Poor Eyesight Chess AI Report

Alex Fish

1 Overview

Oh no, Stockfish forgot its glasses at home! The state of the art chess move-suggestion engine can only recognize the **location** and **color** of pieces on the board and must guess the piece types based on previously-observed games before it thinks of a move to make. While Stockfish with perfect vision (i.e., being told the exact board state) is far better than any human chess grandmaster, how well can it perform when it has "poor eyesight" and might incorrectly guess which pieces are on the board, influencing its decisions?

The primary goal of this project is to build a deep-learning model which uses only chess piece **location** and **color** information to predict the exact board state, trained on a history of chess games played by real people.

Even the most interesting models are mere curiosities without proper deployment, so I created a small chess program to play against this "Poor Eyesight Stockfish" locally on the command-line, instructions for which can be found on the GitHub page. Additionally, I created a Lichess "bot" account, PoorEyesightBot, where you can play against Poor Eyesight Stockfish online with no account, downloads, or setup required!

In Section 2, I will provide a brief overview of the relevant terminology. In Section 3, I will describe the model motivation and structure in more depth. In Section 4, I will discuss the training and testing data source, mining, storage, and volume along with a brief analysis of the data. In Section 5, I will go into the technical details of the model training, testing, and tuning process. In Section 6, I will discuss implementation considerations for using this model together with Stockfish. Finally, in Section 7, I will assess model performance and give conclusions.

1.1 Keywords

data parsing, feature engineering, classification, class imbalance, deep learning, parameter tuning, ensemble modeling, model deployment.

1.2 Libraries Used

pandas, python-chess, Matplotlib, PyTorch, NumPy

2 Chess Terminology

In this Section, I'll define the relevant terminology. First, the relevant chess terminology to building the model.

Term	Description
Piece	Both players in a chess game start with the following pieces: 8 pawns, 2 knights, 2 bishops, 2 rooks, 1 queen, 1 king.
Square/Piece location	A chess piece is located on a square of the chess board. There are 64 squares in an 8x8 grid, with columns or "files" labeled "a" through "h" (left to right), and rows or "ranks" labeled "1" through "8" (bottom to top). For example, the white queen starts on "d1."
Piece color	One player controls the white pieces, while the other controls the black pieces. The white pieces start on the first and second ranks, while the black pieces start on the seventh and eighth ranks.
Piece type	For example, "pawn," "knight," or "rook."
Board state	The complete set of information of piece locations, colors, and types across the whole board. When any of this information changes (such as after a piece moves), it is considered a different board state.
Square occupant	The piece that resides in a square, or "empty." For example "white rook" or "black pawn."

Next, some auxiliary terminology.

Term	Description
Stockfish	A chess AI which is the state of the art in chess move decision-making.
Poor eyesight	Limiting the information available to only the piece location and color, simulating blurry vision in real life.
Perfect vision	No limitations on board state information.
Pawn promotion	If a pawn moves 7 squares forward, reaching the end of the board, it promotes to another piece. This piece can be a knight, bishop, rook, or queen and the choice is up to the player who controls the pawn.

3 Model Motivation and Structure

3.1 Motivation

As mentioned in Section 1, the goal is to build a model which can only observe chess piece location and color information to predict the full board state (i.e., predicting which exact piece is on each square).

This project is inspired by one of the algorithms in the YouTube video 30 Weird Chess Algorithms by Thomas Murphy VII, also known as "tom7". However, his model only used piece location information without considering piece color, which struck me as unmotivated and arbitrary. Tom's model, as mentioned in the video, has deep faults in that it tends to think any piece deep into a given side of the board belongs to that side's starting color, allowing his queen to infiltrate the AI's side of the board without any resistance.

3.2 Structure

The modeling is actually done through an ensemble of 64 smaller classifiers, one dedicated to each square on the board. Each of these 64 models uses the entire board state to predict which piece is on its dedicated square (with a "no piece" designation in case there is no piece on the square).

These models have 128 binary inputs: a binary input for whether a white piece is on a given square and a binary input for whether a black piece is on a given square, for each square on the chess board. When the given square is empty, both inputs for the square are set to zero.

One sample of input data to the model (representing the poor eyesight board state information) for an unmoved board has the following structure.

feature	value
a1_white	1
a1_black	0
a2_white	1
a2_black	0
...	...
e4_white	0
e4_black	0
...	...
h7_white	0
h7_black	1
h8_white	0
h8_black	1

The models have either 11 or 13 possible outputs. A given square can contain a white or black pawn, knight, bishop, rook, queen, king, or can be empty. However, the squares on the first and eighth ranks cannot ever contain a pawn because pawns start on the second and seventh ranks, only move toward the other side of the board, and promote to another piece when they reach the opposite end of the board.

One sample of output data (representing the correct piece) for the "a1" square on an unmoved board has the following structure, note that there are 11 total possible occupants of the square, due to the impossibility of pawns.

output class	value
a1_white_knight	0
a1_black_knight	0
a1_white_bishop	0
a1_black_bishop	0
a1_white_rook	1
a1_black_rook	0
a1_white_queen	0
a1_black_queen	0
a1_white_king	0
a1_black_king	0
a1_empty	0

One sample of output data (representing the correct piece) for the "e4" square on an unmoved board has the following structure. Note that there are 13 total possible occupants of the square, due to the possibility of pawns.

output class	value
e4_white_pawn	0
e4_black_pawn	0
e4_white_knight	0
e4_black_knight	0
e4_white_bishop	0
e4_black_bishop	0
e4_white_rook	0
e4_black_rook	0
e4_white_queen	0
e4_black_queen	0
e4_white_king	0
e4_black_king	0
e4_empty	1

The hope is that these models they will learn common board patterns and structures. For example, if only a few pieces have been moved from the second rank, then those pieces are likely pawns, if there is a piece far beyond the back rank while the second-from-back rank is still occupied and not many pieces have moved, then the further piece is probably a knight or bishop, or if there are diagonally-adjacent chains of pieces, then they are probably pawns, and so on.

One fault of this model is that the squares do not work together to form a sensical, legal board state. The individual square models predict their occupant independently. It is possible no king is predicted, multiple kings could be predicted, a large number of queens could be predicted, etc. The predicted board state is minimally algorithmically fixed to produce a legal board state for playing against Stockfish.

The better this model performs, the better Stockfish's idea of the exact board state, the tougher it will be to beat!

Because there are an incredible amount of possible board states (the best estimates place this number around 10^{45}), the models cannot possibly be trained on all of them. I focus the training on positions played in actual games. These will be more useful when using the model to play against humans as it has observed relevant positions.

3.3 Why not predict a whole boardstate at once?

It is possible to predict an entire board state at once with careful setup of the model's cost function, using only segments of the then-800-length output layer (considering 11 possible occupants for the 16 squares on the first and last ranks, and 13 possible occupants for the rest of the 48 squares) to predict a given square's occupant. However, the network would have to be quite large to accommodate for the vastly increased output layer size. Also, as will be discussed in Section 5.2, each square is almost always empty so dealing with the large "class imbalance" square-by-square would be a huge challenge when predicting an entire board at once.

3.4 Incorporating the Model with Stockfish

Figure 1 shows the workflow chart of a game when Poor Eyesight Stockfish plays against a game against an opponent. The opponent makes a move, then the poor eyesight board state (piece location and color data) is sent to the poor eyesight model. The poor eyesight model predicts a board state, then this predicted board state is fixed as discussed in Section 6, ensuring one king of each color is on board. Stockfish uses this fixed predicted board state to make a move. This move is made, and it is the opponents turn again. This repeats until the end of the game.

4 Data

4.1 Data Source and Parsing

The training and testing data come from Lichess's database of rated games. I semi-arbitrarily chose the March, 2016 database. This database contains PGN data for 5,801,234 games. When unzipped, this database file is 5.07GB. A PGN (Portable Game Notation) file contains information on the chess game played, including player usernames, player ratings, the game result, the moves played, etc. I used the library python-chess to parse these files and analyze individual board states by progressing a chess board through a each game's list of moves.

4.2 Data Storage and Volume

For each game in the dataset, I parsed board state information from every turn. After every turn, I stored the piece and color on every square (with an empty designation) and stored the result as one row in a table.

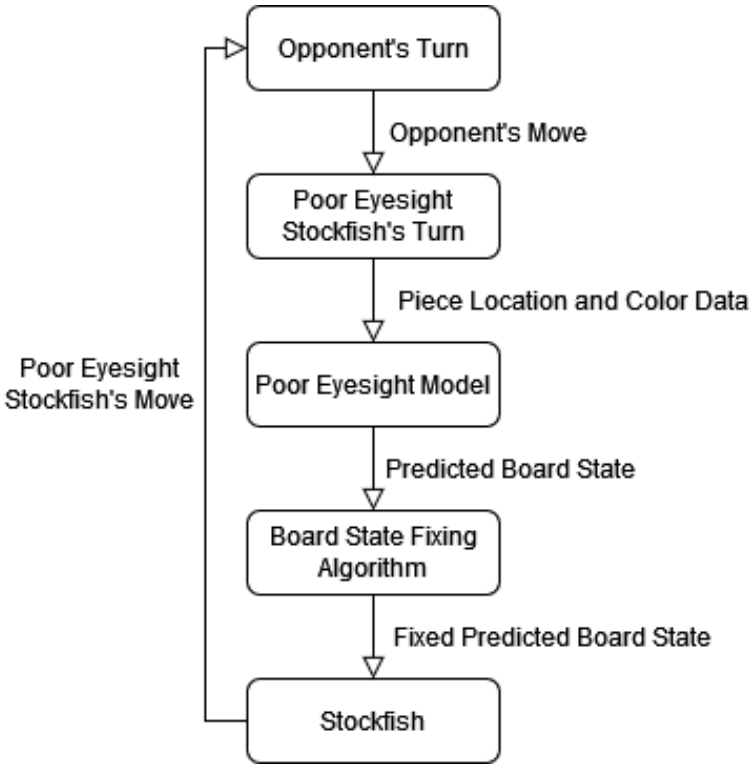

Figure 1: Chess game workflow chart with Poor Eyesight Stockfish

Because, especially in the first few moves of the game, there are very common board states, I only stored unique board states. It did not make sense to store an unmoved board or one with a single pawn moved thousands of times.

Additionally, after the first 10 moves or so, it is very *unlikely* a board state had been seen before. Thus, a huge number of board states were observed. Because I have limited storage and RAM on my computer, I had to limit the total board states stored in some structured way. I decided to only look at 100,000 total games in which both players were rated at least 2000. The board states will be observed from skilled players and thus be more likely or more at least important to get right. I figured observing lower-rated players' games would be more likely to lead to fairly nonsensical and unlikely board states.

This produced just over 6.5 million unique board states. Loading all of these into a pandas dataframe at once used approximately 20GB of my 32GB total RAM. When simply loading the dataset takes so much RAM, it is difficult to fit in model training. I trimmed off a minor amount of board states from the end to bring the dataset to 6.5 million unique board states, randomly shuffled them, then stored them in 65 100,000-board state files. Splitting the files made it easier to manage analysis and testing than loading one large file in chunks

every time.

Beyond issues of storage, using a much larger dataset would lengthen training times for questionable increases in performance. Intuitively, how important is it to train on the 7 millionth unique board state, if it had not been observed in any of the many thousands of prior games?

If I had access to larger computing nodes, more RAM, more storage, more time, etc. combined with a more suitable distributed database library such as Apache Spark, I would feel more comfortable training on more board states.

5 Training, Testing, and Tuning

I used the first 5.5 million unique board states for training and the remaining 1 million unique board states for testing.

5.1 Parameter Tuning

I used deep learning models, built with PyTorch, for each square's classification model. Due to lengthy training times bogging down my personal computer, I decided to tune parameters manually. I examined performance on a few squares, "f5" and "g4", squares on which a variety of pieces commonly land. If I had access to larger computing power, I would use a hyperparameter tuning library such as Optuna to find the most optimal parameters.

I started with a similar model structure to that of tom7 in 30 Weird Chess Algorithms: a three layer neural network with a moderately large first layer, a much larger second layer, then a third layer which was equal in size to or slightly smaller than the second layer. I kept all layer sizes equal to a power of two, with all three layers containing at least 128 nodes, the size of the input layer. I examined a variety of structures, with the first layer ranging from 256 to 2048 nodes, the middle layer ranging from 512 to 8192 nodes, and the last layer ranging from 128 to 2048 nodes. I found that the performance of the (512,2048,2048) had the best performance. All of these evaluations were performed using a batch size of 1000 and 5 training epochs. A learning rate of 0.1 provided the fastest convergence without any divergence issues after testing with 0.1, 0.05, 0.01, 0.05, 0.001. Using a step size momentum of 0.9 sped up convergence drastically.

Because I expected the last layer to be significantly smaller than the second layer, somewhere between the prior layer and the output layer, I decided to add fourth layer. I kept the first 3 layer sizes and evaluated with a fourth layer sizes of 128 through 1024. A layer size of 256 provided the best performance.

Finally, I evaluated performance on the test set across a number of total training epochs to avoid over-fitting. I found that 20 epochs had good performance, with more epochs leading to overfitting issues and decreased test set accuracy.

Figure 2 shows the final neural network architecture for a single square's model.

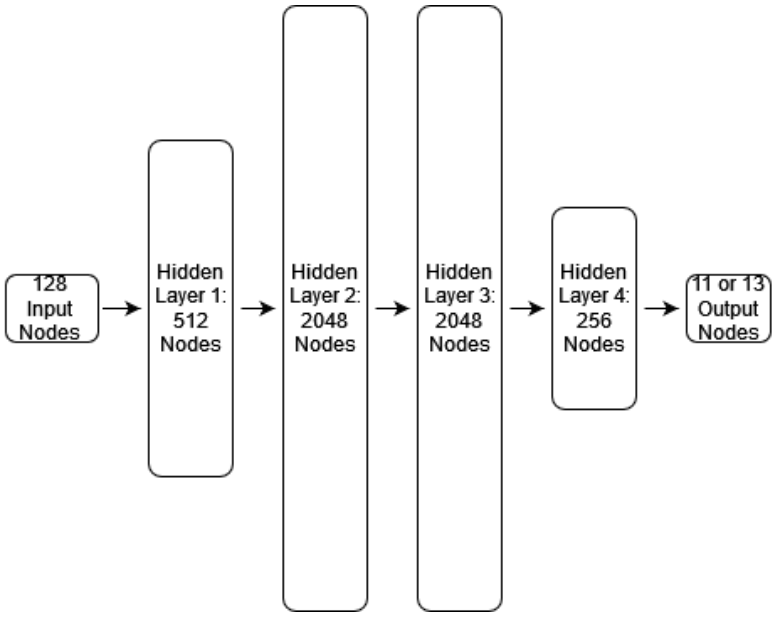

Figure 2: Poor Eyesight neural network model for predicting the occupant of a single square.

5.2 Class Imbalance

For any given square, there is no piece on the square for between 60-95% of the board states. This provides a challenge for training, as the models tended to predict "empty" every time and achieve low training cost. To solve this, I over-sampled the board states, split by the corresponding square occupant, by repeatedly training on them in inverse proportion to how frequent they were. The number of times to repeat training on board states corresponding to a given output class was given by 40,000 divided by the number of board states associated with given output class, rounded up. This process of repeating samples was performed for each of the files originally containing 100,000 board states used in training.

This led to an overall minimum of approximately 2.6 million samples of each square occupant, and an overall approximately five times increase in the number of samples trained per epoch, for a total of 31 million samples per epoch. This roughly balances the number of dataset entries of each square occupant such that each square occupant comprises approximately 1/13 of the dataset, where 13 is the typical number of possible square occupants.

This over-sampling process during training drastically improved test set accuracy, going from predicting "empty" every time to a per-piece accuracy over 60%, sometimes over 90%. Note that the test set was *not* over-sampled.

There are methods of creating interpolated data to oversample and reduce

overfitting and often involve some sort of K-means clustering. Because of a variety of factors—binary input data, desire to include only board states seen in real games, the finicky nature of having 13 output classes, the drastic nature of the class imbalance—I decided to do a simple repeated-sample technique. The performance increase on the test set speaks to the success of this technique in this case.

6 Implementation Considerations

As mentioned in Section 3, the individual models predicting the piece on each square do not work together to ensure a legal board state. When assessing accuracy and performance of the individual models, this is not important. This becomes important when feeding a predicted board state into Stockfish, which requires a legal board state to suggest a next move. Fixes need to be manually/algorithmically implemented to issues preventing illegal board states.

The primary issue to fix is the amount of kings on the board. For example, if there are no kings predicted, the square which is *most likely* a white king is set to contain a white king. If there are multiple white kings predicted, the square which is most likely a white king is kept, while the less likely squares are set to the next-most likely piece. The same process is performed for black kings.

7 Performance