Playing Chess with RL

Michał Kurek, Joel Runevic, Vladimir Petrov
Harvard College

1 Introduction

1.1 Chess as an Algorithmic Challenge

Invented in the 6th century [3], various strategies and algorithms for the game of chess have since been developed. The 1950s and 1960s saw the first fully developed chess programs that leveraged alpha-beta search algorithms and various heuristics, being able to beat humans in tournament play [6]. Most recently, neural networks and Monte-Carlo methods (such as MCTS) have been used to develop and/or learn evaluation functions that have enabled chess-engines to attain superhuman levels of performance, such as in the case of Stockfish and AlphaZero.

1.2 Our Contribution

Our work explores the performance of the baseline MCTS with Upper Confidence Bounds (UCB) algorithm for the purposes of playing chess, as well as its variations where we calibrate exploration vs. exploitation trade-off with self-play. We also compare the algorithms to baseline models, such as a pure Alpha-Beta search to analyze which particular variation is empirically the most effective at playing the game of chess. In our analysis, we also consider various implementation tradeoffs (e.g. time and memory efficiency). The core infrastructure of our chess engine (move generation, maintaining board state et cetera) was written by Michał in C++ over the summer. Thus, our work focused on developing various models for the evaluation and search functions for the engine. Interestingly, our MCTS-based approach was able to obtain a good grasp of the chess fundamentals (e.g. material values of individual pieces and how they change over time as the game progresses).

2 Related Methods

Heuristic-based methods of tree search for the game of chess include performing an exhaustive breadth-first search of the game tree for the first few several layers before utilizing heuristic thresholds at deeper levels of the game tree [2] in order to prune unpromising branches, thereby allowing the algorithm to search deeper levels of the game tree, whilst circumnavigating the problem of combinatorial explosion. Fogel et al [7] utilized an evolutionary algorithm, allowing the model to learn to play chess by simply playing games against itself. Moreover, alpha-beta pruning was used to search the game tree itself and was done so in combination with a neural network trained to assess specific sections of the chessboard (i.e. the output of a neural network was used to determine the evaluation score assigned to a particular board state s). Bandit-based algorithms have been applied to the problem of tree search, particularly in the context of games such as Chess and Go; Coquelin et al [5] discuss UCT - a UCB-based Bandit Algorithm applied to game trees - which allows for a tree-based algorithm that statistically balances exploration and exploitation. Takeuchi et al [10] analyze various methods in which evaluation functions (in the context of game programming) can be tuned in order to strongly correlate with win probabilities (i.e. a low evaluation score corresponds to a low probability of winning and vice versa).

2.1 Evaluation Functions

An evaluation function is a function $f_{\texttt{eval}} : \mathcal{S} \to \mathbb{R}$ that assigns a real value $r \in \mathbb{R}$ to each state $s \in \mathcal{S}$. For any arbitrary state $s \in \mathcal{S}$, we note that $f_{\texttt{eval}}(s) \in \mathbb{R}$ corresponds to the *centipawn* score of state s. This is simply a unit of measure in chess that quantifies the current advantage of a given board state with respect to the side that is to move. If $f_{\texttt{eval}}(s) > 0$, then the side-to-move is winning and $f_{\texttt{eval}}(s) < 0$ is thus a losing state with $f_{\texttt{eval}}(s) = 0$ corresponding to a draw.

Suppose that it is Player A's turn to move, where the chess board is in state s. Ideally, the evaluation function should be close to the *true* reward function r of the chess game itself, so that Player A would simply select the action a that maximizes expected reward:

$$s^* = \underset{s' \in \Delta(s)}{\operatorname{argmax}} f_{\texttt{eval}}(s')$$

where $\Delta(s)$ denote the set of all of the possible states $s' \in \mathcal{S}$ that can be transitioned to from state s in one legal move. That is to say, Player A will simply take the unique move a that transitions the game state from the current state s to the state s^* that is most favorable according to the evaluation function.

2.2 Tree Search + Evaluation Functions

In reality, learning such an "ideal" $f_{\texttt{eval}}(s)$ just as a function of the present state of the board s is practically impossible, as the evaluation itself cannot (usually) predict things like traps or some nuanced lines of play, especially in a game like chess. Therefore, it is essential to "look ahead" and attempt to simulate how the game could play out from next states s'. Ideally, we would like to explore all potential future outcomes (i.e. sequences of moves) with the "look-ahead". However, as we know the game of chess suffers from *combinatorial explosion*: looking ahead by just a few moves in a brute-force manner is computationally infeasible.

Ultimately, evaluation & search are complementary: We first develop a reasonably good heuristic for how good the states are without any look-ahead – our actual $f_{\texttt{eval}}$. Then, we attempt to improve the look-ahead exhaustive tree search by not exploring branches where we already have sufficiently-high confidence about the state's outcome due to $f_{\texttt{eval}}$. Luckily, RL approaches, namely self-play, can help us both to tune $f_{\texttt{eval}}$ and to efficiently incorporate it in the MCTS search.

The subsequent sections of the paper are organized as follows. In the remains of this section, we describe current State-of-the-Art (SOTA) chess engines and how they relate to our work. Section 3 goes into technical details of our methods for both $f_{\texttt{eval}}$ and the MCTS search strategy. Section 4 outlines implementation, while Section 5 presents results and our analysis. Finally, Section 6 discusses future directions and concludes.

2.3 Chess SOTA

Stockfish: utilizes Sequential Probability Ratio Testing (SPRT) in its tuning, determining through self-play whether one version of the engine is stronger than another. In SPRT applied to chess engine testing, we set up a null hypothesis (e.g. the new version of the engine is at least 0 Elo stronger than the old one) and an alternative hypothesis (e.g. the new engine is stronger by 5 Elo), with a certain confidence level (typically 95%). The number of self-play games needed to get statistically significant results often turns out to be very large (in the order of hundreds of thousands of games for each little change to the engine's parameters), especially if the performance difference between the two versions of Stockfish tested is subtle. As such, Stockfish uses a distributed framework (Fishtest) where volunteers donate CPU time to run self-play games between new and old versions of Stockfish. As of December 2023, it's reported that Fishtest has used over 11,600 years of CPU time to play more than 6.8 billion chess games [1].

AlphaGo Zero: though not a chess engine, AlphaGo Zero takes an approach much closer to ours. Unlike its predecessor AlphaGo, it doesn't rely on human game data at all, instead learning Go from scratch through self-play. These games are generated by the most effective player (version of the engine) from all previous iterations. The performance of a new player iteration was measured against the best player to date; if the new player won by a margin of 55% or more, it would replace the best player. Subsequent self-play games would then be generated by this new, superior player.

AlphaZero: a significant departure from the way traditional (Alpha-Beta based) chess engines were tuned in the past, the DeepMind developed engine utilizes self-play without reliance on historical game data or opening books to train a combination of two deep neural networks (separate policy and value networks) and MCTS hyperparameters. Unlike AlphaGo Zero, which created distinct generations of players, AlphaZero maintains a single neural network during training that gets updated continually with each iteration. AlphaZero's training involved 5000 first-gen Google TPUs (for self-play) and 64 second-gen TPUs (for training the neural networks) [9].

3 Our Methodology & RL Components

Drawing inspiration from some of the most successful chess and Go engines described in the previous section, we decided to incorporate self-play to derive both a good $f_{\texttt{eval}}$ and tune the MCTS search hyperparameters. This section first goes into detail of fitting $f_{\texttt{eval}}$ with self-play using SGD and self-play tournaments between different instances of our policies (as elaborated upon in Section 3.1). Then, we introduce our initial MCTS and strategies that helped to improve our MCTS implementation further.

3.1 Self-play Setup

Lishex (Ours)

We adopt a DAGGER-like (Dataset Aggregation) approach for iterative improvement of our engine. Self-play games are generated by the current best iteration of our player (engine), and from every game a set of chess positions is randomly selected and annotated with a label (we elaborate on the specifics of the labels later on in this section). After each training iteration, we evaluate the player's performance based on the mean squared error (MSE) against our dataset of self-play games, rather than direct win-rate comparison as in the case of AlphaZero. This helps reduce the computational needs of training; namely, we do not require potentially hundreds of thousands of games to get statistically significant results. One theoretical disadvantage of our approach, however, is that the respective method can potentially lead to less accurate results (i.e. there is no statistical guarantee), but we found this approach very effective in practice.

We periodically play a large number of games with the new player, using positions from these games to augment our dataset. This approach ensures our model is always trained on fresh, up-to-date data. To improve the rate at which our model learns (as well as to combat memory requirements), as we aggregate new data into the dataset we discard all training data, generated by past players. We utilize chess tournament software, 'cutechess' to perform and record the results of N self-play games of our engine (with N sufficiently large, e.g. $N = 10000$, all played in parallel on an 8 core machine). To manage time constraints, we set short time controls for these games, such as 10 second time limit with 0.05s per move.

The resulting dataset consists of N game results, stored in the Portable Game Notation (PGN) format: a human-readable text format keeping track of each move performed in the game, as well as the final result, which can be a win for white (1-0), draw (1/2-1/2), or loss (0-1). From each game, we randomly select a small subset of positions, say $n = 10$, for further analysis. These positions form the input of our training data. Correspondingly, the actual game outcomes form the output 'label.' Each row of our dataset thus represents a unique position paired with the game's final result. For the labels, we encode the label with respect to the side-to-move in a given position. For example, if in a given position X_i white is to move, and white wins the game, the corresponding label y_i is $y_i = 1$, whereas if black was to move the label would have been $y_i = 0$ (we encode draws as 0.5).

This paradigm of self-play was primarly used to tune evaluation function $f_{\texttt{eval}}$, as described in the following section.

3.1.1 Tuning (Training)

To take a step-back, currently the evaluation function consists of numerous heuristics as the earlier mentioned "centipawn scores" and others. The question is how to combine them all together into a working, effective $f_{\texttt{eval}}$, which is just a tuning process of choosing the "weights of importance" for those different heuristics. During this tuning process, the parameters of our evaluation function are updated through stochastic gradient descent. The training is carried out in a mini-batch fashion, where gradients are estimated numerically.

Each position is fed into our static evaluation function, resulting in some centipawn score P. We then transform this score into a perceived probability W of winning for the player whose turn it is using the following formula (for an in-depth explanation as to why this approach works, see [10]):

$$W = \frac{1}{1 + 10^{-P/4}}$$

Our loss function then becomes the Mean Squared Error of our evaluation function's predictions and the actual outcomes of the games.

Care has to be taken to maintain a balanced distribution of classes in our dataset. Chess is considered a *drawish* game, where a vast majority of games between top players results in draws. The high frequency of draws can skew our evaluation function towards predominantly predicting draws (to minimize the loss). To counter this, we trim our dataset to maintain a balanced ratio of 30% wins, 40% draws, and 30% losses (W-D-L) in our training dataset.

3.2 MCTS + UCB: baseline

As discussed earlier, the game breaks down to the effective decision of which move to make at each timestep from the current position r ("root"). In our tree, each node p represents a position in the game reachable from the root r with a indicator of whose turn it is. Our goal is to understand how promising are the children **from the perspective of the player** whose turn it currently is in the node p. We accomplish this by estimating $AveValue(s)$, which importantly stands for the probability of **the parent** of s to win if they were to make a move into s:
$\pi($ win for p $\mid$ move p -> s$) \approx AveValue(s)$.

In this way, our version of MCTS is closer to the spirit of AlphaZero rather than original MCTS. Below we provide the pseudocode for this search, while the actual code can be found in the "MCTS_Search" function in our GitHub repo. Figure 1 also shows the schematic presentation of the algorithm.

Repeat t = 1 to N ($\approx 100k$):
(1) **Selection.** Start in the root, go down until current node s has all children explored.
Every time choose the child with max UCB score of winning probability for the parent s:
Define s' = nextState(s, a) and define

$$UCB(s,a) = AveValue(s') + C \cdot \sqrt{\left(\frac{log(\#visits(s))}{\#visits(s')}\right)}$$

choose the action $\hat{a} = argmax_a(UCBscore(s,a))$
set $s = nextState(s, \hat{a})$

(2) **Expansion + Simulation**. Simulate the game to get estimated *reward* at s.
This requires obtaining a rollout deeper in the tree until the terminal node (reward = ± 1 or 0 when ties) or until we ran out of budget[1], in which case we estimate the reward with $f_{\texttt{eval}}$.

(3) **Back-propagation**. For all board states "visited" on our path to s, update the stats: let $V(s') = visits(s')$, then

$$AveValue(s') = \frac{V(s')}{V(s')+1} \cdot AveValue(s') + \frac{1}{V(s')+1} \cdot (\pm 1) \cdot reward$$

$$V(s') = V(s') + 1$$

where the sign $\pm$ depends on whose turn it was in the s' node.

(4) **Output**. Finally, in the root we return the action that maximizes the "empirical" score across root's children $argmax_s AveValue(s)$ for all s children of r – without the exploration component.

The overall idea is very much of the RL spirit: balance between exploration of the moves that give high empirical probability of winning (based of what we have seen so far) and the moves that we have not visited much yet. This exact trade-off is controlled by the const C in the UCB score (which we tune), while the "learning" of empirical probabilities happens via rolling-out policies again and again.

3.3 MCTS limitations

There are several limitations of the initial MCTS + UCB approach described in the previous section. (1) First, as mentioned, in the game of chess the tree "explodes", i.e. already on the 4th level (4 moves

[1]Time and memory budget constraints are discussed in the implementation section.

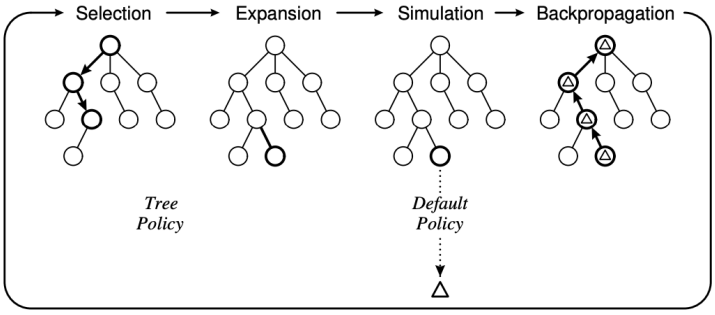


Figure 1: Diagram of a single MCTS search loop. Tree represents chess board positions reachable from the root, default policy can be random or fully deterministic determined by $f_{\texttt{eval}}$.

ahead) we deal with ≈1M of potential nodes. Initial algorithm's **selection** stage always prioritizes unexplored moves: we traverse the tree until the first moment we find the node s who has at least one unvisited child. Thus, most probably, we spend lots of time and memory on moves that are clearly unpromising.
(2) Second, the **simulation** stage as it's proposed originally is very *slow* and very *noisy*. It's slow because we choose to traverse the tree until the very end (terminal node), which might require up to 100 moves. It's noisy because our default policy is completely random (all children are unvisited and so we choose actions by random), and thus the game's output *reward* after 100 moves is very uninformative of initial's winning probability for s.

To address concern (1), we tried two approaches:
Epsilon sampling: during the selection stage, with some probability ϵ we decide to exploit current children, and with probability $1 - \epsilon$ we stick to the previous approach of exploration (i.e. if there are unseen children – explore them). ϵ is a hyperparameter that we tune.
Evaluation based sampling: when exploring unseen children s of current node p, instead of picking them by random, we sample from the distribution *inverse*-proportional to $f_{\texttt{eval}}(s)$. In this way, children with high values of $f_{\texttt{eval}}(s)$, i.e. promising positions for the opponent (since at the position s it's opponent's turn!), would get decreased chances of being explored – because they are less lucrative for the player at parent node p.

To address concern (2), we tried the following ideas:
Budget limits for depth: our simulation rollouts do not go deeper than some *BUDGET* hyperparameter. Then, we output $reward = f_{\texttt{eval}}(s_{end})$ for the node at which a given rollout stopped. This speeds up the simulation part (no need to wait until the end of the game), but also puts more trust on the $f_{\texttt{eval}}$ correctness.
Heavy Rollouts: during the simulation assume that players have preference towards nodes that have lower $f_{\texttt{eval}}$ scores of the children (i.e. opponents is in disadvantage). We tried both probabilistic sampling with children's $f_{\texttt{eval}}$ scores (similar to 1.2 idea), and deterministic $argmin_s(f_{\texttt{eval}}(s))$. This reduces "the noise" of completely random, "light" rollouts, but also slows downs the process since $f_{\texttt{eval}}$ is quite time-expensive to compute.
Averaging out Rollouts: Finally, another option to reduce the noise is simply to do numerous rollouts and output the average reward among all of them.

3.4 Tuning MCTS hyperparameters: manual self-play

In addition to utilizing self-play to tune our evaluation function, we utilize it to tune the hyperparameters of our MCTS algorithm: the rollout budget, the UCB exploration constant C, and the ϵ noise constant used during the selection phase, as well as the rollout policy used. Notably, the methodology we used for hyperparameter tuning is akin to a regular hyperparameter search combined with the methodology employed by AlphaGo Zero: we start with an existing version of our engine, using some initial set of hyperparameters. We then manually[2] adjust these parameters and create a new version of the engine for testing. We then perform a couple hundred self-play games between the two versions

[2]That's why we call it "manual" self-play

of the engine and pick a winner to move onto the next iteration according to the win-draw-lose ratio. Despite its simplicity (and lack of more sophisticated approaches like SPRT used by Stockfish), considering the initial suboptimality of our hyperparameters and the limited computational resources and time, we were happy with the results.

4 Implementation

We have implemented a single-threaded pure MCTS search routine, as well as a UCT variant (UCB applied to Trees). The code for our implementation can be found in our GitHub repository. As discussed in section 3.3, we experimented with multiple rollout policies for our MCTS routine: fully random, evaluation based greedy, and evaluation based sampling. We discuss the comparative performance of these strategies in section 5 (results).

We implement a *Node* class, as well as multiple helper functions for helping perform the MCTS search. Note that we structured our code in a plug-n-play fashion: allowing us to quickly switch between utilizing AlphaBeta, Pure MCTS, or MCTS w/ UCB for our experiments further down the line.

4.1 Time management

In the original formulation of MCTS we perform some number K of iterations, during each iteration performing each step of MCTS: selection, simulation, playout, backpropagation. In practice however, a game like chess imposes an additional constraint: time controls. To address this, our implementation periodically checks during the MCTS process whether the allocated time for a given move has been exhausted, and halts the search if necessary. Then, the best move is reported to the user according to the win/visit ratios accumulated at the root during the search. Note that this is safe to do with MCTS, since it's an *anytime* algorithm: it can be halted at any time during execution and still return a valid move (though not necessarily the most optimal one!).

4.2 Memory management

Due to the anytime nature of MCTS and the way it grows the search tree it's performance improves the longer it runs. As it operates, it allocates more nodes and progressive builds up the search tree. In practice, given constraints of finite memory resources, especially in a game of chess where numerous searches are conducted, it's essential to bound the memory usage of MCTS. In our implementation, we utilize an approach of *stunting*: we impose an upperbound on the memory usage of MCTS and after we run out of this memory, we stop expanding the tree: the selection, simulation, and backpropagation steps continue to be executed until the time limit is reached [8]. To achieve the above we implement a custom arena memory allocator: a big chunk of memory, for instance 2GB, is preallocated for the entire tree during the startup of the engine and each new node allocation during the expansion step utilizes only this part of memory. This not only bounds the memory used by the engine, but also drastically improves our search performance by eliminating the need for dynamic memory allocation at each expansion step: our engine experienced a fivefold improvement in its nodes (positions) searched per second (NPS) rate after switching to our arena allocator.

4.3 Evaluation function

Lishex employs a traditional handcrafted evaluation function (HCE), as opposed to neural-based approaches of modern engines like Stockfish and AlphaZero. The evaluation at its core is a linear combination of various independent features, each assigned a specific weight. These features range from human-derived chess concepts (such as king safety and mobility of pieces) to direct features of the chessboard like the number and types of pieces and their spatial distribution. We also consider more complex interactions between pieces: a pawn supporting a minor piece or a queen attacking a hanging pawn.

In addition we split our evaluation function into two distinct phases: the middlegame and the endgame, each with a separate set of weights. For any given board state s, we calculate two separate scores: one for the middlegame and another for the endgame. As the game progresses, we linearly interpolate between the two to reflect the transition from one phase to the other. We use a material balance based

heuristic to determine the current phase of the game: the less pieces there is on the board the closer to the endgame we expect to be.

5 Results

We split this section into two parts: sole analysis of learned patterns in the evaluation function and best strategies of MCTS as identified by our manual self-play.

5.1 Evaluation Function analysis

We rescale the centipawn scores assigned by Lishex so that one pawn is equivalent to 100 centipawns, as detailed in Table 1. Note that our results (especially for the endgame) align well with the commonly held human scale of material values: 1 for a pawn, 3 for knights and bishops, 5 for rooks, and 9 for queens [4].

In figure 2 we observe quite a few interesting trends. Though initially random, over time Lishex steadily increases its belief in the high value of queens and the relatively lower significance of pawns. Notably, Lishex learns that pawns gain value in the endgame, likely due to their potential for promotion (playing a decisive role in the outcome of many games). It also realizes that bishops are marginally stronger than knights and interestingly suggests that queens and rooks become slightly less valuable in the endgame. This latter finding is particularly intriguing: for queens we hypothesize that Lishex learns that their higher mobility becomes less and less useful as there are fewer pieces on the board to be attacked. For rooks however, the typical human heuristic says that they become more and more valuable as the game progresses, as their movement and attacks are less obstructed than in the middlegame. That said, the values of the parameters don't deviate much from the human nor the AlphaZero values, making us confident in Lishex learning correctly!

Arguably our engine didn't learn chess from scratch the way AlphaZero did: except material values, sensible defaults were used for all other parameters of the evaluation function, giving us a head start. Despite that, we were happy to see our self-play work out and Lishex learning sensible material values for the different piece types.

Table 1: Comparison of Material Values (pawn relative)

Piece	**Human**	**AlphaZero**	**Lishex (middle game)**	**Lishex (endgame)**
Pawn	1	1	1	1
Knight	3	3.05	5.75	2.72
Bishop	3	3.33	6.42	2.84
Rook	5	5.63	9.29	4.95
Queen	9	9.5	17.51	9.56

5.2 MCTS analysis

Overall, the best choice of strategies was **Epsilon sampling** with $\epsilon \approx 0.5$ and utilizing **light rollouts** for the simulation part. Below we provide the discussion of the comparative performances between different policies and why believe the final choice turned out to work the best.

5.2.1 Choice of selection

As described in the methodology section 3.3, Epsilon sampling approach tends to **exploit** more: in roughly half of the cases we use the children we have seen already (and thus we are confident about their scores), rather than unseen children. The problem with aggressive **exploration** approach proposed initially, i.e. when we prioritize any child we have not seen yet (which also corresponds to $\epsilon = 0$) is that we spent too much of compute time and memory to explore all the children *in breadth* from the root. As a result, our tree depth practically never exceeded a value of 6 (with the searching time limit of 10 seconds), which corresponds to only looking 3 steps ahead for both of the players. This means that for child nodes 3 steps ahead we had just a small number of estimates of rewards from simulation rollouts, perhaps implying that those estimates were still very noisy and uninformative.

Figure 2:

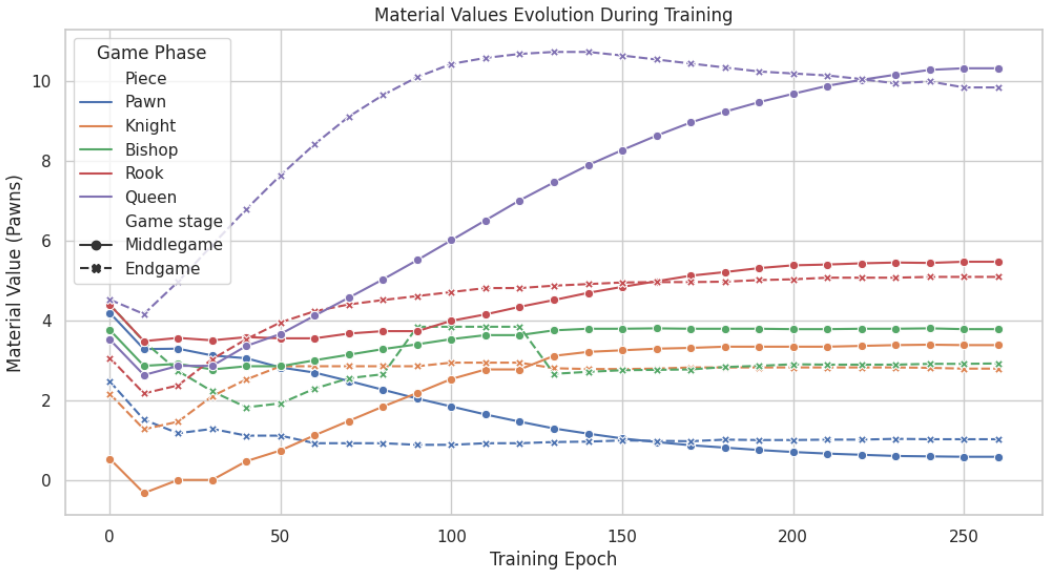
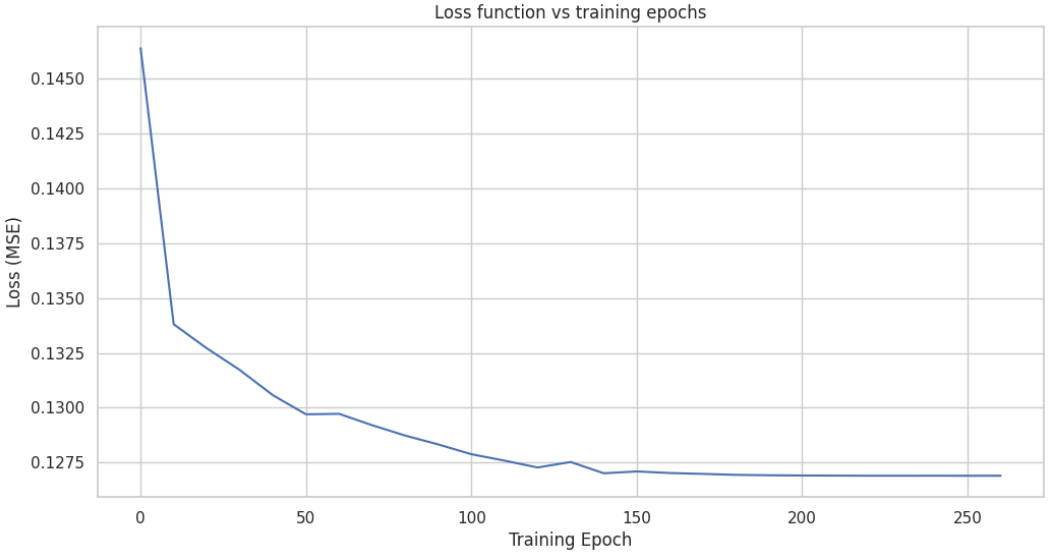


Figure 3:



On the contrary, with $\epsilon \approx 0.5$ our tree depth was always at least 12 (and it increased up to 18 as we approached the endgame), because we were exploring less. In such a scenario, child nodes 3 steps ahead have a much larger number of simulation rollouts, and thus estimated rewards / winning probabilities are more informative, explaining the improved performance.

Evaluation based sampling suggests sampling based on $f_{\texttt{eval}}$ scores to make decisions about moves during the selection stage. However, it turns out that in most cases these scores tend to be fairly close to each other numerically. That's because two children of any particular node would differ just a little bit in terms of centipawn score (say, some player loses a pawn), which results in final winning probabilities difference on the magnitudes of $1 - 5\%$. In other words, such approach would be very similar to that of just random sampling, which we also observed empirically. To address this issue one could do transformations of obtained scores (say, square the winning probability estimates) to let the underlying distribution deviate more from the uniform (i.e. random choice). Further investigation is required for to calibrate the proper approach and see whether it could outperform the **epsilon sampling**.

5.2.2 Choice of simulation

First, **heavy rollouts** (i.e. assuming that players make their moves in the simulation based on $f_{\texttt{eval}}$ values) tended to underperform. Using $f_{\texttt{eval}}$-based sampling suffered from the same issue to that of described in the previous paragraph: the distribution is very close to uniform, and, to add to this, search is slowed down because computing $f_{\texttt{eval}}$ is not cheap. On the other hand, fully deterministic $argmax_s f_{\texttt{eval}}(s)$ (among all children s) strategy seems to be too greedy and over-reliant on the $f_{\texttt{eval}}$. In other words, there was no exploration in the simulations at all, so the search was missing out some non-obvious moves two-three steps ahead.

Finally, **averaging out** several rollouts during the simulation (instead of using just one of them) didn't produce neither positive nor negative improvements. We found a good discussion here of why this is the case. In short, even though one play-out per selection step might be a very noisy estimate of winning probability, subsequent selection steps can go through the same node multiple times. This is especially true when using the **epsilon based** sampling as we have higher probabilities to visit a node and perform more rollouts from it.

Engine	**NPS (k)**
Stockfish	932
AlphaZero	80
Lishex AlphaBeta	524
Lishex MCTS	22

Table 2: Nodes (positions) per Second (NPS) Comparison. Note: AlphaZero ran on DeepMind hardware, whereas we benchmark the other three engines on an Intel Core i7-7700HQ CPU.

6 Conclusion & Future Work

Inevitably, we never expected our results to be close to the state-of-the-art or a carefully tuned alpha beta approach. Our main goal was to see just how far a combination of RL + MCTS could take us within the game of chess given the scope that we had for this project. All in all, we are satisfied with the results; it seems as though our chess engine was able to learn a solid grasp of the chess fundamentals and perform some good openings and moves. As for future work, given our analysis, we would be interested to explore a combination of MCTS and Alpha Beta search. This could be done by deciding which branches to prune during the MCTS search, using α and β values, a proposed method we denote $\alpha\beta MCTS$. We think that this combined approach could pose as an interesting avenue for future work.

References

[1] Stockfish testing framework.

[2] G.M. Adelson-Velskiy, V.L. Arlazarov, and M.V. Donskoy. Some methods of controlling the tree search in chess programs. *Artificial Intelligence*, 6(4):361–371, 1975.

[3] Yuri Averbakh. *A history of chess: From Chaturanga to the present day*. Russell Enterprises, Inc, 2012.

[4] J.R. Capablanca and N. de Firmian. *Chess Fundamentals: Completely Revised and Updated for the 21st Century*. McKay chess library. Random House Puzzles and Games, 2006.

[5] Pierre-Arnaud Coquelin and Rémi Munos. Bandit algorithms for tree search. *arXiv preprint cs/0703062*, 2007.

[6] Carl Felstiner. Alpha-beta pruning, 2019.

[7] D.B. Fogel, T.J. Hays, S.L. Hahn, and J. Quon. A self-learning evolutionary chess program. *Proceedings of the IEEE*, 92(12):1947–1954, 2004.

[8] Edward Powley, Peter Cowling, and Daniel Whitehouse. Memory bounded monte carlo tree search. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 13(1):94–100, Jun. 2021.

[9] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[10] Shogo Takeuchi, Tomoyuki Kaneko, Kazunori Yamaguchi, and Satoru Kawai. Visualization and adjustment of evaluation functions based on evaluation values and win probability. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 22, page 858. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.