found in Section A.

4 Discussion

This section discusses the results obtained in Section 3.

4.1 Alpha-Beta

This section discuss the results of the two Alpha-Beta search engines: the one using handcrafted features and the one using the NNUE for evaluation of positions.

4.1.1 Handcrafted heuristic

Even though the handcrafted heuristic only received an estimate of 1350 Elo, it still performed well against humans players which it was tested against. For a Blitz game where the engine and the opponent had 10 minutes each the engine spent around 10 seconds on each move and reached a search depth of around 4. The consequence of this is that it will play perfectly, in respect to it's own evaluation function from 4 moves in the future, so it will catch any obvious piece blunders which are common by human beginners. However beyond the 4 moves which it has searched it is fairly naive. Quiescence search limits this naivety since it avoids finishing an evaluation in an unstable state, but it is possible that the opponent has a checkmate available 5 moves in the future which is entirely missed by the algorithm.

Additionally, the evaluation function is biased and completely based on human intuition. No machine learning has been utilized, so no complex patterns are detected, only a simple formula that combines piece and position values is returned. However, the upside of this is that the function is extremely fast which allowed it to reach a search depth of 4 on average laptop hardware. This is an enormous advantage compared to more advanced but slower methods as much more can be inferred from a deeper search tree then a shallow one. Additionally, evaluating a chess position is a very complex task, so creating a machine learning method that can do this fairly accurately is not straight forward.

4.1.2 NNUE evaluation

When the handcrafted heuristic was exchanged for the NNUE model the engine played much worse. This was unfortunate but not very unexpected, due to several reasons. Firstly, since chess is a complicated game, in order to achieve good results the network had to be trained for many epochs on a large dataset. This took a long time and since the hardware resources were limited hyperparameters had to be chosen such that the training/validation loss converged in a reasonable amount of time. By lowering the learning rate and increasing the number of epochs there is a chance that the model would have found better relationships between positions and evaluations (converged to a better global minimum).

The model could probably also have been further improved by spending more time on testing and comparing hyperparameters, input encodings, and loss functions. For instance, in the Stockfish implementation of the NNUE, the actual outcome of the game is included in the loss function. Also, virtual feature encodings (over-representations of information) are concatenated with the HalfKP encodings during training. This is said to make the NNUE less sensitive to hyperparameter changes and reduce the convergence time.

Finally, in order for the NNUE to outperform the handcrafted evaluation function extremely fast inference time is required. Even though the NNUE architecture was constructed for that purpose, an efficient enough implementation is difficult to implement. As mentioned in section 3.2, to speed up inference, intermediate results from earlier evaluations were stored and reused. During this process, the weights of the moving pieces were added or subtracted batch-wise (vectorized calculations) by using PyTorch tensors allocated on the GPU. This was probably faster than updating the weights iteratively on the CPU, but an even more efficient approach would have been to perform vectorized calculations on the CPU (to avoid latency from moving memory between devices). However, this is complicated to implement and would have required a lot of hours spent on learning about SIMD optimization, which was considered out of topic for this project.

4.2 SigmaZero

It is somewhat difficult to come to any kind of conclusion based on our results. As mentioned earlier, the loss graphs tell us very little about how well the two different methods actually play chess. The games Justus played against both engines can however give us some limited insight into the quality of the two engines' play. It is clear that the classical end-outcome approach, losing all three games, has not really had the time to learn much. It often seems to behave randomly and frequently blunders pieces. It also often misses free material. The reason behind these results will be discussed further later on.

The material-value approach performed significantly better, beating Justus in all three games. This agent played significantly more aggressively, exploiting all situations where it could gain material, even if the move had long-term negative consequences. This aggressive goal of taking material makes the agent good at exploiting blunders where the opponent player leaves pieces hanging (i.e. undefended), which subsequently makes the agent good at gaining a material advantage over inexperienced chess players. However, a downside of the materialistic attitude of the agent is that it becomes bad at finding checkmates and finishing games.

It was clear that in both approaches, the amount of training resources we had access to was a limiting factor when it came to the performance of the engines. The team at Deepmind used over 5000 TPUs to train AlphaZero over 9 hours [6], compared to the 16 GPUs we used to train over 48 hours. The resulting discrepancy in the number of games played is massive. During its training, AlphaZero played 44 million games, while SigmaZero only played around 100 thousand. We were aware that this would be a problem, but we were initially hoping that the training resources we had would be enough to at least get some intelligent level of chess play using methods similar to AlphaZero.

One central issue with AlphaZero and its end-outcome approach is that reward is uniformly assigned to all moves of a given player. Meaning that a player could play terribly all game, but if it successfully exploits a blunder from the opponent and wins the game, all of its moves, including the terrible ones will be marked as winning moves. This makes it hard to distinguish good moves from bad moves and properly decide which moves made the agent win, presenting an instance of the credit assignment problem. This issue makes the approach quite sample-inefficient, as the agent will require a massive number of games to counteract this problem and properly identify which moves regularly lead to a winning position.

One way to mitigate this problem is to introduce reward shaping. Reward shaping is a classical extension in reinforcement learning, where the reward function is modified using domain knowledge to give the agent intermediate rewards, guiding it towards winning states. Reward shaping has been successfully applied in domains where an agent trained through simple self-play (as in our end-outcome approach) performed no better than a random agent [11]. We had similar results, as our material-value implementation that applied reward shaping gave us much better results in the same limited training time than the end-outcome approach. Reward shaping partially addresses the sample inefficiency of the end-outcome approach.

A Justus vs SigmaZero

A.1 SigmaZero material version wins by 3-0

Game 1 - Black (bot) wins Link to game

1. d4 Na6 { A40 Australian Defense } 2. Nc3 h6 3. e3 Nb8 4. Ba6 bxa6 5. Nb5 axb5 6. Qd3 b4 7.
Qb3 e6 8. Qxb4 Bxb4+ 9. Bd2 Bxd2+ 10. Kxd2 Bb7 11. Nf3 g5 12. c4 g4 13. Ne5 f6 14. Nxg4 h5
15. Nxf6+ Nxf6 16. Rac1 Ne4+ 17. Kd3 Nxf2+ 18. Ke2 Nxh1 19. Rxh1 Rh6 20. Rd1 a6 21. d5 Rg6
22. dxe6 Ke7 23. exd7 Rxg2+ 24. Ke1 Rg8 25. b4 Rf8 26. c5 Ra7 27. a4 Rf6 28. b5 Ra8 29. c6
Bxc6 30. bxc6 Ke6 31. e4 Nxc6 32. a5 Nxa5 33. Ra1 Nc4 34. Rc1 Ne5 35. Rc5 Nxd7 36. e5 Nxe5
37. Rxe5+ Kxe5 38. h4 Rg6 39. Kf2 Rb6 40. Kf3 Rb2 41. Kg3 Qc8 42. Kf3 Qh3# { Black wins by
checkmate. } 0-1

Game 2 - Black (bot) wins Link to game

1. Nc3 { A00 Van Geet Opening } f5?! { (0.00 0.54) Inaccuracy. d5 was best. } (1... d5 2. d4
Nf6 3. Bf4 e6 4. Nb5 Na6 5. e3 Be7 6. h4) 2. e4 fxe4 3. Nxe4 Nf6 4. Nxf6+ exf6 5. d4 Nc6 6.
Nf3?! { (0.75 -0.13) Inaccuracy. d5 was best. } (6. d5 Ne5 7. Qh5+ g6 8. Qe2 Qe7 9. Bd2
b6 10. O-O-O Bb7) 6... Qe7+?! { (-0.13 0.75) Inaccuracy. d5 was best. } (6... d5 7. Bd3
Qe7+ 8. Be3 Be6 9. O-O Qd7 10. c3 Bd6 11. b4) 7. Be2 d5 8. O-O Bg4 9. h3 Bf5 10. Bb5 g6?? {
(0.31 3.87) Blunder. O-O-O was best. } (10... O-O-O) 11. Re1 Be4 12. Bf4?! { (3.84 2.92)
Inaccuracy. Nd2 was best. } (12. Nd2 O-O-O 13. Bxc6 bxc6 14. Qe2 Qb4 15. Nxe4 dxe4 16. c3
Qa4 17. Qxe4 Bd6 18. Qe6+ Kb7) 12... g5?! { (2.92 3.75) Inaccuracy. O-O-O was best. }
(12... O-O-O 13. Bxc6) 13. Bg3?! { (3.75 2.72) Inaccuracy. c4 was best. } (13. c4) 13...
a5?? { (2.72 6.67) Blunder. O-O-O was best. } (13... O-O-O 14. Bxc6 bxc6 15. Nd2 Qd7 16.
c3 Bd3 17. Re3 Bb5 18. a4 Ba6 19. b4 f5 20. Be5) 14. c3?! { (6.67 4.86) Inaccuracy. Nd2
was best. } (14. Nd2) 14... Rc8?! { (4.86 8.95) Inaccuracy. O-O-O was best. } (14... O-O-O
15. Bxc6) 15. Rc1 a4 16. c4 dxc4 17. Rxc4 a3 18. bxa3 Ra8 19. Bxc6+ bxc6 20. d5 cxd5 21.
Rcxe4?? { (8.90 2.92) Blunder. Rxc7 was best. } (21. Rxc7) 21... dxe4 22. Rxe4?? { (3.12
-7.61) Blunder. Qc2 was best. } (22. Qc2 exf3) 22... Qxe4 23. Bxc7 Qc6 24. Nd4 Qxc7 25. Qe1
+ Kd8?? { (-10.47 0.00) Blunder. Be7 was best. } (25... Be7 26. Qb1) 26. Ne6+ Kd7 27. Nxc7
Kxc7 28. Qc3+ Kb8?? { (0.00 5.86) Blunder. Kd7 was best. } (28... Kd7 29. Qd3+ Bd6 30.
Qb5+ Kc7 31. Qc4+ Kb6 32. Qb3+ Kc6 33. Qc4+) 29. a4?? { (5.86 -0.38) Blunder. Qxf6 was
best. } (29. Qxf6 Bg7 30. Qxg7 Rd8 31. Qxg5 Rd1+ 32. Kh2 Rd7 33. a4 Kc8 34. Qg8+ Kb7 35.
Qb3+ Kc7) 29... Ra6 30. a5? { (0.00 -1.02) Mistake. Qb3+ was best. } (30. Qb3+ Kc7 31. Qf7
+ Kd8 32. Qb7 Rd6 33. a5 Rd1+ 34. Kh2 Rd7 35. Qb8+ Ke7 36. a6 Kf7) 30... Ra8?? { (-1.02
6.49) Blunder. Be7 was best. } (30... Be7) 31. Qb4+?? { (6.49 -15.07) Blunder. Qxf6 was
best. } (31. Qxf6 Bg7 32. Qxg7 Rd8 33. Qxg5 Kc8 34. g4 Ra6 35. Qc5+ Kd7 36. Qf5+ Ke8 37.
Qxh7 Rd1+) 31... Kc7?? { (-15.07 8.07) Blunder. Bxb4 was best. } (31... Bxb4) 32. Qb6+ Kd7
33. a6?? { (9.72 0.00) Blunder. Qb7+ was best. } (33. Qb7+) 33... Be7 34. Qb7+ Kd6 35. a7
f5 36. a4 Rhf8 37. g3 g4 38. hxg4 fxg4 39. f4 Rfe8 40. f5?! { (-0.08 -0.73) Inaccuracy.
Kg2 was best. } (40. Kg2 Rec8) 40... h6?! { (-0.73 -0.17) Inaccuracy. Ke5 was best. }
(40... Ke5 41. Qb5+) 41. Qb6+ Kd5 42. f6?? { (0.00 Mate in 9) Checkmate is now unavoidable
. Qe6+ was best. } (42. Qe6+ Kd4 43. Qd7+ Kc3 44. Qc6+ Kb3 45. f6 Bb4 46. f7 Rec8 47. Qe6+
Ka3 48. Qxg4 Kxa4) 42... Bf8?? { (Mate in 9 -0.54) Lost forced checkmate sequence. Bc5+
was best. } (42... Bc5+ 43. Qxc5+ Kxc5 44. f7 Rf8 45. Kf2 Rxa7 46. Ke2 Rxa4 47. Kd2 Ra2+
48. Kc3 Rxf7 49. Kb3) 43. f7?? { (-0.54 -56.09) Blunder. Kg2 was best. } (43. Kg2 Red8 44.
Qe3 Kc6 45. Qe6+ Kb7 46. Qf7+ Kb6 47. Qe6+ Kxa7 48. Qe3+ Kb7 49. Qb3+ Kc7) 43... Re7?? {
(-56.09 0.00) Blunder. Bc5+ was best. } (43... Bc5+) 44. Qb3+ Kc5 45. Qb4+?? { (0.00 Mate
in 6) Checkmate is now unavoidable. Qc3+ was best. } (45. Qc3+ Kd5 46. Qb3+ Kd4 47. Qb4+
Kd5) 45... Kxb4 46. a5 Rxf7 47. a6 Rfxa7 48. Kf2 Rxa6 49. Ke3 Rd8 50. Kf4 Ra4 51. Kxg4 Rd7
52. Kh5 Ra3 53. g4 Ka4 54. g5 Ra2 55. g6 Rb2 56. Kg4 Rb4+ 57. Kf5 Rg7 58. Kf6 Ka3 59. Ke6
Rc4 60. Kf6 Rc1 61. Kf5 Be7 62. Ke6 Rg1 63. Kd7 Rg4 64. Ke8 Bf6 65. Kf8 R4xg6 66. Ke8 Rg5
67. Kf8 Rd5 68. Ke8 Rd8# { Black wins by checkmate. } 0-1

Game 3 - White (bot) wins Link to game

1. Nf3 d5 { A06 Zukertort Opening } 2. Na3 Nf6 3. h3 e5 4. Nxe5 d4 5. g3 c5 6. Nb5 Bd6 7. Nf3 Bd7 8. Nxd6+ Ke7 9. Nxb7 Qc7 10. b4 Qxb7 11.
bxc5 Na6 12. Ba3 Rac8 13. c6+ Qb4 14. Bxb4+ Nxb4 15. cxd7 Nxd7 16. Nxd4 Rhd8 17. c3 Nc6 18. Nxc6 + Rxc6 19. Bg2 Rcc8 20. Bf3 g5 21. Bg2 f5
22. Rh2 Rh8 23. Rc1 h5 24. Bb7 Rc7 25. Bf3 Rcc8 26. Bb7 Rb8 27. Bf3 g4 28. hxg4 fxg4 29. Bd5 Rh7 30. Be4 Rh6 31. Rh1 Nf6 32. f3 Nxe4 33.
fxe4 Rbh8 34. e5 h4 35. Rxh4 Rxh4 36. gxh4 Rxh4 37. d4 Rh1+ 38. Kf2 Rxd1 39. Rxd1 a5 40. Rc1 Kf7 41. Kg3 Kg6 42. Kxg4 Kf7 43. Kf5 Ke7 44.
Kg5 Kd7 45. c4 Kc6 46. Kf4 Kb6 47. Kg3 a4 48. d5 Ka5 49. d6 Kb4 50. d7 Ka3 51. d8=Q Kxa2 52. Qh4 a3 53. Qd4 Kb3 54. e6 a2 55. Qa1 Ka3 56.
e7 Kb3 57. e8=Q Ka3 58. Rh1 Kb4 59. Qe3 Kxc4 60. Qd2 Kb3 61. Qaxa2# { White wins by checkmate. } 1-0

A.2 SigmaZero end outcome version loses by 0-3

Game 1 - Black (human) wins Link to game

1. Nh3 { A00 Amar Opening } Nf6 2. a4 Nc6 3. Ra3 d5 4. b3 e5 5. e4 Bxa3 6. g4 Nxe4 7. Bxa3 Qf6 8. Bg2 Nxf2 9. Ke2 Nxd1 10. Bd6 Bxg4+ 11. Ke1 Bxh3 12. Be7 Kxe7 13. Rg1 Qf2+ 14. Kxd1 Qxg1+ 15. Ke2 Qxg2+ 16. Ke3 d4+ 17. Kd3 Be6 18. c3 Qf3+ 19. Kc2 d3+ 20. Kb2 Rad8 21. h3 Qxh3 22. c4 Nb4 23. Na3 Qh2 24. Nb1 Rd4 25. Na3 Qxd2+ 26. Kb1 Qa2+ 27. Kc1 d2+ 28. Kd1 Qa1+ 29. Ke2 d1=Q+ 30. Kf2 Rd3 31. Nb1 Qa2+ 32. Nd2 Qaxd2# { Black wins by checkmate. } 0-1

Game 2 - White (human) wins Link to game

1. d4 g6 { A40 Modern Defense } 2. Nc3 e6 3. e4 Qh4 4. Nf3 f5 5. Nxh4 a5 6. exf5 h6 7. fxg6 Ra7 8. Qh5 a4 9. Bg5 Be7 10. g7+ Kd8 11. gxh8=Q hxg5 12. Qxg8+ Bf8 13. Qxf8# { White wins by checkmate. } 1-0

Game 3 - White (human) wins Link to game

1. Nh3 { A00 Amar Opening } e5 2. e4 Nf6 3. Qh5 Nxh5 4. g3 d5 5. Bb5+ c6 6. exd5 Qxd5 7. Bxc6+ Nxc6 8. a3 Nd4 9. b3 Nxc2+ 10. Kf1 Nxa1 11. a4 Qxh1+ 12. Ke2 Bxh3 13. Ba3 Qf1+ 14. Ke3 Bg2 15. d4 Nc2+ 16. Kd2 Qxb1 17. g4 Bxa3 18. h3 Bb4+ 19. Ke2 Be4 20. a5 Qe1# { Black wins by checkmate. } 0-1

References

[1] H. W. Kuhn and A. W. Tucker, "John von Neumann's work in the theory of games and mathematical economics," *Bulletin of the American Mathematical Society*, vol. 64, no. 3.P2, pp. 100–122, 1958. DOI: `bams/1183522375`. [Online]. Available: `https://doi.org/`.

[2] L. V. Allis, "Searching for solutions in games and artificial intelligence," 1994, p. 171.

[3] C. E. Shannon, "Programming a computer for playing chess," *Philosophical Magazine*, vol. 41, no. 314, Mar. 1950.

[4] *18 best chess engines of 2022 –dis based on their ratings*, `https://www.rankred.com/chess-engines/`, Accessed: 2022-01-03.

[5] Y. Nasu, "Efficiently updatable neural-network-based evaluation functions for computer shogi," *The 28th World Computer Shogi Championship Appeal Document*, 2018.

[6] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, *Mastering chess and shogi by self-play with a general reinforcement learning algorithm*, 2017. arXiv: `1712.01815 [cs.AI]`.

[7] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *International conference on computers and games*, Springer, 2006, pp. 72–83.

[8] D. J. Wu, *Accelerating self-play learning in go*, 2020. arXiv: `1902.10565 [cs.LG]`.

[9] M. Minsky, "Steps toward artificial intelligence," *Proceedings of the IRE*, vol. 49, no. 1, pp. 8–30, 1961.

[10] A. E. Elo, *The Rating of Chessplayers, Past and Present.* 1978, ISBN: 0668047216 9780668047210.

[11] M. Jaderberg, W. M. Czarnecki, I. Dunning, L. Marris, G. Lever, A. G. Castañeda, C. Beattie, N. C. Rabinowitz, A. S. Morcos, A. Ruderman, N. Sonnerat, T. Green, L. Deason, J. Z. Leibo, D. Silver, D. Hassabis, K. Kavukcuoglu, and T. Graepel, "Human-level performance in 3d multiplayer games with population-based reinforcement learning," *Science*, vol. 364, no. 6443, pp. 859–865, 2019. DOI: `10.1126/science.aau6249`. eprint: `https://www.science.org/doi/pdf/10.1126/science.aau6249`. [Online]. Available: `https://www.science.org/doi/abs/10.1126/science.aau6249`.