Algorithmic Chess Engine Development CSE M.Eng Capstone - Final Report

Adam Abdelrehim

August 2024

1 Abstract

In this paper, we present an exploration into the development of chess engines utilizing neural network models trained on different datasets. Two primary models are investigated: one trained on Stockfish evaluations and another based on personal game data from Chess.com. The Stockfish-trained model struggled with the inherent complexity of chess, exhibiting limitations in capturing nuanced strategic and tactical aspects due to its reliance on abstract evaluation scores and lack of deep search capabilities. Conversely, the personal game-based model excelled in replicating familiar openings and early-middle game strategies but faltered in complex middle-game and endgame scenarios due to limited exposure to diverse positions and tactical motifs.

Additionally, a model trained on high-level games from Lichess, featuring players with ratings above 2300, demonstrated similar weaknesses in tactical understanding and endgame precision despite the strength of its training data. The challenges faced by these models underscore the limitations of neural networks in handling the full spectrum of chess complexities, emphasizing the need for a hybrid approach that combines neural network capabilities with traditional search algorithms and reinforcement learning. This approach aims to integrate the pattern recognition strengths of neural networks with the deep strategic insights and tactical precision offered by established chess algorithms. The paper concludes with recommendations for future improvements, including the integration of advanced search techniques, dynamic evaluation adjustments, and comprehensive training datasets from high-level engines like Stockfish and AlphaZero, to enhance the performance of chess engines in both pattern recognition and strategic depth.

2 Introduction and Background

2.1 Introduction

The evolution of chess engines has significantly transformed the landscape of both competitive and casual chess. Traditionally, chess engines have relied on rule-based systems and extensive opening books to evaluate and select moves. However, recent advancements in artificial intelligence and machine learning have paved the way for more sophisticated approaches, particularly through the use of neural networks. These methods offer the potential to enhance the accuracy and efficiency of move predictions by learning from vast datasets of historical games rather than adhering to predefined rules.

This project focuses on my attempt at the development of a chess engine that leverages neural network-based techniques to evaluate chess positions and return the move that it determines is the best. The primary objective is to create a chess engine trained on a comprehensive dataset of chess games, utilizing a neural network to learn and predict optimal moves based on board states. Unlike a more traditional engine, which might rely upon a combination of opening databases, minimax algorithms, alpha-beta pruning, and endgame tablebases, this approach aims to explore the efficacy of neural networks in capturing the complexities of chess strategy and comparing the engine's performance compared to traditional methods.

The significance of this project lies in its potential to capture the methods built upon in my graduate studies, and provide a stepping stone to future projects, potentially in the chess field, that integratE modern machine learning techniques. Neural networks, particularly convolutional neural networks (CNNs), have demonstrated remarkable success in various domains, including image recognition and natural language processing. Applying these techniques to chess engine development could offer me new insights into game strategy and move evaluation and selection. By training the neural network on extensive game data, my project seeks to test an engine's ability to make strategic decisions and adapt to various playing styles when built upon such a structure.

In the following sections, I will delve into the methodology used to develop and train the chess engine, the results obtained from its performance evaluation, and the broader implications that my results represent. Through this exploration, I aim to gain a truly comprehensive understanding of convolutional neural networks and its applications in strategic game play.

2.2 Project Motivations

My journey with chess began unexpectedly in the 6th grade, sparked by a challenge from my middle school science teacher: anyone who could defeat him in a game would earn \$10, with a caveat: for every subsequent win, the prize money was halved, meaning that, no matter how many times you won, the most you would ever earn was a measly \$20. Though the reward was modest, the most anyone had ever managed was beating him just three times. By the end of 8th grade, however, I had accumulated 50 victories—a record that still stands to this day. This teacher was not only my first serious chess opponent, but also the one who inspired me to pursue engineering, even urging me to consider the University of Connecticut for my future studies.

Chess quickly became an integral part of my middle school years. As an active member of the chess club, I earned a series of trophies in scholastic tournaments (7 of which still remain perched upon my bookshelf), including Top Unrated in the novice section for a K-8th grade tournament, Team MVP for a 4th place team finish, and first place in a U900 section. However, as high school approached, my enthusiasm for chess began to falter, and I only ever played in my town's annual chess tournament. It wasn't until the release of The Queen's Gambit that my passion was reignited, part of the global "Chess Boom" that followed the show.

Returning to the game, along with several other of my closest friends, I found myself repeatedly outplayed by one in online matches. Only later did I discover he had been using the chess engine StockFish to defeat me—an introduction to the world of chess engines that intrigued me deeply. My initial goal when restarting my playing career was simply to surpass my friends in skill, but I soon found myself climbing the ranks, eventually reaching a place within the top 20,000 players on Chess.com's Rapid format, out of over 50 million active users.

As I now conclude my master's degree in computer science and engineering, I am eager to bring together my passion for chess and the skills I've developed in my graduate studies. Building my own chess engine feels like a natural culmination of these passions—a project that not only challenges me but also allows me to contribute something meaningful to the game that has given me so much.

2.3 Rules of the Game

In order to build an effective engine, it is important to understand all of the rules of chess. Chess is a two-player strategy game played on an 8x8 board with 64 squares of alternating colors, usually black and white. Each player begins with 16 pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. The objective is to checkmate the opponent's king, placing it in a position where it is under direct threat of capture ("in check") and has no legal move to escape. Each piece has unique movement rules: pawns move forward one square but capture diagonally; rooks move any number of squares horizontally or vertically; knights move in an L-shape (two squares in one direction and then one square perpendicular); bishops move diagonally any number of squares; the queen can move any number of squares in any direction—diagonally, horizontally, or vertically; and the king moves one square in any direction. Pieces cannot move through or capture friendly pieces, with the exception of knights, which can jump over other pieces. Special moves include castling, where the king moves two squares towards a rook, and the rook moves to the square the king crossed, provided neither piece has moved before, and the squares between them are unoccupied; en passant, where a pawn that moves two squares forward from its starting position can be captured as if it had moved only one square; and pawn promotion, where a pawn reaching the opponent's back rank can be promoted to any piece except a king. The game concludes with checkmate, a draw (due to conditions like stalemate, insufficient material, threefold repetition, or mutual agreement), or resignation.

2.4 Chess Notation

Chess notation is a standardized method for recording and communicating the moves made during a chess game. The most commonly used system is algebraic notation, which represents each move using a combination of letters and numbers. The board is divided into 64 squares, with columns labeled "a" through "h" from left to right, and rows numbered "1" through "8" from bottom to top. Each piece is represented by an initial: "K" for king, "Q" for queen, "R" for rook, "B" for bishop, and "N" for knight, while pawns are typically indicated by the absence of an initial. A move is noted by specifying the piece and the destination square, for example, "e4" indicates a pawn move to e4, and "Nf3" indicates a knight moving to f3. Captures are denoted by an "x" before the destination square, such as "Qxf6" for a queen capturing on f6. Special moves have their own notations: castling is recorded as "O-O" for kingside and "O-O-O" for queenside, promotion is indicated by the piece the pawn is promoted to (e.g., "e8=Q"), and checks and checkmates are marked with "+" and "#" respectively. Chess notation is essential for analyzing games, studying past matches, and communicating strategies, making it a crucial aspect of the game for players at all levels.

Original chess notation, often referred to as descriptive notation, was the standard method for recording chess moves before algebraic notation became widely adopted. In descriptive notation, the chessboard is divided into two halves, each described from the perspective of the player making the move. The files (columns) are named according to the piece that starts on them, such as "QR" for the queen's rook file and "KB" for the king's bishop file, while the ranks (rows) are numbered 1 through 8 from each player's perspective. A move is described by stating the piece, the file or rank it moves to, and sometimes the file or rank it came from to avoid ambiguity. For example, a move might be written as "P-K4" for advancing a pawn to the fourth rank on the king's file, or "N-QB3" for moving a knight to the third rank on the queen's bishop file. Captures are indicated by "x" (e.g., "BxP" for a bishop capturing a pawn). Castling is written as "O-O" for kingside and "O-O-O" for queenside, similar to modern notation. Though descriptive notation was popular, it had limitations, such as being more cumbersome and less consistent across different languages. Its complexity led to its gradual replacement by algebraic notation, which offered a more universal and concise way to record chess moves.

In addition to these traditional methods, recent advancements in chess engine development and the rise of online play have popularized two key forms of notation: Portable Game Notation (PGN) and Forsyth-Edwards Notation (FEN). PGN is a widely used format for recording and sharing chess games. It provides a standardized way to notate the moves of a chess game, along with meta-information such as player names, game result, and opening variations. PGN files are text-based, making them easily readable and accessible for analysis, storage, and sharing. This notation has become an essential tool for documenting and studying games, especially in the context of online play where vast quantities of game data are generated. FEN, on the other hand, is a notation system designed to describe a particular board position in a concise and standardized manner. It encodes the arrangement of pieces, whose turn it is, castling rights, en passant targets, and other relevant details. FEN is particularly valuable for representing individual positions within a game, allowing engines and players to analyze specific scenarios in isolation. Its compact format facilitates easy transmission and processing of board states, which is crucial for both manual analysis and automated chess engines.

Both PGN and FEN have become integral to modern chess practice, enabling efficient handling and analysis of game data. Their widespread adoption has significantly contributed to the development and refinement of chess engines by providing a standardized way to input and evaluate game positions, making them invaluable tools for both players and developers alike. Both of these were essential over the course of the development of my engines for building and testing phases.

2.5 Engine History

The earliest attempts to create a chess-playing machine date back to the mid-18th century with "The Turk," an automaton that allegedly played chess against human opponents. However, The Turk was later revealed to be a hoax, operated by a human hidden inside the machine. Despite this deception, it sparked interest in the possibility of automating chess play, leading to the first real strides in computer chess in the 20th century.

The development of genuine chess engines began in the 1940s and 1950s, coinciding with the birth of modern computing. Pioneers like Alan Turing and Claude Shannon, who are often credited as fathers of artificial intelligence, laid the theoretical groundwork for computer chess. Turing, in particular, developed one of the first algorithms capable of playing a complete game of chess, though it was never implemented on a computer. Shannon, on the other hand, introduced the idea of "minimax" strategy, which became a fundamental concept in game theory and is still used in modern chess engines to evaluate positions by minimizing the opponent's maximum possible gain.

The first actual chess program was written by Alex Bernstein in 1957 on an IBM 704 computer, marking the beginning of serious attempts to create competitive chess engines. Over the next few decades, advances in both hardware and software led to the creation of increasingly powerful engines. By the 1970s, engines like CHESS 4.5, developed by David Slate and Larry Atkin, were able to compete in human tournaments, achieving respectable results. These early engines used brute force algorithms combined with rudimentary positional evaluation to play at a high level, though they still lagged behind the best human players.

The most significant breakthrough in the history of chess engines came in 1997 when IBM's Deep Blue defeated the reigning world champion, Garry Kasparov, in a six-game match. Deep Blue was a supercomputer capable of evaluating 200 million positions per second, using a combination of brute-force search and carefully tuned heuristics to play at an unprecedented level. This match marked the first time a computer defeated a world champion in a standard chess match, signaling a turning point not only in the world of chess but also in the broader field of artificial intelligence.

In the years following Deep Blue's victory, chess engines continued to evolve, becoming more sophisticated and accessible. The advent of personal computers and later the internet made powerful engines like Fritz, Shredder, and Stockfish available to the public. Stockfish, in particular, has become one of the most dominant engines in the world, thanks to its open-source nature and the collaborative efforts of developers worldwide. It routinely outperforms even the best human players and is used extensively for analysis and training.

The latest revolution in chess engines has been driven by artificial intelligence, particularly with the development of AlphaZero by DeepMind in 2017. Unlike traditional engines that rely on brute force and human-crafted evaluation functions, AlphaZero uses deep learning and self-play to teach itself chess. Within just a few hours of training, AlphaZero was able to defeat Stockfish, demonstrating a new level of strategic understanding and creativity in its play. This breakthrough has opened new possibilities for AI in chess, blending machine learning with the centuries-old game in ways that continue to push the boundaries of what is possible.

2.6 Different Engine Types

Chess engines come in various types, each with different methods for evaluating positions and selecting moves. Traditional engines, like Stockfish and Komodo, rely on brute-force search algorithms, such as the Minimax algorithm with alpha-beta pruning, to explore millions of potential moves. These engines use a combination of hand-crafted evaluation functions, which assign values to different board states based on material balance, piece activity, king safety, and other factors. Another type of engine is based on neural networks, as seen in AlphaZero, which employs deep learning techniques to evaluate positions. Instead of relying on brute-force calculations, these engines analyze patterns learned from millions of games to predict the best moves. Hybrid engines, such as Leela Chess Zero, combine elements of both approaches, using a neural network to evaluate positions and a traditional search algorithm to find the best move. These engines illustrate the diversity of strategies in computational chess, each contributing unique strengths to the way machines play the game.

3 Survey of Related Literature

In developing convolutional neural network (CNN) models for chess move prediction, prior research has demonstrated various approaches and outcomes. One study employed CNNs to classify chess pieces and predict moves based on board states, leveraging techniques such as ReLU activation functions, which outperformed tanh despite its inherent limitations in handling negative values. The study detailed a preprocessing pipeline converting Algebraic Chess Notation (SAN) into coordinate-based move representations, essential for effective training. The CNNs were trained over extensive datasets, emphasizing the importance of weight initialization and regularization. Findings revealed that local piece movements, like those of pawns and knights, were predicted with higher accuracy compared to global pieces, such as queens and rooks. Notably, CNNs' performance was influenced by design choices like pooling and dropout, with no significant benefits from dropout and pooling, which were hypothesized to affect the model's ability to learn chess-specific features. Another experiment compared the CNN model against established chess engines, noting strengths in basic tactics like piece captures but struggles in complex, strategic positions. This research underscores the potential of CNNs in chess move prediction while highlighting the limitations in handling intricate strategic considerations, suggesting future work could benefit from integrating evaluation functions to enhance the model's strategic depth and adaptability.

4 Methodology

My first attempt began with the generation of training data, specifically random chess positions that were evaluated by the Stockfish engine. This approach aimed to create a diverse set of board states, capturing a wide range of scenarios that might occur during a game. The randomization process involved simulating a sequence of legal moves, starting from the initial position on a standard chessboard. A maximum depth of 200 moves was set, and a random depth was selected for each board to ensure variety. Once the board reached this randomly selected depth or a terminal game state, it was used as a data point.

The Stockfish engine was then utilized to evaluate these randomly generated positions. Stockfish provided a numerical evaluation of each board state, reflecting the advantage or disadvantage of one side. This evaluation served as the target output for the neural network model during training. The boards were encoded into a 3D tensor representation, where each layer of the tensor corresponded to specific features of the chess position, such as the presence of different types of pieces and the available legal moves for both sides. This representation allowed the neural network to process the spatial relationships and dynamics present on the board.

This first model consisted of several convolutional layers, designed to extract patterns and features from the board's spatial structure. The convolutional layers were followed by dense layers, which served to integrate the extracted features and produce a final evaluation of the position. The model was trained to minimize the difference between its predicted evaluation and the evaluation provided by Stockfish. By iterating over many randomly generated positions and their corresponding Stockfish evaluations, the model aimed to learn to approximate the evaluation function used by Stockfish.

The model was trained using TensorFlow and Keras, with the Adam optimizer used to adjust the weights of the network based on the error between the predicted evaluations and the Stockfish evaluations. The first model, which used a straightforward convolutional approach, provided a baseline for comparison with the more complex residual model. The goal was to assess whether the additional complexity of the residual connections would lead to better performance in evaluating chess positions and ultimately in selecting the best move in a given scenario.

In my second attempt to create a chess model, I tried to create a model that was more aimed at replicating the moves and ideas of both myself and more advanced players. I sourced games in PGN format from two distinct origins. The first dataset comprised games from my own Chess.com account, representing my personal style of play. This dataset reflected the strategic decisions and tendencies I exhibited in various games. The second dataset was drawn from a more advanced pool—Lichess players with ratings exceeding 2300. This high-level dataset represented a different caliber of play, characterized by more sophisticated strategies and tactics that are typically employed by strong players.

I implemented a script to load these PGN files, iterating through each game to extract valuable data. The script utilized the Python Chess library to read and parse each game from the PGN files. The result was a substantial collection of games that served as the foundation for training the model.

Next, I focused on converting the raw chess data into a format suitable for feeding into a neural network. Chess board positions were transformed into 8x8x12 matrices, where each matrix represented the state of the board. This three-dimensional matrix captured the spatial arrangement of all the pieces on the board. Specifically, the 8x8 grid represented the chessboard itself, while the 12 channels encoded the different types of pieces (six for white and six for black). This encoding allowed the neural network to process the complex patterns and structures that emerge during a chess game.

For each move in the games, I extracted the corresponding board position and encoded it as an input sample for the neural network. The output labels were the moves themselves, represented in Universal Chess Interface (UCI) format. To facilitate the neural network's training process, I encoded these moves into categorical data, creating a mapping between each move and a unique integer. This step was crucial for training a classification model that could predict the next move based on the current board state. Unlike the previous model, which would try to play the move it thought Stockfish determined was the best move, this one was designed to play the move that most fit the extensive dataset it was trained on. If done correctly, theoretically, it should play much stronger moves.

With the data prepared, I moved on to designing the neural network architecture. I opted for a CNN due to its proven ability to capture spatial hierarchies in data, which can be particularly useful in the context of a chessboard. The architecture started with a series of convolutional layers, each employing filters to detect patterns such as piece arrangements, potential threats, and possible strategies. These convolutional layers were followed by ReLU activation functions, which introduced non-linearity into the model, enabling it to learn complex decision-making processes.

The convolutional layers were designed to gradually increase in depth, starting with 64 filters in the first layer and doubling to 128 in the subsequent layer. The larger number of filters in deeper layers allowed the model to recognize more abstract patterns, such as long-term strategic plans or potential combinations of moves. After the convolutional layers, the network flattened the 3D matrices into a 1D vector, which was then passed through dense layers. The final dense layer had a softmax activation function, outputting a probability distribution over all possible moves, allowing the model to predict the most likely move given the current board state.

I used the Adam optimizer, a popular choice for training deep learning models due to its efficiency and adaptability. The loss function selected was categorical cross-entropy, which is well-suited for classification tasks. I trained the model for 50 epochs, with a batch size of 64, balancing the need for sufficient learning iterations with computational constraints.

To evaluate the model's performance during training, I employed a validation split of 10%, meaning a portion of the dataset was held back to test the model's accuracy after each epoch. This helped in monitoring over fitting, ensuring that the model generalizes well to unseen data. After training, the model was saved for later use, allowing for easy deployment and testing on new chess positions. With the model trained, I tested its ability to predict moves on new chess positions. The prediction process involved taking the current board state, converting it into the 8x8x12 matrix format, and feeding it into the trained model. The model outputs a probability distribution across all possible moves. I then sorted these probabilities to identify the most likely move, ensuring that the move was legal in the given board position.

5 Results and Analysis

5.1 Model Trained on Stockfish

When creating this model, I believed it had the highest chance of yielding positive results, due to strong ability of neural networks to capture patterns in other tasks. However, this model's performance fell short of expectations, and several factors likely contributed to this outcome. One primary issue is the inherent complexity of chess, which presents a vast and diverse array of positions.

Stockfish's evaluations are derived from a combination of deep search algorithms and finely-tuned heuristics, which consider both immediate tactical threats and long-term strategic considerations. The neural network, despite being trained on a large dataset, may have struggled to capture the nuanced understanding that Stockfish achieves through these methods. Instead of developing a deep comprehension of chess, the model likely ended up with a superficial approximation of Stockfish's evaluations, which may work well in familiar scenarios but fails in novel or less common positions. For example, consider a position such as the following:

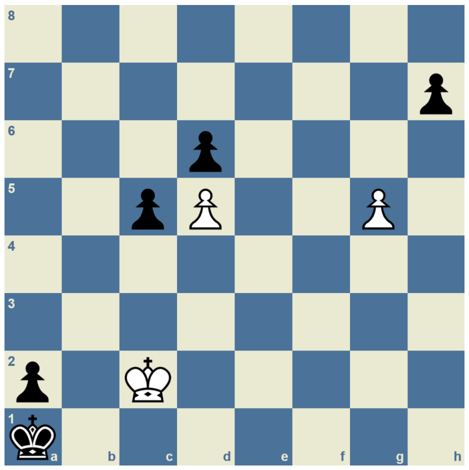


Figure 1: White to move

In this position, Stockfish recognizes M9, or forced checkmate in 9 moves, for White. The reason for this is that it recognizes that Black has no moves for its King, meaning it must move either the c5 or h7 pawns. If Black chooses to move the c5 pawn, White will simply move their King back and forth between c1 and c2, keeping the Black King isolated on a1. After the c5 pawn moves down the board and is captured with the following sequences of moves - 1. Kc1 c4 2. Kc2 c3 (note: the White King cannot capture the pawn on c3, as that would allow the Black King to escape a1) Kc1 c2 Kxc2 - The game is over, as Black must play h6 or h5 and allow the white pawn to Queen on g8 before delivering mate.

However, the engine I created struggled to accurately analyze this position. The move it recommended was Kd3. While this move makes sense in terms of general play, activating the King, moving to the center of the board, this fails to recognize that the position is winning. The evaluation for this position was calculated at -1003 centipawns, which means the engine analyzes this as heavily in Black's favor. This is understandable, as the engine has come to recognize that pawns further up the board are better than pawns closer to their home square. Although it had been trained on a substantial number of games, including many King and Pawn endgames, it had developed a simplistic understanding that having more pieces in better positions generally led to a better evaluation for that side. This approach, however, is flawed, as it doesn't apply universally to all positions. The engine failed to grasp the nuances of certain positions, often overvaluing elements like piece activity—factors that might routinely appear in evaluations—while overlooking concrete strategic or tactical ideas that are crucial in specific scenarios.

Another critical challenge lies in the way the model was used to select moves. After training, the model was tasked with evaluating every possible move in a given position and then choosing the move with the highest predicted evaluation. This process assumes that the model's evaluations are accurate enough to make such a determination, but even small inaccuracies in the predictions can lead to significant errors in move selection. Unlike traditional engines that use iterative deepening and other search techniques to validate and refine their move choices, the neural network's move selection was a one-step process, heavily reliant on the accuracy of its evaluation. This lack of a robust search mechanism likely exacerbated the model's weaknesses, as it had no way to correct or reconsider its initial evaluations.

The engine also showed tremendous bias towards piece captures, even at the expense of an overwhelming attack, piece loss, or checkmate. Consider the position shown below:

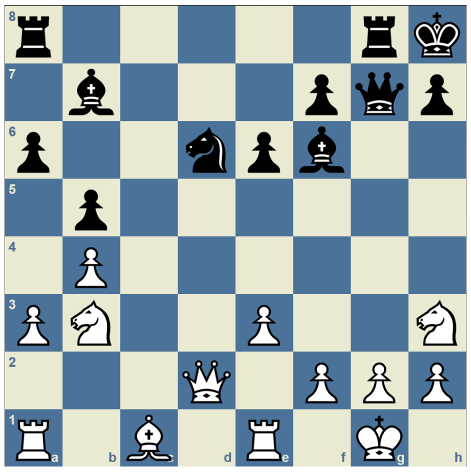


Figure 2: White to move

In this position, the Engine recommends the move Qxd6. At first glance, this seems like it makes a lot of sense. After the capture, White has two Knights and a pawn for Black's Bishop. However, Black can respond with Qxg2# - checkmate. Because this engine is purely focused on evaluations, but the evaluations are the result of training and not brute force calculation, the engine doesn't consider the opponents next move when making a selection for its own move. A potential solution to this could be to add a second layer of calculation in the move selection algorithm, which, after calculating the evaluation for all legal moves, it then calculates all of the legal moves and their respective evaluations for the opponents corresponding move. This might allow the engine to be more accurate in understanding opponents threats.

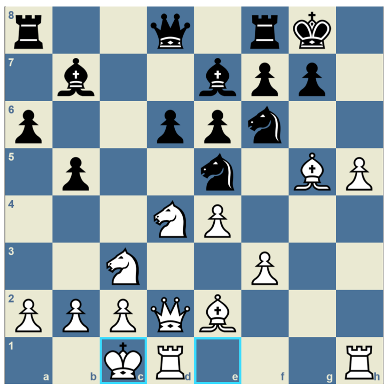
Additionally, the model's architecture and training process may have contributed to its limitations. Neural networks, while powerful, are often data-hungry and require careful tuning of hyperparameters to perform optimally. If the model was not sufficiently complex or if the training data was not diverse enough, the network might have struggled to generalize beyond the specific types of positions it saw during training. This could lead to overfitting, where the model performs well on training data but poorly on unseen data, or underfitting, where the model fails to capture the complexity of the task even during training. This can be best witnessed when comparing the strength of how the model performs in the opening of a chess game versus the middle game or the end game. Since the positioning of the pieces relative to one another is much more prevalent in opening positions (e.g. King is still on the home square, Knights are on c3/f3/c6/f6, common squares in openings), the model has a better understanding of those positions, while it fails to have much data on a position with fewer pieces, as its training data might not have covered a similar position.

Moreover, the approach of training the model solely on Stockfish evaluations may have introduced an additional layer of abstraction that hindered the model's learning. Stockfish evaluations are not raw assessments of a position's value but are influenced by the engine's deep search, which considers future moves and countermoves. The neural network, lacking this search capability, was trying to replicate an evaluation process without access to the underlying reasoning that produced those evaluations. This disconnect likely led to the model making evaluations that, while superficially similar to Stockfish, lacked the depth and accuracy needed for effective move selection.

5.2 Model Trained on My Games

Unlike the previous model, this one was not trained on random positions, but rather the total library of all 12,000 games I played on Chess.com at the time of downloading my game history. While I currently am at a much higher level than most Chess.com players, this encompasses the entire history of my skill, meaning it should feature a more comprehensive set of games, rather than just stronger level ones.

As a first test of my engine, I player against "myself". As Black, it opted


(a) 15. O-O-O

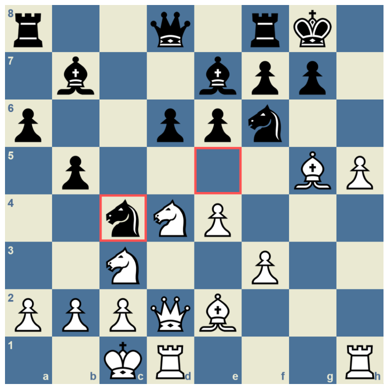

(b) 15. ... Nc4

Figure 3: Position after Nc4

for my traditional opening against 1. e4, which is c5, or the Sicilian Defense. Up until move 13, it played every single move I would have played against the opening I played against it, showing extremely good accuracy.

However, on move 14 and 15, it opted for an idea that was not considered the best by the real Stockfish engine. It went for 14. Ne5 and 15. Nc4. Usually, this is an idea I play in some of my games in a different position. The point is to fork the Queen that is usually on d2 and the Bishop which is usually on e3 (you might notice, it is currently on g5 instead). In addition, the Rook is typically on C8, ready to capture the Bishop if it captures the Knight on c4. However, since the Rook is still on a8, after 16. Bxc4, the engine was forced to respond with bxc4, not as strong of a move.

At this point, the engine is considered losing (Stockfish evaluates the position as about +2.67 at this point in favor of White), but the game is not over yet. However, after 17. h6 and gxh6?!, the game is lost for Black. After 19. Kh7, it is forced mate in 3 (which I didn't play, opting for mate in 6 instead.

When examining the strengths and weaknesses of this engine, there are several things that stand out. One of the notable strengths is the engine's ability to mimic my playstyle, especially in familiar openings and early middle-game strategies. This is a direct result of the engine being trained on a vast dataset that includes a diverse array of positions I've encountered over the years. As a result, the engine is highly adept at recognizing and replicating the patterns I've employed successfully in the past. This makes it particularly strong in the opening phase, where established theory and common strategies come into play. The engine's replication of my Sicilian Defense and subsequent moves up to move 13 in the test game against "myself" demonstrates its strength in mirroring my tactical preferences during the early game.

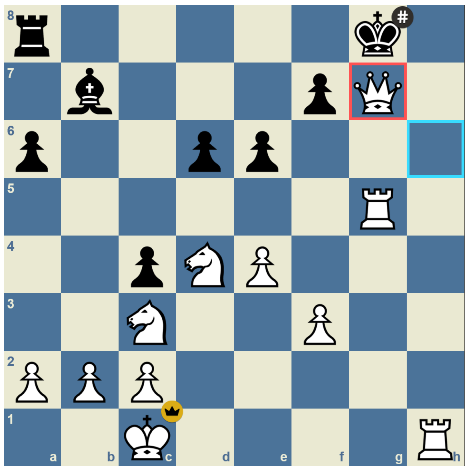


Figure 4: Final Position

However, despite these strengths, the engine also exhibits significant weaknesses, particularly in areas that require deep tactical understanding and precise calculation. One of the most glaring weaknesses is its tendency to fail in recognizing critical captures or tactical opportunities that fall outside of the patterns it has learned. This highlights a fundamental issue: the engine's learning process is based on pattern recognition rather than a deep understanding of the underlying tactics. As a result, it can misapply familiar ideas in positions where they don't fit, leading to sub-optimal moves.

This weakness becomes even more pronounced in complex middle-game positions and endgames, where precise calculation and a deep understanding of positional nuances are crucial. Because the engine is trained to learn and adapt to patterns from my games, it may struggle in situations that require a departure from these patterns. For example, it might fail to find a critical capture simply because it hasn't encountered a similar position in its training data. This lack of exposure to specific tactical motifs can cause the engine to miss winning opportunities or fall into traps that a more tactically aware engine would avoid.

Additionally, the engine's performance in endgames is notably weaker. Endgames often require a precise understanding of material imbalances, king activity, and pawn structure—concepts that go beyond pattern recognition and delve into deeper strategic planning. Since my training data likely contains fewer endgames and more diverse middle-game positions, the engine is less proficient in handling the unique challenges that arise in this phase of the game. It tends to rely on learned patterns rather than an understanding of endgame principles, which can result in sub-optimal moves, particularly in critical moments where accurate calculation is essential.

5.3 Model Trained on Lichess Elite Games

This model was not tuned upon any specific players games, but rather a database I downloaded encompassing 20,000 games from players on Lichess with a rating of 2300 or higher during the month of June 2024. Despite this advantage, the new engine demonstrates similar weaknesses, particularly in tactical understanding and endgame precision.

The model trained on the Lichess dataset, while benefiting from stronger and more varied high-level game data, still fails in crucial areas. It struggles with recognizing and exploiting tactical opportunities that fall outside the learned patterns. This issue mirrors the previous model's difficulty in applying familiar strategies to novel or complex positions. The variability in openings and the wide range of positions encountered in the Lichess dataset contribute significantly to this problem. The engine faces positions it has not been extensively trained on, resulting in a tendency to make sub-optimal moves.

In particular, the high variability of openings in the Lichess dataset introduces diverse and less familiar positions that challenge the engine's ability to generalize effectively. Unlike the more predictable openings found in the personal game dataset, the Lichess games present a broader spectrum of opening theories and subsequent positions. This diversity means that the engine often finds itself in scenarios where it lacks adequate training, leading to poorer performance in tactical and strategic play.

In an example game I played against this engine, I sought to intentionally play a dubious opening. I pushed both my a2 and h2 pawns all the way down to a6 and h6 respectively, as seen below:

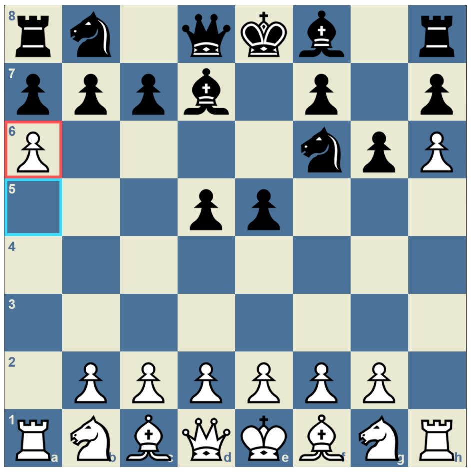


Figure 5: Position after move 6

Since this confused the engine, which had never seen such an approach before, it continued on its path of developing its pieces before abrutly hanging both of its center pawns by move 10.

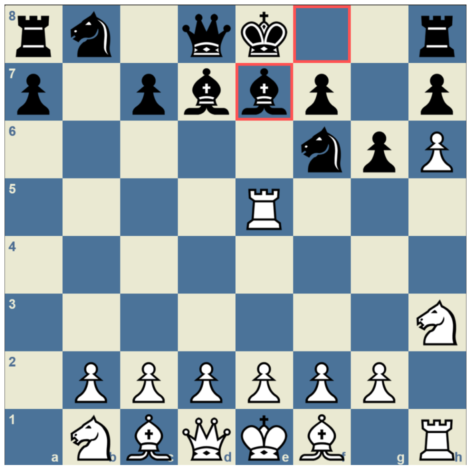


Figure 6: Position after move 10

And by move 15, thoroughly not understanding my lack of development, the engine hung its Queen on d5.

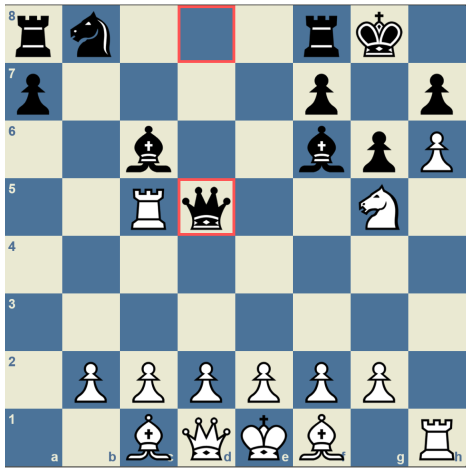


Figure 7: Position after move 15

What this shows me is that the engine gets very easily put out of its preparation by simply playing abnormal openings. Using openings it has not analyzed in its neural network make it much more likely to make a sub-optimal move

In addition, the engine's performance in endgames remains weak, similar to the previous model. The complex nature of endgames requires precise calculation and a deep understanding of material imbalances and strategic nuances, areas where pattern recognition alone is insufficient. Despite the high-level nature of the training data, the engine's reliance on learned patterns rather than comprehensive strategic understanding results in sub-optimal play during critical endgame moments.

Overall, while the Lichess-trained model benefits from exposure to stronger data, the inherent variability and the engine's reliance on pattern recognition continue to limit its effectiveness, especially in unfamiliar positions and complex endgames. This underscores the challenge of building a chess engine that can handle both highly varied positions and the intricate demands of endgame play.

6 Conclusion

During this project, I attempted to integrate neural networks to assess a chess position and recommend what it determined was the best move. Two distinct models were explored: one trained on Stockfish evaluations and another based on personal game data. The first model, trained on Stockfish evaluations, demonstrated a capacity for learning positional patterns but struggled with specific tactical and strategic nuances. Notably, the engine failed to accurately assess endgame positions and complex middle-game scenarios, often resulting in suboptimal move choices. This shortcoming stemmed from the model's simplistic evaluation understanding, reliance on pattern recognition, and lack of a robust move selection process. The absence of deep search capabilities and an overreliance on evaluation scores without considering opponent responses further exacerbated these issues.

The second model, based on a dataset of personal games, showcased strengths in mimicking familiar opening strategies and early middle-game tactics. However, it displayed weaknesses in handling complex positions and endgames, largely due to a lack of exposure to diverse endgame scenarios and tactical patterns. This model's performance highlights a key limitation of neural network-based engines: their tendency to rely heavily on learned patterns and historical data, which may not always translate into accurate evaluations in novel or critical positions.

These results underline a specific weakness of neural network engines that rely solely on neural networks for position calculation. While these models can excel in pattern recognition and replicating known strategies, they often fall short in areas requiring deep strategic understanding and precise calculation. The disconnect from traditional evaluation methods and search algorithms contributes to this limitation.

To address these weaknesses and achieve a more robust chess engine, a hybrid approach could be employed. Combining neural networks with traditional methods such as minimax algorithms, alpha-beta pruning, and reinforcement learning can leverage the strengths of both approaches. Neural networks can provide sophisticated pattern recognition and evaluation, while search techniques and reinforcement learning can enhance strategic depth and tactical accuracy. Reinforcement learning, in particular, can help the engine learn from self-play and adapt its strategies over time, mitigating some of the limitations of static training datasets.

A promising future direction for this project involves training the engine using data from advanced engines like Stockfish and AlphaZero. These engines, known for their high accuracy and innovative play styles, can provide a richer and more accurate dataset for training. Integrating an opening database and endgame tablebase into the engine's architecture would further strengthen its performance by providing comprehensive opening theory and precise endgame knowledge.

Additional enhancements could include the integration of advanced search algorithms, dynamic evaluation adjustments, and hybrid learning techniques to continually refine the engine's performance. By incorporating these elements, the engine can achieve a balance between the intuitive strengths of neural networks and the rigorous analytical capabilities of traditional chess algorithms.

In summary, while the project highlighted specific limitations of neural network-based chess engines, it also demonstrated the potential for creating more effective engines through a hybrid approach. The integration of neural networks with traditional search techniques and advanced training methodologies offers a promising path toward developing chess engines that excel in both pattern recognition and strategic depth.

7 References

Chess Analysis Board and PGN editor. Chess.com. (n.d.). https://www.chess.com/analysis?tab=analysis

Chessboardjs.com Documentation. (n.d.). https://chessboardjs.com/examples

Download the lichess elite database. Lichess Elite Database. (n.d.). https://database.nikonoel.fr/

Encyclopædia Britannica, inc. (2024, July 22). Chess. Encyclopædia Britannica. https://www.britannica.com/topic/chess

Silver, David et al. (2017). Mastering Chess and Shogi by Self-Play with a

General Reinforcement Learning Algorithm.

FIDE laws of chess. (n.d.). https://www.fide.com/FIDE/handbook/LawsOfChess.pdf

Klein, D. (2022). Neural Networks for Chess.

Making a simple chess engine. lichess.org. (n.d.). https://lichess.org/@/JoaoTx/blog/making-a-simple-chess-engine/fGBIAfGB

Networks. Game Theory: How Stockfish Mastered Chess: Networks Course blog for INFO 2040/CS 2850/Econ 2040/SOC 2090. (n.d.). https://blogs.cornell.edu/info2040/2022/09/30/game-theory-how-stockfish-mastered-chess/

Oshri, B., & Khandwala, N. (n.d.). Predicting Moves in Chess Using Convolutional Neural Networks.

What is a chess engine? types and winning practices unveiled. (n.d.). https://chessify.me/blog/what-is-chess-engine