Stock Reduction: A Shallow Implementation of a Classical Chess Engine

Connor Lennox

Spring 2021

1 Introduction

Standard chess engines operate by performing a state-space search on a game tree of all possible moves from a given position. Although this search uses several techniques such as alpha-beta pruning and move ordering to limit the size of the search, chess has a significant branching factor (on average 35) which in some circumstances limits the effectiveness of these techniques. To compound on this, many moves in chess have no direct impact on the state of the game and their utility will be realized several moves into the future. Due to this, tree searches that are insufficiently deep may fail to see potential moves that have no immediate benefit but will lead to openings later.

Stockfish is a chess engine that does the aforementioned search and can predict what the best move for a player in any given position. For the purpose of this project, Stockfish will be allowed to search to a tree depth of 22. As such, the Stockfish evaluation of a position is equivalent to the mini-maxed evaluation 22 levels deep.

The goal of this work is to condense the state space search into a single layer, by learning a function that can approximate the Stockfish evaluation at depth 22 by looking only at the top-level state. In doing so, the search will be considerably shorter, as the evaluation of only each potential move from a given position will need to be made (leading to a tree height of 1 or 2 as opposed to 22).

This is a prediction problem: given a state the result of the evaluation function must be determined. In addition, it is a regression problem, as the evaluation function returns continuous values. The unit of the evaluation function is "centi-pawn advantage," which can be considered a value that determines which player is winning in any given state. Large positive values favor white, while large negative values favor black. Values near zero are a neutral game. While this problem in and of itself may not be of much significance, the concept of accelerating state-space searches through the use of machine learning may be beneficial to many other domains. In this scenario, a successful model would be able to clearly approximate the Stockfish evaluation function, within a certain degree of error.

Given the popularity of chess, it is very easy to find applicable datasets. For the purposes of this project, a dataset containing 12.9 million chess board layouts with the corresponding Stockfish evaluation at depth 22 will be used.

2 Related Work

The idea of computers playing chess is one that has been approached with several different methods. By far the more popular method is state-space search over possible moves, dating back to 1950 [2]. The high branching factor of chess makes this a difficult problem, however improvements in game-tree search such as alpha-beta pruning and move ordering help alleviate this somewhat. In addition, work has been done to observe the impact on search depth on performance [1]. One of the most widely known and best performing chess engines employing this strategy is Stockfish [4].

The other ideology within the field of automated chess lies in machine learning. This method is more recent than the traditional approach, dating back to 1995 [5], and employs a learning procedure to determine what next move to select. Since then, has been expanded upon significantly, with AlphaZero marking a culmination of much of this work [3]. AlphaZero used a reinforcement learning technique to learn how to play chess with no external stimulus: an approach that was also applied to several other similar turn-based games with very little need for adaptation. Stockfish has also recently begun to use a neural component, although it is secondary to the state-space search that the engine has used in the past.

3 Methodology

Above all else, a successful model will be one that is able to effectively predict the quality of a chess position based on the potential centipawn advantage that can be gained in the future. Given that this is an incredibly complex problem, it is unlikely that the exact advantages will be able to be predicted, however in the situation where multiple states are being compared the model should be able to properly order them. This problem of ordering several different positions is critical for a chess engine because it drives the decision making process that leads the engine to select one move over another.

The proposed technique for properly calculating centipawn advantages is a deep learning method. Given that states are seen without context (predictions must be made off of a single board state, not a full history of moves), a complex model will not be necessary but many degrees of freedom will be beneficial. The model design consists of several stacked linear matrix transformations, all of which are learned. Features are designed to give a sufficient representation of the board state: first features are extracted representing the current pieces on the board via flattened one-hot vectors. There are 832 features that are made in this way: for each of the 64 squares on the board, a 13-length one-hot vector is created. The reason for a length 13 vector as opposed to length 12 (there are 6 piece types and 2 colors, and as such 12 distinct pieces on the board) is that the empty space is critical for chess. Since so many pieces can only move to squares they have line-of-sight to, explicitly indicating empty squares is preferred to the alternative of implicitly indicating them by having no elements of the one-hot vector set to 1. In addition to these board layout features, there are additional features that represent whose turn it currently is, player castling rights, and the actual counts of each piece type on the board (this final feature can be inferred from the board layout, but was decided to be important enough to justify providing the value directly to the model). In total, there are 851 features used in the model, but the majority of these are 0 in any given example due to the one-hot nature of the board layout representation.

The data used for this project consists of approximately 13 million chess positions and the associated 22-depth Stockfish evaluation. This data is split into a training and testing set at a 80/20 ratio. Training data is used to train the deep learning model via gradient descent, and then performance is recorded on the testing data to check for model performance on unseen positions. Due to the vast amount of data, it is unlikely that overfitting occurs, and it is likely that the model will generalize well to new data.

Beyond the main task of predicting depth-22 evaluations of board states, the purpose of this project is to create an engine that plays the game against humans (or other engines). The process for move selection is simple: whereas a traditional chess engine might do a minimax search with complex optimizations to ensure fast performance, this machine learning approach can instead do a search on a much shallower tree. By doing a shallower search there are exponentially fewer states that need to be inspected, and accuracy does not necessarily need to be decreased since an optimal model will be able to correctly predict the expected result of a 22-depth tree.

4 Results

The model used for this problem is relatively simple - it consists of a stack of learned linear matrix transformations with LReLU activation functions. Both the number of hidden layers and the size of these layers are hyperparameters of the model, and several were tested via a validation set. The leaky variant of the ReLU activation function was selected due to the model having difficulty predicting negative centipawn advantages (indicating that black is at an advantage in the current state) when only a ReLU function was used.

Index	Model Structure	R^2	RMSE	MAE
1	2 x 2048	.565	401.9	121.01
2	3 x 2048	.526	420.1	115.3
3	2 x 4096	.620	376.0	97.5
4	2 x 4096, 1 x 2048	.633	369.7	98.7

Table 1: Results of the hyperparameter test process. Reported results are all generated via predictions over the validation set. Model structure is in the format of (number of hidden layers) x (hidden layer size).

All models that were tested were trained with 160000 samples and evaluated with a validation set of 40000 samples. All models were trained for 100 epochs. Looking at the result table, it is clear that the model with 2 hidden layers of size 4096 and a single hidden layer of size 2048 performed the best. This makes sense as this model is the most flexible of all those tested.

The best performing model was trained again over a larger training set of 800000 samples and evaluated over a separate test set of 200000 samples. The rest of the results were refer to this model in particular. The final model has an R^2 score of .569, an RMSE of 432.2, and an MAE of 108.6. These values are higher than they were for the smaller dataset, but this is to be expected: with five times the amount testing samples it is inevitable that higher errors will be introduced (even with the offset of additional training data).

The following residual plot relates the true values to the predicted values by the model. There is a clear linear trend in the residuals, which is unusual but explainable in the context of this problem. Many of the data points have very small centipawn advantages: only on the scale of approximately (-100, 100). So, when the rare value gets into a range that is well beyond this small range, it is difficult for the model to accurately predict. It should be noted, however, that the model still predicts in the right "direction" relative to zero. This is critical for the application of the model, since the true accuracy of the prediction is less important than the relative accuracy when comparing any two states.

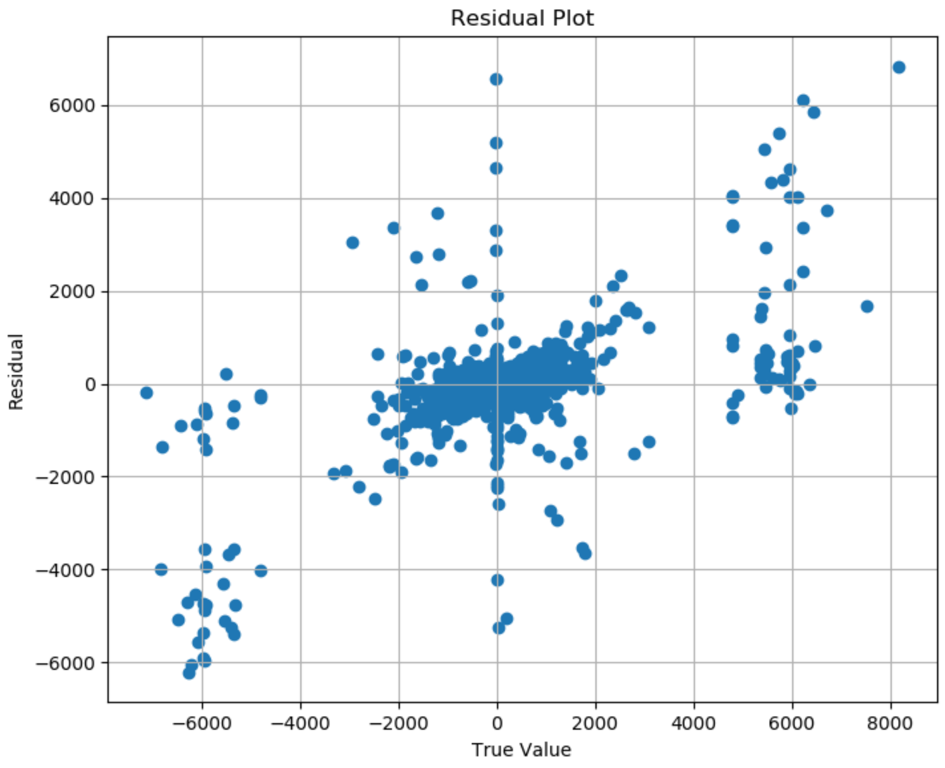

Figure 1: The residual plot for the final model. The linear trend in residuals is unusual, but explained by the models preference to predict smaller values in the correct "direction" as the true value.

In order to specify an evaluation metric that display the ability of the model to play chess, a pairwise comparison was used. For any pair of states, the model should be able to select which one has the higher evaluation function. When playing chess, the model should select the move that will put it in as best a scenario as possible - this is accomplished by either minimizing or maximizing the evaluation function given the possible set of resultant states from making a move. If the model is able to accurately rank states, or select the better from a pair, then it should be able to make good moves when playing chess.

To evaluate this, 100000 pairs of states were gathered from the test data. For each pair of states, the model was asked to predict which of the two had a higher evaluation function. The model was able to correctly identify the preferred state in 83.415% of cases. This is a good enough result to confidently say that the model is capable of selecting good moves while playing chess.

5 Conclusion

The goal of this project was to create a regression model that could accurately predict the result of a deep Stockfish analysis on a chess board. In order to achieve this, a multi-layer perceptron was used. Having tested the model, it was found that there are difficulties when the true evaluation has significantly high magnitude. However, the model was capable of accurately ranking pairs of states, which suggests that adequate performance in gameplay can be achieved.

References

[1] Diogo R Ferreira. The impact of the search depth on chess playing strength. *ICGA Journal*, 36(2):67–80, 2013.

[2] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[4] Stockfish. https://stockfishchess.org/. Accessed: 2021-3-13.

[5] Sebastian Thrun. Learning to play the game of chess. *Advances in neural information processing systems*, 7, 1995.