Chess Engine Inspired by AlphaZero

Jakub Krakovský, Dominik Liberda

June 24, 2025

1 Introduction

Today, there are many powerful chess engines, thanks to decades of improvements. The first breakthrough moment came in 1997 when IBM's Deep Blue defeated world chess champion Garry Kasparov. Since then, conventional chess engines have become increasingly powerful, while human performance has remained practically the same.

In 2017, DeepMind introduced their self-play reinforcement learning chess engine AlphaZero [6], representing a second big leap in the world of chess engines. This engine revolutionized how modern engines are built and introduced novel strategies and tactics previously unseen in traditional chess engines. So it not only elevated engine performance but also offered valuable insights that influenced how human players approach the game.

We decided to inspire ourselves with this chess engine and build our own self-play reinforcement learning chess engine as an AI project.

2 Related Work

First way to design a chess engine is in the traditional way, where a large tree is created and an algorithm such as minimax with alpha-beta pruning is used to prune the tree to find the best possible move. Some examples of such engines are Houdini [2], Fritz [8], and originally Stockfish [5] which now uses combination of neural networks and the traditional approaches.

The second class of chess engines was created with the introduction of AlphaZero [6], which marked a shift from manually crafted evaluation functions to deep learning and self-play reinforcement learning. These engines learn to play by repeatedly playing games against themselves, gradually improving without any prior knowledge of chess beyond the basic rules. Instead of relying on traditional search trees and hand-tuned heuristics, they use deep neural networks to evaluate positions and suggest promising moves. An examplee of this type of engines is Leela Chess Zero (Lc0) [1], an open-source project that replicates AlphaZero's methodology and continues to evolve through distributed training by a global community.

3 Algorithm and Theory

Deep learning chess engines typically consist of three main parts: Value network, Policy network, Monte Carlo Tree Search(MCTS).

3.1 Value Network

The value network takes the current board state as input and outputs a score that reflects the expected outcome of the game from the perspective of the current player. A score of $+1.0$ indicates a win, 0.0 represents a draw, and -1.0 signifies a loss.

Training the network involves using board positions from played games and minimizing the difference between the network's predicted value and the actual outcome of the game. For each training sample, the final result (win, loss, or draw) is used as the target label corresponding to that board state.

3.2 Policy Network

Policy network accepts board state as an input and returns the probability distribution over all the possible moves in the current position. The better the move, the higher the probability should be. The role of the policy network is guiding the Monte Carlo Tree Search through the tree by suggesting promising moves. Policy network gives non-zero values to invalid moves, so a mask which keeps only valid moves is created, and the new probability distribution is normalized to sum up to 1.

The policy network is trained to get as close to the improved policy created by Monte Carlo Tree Search. As the games are played, the policy network suggests moves to MCTS. MCTS uses this suggestions to explore the tree and get better set of probabilities. For each position along the played game, we save the board state along with the MCTS enhanced probability distribution. The policy network is then trained to minimize the distance between its own output and the MCTS-generated distribution.

3.3 Monte Carlo Tree Search

Actual search algorithm of the engine is MCTS.

The search tree consists of nodes, each node representing a reachable board state and the edges represent actions that a player can take in that state. Each node stores some information:

1. Prior probability given by policy network.
2. Whose turn it is.
3. Legal children positions.
4. Visit count.

5. Total value of this position from all visits.

Starting with an empty tree, MCTS builds up a tree by running a number of simulations. Each simulation adds a single node and has three stages:

1. Select
2. Expand
3. Backup

First, we create the root node of the current board state and expand it. When a node is expanded, the policy network gives us distribution of all valid moves from this position. We then add these positions as children of this node. The children are not yet explored, we just know that we are allowed to make these moves.

After expanding the root node, we make our first MCTS simulation. Our goal is to select an unvisited leaf node and expand it. For that, we use Upper confidence bound (UCB) score. UCB score takes three things into account when producing a score:

1. The prior probability for the child action.
2. The value for the state the child action leads to.
3. The number of times we've taken this action in past simulations.

UCB score can be seen in Figure 1.

$$UCB = Q(s,a) + \left(c \cdot P(s,a) \cdot \sqrt{\frac{lnN}{1+n}}\right)$$

Q = value
P = policy
N = parent node visits
n = action visits
c = exploration hyperparameter
a = current action
s = current state

Figure 1: Upper confidence bound score.

We repeatedly select child nodes using the Upper Confidence Bound (UCB) to guide the search through the tree until a leaf node is reached. Once a leaf node is encountered, we check whether the game has ended in that state. If the game is over, we do not expand the node further. If the game is still ongoing, we expand the leaf node by adding its child nodes based on the current policy, just as we do during regular expansion.

After expansion, we enter the backup stage of the MCTS simulation. In this stage, we update the statistics (visit count and value sum) for every node in the

search path. Updating these statistics will change their UCB score, and will guide our next MCTS simulation.

When the maximum number of simulations is reached, we return the root of the base game tree. The child of this root with the highest visit count is regarded as the best action. When playing against a real opponent, we use the best action directly. During training, we use the probability distribution made from visit counts to select the best child to encourage exploration of longer variations, which could be better.

This chapter was inspired by the explanations of John Varty [7], which helped us a lot to understand the problem and gave us solid theoretical foundation.

4 Implementation

We implemented this project purely in Python. For representing the chessboard and handling all chess-related functionality, we used the python-chess library.

4.1 Neural network

The neural network components were developed using PyTorch.

4.1.1 Input

The input to the network consists of 21 8×8 planes, where each plane encodes an aspect of the chess game state. Specifically, 12 planes represent the positions of the pieces, with 6 planes for each player, one for each piece type (pawn, knight, bishop, rook, queen, king). In addition, 2 planes track threefold repetition, indicating positions that have occurred previously to detect potential draws. One plane represents the color of the player to move, one plane encodes the total move count to provide the network with temporal information about the game's progress. Four planes are used to represent castling rights, with two planes per player (one each for queenside and kingside castling availability). Finally, a plane is dedicated to the no-progress count, tracking the number of half-moves since the last capture or pawn move, as required by the fifty-move rule. The concise view can be seen in Figure 2. The original also used 8 moves history windows, but we decided to keep only the current node as the input for speed reasons.

4.1.2 Architecture

We used Convolutional Neural Network(CNN) with residual blocks, inspired by ResNet architectures. The network processes input tensors with 21 channels using an initial convolutional layer followed by two residual blocks. The number of residual blocks is adjustable, allowing a trade-off between model capacity and computational efficiency-the deeper the network, the stronger its representational power, but at the cost of increased inference time. As we had problems mainly with speed in our implementation, the blocks are kept at minimum.

Chess	
Feature	Planes
P1 piece	6
P2 piece	6
Repetitions	2
Colour	1
Total move count	1
P1 castling	2
P2 castling	2
No-progress count	1

Figure 2: Board input [6].

For reference, the AlphaZero [6] used 19 residual blocks. Each residual block consists of two convolutional layers with ReLU activations and includes a skip connection. At the end, the network splits into the policy head and value head. This architecture also uses proper weight initialization to ensure stable training. Additionally, masking is applied to only keep the valid moves.

4.1.3 Masking

For me, move masking turned out to be a surprisingly interesting aspect of the project, which is why I highlight it here. I drew inspiration from the original AlphaZero paper, where their method of encoding all possible chess moves was something I had not previously considered. They represented moves using 73 distinct 8×8 planes, where each plane corresponds to a specific type of move. In this encoding, each square on the 8×8 grid represents the origin square of a potential move.

- The first 56 encode queen moves: a number of squares [1..7] which to move along one of the compass directions $\{N, E, S, W, NE, NW, SE, SW\}$.
- The next 8 planes encode the possible knight moves from a particular square.
- Finally, the 9 planes encode underpromotions, so promoting a pawn by a move or by 2 diagonal captures either to knight, bishop, or a rook.
- Promotion to the queen in implicitly encoded in the first 56 planes, which could be a little confusing, but really smart.

Concise view can be seen in the figure 3.

Chess	
Feature	Planes
Queen moves	56
Knight moves	8
Underpromotions	9
Total	73

Figure 3: Masking of chess moves [6].

4.1.4 Training

The network is fed the moves generated from given number of games. For every move it needs the:

- MCTS distribution given on the current node for the policy head.
- Result of the game for the value head.
- The input features representing the board state at the move, encoding piece positions and game context.

For policy loss, the cross-entropy is used seen in Figure 4, for value head the mean squared error is used. It can be seen in 5.

$$L_{\text{policy}} = -\sum \pi(s) \log P(s)$$

Figure 4: Cross entropy loss.

$$L_{\text{value}} = (V(s) - z)^2$$

Figure 5: Value head loss.

4.2 MCTS

We implemented the algorithm based on the theory described in Section 3.

4.3 Datasets

In self-play training, the dataset is generated continuously as games are played. A key consideration is determining how to train the neural network effectively. Specifically, how many games to use for each training iteration. Additionally, there is the question of whether to train the network on the entire history of

games or to keep only a recent subset of the data. Since the original AlphaZero paper does not provide details on these parameters, we had to establish our own values through experimentation.

We have used dataset of 1 000 - 10 000 most recent chess positions. For comparison, AlphaZero used a dataset of 500 000 most recent chess games.

5 Performance optimizations

We have used the Python Line profiler [3] tool to identify bottlenecks inside the code. In the initial implementation, the main bottleneck were the calculations inside the monte carlo tree search. This was mainly caused by unnecessary conversions between Python ChessMove objects and their UCI string representations. In addition, we have implemented caching and lazy evaluation to avoid repeated calculations.

After these optimizations, the main performance bottleneck shifted to neural network inference. We have enabled PyTorch model compilation [4] and set it to `max-autotune` which aggressively trades training time for improved inference time. Additionally, we have enabled several PyTorch flags which slightly degrade precision but improve inference time:

```
torch.set_float32_matmul_precision('high')  # TF32
torch.backends.cudnn.benchmark = True
torch.backends.cuda.matmul.allow_tf32 = True
torch.backends.cudnn.allow_tf32 = True
```

After these optimizations, the runtime is roughly evenly split between MCTS and neural network code:

1. MCTS takes roughly 40-50% of run time
2. Generating network input for neural network and legal moves mask takes roughly 20-30% of run time
3. Neural network takes roughly 20% of run time

Country List			
Chess engine	Games played	MCTS Iterations	Training time
Our	1200	50	1 GPU-hour
AlphaZero	44M	800	41 TPU-years

Figure 6: Achieved number of training games and MCTS iterations.

Unfortunately, even with those improvements the achieved performance was far below what training a proper chess engine would require as shown in the comparison in Figure 6. The code could likely be further optimized to achieve

a 2x improvement, but that would still not be enough to train a proper chess engine.

The following subchapters will explore what changes would likely be needed to bring performance to an acceptable level.

5.1 Improving speed of neural network

The speedup obtained by running the neural network on GPU compared to running it on a CPU is surprisingly small. On a CPU, the neural network inference takes roughly 45% of run time. On a GPU, the neural network inference takes 20% of run time - a 2x speedup compared to running on a CPU.[1]

This suggests that the current implementation does not expose enough parallelism to fully exploit the GPU and the overhead of transporting results between CPU and GPU almost dwarfs the speedup.

This is likely caused by lack of batching at inference time. The current implementation performs evaluation of only one chess board at a time. This results in a lot of shuffling data between CPU and GPU.

A better implementation would be for MCTS to collect several board positions and then process them in one batch on the GPU. This would better take advantage of the compute resources on the GPU and make the neural network faster. Chess engines like Leela [1] process 8 to 256 board positions in one batch during inference.

5.2 Replace Python with C++

The MCTS code and generation of neural network inputs and masks could likely be sped up by employing more aggressive caching and lazy evaluation, but this would pose a significant effort which would still not achieve satisfactory performance.

In order to achieve satisfactory performance, the code would have to be rewritten in a more performance focused language like C++.

5.3 Parallelization

The current implementation performs always only one game of chess at a time and therefore runs only on one core and one GPU. Chess engines like Leela [1] use a more distributed architecture, where thousands of games are played in parallel.

6 Installation and startup instructions

Use a recent version of Python (3.12 tested) and install the required dependencies from the supplied `requirements.txt` file. The Chess Engine training can

[1] Without the PyTorch model compilation and PyTorch optimization flags, GPU inference was same speed as CPU inference.

be launched as shown in figure 7.

```
$ : python3 main.py
cuda
ChessNet running on: cuda
Attempting to compile the model for device: cuda...
Model compiled successfully.
Training games (iteration 1/50): 5/5 [Total games=5, Remaining=49
Training iteration 1 with learning rate 0.001000 (warming up)
Training on 2502 positions
Game outcomes: White wins: 5, Black wins: 0, Draws: 5

Training Epochs: 0/5 [00:00<?, 1/5, Avg Loss: 7.1086
Training Epochs: 1/5 [00:01<00:04, 2/5, Avg Loss: 6.4346
Training Epochs: 2/5 [00:01<00:01, 3/5, Avg Loss: 6.1827
Training Epochs: 3/5 [00:01<00:00, 4/5, Avg Loss: 5.9112
Training Epochs: 4/5 [00:01<00:00, 5/5, Avg Loss: 5.6139
Training Epochs: 5/5 [00:01<00:00]

Training games (iteration 2/50): 5/5 [Total games=10, Remaining=48
Training iteration 2 with learning rate 0.001000 (warming up)
Training on 5485 positions
Game outcomes: White wins: 8, Black wins: 0, Draws: 2
Win trend: (+3 compared to previous iteration)
```

Figure 7: An example output of the chess engine.

7 Goal & Evaluation

Our original goal was to train a network reaching an approximate ELO strength of 2000, which, as it turned out, was a very ambitious task. Primarily, we wanted to see if the trained engine could independently discover at least one human-known opening strategy, similar to what AlphaZero achieved during its training process. I believe the fact AlphaZero came up with the same strategy as humans by playing self-play games confirms that the humans have good approach to play the game.

We were able to code all the functions we needed, but through the whole process of implementation we dealt with performance issues. We wanted to get as close as possible to the number of games AlphaZero used for their training, however we did not get nowhere near that. They trained with 44 million self-play games with 800 iterations. For us, even after all the performance improvements we made, one game on average in this setting runs for approximately one minute. This means it would take us 44 million seconds approximately, which is too much to handle (it is approximately 12 222 hours/510 days).

The performance struggle led us to realization that we do not have sufficient compute resources (partly due to selecting Python as the language as choice) to train a proper chess engine. Therefore we have shifted our focus to evaluation of the chess engine on simpler positions.

We have focused on various endgame positions where the game was already won by one of the players with the intention that the network should reliably learn to finish them. Unfortunately we have faced difficulties with training of the neural network, where the neural network reliably learns to draw the games (usually by move repetition or 50 move rule) and eventually all games end in a draw.

It is unclear what causes the network to prefer draws:

1. A bug in the MCTS code, where the evaluation of the position is not handled correctly.
2. A bug in the neural network training which leads to the network preferring draws.
3. Not enough training data - due to the performance issues, we are not able to train the network on many positions. Thus we are not sure if simply training on more positions would eventually lead to the network learning to reliably win, and the preference for draws is there only because draws are quick and easy to learn.

8 Results & Future work

We have not been able to confirm that the chess engine works correctly, other than it being successfully able to learn how to draw games. The selected programming language and architecture decisions limit the achievable performance. Due to the nature of self play (neural network has to learn the rules of the game from scratch), many games are required to successfully train a neural network. It is unclear whether the implementation is wrong or whether we have not reached the sufficient amount of data to train a chess engine.

If we were to start from scratch, we would change several things:

1. Start with a simpler game that requires much less compute resources to learn. Make sure that the code for neural network and monte carlo tree search works, and only then scale up to a more complex problem (chess).
2. Design the chess AI with focus on performance. Our initial thought was that majority of the run time would be spent in the neural network and we did not realize how much book keeping and calculations there would be around the network. We would change the architecture of the system:

   Neural network inference, monte carlo tree search and the chess simulation in C++/Rust. Launch multiple chess games in parallel and then collect the played games and save them to a database. Use the database to train neural network in Python.

This architecture would remove the faced performance bottlenecks and allow us to scale up to more compute resources.

References

[1] Ankan Banerjee. *Leela chess zero*. 2025. URL: https://lczero.org/.

[2] Cruxis. *Houdini*. 2025. URL: https://www.cruxis.com/chess/houdini.htm.

[3] Line profiler. *Line profiler*. 2025. URL: https://kernprof.readthedocs.io/en/latest/.

[4] PyTorch. *Torch Model Compilation*. 2025. URL: https://docs.pytorch.org/tutorials/intermediate/torch_compile_tutorial.html.

[5] Tord Romstad, Marco Costalba, and Joona Kiiski. *Stockfish*. 2025. URL: https://stockfishchess.org/about/.

[6] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. arXiv: 1712.01815 [cs.AI]. URL: https://arxiv.org/abs/1712.01815.

[7] Josh varty. *AlphaZero, A step-by-step look at Alpha Zero and Monte Carlo Tree Search*. 2025. URL: https://joshvarty.github.io/AlphaZero/.

[8] Matthias Wuellenweber. *Presenting: Fritz 19*. 2025. URL: https://en.chessbase.com/post/presenting-fritz-19.