Machine Learning in Chess

Making a neural-network based chess evaluator

Nasif Hossain

6/10/2022

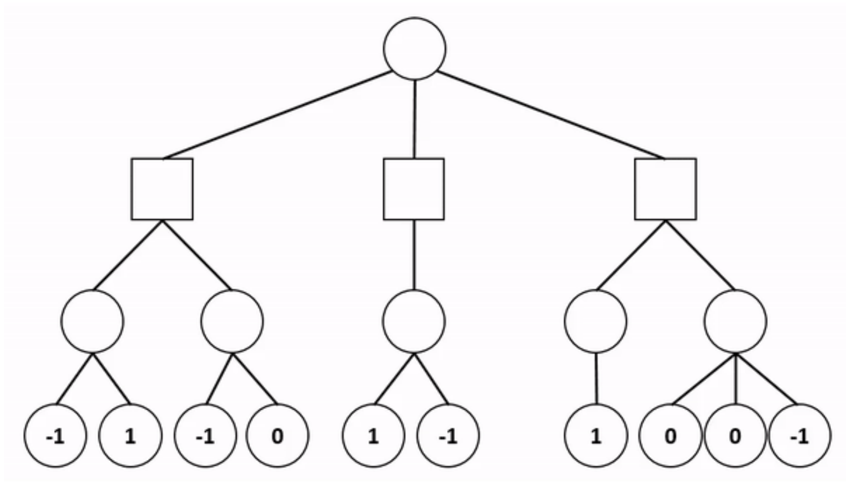
Introduction/Motivation:

Chess has been a very popular game since written history where the premises of the game made players have to think ahead. Recently, in this world of computation, there have been a number of Chess engines developed specifically to play against Human players. An average chess player usually thinks ahead 2-3 moves to decide whether that move is good for them in the game. As players play more games and learn from them, they develop an intuition/gut-feeling to what a good move is which acts as a 'heuristic' in the world of AI-Programming. In this project I wanted to code a chess engine which deals with the problem of having a human think through all the moves and computes the best move for the game. Since Chess is a game that has been around for a while and to scale that, in America the first national championship, the First American Chess Congress, was held in New York City in 1857.(1) It has resulted in creation of multiple chess engines after 1951, when the University of Manchester developed a chess AI program for the Ferranti Mark 1 computer. For a benchmark test, the AI played a game of a 6x6 boarded chess against a young woman who had trained for a week where the AI had won the game.(2) As of now the chess engine with the highest chess ELO rating is 'Stockfish 13' with around 3500 ELO. So, I will be making a chess engine through the overlapping of my chess knowledge and some fundamentals of chess programming that have been used throughout the years.

Creation:

To set up the board in python, I used the python-chess library which has all the functions needed to run a chess game (moving a piece, getting a list of possible moves for the current player and deciding whether the game is over). Making a chess engine requires an evaluation function such that when the position data is passed in the function it returns an evaluation number which is used to decide who is winning and by how much. A chess piece is said to be good in most situations if they are in a certain position on the board, e.g protected Bishops covering many squares, Knights in the center covering many squares, these are said to be intuitions that a good chess player should have. This knowledge comes from experience as players develop ‘gut’ moves which are good most of the time. I wanted my Neural Network to learn positional advantage through the whole learning process. But, having to compute evaluation for different positions would result in a very big tree, hence I decided to implement some efficient search algorithms for this type of problem and optimize it with another algorithm. My main search algorithm is a variant of the Mini-Max search algorithm called Negamax. The MiniMax algorithm is very popular in the world of AI for two player games like ‘GO’ or ‘Chess’. The algorithm assumes that the opposition will play the best move on their turn. Hence, the search is conducted such that when it is the AI’s turn to move, it chooses the maximum evaluation value in that specific depth, then for the immediate opposition’s turn it chooses the minimum evaluation value (assuming that the opposition will play the best move resulting in the lowest evaluation value for the chess board). Note that the higher evaluation value means that the Chess AI is winning and the lower evaluation value means that the opposition is winning. In Chess, the evaluation value of player A is the negated evaluation value of player B, *(max(a, b) =*

-min(-a, -b)). Negamax was implemented here because there is no need to have two subroutines to calculate the evaluation score for each of the players, instead it just finds the evaluation value of one player and negates it to find the evaluation value of the other player. This resulted in lower search time.



Animated representation of MinMax search, 1 = winning state, 0 = draw state, -1 = losing state

To further improve the search time I also implemented **alpha beta** pruning in the search algorithm. It is a search algorithm that decreases the number of nodes in its search tree that is evaluated by the MinMax algorithm. Alpha-Beta pruning seeks to stop evaluating through a node to its successor (chess position in this case) when it finds a better move. It reduces search time by not evaluating further when there is a better path to go down to. In addition, I also added Quiescence search which evaluates through positions where there are moves to be made that drastically increases the evaluation value. This is done because of the danger of missing a better move just because of the depth restriction.

Machine Learning

Introduction:

Due to the problems mentioned above and the huge amount of time taken to evaluate hundreds of thousands of chess positions through computation, I decided to implement deep learning for the evaluation of chess positions. In the paper about deep learning[5], the research group used 4 different deep learning algorithms with 2 types of input for 4 different datasets. 1 study was particularly interesting to me which involved using 3 multi layered perceptrons for the Neural Net and training it with regression and back-propagation. This paper motivated me to implement a neural network for the evaluation function.

The evaluation data was taken from Kaggle[3]The data I found has ‘Chess Board Positions’ in FEN format and evaluations for the position done by Stockfish AI with 22 depth search. Regression learning seemed appropriate for this dataset so I decided to use Keras to perform it. The input for my learning algorithm would be the position of the chess pieces, so in order to do that I had to transform the FEN formatted data to numerical arrays for our Machine Learning input (the x values). The data was transformed into a 8x8x12 2d matrix array for the board-position representations for the input of the neural network. The 8x8 array would represent the 64 chess squares and the 12 would represent 6 different types of pieces for both black and white. The input of the neural network would result in 768 inputs for the neural network. In order to extract the features from the board-position data I had to prepare the data for the input of the Neural Network in such a way so that none of the features of the data is lost, which made me choose this input architecture.[Code in Jupyter Notebook]. The y values (evaluation values) were in range between -300 to +300 hence I normalized all the values to be between 0 and +1 so that the predictive model can better approximate the y (evaluation) numbers given the chess board data.

Looking over the structure of the dataset, I used my judgment call to decide to use linear regression to approximate the output evaluation value. My general architecture consisted of 3 hidden layers of MLP with the input layer consisting of flattened input data (flattening the 8x8x12 array to just 1 list of 768 numbers). The output layer for training the Neural Network has 1 unit since we are trying to approximate the output of Stockfish's evaluation score. The batch size was set to **512** to greatly reduce the amount of time it takes for the model to learn and this did not seem to impact the learning by a lot.

Results:

On the first epoch, the model's mean loss between the actual and predicted value was 0.0194 and a mean absolute error of 0.0918. After 200 epochs the loss was 0.0032 and the mean absolute error was 0.0397. This had suggested that the NN model definitely learned to predict an evaluation value that was very close to the actual value. The model was then saved and exported to a testing program for data collection and evaluation of the model.

Testing:

The NN model was evaluated through a test data set 'rand_evals.csv' which consisted of 10,000 random evaluations computed by Stockfish. For the test data, the board representation data was massaged (in the same way in learning) for the proper input of the neural network and the evaluation values were normalized for comparison between real and predicted values.

The model was then loaded and used to predict 10,000 evaluation values from the test dataset. However, the NN only predicted 9049 samples. The mean loss between the test evaluation values and the predicted evaluation values was 0.0030 and the mean absolute error was 0.0345. For each of the samples, the difference between the actual and the predicted value was calculated for further analysis. For data representation I rounded down the value to 9000 for the graphs.

The highest difference between the predicted values and the actual values was **0.52**. But 8694 samples out of 9000 (**96.6%**) had a difference of less than or equal to **0.1**.

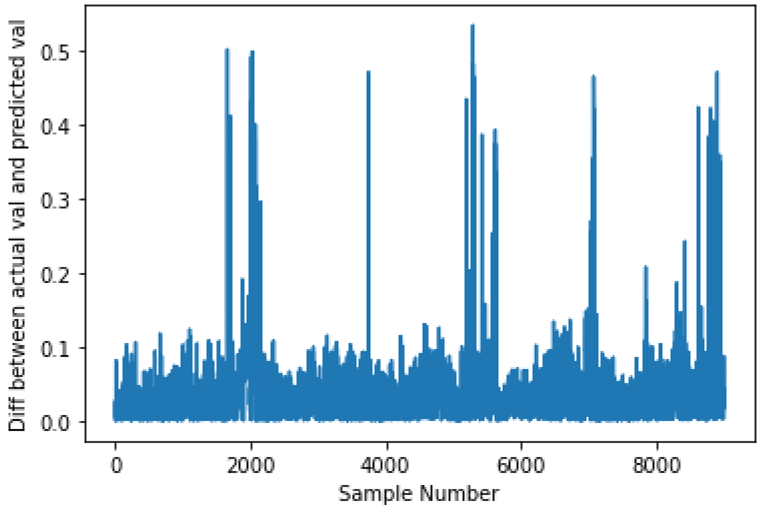


Fig: Difference between actual evaluation values and predicted evaluation values for each of the 9000 sample

Looking closely at the 8694 samples which resulted in <= **0.1** difference values, we can see that the majority of the difference values are very low in Fig (1). The graph tells us that there are more samples with lower difference values than there are for higher difference values. For further analysis, the cumulative function of the histogram in Fig (2) was computed, and from there we can see a sharp decline (pre-flattened) around 0.05 difference values and the curve starts to flatten out at **0.1**. For these reasons, we can assume most of the samples with higher difference values are from chess positions that consist of extreme cases such as “pinning a piece”, “multiple threats on a piece” which the NN doesn’t really account for. Hence, we can ignore that since our learning was based on chess piece positions only. This data suggests that for any chess position the Neural Network model can output an evaluation value within 0.1 difference from the correct value with 95% accuracy.

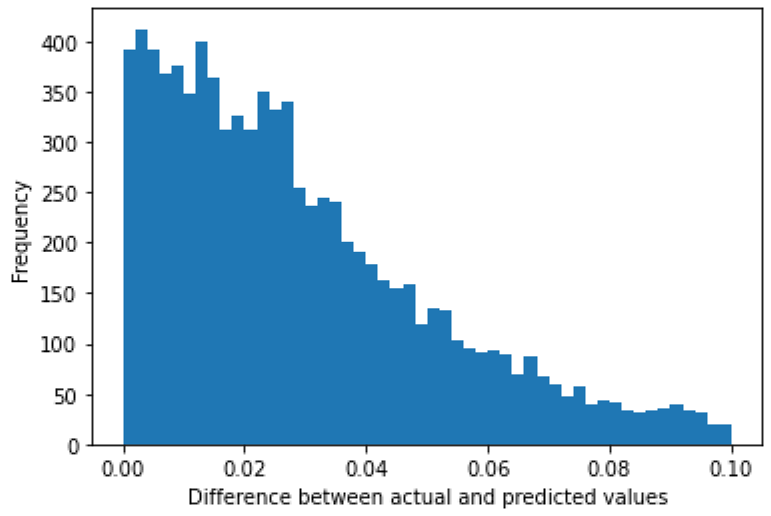


Fig 1: Histogram of frequency of difference values from 0 to 0.1

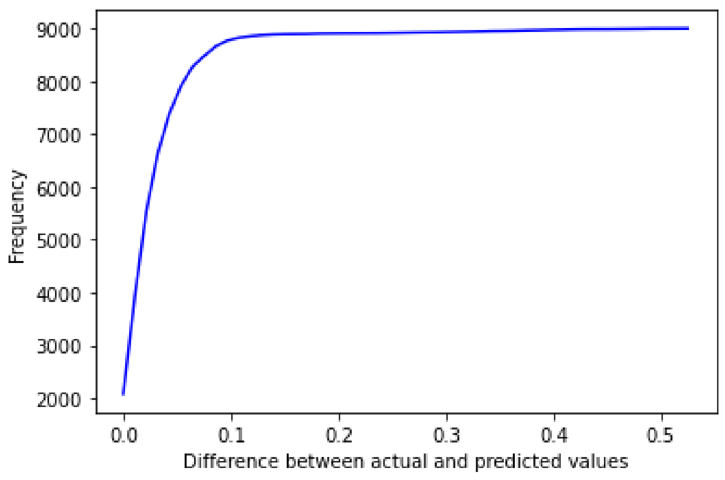


Fig 2: Cumulative function of the frequency histogram

Game Testing:

The NN was loaded into a chess game where it had to play against stockfish. I used the NN to evaluate positions and the search algorithms mentioned above to create a search tree along all the legal chess moves. Analyzing the moves that were made. The NN model played very good positional moves in the first 4 moves and started to play attacking moves after the 4th move when Stockfish started attacking with pawns.

White = NN, Black = Stockfish

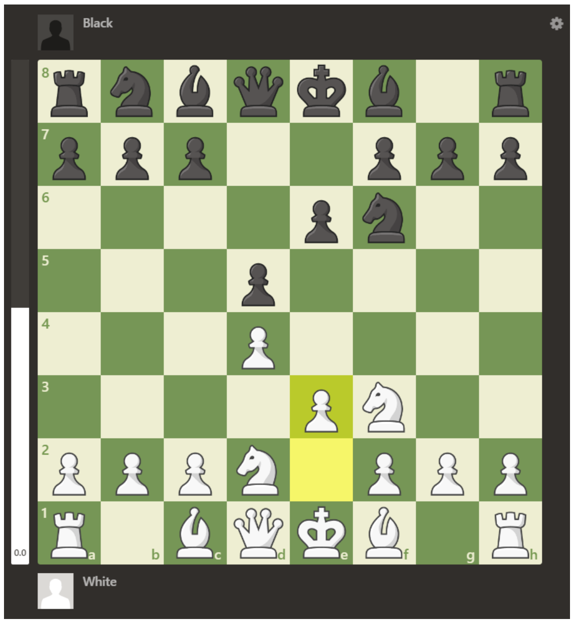


Fig: After 3rd move, good positional moves by NN

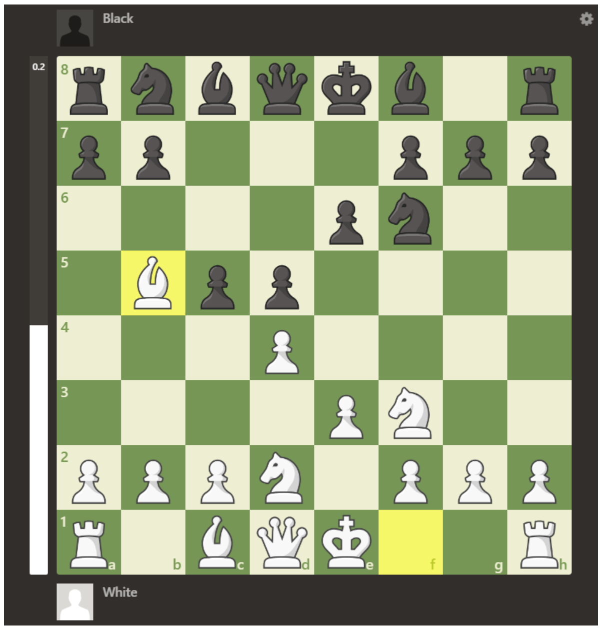


Fig : After 4th move, NN attacks when being attacked

By the 8th move, the model started to lose against stockfish by playing losing moves.

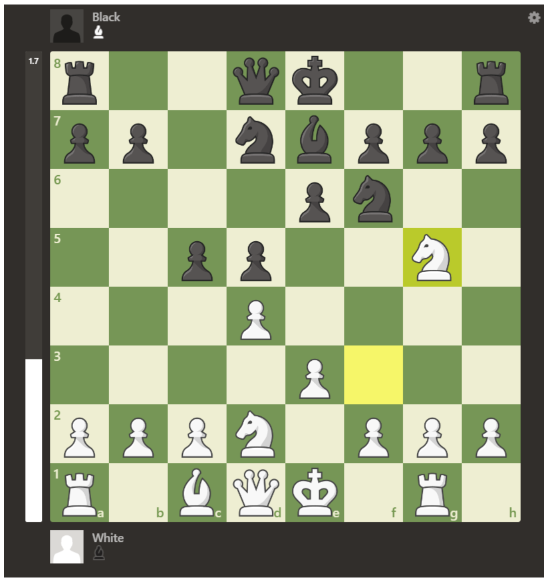


Fig: Blunder at 8th move

The NN lost after 20 moves by playing very unconventional moves at the endgame of the chess game and got checkmated.

Moves made by engine - Nf3 d4 e3 Bb5+ Bxd7+ Nbd2 Rg1 Ng5 Qf3 Nxe6 Rh1 c4 Qxd5 e4 Bxd2 Bxb4 Ke2 Kd3 Raf1 -- --

Conclusion:

The results from the testing suggests that the NN model definitely learned through approximation as it was able to reduce the error down to very low. The low median value of 0.0249 and the low mean value of 0.0345 tells us that the NN model can approximate the output value of the evaluation function of Stockfish AI very well, of course with some outliers for the extreme cases. However, this model was only trained with the chess pieces positions, and analyzing the game of the NN vs Stockfish we can understand that it lacks understanding of “pieces under attack”, “pinned pieces”. Improvements can be made further on this NN model by training it under specific cases. Additional input of what piece is attacking what piece (chance to take a piece) will further improve the accuracy of the NN model when played in a real chess game. Specific chess positions can be used with increased learning rate to make the NN learn about some core ideas of chess strategy. Completing chess puzzles is a way for human players to learn some game strategies, it consists of a series of correct moves that a player must make to gain advantage in the game. The Neural Network can be expanded such that it is trained with the puzzles and has to predict the correct move.

Work Cited

*. Chess Programming Wiki heavily influenced my ideas and code for the chess engine.

1. “Chess History.” *Britannica*,

3. “Chess Evaluations”, *Kaggle.*

5. Playing Chess with limited look ahead