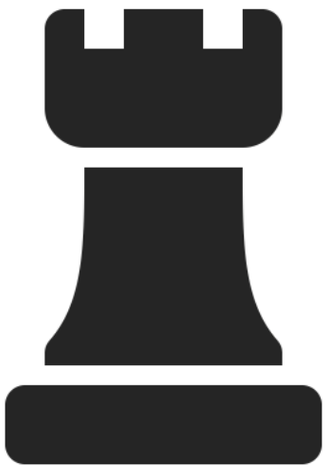
Lunds University
Department of Statistics / LUSEM
Advanced Machine Learning

Creating a Chess AI

Viktor Sjöberg, Asger Bo Rønsholdt

January 5, 2024



Contents

1 Introduction

In the ever-evolving field of computer science, machine learning stands as a pivotal area of study, blending the intricacies of algorithmic design with the dynamic nature of artificial intelligence. The course "Advanced Machine Learning," offered by Lunds University, ventures deep into this domain, exploring a comprehensive array of machine learning methods. Throughout this course, we have not only delved into the underlying theory of these methods but also gained hands-on experience in applying them to solve complex real-world problems.

In this final project, we wanted to be ambitious, and chose to create a chess engine. The game of chess requires deep strategic knowledge, since it inherently has a great deal of complexity. Not only did it require to be able to understand chess board positions in an entirely new format, but also how these observations should be converted into something a computer could read. Next the methods applied had to be applicable to this domain, we found two methods, the minimax algorithm and a neural network with a rather new method in deep learning for optimizing a neural network, NAdam.

1.1 Chess

Chess is a two-player strategy game with a history that spans over centuries, recognized as one of the world's most intellectually demanding and classic board games. Played on an 8×8-square board with alternating light and dark squares, each player begins with sixteen pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. The objective of the game is to checkmate the opponent's king, which means placing it under an inescapable threat of capture. Each type of piece moves in a distinct pattern: bishops move diagonally, rooks horizontally and vertically, knights in an L-shape, and the queen combines the abilities of a rook and bishop. Pawns have the most complex rules for movement, moving forward but capturing diagonally, with the option for a two-square advance on their first move and 'en passant' captures. Chess games can also end in a draw through several scenarios like stalemate, insufficient material, or mutual agreement. With its deep strategy and tactical complexity, chess is a test of foresight, planning, and adaptability (Wikipedia contributors 2023b).

1.2 Goal

Our objective is to harness our collective understanding of chess strategies and our computer science expertise to develop the best possible chess engine within the confines of our available computational resources. Although our processing capabilities may not rival those of high-end engines, we are committed to optimizing our engine's performance by applying advanced algorithms and efficient coding practices. This project is not just about creating a competitive chess engine, it's also an opportunity for us to deepen our programming expertise and broaden our strategic grasp of chess.

2 Data set

Before diving further into the development of our chosen methods. Lets have a look at the data and what a FEN notation means when it comes to board positions. The data set are observations for board positions denoted in FEN(Forsyth-Edwards Notation). Here is the board starting position in FEN; rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1.

The first 8 characters, which are encased in a "/" represents the first rank (row) of the board. Each letter represents a piece (r: rook, n: knight, b: bishop, q: queen, k: king). Lowercase are black and uppercase are white.

The next is the second rank from blacks perspective (7th from white), and here there are 8 pawns. The four 8's are the four ranks which are empty in the beginning of the game and then whites pieces are shown.

The next is either a "w" or "b" which indicates whether or not it's white or black's turn to move.

The next is castling availability → KQkq both sides can castle.

The dash represents no possibility for en passant, which a special case move, where a pawn has the possibility of moving at an angle instead of only forward.

The next two numbers are half moves and full moves, half moves are for counting when only one side has made a move and full moves are if both sides hace made a move.

Lastly we have the evaluation metric, which can either be a numerical value or it can start with "#" indicating a decisive advantage for black(-) or white(+), which would typically be when there is checkmate in a given number of turns.

3 Methods

3.1 Evaluation Functions

In the realm of chess engines, evaluation functions play a crucial role in assessing the strength of a given position on the chessboard. These functions are algorithmic representations that analyze a chess position and assign it a numerical value, typically indicating how favorable the position is for one player over the other. The evaluation considers various factors such as material balance (the relative value of the pieces on the board), piece mobility, king safety, control of the center, pawn structure, and other positional elements.

Having a well-designed evaluation function is vital because it guides the chess engine in making decisions. It helps the engine determine which moves lead to more advantageous positions and which should be avoided. This decision-making process is fundamental to the engine's overall strategy and effectiveness in the game.

Interestingly, a chess engine can be quite potent even with a relatively simple evaluation function. This is because the strength of a chess engine also depends heavily on its ability to explore different move sequences (search algorithms) and predict outcomes. A simple evaluation function, if well-constructed, can provide sufficient insight for the engine to make strong moves, especially when combined with a robust search algorithm. Such a function can effectively balance computational efficiency with strategic depth, making it possible to create a chess engine that is both competent and fast, capable of challenging human players or even more sophisticated engines.

This approach highlights the elegance of chess programming, where a seemingly straightforward evaluation can be the cornerstone of a powerful and strategically adept chess engine.

3.2 One-Hot-Encoding and Scale Evaluation

In order for the *Neural Network*, (NN), to be able to read the data, which is currently in a *FEN*-string format, it needs to be transformed into numerical digits. For this, we define a function in our program, *one_hot_encoder*(). This function takes the FEN-string as an argument and returns an 8×8-matrix. More specifically, the function takes a FEN-string and iterates over the elements in the string, and if the element is a digit, it appends the digit as the number of empty squares, in our case set to 0. If the iteration is a character, it uses our predefined list as the number to append. As mentioned earlier, each piece has a heuristic value indicating the worth of the piece; a larger absolute value indicates pieces with the largest isolated value in a game. Below, in figure 1, we see an example of a matrix outputted from the initial board state when starting a game and showcasing the function use case.

```
one_hot_encoder('rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq')
✓ 0.0s                                                                Python
array([[[ -5.,  -3.,  -3.,  -9., -20.,  -3.,  -3.,  -5.],
        [ -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.],
        [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
        [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
        [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
        [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
        [  1.,   1.,   1.,   1.,   1.,   1.,   1.,   1.],
        [  5.,   3.,   3.,   9.,  20.,   3.,   3.,   5.]]], dtype=float32)
```

Figure 1: One-hot-encoding use case for initial board state

Next, we set up the evaluation column in the data set. We do this by first iterating through the data to find values that start with "#." As mentioned in Section 2, these indicate a decisive

advantage for either black or white. We want these to be the extreme values, so for black, a decisive advantage will correspond to a value of 0. More specifically, closer to this value indicates that the checkmate is closer, and what is meant by closer? It does not take many moves before checkmate is accomplished. Conversely, for whites, a decisive advantage will be values equal to or close to 1. Therefore, having no clear advantages for either black or white is valued around or equal to 0.5. The "#"-values are special cases, although in general we have heuristic values attached to the evaluations since we do not always have decisive advantages. In the beginning of our class, we set the evaluations to be in the range $[-1000, 1000]$, but in reality, our data set can exceed these values, so we want to normalize the evaluations.

3.3 A Simple Neural Network

Having set up the preprocessing steps, the network is ready to be trained. We have chosen to train the network using the PyTorch library, and although this has no influence on how the network performs, it has allowed us to delve deeper into one of the most widely used libraries for deep learning.

First, we define a device; this is not a crucial step, but if, i.e., we wanted to use an external GPU instead of our internal CPU, this would be necessary. After this, we define a *train_loader*() variable that uses the function *DataLoader*() to iterate over the data set in batches of 32 and also shuffles the data. The class we have set up for the data set is a crucial step for using the *DataLoader*(). We have chosen a batch size of 32 for the *DataLoader*(), in essence, this is a hyperparameter that can be tuned, but conventionally, 32, seems to be a good selection. The need for batch sizes comes from the fact that loading the entire data set each iteration will be computationally heavy and be intensive in its use of memory.

Then the architecture of the network is set up; it is important to understand that this network is *very* simplistic. First, we define an input layer, which takes a board position of an 8×8-matrix and converts it into 16 neurons, which are then passed into the first and only hidden layer. This layer consists of 16 neurons which then pass into a single output neuron. This type of NN can be very effective for evaluating single board positions, but will perform worse when faced with entire games and can struggle with the complexities of chess.

This next paragraph on optimization strategies is based upon (Ruder 2016).

After having defined the architecture, we can impose an optimization strategy to the network. Usually we have chosen *Adam*, which consists of having gradients depending on the history of gradients, also referred to as *momentum*. A paper from (Dozat 2016), implemented a more efficient algorithm built on top of *Adam*, which uses *Nestorov's Momentum*. Traditional momentum depended upon past gradients by adding a fraction, γ, of the update vector of the past time step to the current update vector. This contributes to lowering oscillation, but even so, the gradient will not slow down towards the optimum; therefore, we want to incorporate a momentum that can accommodate this. This is where *Nestorov's Momentum* comes in. It does this by calculating the gradient w.r.t. future parameters instead of the current; this ensures that we make huge leaps at the beginning of the process and slows down towards optimimum. Below, in equation (1) the *NAdam* parameter update equation is shown, which incorporates *Nestorov's Momentum* together with *Adam*.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{\upsilon}_t} + \epsilon}\hat{\beta}_1\hat{m}_t + \frac{(1-\beta_1)g_t}{1-\beta_1^t} \tag{1}$$

Where θ is our parameter, $\hat{m}_t$ is the current momentum vector, η is the learning rate, $\sqrt{\hat{\upsilon}_t}$ is the square root of the bias-corrected second moment estimate, also referred to as the variance of the gradients, β_1 is the exponential decay rate for the first moment estimates, and g_t is the gradient at time t.

NAdam has shown to perform very well, the paper by, (Dozat 2016), shows that a *convolutional autoencoder* trained on the MNIST data set, outperforms other optimization strategies, even

though it has more hyperparameters to tune. In an ideal scenario, the network we trained would be a *Convolutional Neural Network*, (CNN), since these have the highest performance for chess problems and work well together with the *NAdam* optimization algorithm. Though from a computation standpoint, this was too demanding, which is why, we instead trained a simpler and less computationally demanding *feed-forward network*. The MNIST data set used in the paper (Dozat 2016), coupled with a CNN, mimics the scenario for our chess problem. The MNIST data set is used to train a network to recognize digits from 64 bits, just as our network uses an 8×8-matrix as input, to represent a chessboard.

The next paragraph will cover a single iteration, which can be generalized for each epoch.

As a next step, we create a *for-loop*, and again from a computation standpoint we chose a low amount of epochs (50).

First we need to call previous functions which load the data set, as well as shaping the data set so that the epochs will be able to compute the parameter updates. We then initialize all parameters to 0 through the *zero_grad*() function in *PyTorch*. We also utilize debugging features, since some issues arose surrounding the FEN-string lengths being too short or long. This issue only arose for a handful of observations so we chose to skip these values.

The next three lines initiates the *backpropagation* and then clips the gradients, which ensures that the *norms* of the vector does not exceed 1. Then the *NAdam algorithm* is called to update the step. To see where the training is during computation each computation is printed out with loss and which epoch the iteration is at. Lastly, the model is saved.

3.4 Minimax Algorithm

The core of the decision-making process in the chess engine is the minimax algorithm. This recursive algorithm explores possible future moves, expanding the game tree to a specified depth (maxDepth). At each depth level, the engine simulates all possible legal moves and evaluates their outcomes using the evaluation function. The minimax principle involves alternating between maximizing the engine's advantage and minimizing the opponent's advantage, akin to predicting the best possible move for both players. When it's the engine's turn, it seeks to maximize the score, choosing the move leading to the most favorable position. Conversely, when simulating the opponent's turn, it assumes the opponent will play optimally and thus minimizes the score, representing the least advantageous outcome for the engine. This alternating maximize-minimize strategy allows the engine to navigate through a vast array of possibilities and choose a move that leads to a strategically advantageous position. (Wikipedia contributors 2023d)

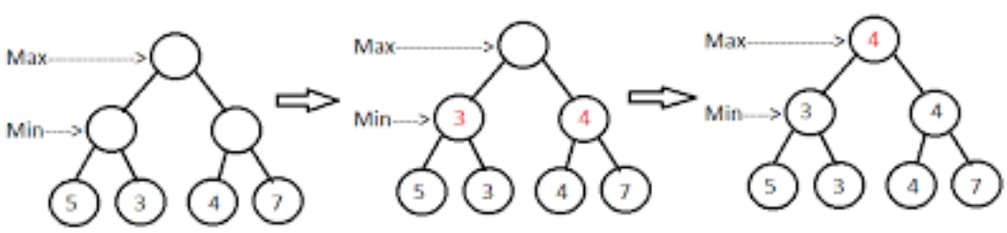


Figure 2: Minimax Algorithm

Figure 2 provides a clear depiction of the minimax algorithm in action, a decision-making process utilized by the chess engine to determine the optimal move. The tree-like structure represents the various outcomes of potential moves, commencing at the top with the initial move by the maximizing player, denoted as 'Max'. The subsequent layer, labeled 'Min', showcases the minimizing player's possible responses. Each branch culminates in terminal nodes assigned specific numerical values, which represent the possible scores of the end positions resulting from the sequence of moves.

As the algorithm commences its evaluation, the Min layer selects the lowest score from the set of immediate possibilities, reflecting the strategy of the opponent aiming to minimize the Max player's

score. These minimum values are then propagated upwards to inform the Max player's decision. In the final step, the Max player assesses the minimum scores and elects the move leading to the highest value, encapsulating the strategy of maximizing their own score. This iterative process of selection and propagation continues until the best possible move is identified at the top of the tree, without any branches being pruned, as the focus here is solely on the minimax decision rule. Figure 2 serves as an instructional snapshot of how the minimax algorithm systematically navigates through a game's decision tree to arrive at the most advantageous move for the initiating player.

3.4.1 Alpha-Beta Pruning

To enhance the efficiency of the minimax algorithm, the engine employs alpha-beta pruning, a technique that significantly reduces the number of nodes evaluated in the search tree. This method involves two parameters, alpha and beta, which represent the minimum score that the maximizing player is assured and the maximum score that the minimizing player is assured, respectively. During the search, if the engine discovers that a move will lead to a worse outcome than a previously examined move (i.e., lower than alpha for maximizing or higher than beta for minimizing), it will prune or cut off further exploration of that branch. This pruning prevents the exploration of moves that won't affect the final decision, thereby reducing computational overhead and allowing deeper searches in the game tree within the same time constraint. Alpha-beta pruning ensures that the engine remains efficient, especially in complex positions with many legal moves, without compromising the quality of the decision-making process (Wikipedia contributors 2023a).

3.4.2 The Evaluation Function

The evaluation function in this chess engine plays a pivotal role in assessing the board's current state. It calculates a numerical score representing the position's favorability from the engine's perspective. This function considers several factors, including the material balance, with individual piece values predefined (e.g., pawn = 100, knight = 320, up to the king = 2000). Additionally, it incorporates Hans Berliner's system of positional values, assigning specific scores to each square on the board for different pieces, reflecting their strategic importance. This system is used to assess the strength of each piece's position, with different tables for each piece type and color. The function also evaluates dynamic aspects of the game, such as the opportunity for checkmate and the number of legal moves in the opening phase, adding a small random factor for variability. Such a comprehensive approach ensures a balanced evaluation, combining both material and positional considerations (Wikipedia contributors 2023c).

4 Results

4.1 Minimax

In chess there is no known number of positions, however its estimated to be $4.59 \pm 0.38 \times 10^{44}$ with a 95% confidence level, with a game-tree complexity of 10^{123}. An average position typically has thrity to forty possible moves. In the opening stages of the game, there are 5,362 distinct chess positions or 8,902 total positions after three moves (White's second move). There are 71,852 distinct chess positions or 197,742 total positions after four moves (two moves for White and two moves for Black). (Wikipedia contributors 2023a)

These values could be compared with how many position the Minimax algorithm evaluate. It will establish how well the alpha beta pruning works and understanding the computational power needed for a engine with a higher degree of depth.

In figure 3 (the engine will allways play as black) the engine after one human move evaluates 10 621 distinct positions and with the evaluation function thinks that moving the knight to attack the pawn is the best move, with a depth of 5. Which is significantly less then the number of positions after 4 moves which conclude that the alpha beta pruning works really well to minimize the number of positions that the engine evaluate and safes computational power.

Figure 4 shows the end of a game where the engine outpreformce the human player. The game lasted 26 moves and the engine evaluated 1 064 453 distinct positions which averages about 40 000 evaluations per move.

-

Figure 3: Opening

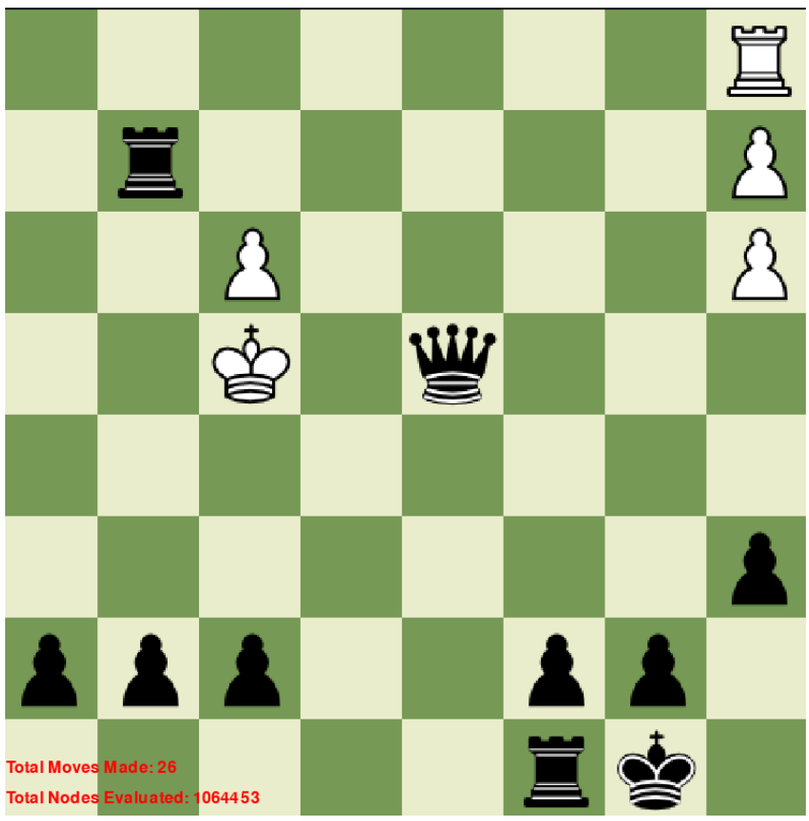


Figure 4: Checkmate

4.2 Neural Network

The evaluation of the NN is done through average loss on an evaluation data set, which is separate from the training data set, which has around one hundred thousand observations. We achieve an average loss of 0.0712, which seems very good for our model. We also choose to calculated the *root mean squared error*, (RMSE), *mean absolute error*, (MAE), and the R^2-score. RMSE achieves a

value 0.2669 and MAE a value of 0.2069, these values support that the model performs mildly well on the evaluation data set. Lastly, a measure for *out-of-sample* R^2, which measures a staggering value of -2.6563, which indicates the trained model performs **much** worse than a simple regression on the mean. This is what is to be expected in a scenario as this. The reason being that our data set is large and complex, a simple network does not have the capabilities of learning the intricacies of the data, with such a structure. As mentioned earlier, it was not to be expected that the model would perform well within a game of chess, but rather perform increasingly well on single board positions. Referring back to Section 3.3, it was mentioned that a CNN would be superior to the *feed-forward* network trained in this project, which the results seem to support.

4.3 Evaluation

5 Conclusion

In conclusion the project not only succeeded in creating more then one competent chess engine but also served as a valuable learning experience, for both exploring chess more in depth and expand our programming knowledge with new algorithms. It provided insights into complexities of chess algorithms, the importance of efficient computation in AI. It has been shown that a simple minimax algorithm can accurately and efficiently learn to play chess without much computation problems. The project has shown that even though NN's are very efficient and able to correctly specificy the underlying structure of much of the data we have today, for this chess engine, it was not the case. A simple architecture was not nearly enough to capture the underlying complexities in chess and there would be a clear need for more computational ability, as well as, more complex methods such as CNN's.

6 Apendix

During the preparation of this work the authors used ChatGPT 4 in order to debugg code that did not work. After using this tool, the authors reviewed and edited the content as needed and takes full responsibility for the content of the publication.

References

Dozat, Timothy (Feb. 2016). "Incorporating Nesterov Momentum into Adam". In: *ICLR*. URL: https://openreview.net/pdf?id=OM0jvwB8jIp57ZJjtNEZ.

Ruder, Sebastian (Sept. 2016). "An overview of gradient descent optimization algorithms". In: *arXiv (Cornell University)*. URL: https://arxiv.org/pdf/1609.04747.pdf.

Wikipedia contributors (2023a). *Alpha–beta pruning — Wikipedia, The Free Encyclopedia*. [Online; accessed 29-December-2023]. URL: https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=1188156145.

— (2023b). *Chess — Wikipedia, The Free Encyclopedia*. [Online; accessed 29-December-2023]. URL: https://en.wikipedia.org/w/index.php?title=Chess&oldid=1190270257.

— (2023c). *Chess piece relative value — Wikipedia, The Free Encyclopedia*. [Online; accessed 29-December-2023]. URL: https://en.wikipedia.org/w/index.php?title=Chess_piece_relative_value&oldid=1192293759.

— (2023d). *Minimax — Wikipedia, The Free Encyclopedia*. [Online; accessed 29-December-2023]. URL: https://en.wikipedia.org/w/index.php?title=Minimax&oldid=1185694928.