CHESS MOVE PREDICTION MODEL

Andy Phan
School of Engineering
University of Virginia
Charlottesville, VA 22903
tmq6ed@virginia.edu

David Wang
College of Arts & Sciences
University of Virginia
Charlottesville, VA 22903
ztg5ve@virginia.edu

Samay Jamakayala
School of Engineering
University of Virginia
Charlottesville, VA 22903
vvv9ns@virginia.edu

December 13, 2024

ABSTRACT

Machine learning offers powerful tools for analyzing patterns and making predictions in complex scenarios. In this project, we aim to enhance chess analysis by developing a machine learning model that predicts the probability of a white victory given a specific chessboard position. Using the Kaggle Lichess dataset, our model analyzes board states to provide an evaluation of the current game. Unlike traditional chess engines like Stockfish, which rely on computationally expensive search algorithms for in-depth analysis, our model delivers rapid evaluations directly from board configurations. This streamlined approach eliminates the need for extensive computations and complex installations, offering a lightweight alternative for quick and accessible insights. By focusing on board evaluation rather than move recommendations, we leverage the dataset's diversity to identify recurring patterns and provide practical evaluations for players at all skill levels. This project demonstrates the potential of machine learning to complement traditional chess engines with fast, user-friendly tools for gameplay analysis.

1 Introduction

Our project is based on the Kaggle competition dataset, Chess Game Dataset - Lichess, which contains over 20,000 recorded chess games from the Lichess platform. This dataset includes detailed move sequences, player ratings, game results, and more. Initially, our goal was to develop a machine learning model that could predict the next optimal move in a given position. We trained models using sequences of prior moves as input and the corresponding historical next moves as output, while also incorporating player ratings and game outcomes to enhance predictions. However, we encountered significant challenges: even with a large dataset, the sheer variety of possible chess positions made it difficult for our models to produce reliable or meaningful move predictions.

Recognizing these limitations, we shifted our focus to a task better suited to the data's strengths: evaluating individual board positions. Specifically, our model now aims to calculate the probability of a white victory given a single chessboard configuration. This change leverages the dataset's diversity to train the model on similar board positions, enabling it to learn patterns and trends effectively.

By providing immediate probability estimates without the need for extensive setup or prolonged analysis, we aim to offer a practical and accessible tool for players, including members of the UVA chess club. This shift in focus highlights the adaptability of machine learning in addressing challenges and optimizing solutions for real-world applications.

2 Motivation

The motivation behind our project stems from the need for a chess analysis tool that provides rapid and accessible insights into game positions. Traditional engines like Stockfish excel at calculating optimal moves through deep analysis but often lack accessibility due to their computational complexity and reliance on brute-force algorithms. Our project aims to bridge this gap by developing a machine learning model that evaluates a single board position and predicts the probability of a white victory.

This approach not only simplifies the analysis process but also provides immediate, data-driven evaluations that players can use to better understand game dynamics. By breaking games into individual positions and focusing on board state evaluation, our model offers relatable insights that can help players identify patterns and refine their strategic decision-making. This tool is particularly valuable for UVA chess players seeking a lightweight, user-friendly alternative to traditional engines for training and analysis. Through this project, we hope to empower players to make intuitive improvements and gain deeper insights into their gameplay.

3 Hypothesis

We hypothesize that a machine learning model, specifically a Convolutional Neural Network (CNN), can reliably predict the probability of a white victory based on a single chessboard position. By leveraging the spatial structure of the chessboard and additional strategic features such as piece differential, mobility, and king safety, the model will achieve predictive performance comparable to traditional chess engines like Stockfish for quick evaluations. However, we also recognize the limitations of this machine learning approach compared to advanced chess engines — some nuances in complex positions are hard to capture, so we don't expect our model to perform at or better than the level of the more advanced models, but just to be relatively close for a "good enough" estimate on most conditions.

4 Task description and data construction

We are provided with the aforementioned dataset, linked here:
`https://www.kaggle.com/datasets/datasnaek/chess`

In particular, we are most concerned with the columns:

1. winner (black, white, draw)
2. victory_status (mate, resign, timeout, draw)
3. white_rating (ELO)
4. black_rating (ELO)
5. moves (list of moves, seperated by whitespace, in standard notation)

Task modeling. We approach this task as a classification problem. For every sequence of moves leading up to some position, we need to predict the move that is most likely the most beneficial to a player out of the legal moves.

Construct train and test data. We are only given a sequence of moves, so we must transform the data to fit our prediction model. We will transform the sequences into unique positions (pieces on a grid), and determining certain features relating to the game goal in each position. These features include but are not limited to: game phase, material advantage, king safety. Each position will also have a list of legal moves, and our model is aimed at selecting the best legal move based on previous data, accounting for skill level and game outcome.

5 Related Work

Chess models have been widely explored in artificial intelligence. Notable examples include DeepMind's AlphaZero [1], which uses reinforcement learning to master chess through self-play, and Stockfish. Stockfish is one of the most advanced chess engines that can make very accurate predictions based on the most advanced models and algorithms [4]. The problem with it is its speed — to provide a good evaluation, Stockfish needs some time to run its search algorithms in the background (the more time spent, the "deeper" the model analyzes, and the more accurate it is). Furthermore, using the Stockfish engine locally requires the installation of many packages and setup. We want to train a machine learning-based model that, given a chess board, will directly output the predicted states without needing the added time of analyzing and can be installed as one or two files onto a machine. There have been many studies comparing Alphazero to Stockfish, and recent advancements have explored new ways to evaluate chess positions based on game data. Utilizing the Bellman equation in conjunction with the framework allowed by deep neural network is a particular modern practice that is being used to construct new models for maximizing probability based on potential game routes [2].

6 Training Preparation

We began by thoroughly analyzing and validating the chess data from the Lichess dataset [3]. This included examining the dataset's structure, identifying missing values, and gathering basic statistics to ensure the data's reliability. Once validated, we moved into preprocessing, where we split the game data into individual moves and converted each board position in a structured representation using Forsyth-Edwards Notation (FEN). These positions were further filtered to retain essential components, such as the player's ratings and game phases, for our analysis.

After evaluating various machine learning approaches, we determined that a Convolutional Neural Network (CNN) would be the most suitable model to its effectiveness in capturing spatial relationships, which are crucial in chess. The CNN's architecture allows it to interpret the board as a grid, where each cell represents a square on the chessboard, enabling the model to learn patterns based on piece positions and their interactions.

Producing the Game States For the sequence of moves for every game, a simple method was written to take a "snapshot" of the game at every single move and stores the board position in a standard FEN notation. Along with the FEN state, other hyperparameters were calculated, which will be mentioned below, and stored in a new CSV file. Out of the original 20,000+ games in the dataset, we produced over a million individual board states which were eventually used as processed data to train the model.

Constructing the Model We decided to use a convolutional neural network (CNN) to process the game board. Since it's similar to an image with distinctive features (pieces on the board), we decided to encode the pieces into different values on a game board that can be "seen" by a CNN model to evaluate its patterns.

In addition, we also calculated some hyperparameters purely from a given game state. These parameters include piece differential, king safety, and other parameters that can help evaluate the game. The established goal was that these parameters can help the model make predictions by presenting some values that the model couldn't otherwise see from purely the game board. These values are calculated during the "snapshotting" phase shown earlier and stored in the CSV file of the one million game states.

By running the model repeatedly and constructing test to examine the effectiveness of each parameter, we decided to mainly use the following hyperparameters:

1. Piece Differential: White piece values minus black piece values, with PAWN = 1, KNIGHT = 3, BISHOP = 3, ROOK = 5, QUEEN = 9, KING = 0.
2. Mobility: The difference between legal moves of white and that of black.
3. King safety: a simple parameter. According to the square in which the king resides, points will be added to the side that has more pawns surrounding the area.
4. Control of key squares: Differential of the squares controlled by white minus those controlled by black in the given squares: E4, E5, D4, D5.
5. Doubled pawns: difference in pairs of doubled pawns between white and black.
6. Isolated pawns: difference in pairs of isolated pawns between white and black.

Producing Ground Truth The next problem was to come up with ground truth data. For every game, we know who wins in the end, but our dataset did not contain data for any sort of evaluation of a single move. In order to produce a "y" data to train our model, we needed to associate each board position with an existing evaluation to train our model against.

To produce a simple evaluation, we first used a very simple equation that takes advantage of the fact that we know the eventual winner of the game:

$$Y = \frac{0.5 \cdot W}{n} + 0.5$$

In this equation, Y is the decimal prediction of white winning, with values from 0 to 1. The value of n represents the total moves in the game, so it is a whole number. W is an indicator variable where:

- if white wins in the end, W = 1.
- if black wins in the end, W = -1.
- if the game draws in the end, W = 0.

This is essentially a simple calculation where the initial prediction always starts at 0.5, and each move will increase/decrease the score linearly until it reaches 1 or 0, depending on who wins. It will also stay the same throughout all the moves if the game ends in a draw.

This equation's main advantage is its simplicity. It assumes that the matchups of players are on a similar level, and each move will slowly progress the eventual winner to the win. Given this assumption, the simplicity of the calculation enables quick processing of the millions of board states and can be used as a somewhat reliable way or evaluating a given board position.

Better Ground Truth Using Stockfish Evaluation The above algorithm, however, has its flaws. As mentioned, it's mostly based on the assumption that players do not make big mistakes, and each move is somewhat incremental and has the same influence on the result of the game. While this might work for higher rated players, it tends to fall apart as less experienced players make more mistakes that can cause the evaluation to fluctuate throughout the course of a game, instead of following a simple linear pattern. The equation might result in labeling a winning endgame position a very low value because the player eventually lost three moves later due to consecutive blunders that had nothing to do with the previous position. Since we don't know how much this will skew our data, we first tried running a model with this as the ground truth and tested it out. The model training seemed to work well, but its actual performance varied — sometimes it would output good predictions that are inline with other evaluations, but sometimes it will give out some bad predictions.

We didn't want to settle with the inconsistent predictions, so we turned to a new method to generate the ground truth data. As mentioned previously, Stockfish's analysis engine produces very good results. As it is open source, we decided to use its evaluation as the basis for ground truth data. To do this, we once again looped through every saved state, then inputted their FEN states into the Stockfish engine that was installed on our machine. Since there were more than a million game states, even with the sub-second time limit that we set on the evaluation function, this was a long process which took more than a couple dozen hours total. Since Stockfish evaluates it own centipawn score, we converted its evaluation into a white in percentage based on a sigmoid function, where x is the Stockfish centipawn score:

$$P(\text{white win}) = \frac{1}{1 + e^{-0.004x}}$$

Constant k = 0.004 was chosen as it is one of the more accepted ways for converting centipawn score to a percentage. This function is symmetrical about the y-axis, returning 0.5 for a evaluation score of 0, around 0.69 for an evaluation score of 200, and around 0.92 for an evaluation score of 600. Once finished all the processed game states have an associated score ground truth score that can be used for the model training. In addition, in our calculation, if Stockfish calculates a "mate in n moves", the model will assign a 1.0 or 0.0 score accordingly.

7 Training and Analysis

We have our game state data, hyper-parameters, and ground truth data all ready for training. We used a primary CNN architecture with the parameters mentioned above as additional features. The tensors we used consisted of:

- Each chess board: 8x8x13 tensor. 8x8 represents the board squares and 13 represents the 6 possible pieces from each side, plus one for an unoccupied square.
- Features: 6-D vector, representing the six additional features mentioned above.
- Ground truth: separate vector for Stockfish generated ground truth data, which obviously only used in the training set.

The board input undergoes two convolutional blocks with 64 and 128 filters, respectively, using ReLU activations, batch normalization, and pooling to extract spatial features. A global average pooling layer reduces spatial dimensions, followed by dense layers for further processing. The extra features input is processed through fully connected layers with ReLU activations and batch normalization. Both branches are merged via concatenation, followed by additional dense layers for joint feature learning, with dropout layers for regularization. The final output is a single sigmoid-activated node for predicting a value in the range [0,1], optimized using the Adam optimizer and mean squared error loss, with mean absolute error tracked as a metric. In our case, the mean absolute error directly reflects the difference in prediction of white win between our own model and the ground truth Stockfish evaluation.

We set the model to train for 100 epochs, an initial learning rate of 0.001, and mean squared error for evaluation. We also kept track of the mean absolute error for a more intuitive metric, and a *ReduceLROnPlateau* module triggering a halving of learning rate if 5 consecutive epochs pass without a decrease in lowest validation error. The final model training history is shown below. Since we didn't see any significant changes to either metric after around epoch 60, we stopped the model at epoch 80.

Both the training MSE (blue line) and validation MSE (orange line) decrease significantly during the initial epochs, indicating that the model learns effectively in the early phase, which can probably be attributed to the abundance of data (>1,000,000). After about 25 epochs, the training MSE continues to decrease smoothly, while the validation MSE starts to stabilize with some fluctuations. After around 60 epochs, the training MSE also seems to converge.

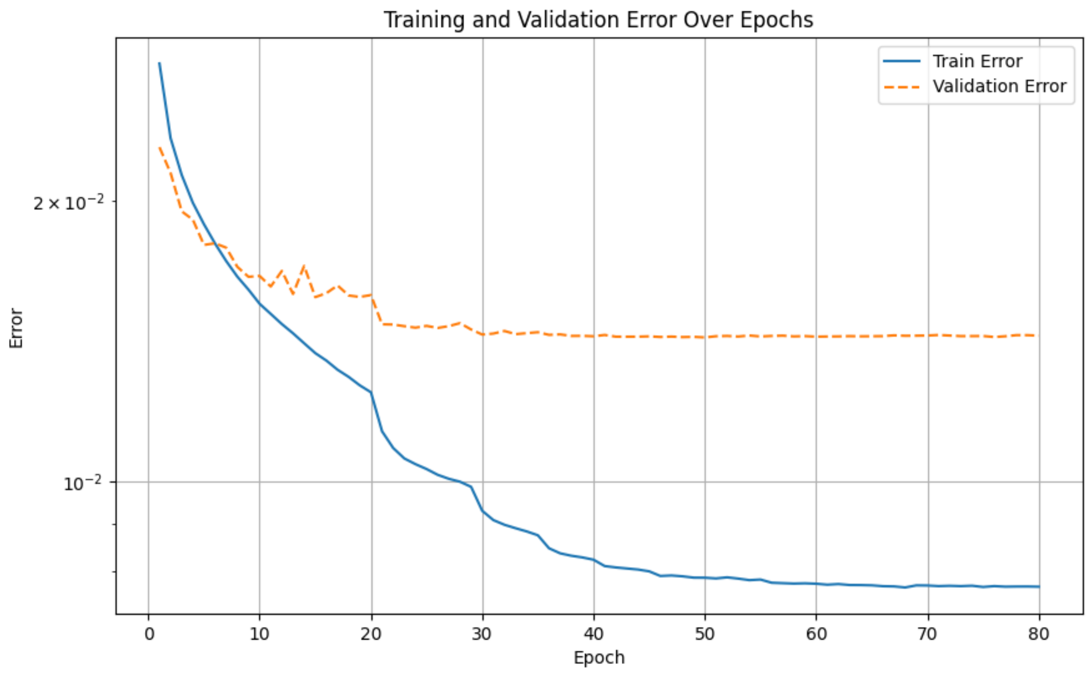
The best model had a training MSE of 0.0079 and a validation MSE of 0.0142 at epoch 50.

Validation loss converged sooner at a slightly higher value, suggesting some overfitting. Despite this, the overall validation performance remains stable.

The training MSE converges to a very low value, indicating the model fits the training data well.

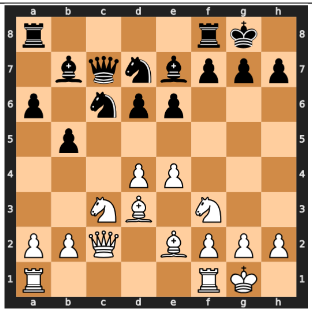
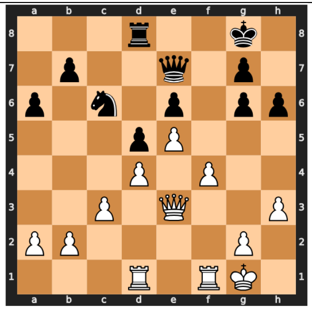
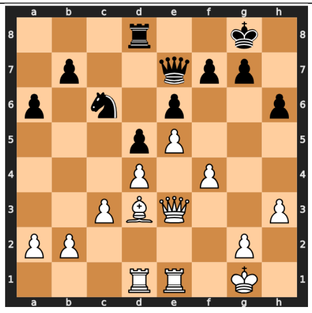
The validation MSE stabilizes at a slightly higher value compared to the training MSE, which is typical in machine learning due to differences in the distributions of the training and validation datasets.

During training, our model saved every 10 epochs as a .h5 file, and continuously updated a best model upon reaching a global lowest validation error. All models can be seen in our GitHub Repository.



8 Model Application

We wrote code to use our trained model. Using a Python script, we loaded our best model to manually test its performance. Below are some examples of our running model. Note that we compared out model (white win probability) to Stockfish's centipawn evaluation, where a higher number indicates more advantage for white. All the following positions are white to move.

Game State	Our Model	Stockfish Evaluation
	0.4869	-0.37
	0.7433	+2.93
	0.9004	+5.55

As one can see, our model is pretty accurate in predicting a win rate — when Stockfish evaluates a near nuetral position (close to 0), our model predicts a close-to 50% win rate, and as Stockfish predicts a higher advantage for white (higher number), our model also outputs an increasing win probability. When testing other board positions, our model seems to align pretty well with Stockfish and other evaluations, including our own intuition. It never produced blatantly bad results and is always in the ballpark.

In conclusion, our model offers an effective and efficient way to estimate chess outcomes by leveraging pre-trained evaluations instead of relying on extensive deep searching. While it may not always produce results as precise as top-tier engines like Stockfish, it comes remarkably close in most scenarios and provides a consistent, intuitive understanding of the board state. This advantage in speed and accessibility makes it a valuable resource for players who want a quick, rough insight into their game progress, guiding them as they learn and improve without the need for time-consuming analysis. We hope that players including member of the UVA chess game can use our tool to analyze multiple different positions quickly and help them improve their own game.

We believe our model was a success, and are looking forward to more advancements in chess and other fields alike with the help of machine learning.

9 Contributions

Andy Phan did the initial analysis and validation of the chess dataset, ensuring data quality by examining missing values, data types, and basic statistics. Andy also handled the preprocessing, which involved segmenting the game data into individual moves, converting board positions into Forsyth-Edwards Notation (FEN) for compatability with the model, and filtering necessary components such as player ratings and game phases. These steps prepared the data for effective training and laid the foundation for subsequent modeling.

Samay Jamakayala worked on implementing the CNN model using TensorFlow and Keras, trying various model architectures to optimize prediction accuracy. The model was structured to accurately predict moves in a reasonable manner utilizing a standard 8x8 board representation with two primary convolutional layers and ReLU activation. Compilation, as mentioned previously, was done with the Adam optimizer and sparse cross entropy loss function. Then the training data was fitted accordingly, utilizing the validation set for additional metric comparisons.

David Wang worked on analyzing CNN model structure to improve performance. Although the current model didn't converge to a acceptable error level, he experimented with different CNN model architecture, including different dropout layers. In addition, he tried different hyper-parameter definitions using the chess data to work on figuring out which combinations of data will likely yield better results for the model. He also started working on implementing heuristic functions (such as piece advantage, move freedom, and king safety) by combining move history and current board state so the model has more data to work off of.

10 GitHub

Our work, including the entire training notebook, application file, log files, and all saved models, can be found on GitHub: `https://github.com/dooodle74/uva-ml4va`

References

[1] AlphaZero: Shedding new light on chess, shogi, and Go. `https://deepmind.google/discover/blog/alphazero-shedding-new-light-on-chess-shogi-and-go/`

[2] Chess AI: Competing Paradigms for Machine Intelligence `https://doi.org/10.3390/e24040550`

[3] Chess Game Dataset (Lichess) `https://www.kaggle.com/datasets/datasnaek/chess`

[4] Stockfish `https://en.wikipedia.org/wiki/Stockfish_(chess)/`