Using Deep Learning to Predict a Player's Rating on Online Chess Websites

CS 230 Final Project

Robert Bennett (Solo Group)

Code Available: `https://github.com/RKABennett/CS_230_project.git`

Abstract

Chess websites quantitatively assess players' strengths by assigning them a numerical rating. In this work, I present a neural net capable of guessing a chess player's rating based the moves from a single game they played. To accomplish this task, I implement a novel encoding method where individual moves in a game of chess are represented as vectors. The final model is trained on ≈ 2 million chess games and achieves a root mean square error of 204, representing 55% of the standard deviation of the dataset.

Contents

1 Introduction

Chess websites use various rating systems to quantify a player's performance. Recently, some top-level players have released popular YouTube videos where they inspect previously played games and attempt to guess the players' ratings [1]. The idea is simple: top-level players have sufficient knowledge of chess to determine whether a player is playing well or not, allowing them to guess the players' ratings based only on a single game. The goal of this project is to use deep learning to accomplish the same task. Specifically, the program will aim to output the average ratings of two players based on a single game that they play without any prior knowledge of the players' skill levels. Practically, such a program could be useful within the chess community for three main reasons: cheating detection (to see if a player is playing well above their actual ranking), rating initialization for new accounts, and gaining understanding of chess players of different ratings.

1.1 Previous Works

Although Chess has been a major topic in artificial intelligence for decades, there have been no previous scholarly works that address the present problem. To my knowledge, the only similar effort made on the present problem is an undergraduate project report where the author attempted to accomplish the same task [2]. In the aforementioned work, the author analyzed ≈ 30000 games from the Lichess database using feed forward NNs and convolutional neural networks (CNNs), as well as ensemble and XGBoost regression models. In this work, of all models tested, the author finds that XGBoost gave the best performance, ultimately achieving a final root mean square (RMS) error of 159 rating points. Unfortunately, a major shortcoming of this previous work (which makes it impossible for me to benchmark against) is that the author does not discuss their dataset in any detail. It is unclear, for instance, if the 30000 games analyzed by the author were randomly chosen from the dataset, or if the games were clustered around similar ratings, which would make the present task easier. More importantly, the author does not separately discuss the RMS error of the training and test set, instead reporting only a single value for the model's overall accuracy. It is therefore unclear if this error corresponds to the training or test set error. To avoid a similar problem in my own work, I make sure to discuss the dataset and its statistics thoroughly in the next section. Finally, it should be noted that the only game features that the author makes extensive use of in their work are the centipawn losses and overall average centipawn loss in the game (I define the meanings of these terms in the next section), although much more data is available for every game played. Thus, to improve upon the aforementioned work, I will incorporate more features of games into my model in an effort to improve its performance.

2 Dataset

This project makes use of Lichess.org's open database [3], which is a free database released under a Creative Commons CC0 license containing detailed information about the vast majority of games that have been played on Lichess.org since 2013. As of November 2022, this database is 989 GB in size and contains data from 3.8 billion games. An example of a game from this database is shown below. Note that the format has been slightly modified to improve readability and the game is truncated for brevity.

```
1      [Site "https://lichess.org/e5ZEY0QD"]
2      [BlackElo "1920"]
3      [ECO "C40"]
4      [Termination "Normal"]
5      [TimeControl "600+0"]
6      [WhiteElo "1446"]
7
8      1. e4 { [%eval 0.33] [%clk 0:10:00] } 1... e5 { [%eval 0.12] [%clk 0:10:00] }
9      2. Nf3 { [%eval 0.19] [%clk 0:09:58] } 2... d5 $2 { [%eval 1.23] [%clk 0:09:59] }
10     3. d4 $6 { [%eval 0.39] [%clk 0:09:47] } 3... dxe4 { [%eval 0.87] [%clk 0:09:58] }
```

The first six lines feature useful information about the game: a permanent URL to the game is shown in line 1 (which is very useful for bug-checking the pre-processed game) and the ratings of both players are shown in lines 2 and 6. Lines 3, 4, and 6 list the opening system featured in the game (i.e., a name given to the first several moves), how the game ended (e.g., checkmate, resignation, draw by agreement), and the time control (i.e., the time allotted to each player, along with additional time the player receives which each move that they play). The final three lines in the file shown above document the actual moves played in the game. The move played by White is given in algebraic chess

notation (e.g., "Re1" means "rook moved to square e1"), followed by the *engine evaluation*[1] of the position after the move was played and the time left on the clock after the move was played.

The lichess database contains data from players of all ratings and across all possible time controls (where a time control refers to how much time each player receives). For the purposes of this project, I will focus specifically on "600 + 0" games, i.e., games where each player starts with 600 seconds (10 minutes) at the beginning of the games and does not get any bonus time for moves they play. Moving forward, all games will refer to 600 + 0 games, and it should be understood that the dataset has been filtered to include only 600 + 0 games for all analyses. For all models, I use Lichess's database from July 2022 (i.e., all games played between July 01 2022 and July 31 2022) as my base dataset ($\approx$ 1 million 600+0 games total) where the training, development (dev) and test datasets followed a 0.95/0.025/0.025 split. All 600 + 0 games in this dataset were randomly shuffled prior to splitting.

The histogram in Fig. 1 summarizes the dataset by showing counts of the average players' rating in each game in the dataset. From this distribution, the average game has a player rating of 1762, and the standard deviation of ratings is 368. The histrogram appears to be somewhat Gaussian in shape and therefore contains fewer games played by players of more extreme ratings, suggesting that we may eventually benefit from data augmentation. It's also worth noting that the average 600+0 rating of players on lichess is $\approx$1500 [4], which is less than the average game's rating in the game database. The reason for this discrepancy is because stronger players will probably play more games than weaker players (or, rather, those who play more games will *become* stronger players).

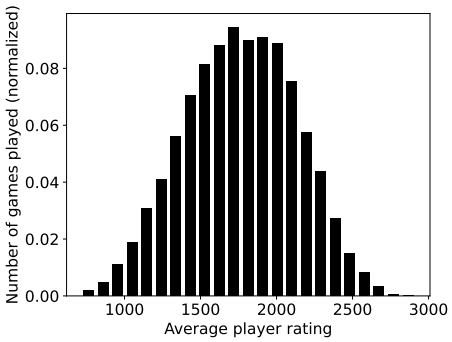


Figure 1: Histogram showing the average player rating as a function of games in the Lichess July 2022 database.

3 Baseline and Best Possible Human Error

A baseline for the task of guessing a player's rating based on a single game could be rooted in the accuracy of the average move that they played throughout that game. The accuracy of a single moved can be judged by its *centipawn loss*,[2] and the average accuracy of the game can be calculated by taking the average centipawn loss of every move. Therefore, the baseline I am using for this project is a simple linear regression on average centipawn loss vs. rating across games. The intuition behind this baseline is simple: stronger players should, on average, play better moves than weaker players. Therefore, games played by stronger players should have lower average centipawn losses than weaker players, suggesting that the strength of a player could be crudely estimated based on their average centipawn loss in a single game. From computing this linear regression, the average centipawn loss in a game does indeed appear to be weakly correlated to the strength of the players. However, this correlation is weak; applying this "model" results in a root mean square error of 344.2, which is 93.5% of the dataset's standard deviation of 368. To assist our analysis later on, it should be noted that the human error for the task is approximately 323, as discussed in Appendix A.

[1] The engine evaluation is a quantitative score, evaluated by a chess computer, that assesses the advantage held by either player. The sign of the score indicates which player is winning, and the magnitude indicates the degree to which they are winning. The engine evaluation is given in the unit of *centipawns*, where one centipawn is equivalent to one one-hundredth of the value of one pawn. For example, if the engine evaluation is +100, then White holds an advantage equivalent to having one more pawn than Black.

[2] Centipawn loss is defined as the change in the engine's evaluation after a single move is played. If the best possible move is played then the engine evaluation should not change, since the best possible move was taken into account while computing the engine evaluation. Hence, "accurate" moves should have very low centipawn losses. For example, a player's Queen is worth roughly 800 centipawns, so a move that gives away a player's Queen will carry a centipawn loss of approximately 800.

4 Method, Results, and analysis

The problem of assigning a rating to a game of chess is conceptually similar to sentiment analysis. In sentiment analysis, we are given a sequence of words and based on these words, attempt to assign a rating corresponding to positivity of negativity of the phrase being expressed. Likewise, in this particular problem, we are given a sequence of moves representing a game of chess and we are attempting to assign a rating to the strength of the players. Therefore, neural network architectures that are used in sentiment analysis appear especially promising for the problem at hand.

4.1 Representing Moves as Manually Embedded Vectors

Taking inspiration from natural language processing (NLP) techniques, I am representing individual chess moves as vectors, just as we represent individual words with embedding vectors in NLP. In this representation, the i^{th} move, $M^{[i]}$, is represented by the following vector:

$$M^{[i]} = \left[E^{[i]}, t^{[i]}, Chk^{[i]}, Cap^{[i]}, Cask^{[i]}, Casq^{[i]}, Pr^{[i]}, K^{[i]}, Q^{[i]}, R^{[i]}, B^{[i]}, N^{[i]}, P^{[i]}, c^{[i]}, r^{[i]}, Term, Res\right]$$

where $E^{[i]}$ is the engine evaluation after the move is played. $t^{[i]}$ is the length of time spent on the move. $Chk^{[i]}, Cap^{[i]}, Cask^{[i]}, Casq^{[i]}$, and $Pr^{[i]}$ are binary variables that describe the move itself, where the variable is given a variable of 1 if the feature is present and 0 otherwise. In order, these variables describe whether or not the move is a check, capture, kingside castle, queenside castle, and pawn promotion. $K^{[i]}, Q^{[i]}, R^{[i]}, B^{[i]}, N^{[i]}$, and $P^{[i]}$ are binary variables that are 1 if the piece moved was a king, queen, rook, bishop, knight, or pawn, respectively. $c^{[i]}$ and $r^{[i]}$ denote the file (column of the chess board, A-H, represented as 1-8) and file (row, 1-8) of the chess board. Finally, $Term$ and Res are quantities associated with the overall game outcome. $Term$ is 1 if the game ended by time forfeit (i.e., if one player lost because they ran out of time) and 0 otherwise, and Res = 1, -1, or 0 if the game was won by white, won by black, or tied, respectively. $Term$ and Res remain constant throughout any given game but of course will vary between games.

4.2 Neural Network Architecture Overview

Because we have sequential data, and because of the similarities of the present problem and sentiment analysis, I am using a long short-term memory (LSTM) as the NN architecture for the present problem. The network architecture uses n_{lstm} LSTM layers, where the ith LSTM layer has $m^{(i)}_{lstm}$ hidden units. The output from the LSTM layers are then fed into a series of n_d dense layers, where the jth dense layer has $m^{(j)}_{d}$ hidden units. Finally, the prediction for the average players' rating is computed using a dense layer, which outputs to a single value. This NN architecture is summarized in Fig. 2 for a sample $n_{lstm} = 2$, $n_d = 2$, $m^{(1)}_{lstm} = m^{(1)}_{lstm} = 32$, and $m^{(1)}_{d} = m^{(1)}_{d} = 16$. The activation function for each layer will be discussed in upcoming subsections.

All models were trained using the Adam optimizer, where all parameters used were default parameters from Tensorflow except for the learning rate, which was varied across models. Finally, batch normalization layers were inserted after every layer in the NN to speed up training. All models are training using early stopping with a patience of 10 and batch sizes of 256, except for where otherwise noted.

Since we are attempting to predict an individual rating as a scalar quantity, our loss function L is simply the average sum of square errors over our entire training set. That is, for N training games, we have:

$$L = \left[\frac{1}{N}\sum_{i}^{N}(Y^{(i)}_{actual} - Y^{(i)}_{predicted})^2\right]^{0.5}$$

Since this is a minimization problem, we do not actually have to take the square root or divide by N. However, doing so means that our metric of interest (the RMS error) will equal our loss function, which is convenient for our analysis.

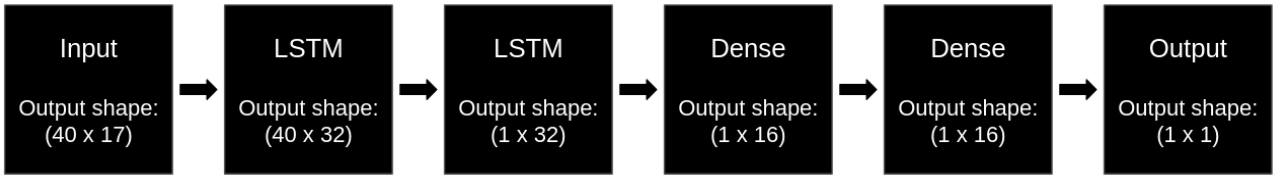


Figure 2: Sample NN architecture for $n_{lstm} = 2$, $n_d = 2$, $m^{(1)}_{lstm} = m^{(1)}_{lstm} = 32$, and $m^{(1)}_{d} = m^{(1)}_{d} = 16$.

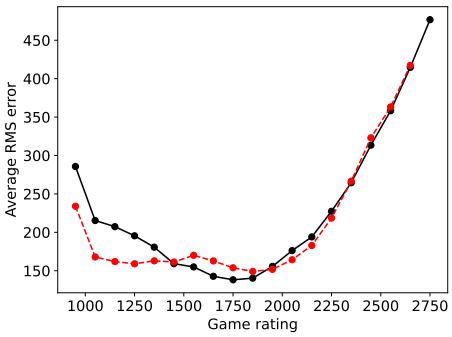


Figure 3: Average RMS error for games in the test dataset as a function of their rating. Games were averaged across ± 50 rating points, e.g., the data point at game rating = 1050 includes all games with ratings between 1000 and 1100.

4.3 Initial Preliminary Modeling

To verify that the proposed methodology/architecture could work, in the earliest stages of the project I performed initial testing using an architecture chosen somewhat arbitrarily as a "proof of concept" before spending time optimizing. Here, I used the architecture outlined above with $n_{lstm} = 2$, $m_{lstm}^{(1)} = m_{lstm}^{(2)} = 32$, $n_d = 1$, and $m_d^{(j)} = 8$. All layers used ReLU activation functions except the final output layer (as the output quantity doesn't fit nicely to a ReLU function). The output layer instead was either a linear or a tanh function; when a tanh function was used, all data was scaled to between -1 and +1, as this is the range of tanh.

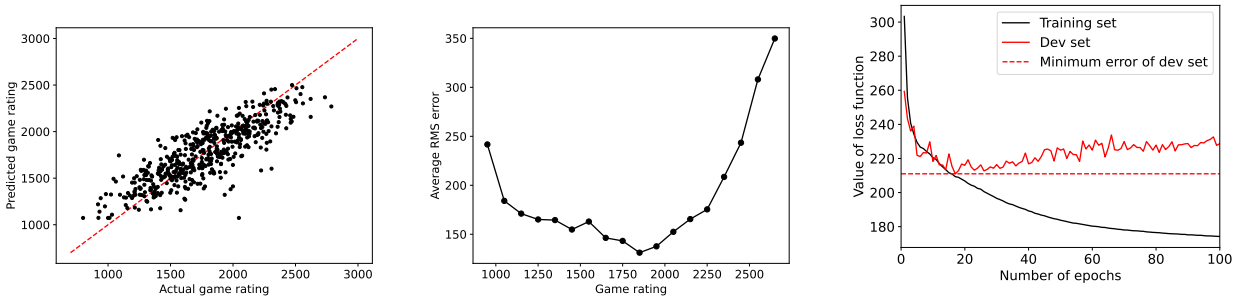
This preliminary model achieved an RMS error of 252 on our test set after 21 epochs. Already this preliminary and unoptimized model significantly outperformed the baseline RMS error of 344 and estimate for human error of 323, suggesting that the proposed methodology is indeed promising for the problem at hand. To further analyze this initial model, I have plotted the error vs. game rating in Fig. 2 (again, using results only from the test set). This plot shows that the model is having an especially hard time predicting games at either extreme ends of our dataset, which may arise because we have fewer games in our training set corresponding to these ratings (see Fig. 1). In an effort to further reduce error, I added all games from the August 2022 with ratings < 1300 and > 1900 (approximately 500,000 games total) to the training set, leaving the dev/test sets unchanged. As shown in Fig. 2, this approach slightly improved performance at the extreme ratings and reduced the average RMS error in the test set to 237. Finally, I noticed that the model output essentially identical results regardless of it was trained using a tanh or linear output function, so I arbitrarily chose to use the tanh output function for all subsequent models.

4.4 Hyperparameter Search and Refinement

Next, to optimize the model, I performed a hyperparameter search. Following the approach described in CS 230 lectures, I used a random search methodology, where many architectures were built by varying relevant hyperparameters randomly. N_{LSTM} was varied between 1 and 3, and each $m_{lstm}^{(i)}$ was individually varied between 16 and 256. N_d was varied between 1 and 4, and each $m_D^{(j)}$ was individually varied between 8 and 128. The activation function for all LSTM layers, as well as the activation function for each individual dense layer, was chosen randomly as either tanh or ReLU. Finally, $\alpha = 10^{-r}$ was varied by randomly varying r between 2 and 6 to ensure even sampling across these orders of magnitude. For all tuning I used the same test/dev sets as above, and I used the augmented training set that contained extra games with more extreme ratings from August 2022.

I tested 100 unique models using the above methodology. Afterwards, I found the three best models (as measured by the RMS error on the test set after training). Then, I performed further optimization using these as starting points. In this second "focused" hyperparameter sweep, the number of LSTM/dense layers and their activation functions were held constant. The number of hidden units in each layer was randomly varied between $n + 20$ and $n - 20$, where n is the number of hidden units in that layer in the original high-performing model, and the learning rate $\alpha = 10^{-l}$ was varied by randomly varying l between $r - 1$ and $r + 1$, where r was the learning rate exponent in the high-performing model. I tested 15 randomly developed models for each starting point using this approach (i.e., an additional 45 models total), and then selected the highest performing model for further analysis and optimization.

Of all models tested, the highest performing model achived an RMS error of 209 (51% of the dataset's standard deviation) on the test set, representing a substantial improvement compared to the baseline and compared to my



(a) (b) (c)

Figure 4: Analysis of the best performing model: (a) loss functions of the training/dev sets, (b) Average RMS error as a function of game rating, and (c) predicted vs. actual rating in the test set for 250 randomly chosen games. The red dashed line shows actual = predicted rating; the farther a datapoint is from this line, the greater its error.

estimation of best human error (344 and 323, respectively). This model used 2 LSTM layers ($m_{lstm}^{(1)} = 101$, $m_{lstm}^{(2)} = 174$) with tanh activation functions and 4 dense layers ($n_d^{(1)} = 62$, $n_d^{(2)} = 100$, $n_d^{(3)} = 95$, $n_d^{(4)} = 15$), with corresponding activation functions of tanh, relu, relu, and tanh, and a learning rate of 8.8×10^{-3}.

4.5 Further Optimization

To further analyze the optimized model, I have plotted the true vs. predicted error in Fig. 4a and error vs. game rating in Fig. 4b. From these results, it is apparent that our model is still performing especially poorly on games whose ratings lie at either extreme. Furthermore, the loss function of the best performing model is shown in Fig. 4c. After 100 epochs, the loss function achieves a value < 180 and is still decreasing (albeit slowly) on the training set, whereas the loss function reaches a minimum value of 211 on the dev set after 17 epochs. This result suggests that I am overfitting my training dataset and that the error on the dev set could be reduced if I could reduce variance.

Based on this analysis, I attempted to further improve performance on the test and dev sets with the following methods: (1) Further augmenting the training dataset to include more games at both extremes (using previously unused data from the Lichess June 2022 database); (2) Further augmenting the training dataset by duplicating game data from both extremes. (3) Increasing the size of the training dataset by adding 500,000 randomly chosen games from the Lichess June 2022 database; (4) Using dropout on (i) LSTM layers only, (ii) dense layers only, and (iii) LSTM and dense layers simultaneously. All three appoaches were tried using keep probabilities of 70 and 85%; (5) Decreasing the learning rate by a factor of 10 after the loss function on the dev set stopped decreasing; and (6) Combining methods 3, 4-i, and 5. Methods 1, 2, 4-i, and 4-iii increased the error by 5-10 points. Methods 3, 4-i, and 5 each decreased the error by 2-3 rating points, and method 6 decreased the error by 6 rating points. Hence, after these final optimization steps, the model achieved an overall RMS of 204 on the test set.

5 Conclusions and Future Work

I have developed and trained an NN capable of guessing a player's chess rating based on a single game that they have played. After training the NN with approximately 2 million games and optimizing the model across a wide range of hyperparameters, the best performing model (archicture/hyperparameters documented in Section 4.4) achieved an RMS error of 204 (55% of the standard deviation of the dataset). This error is significantly smaller than our baseline error of 344 and best human error of 323 (93 and 88% of the standard deviation of the dataset, respectively). If the present work is to be continued or improved upon, I suggest the following two steps/considerations:

First, before attempting to further improve the present model, I recommend first attempting to establish a better estimate of Bayes error to first ensure that we have not already achieved best or near-best performance on the present task. Second, it should be noted that the present model uses a novel vector representation of chess moves as inputs to provide a reasonable estimate of a player's strength. Unlike NLP applications, where vectors representing words are abstract and learned as part of the NN process, the vectors used to represent moves are manually encoded and represent specific attributes of the move, such as the piece moved, the square the piece was moved to, and the time spent on the move. Although representing moves in this way may be convenient and easily accomplished, representing moves through learned encodings may be a necessary next step to further reduce error. However, it is not clear what loss function should be used to train these encodings to determine how moves can be classified in terms of "similarity."

6 Contributions

I completed the project alone have done all components of this project by myself. The only assistance I have received has been from CS 230 TA's during their office hours.

References

[1] GothamChess, "GUESSING MY SUBS' ELO." Accessed 10 November 2022; available online: `https://www.youtube.com/watch?v=0baCL9wwJTA`.

[2] P. Avva, "Guess the elo – predicting chess player rating," *Princeton University SML 310 Report*, 2022.

[3] `Lichess.org`, "Lichess Open Database." Accessed 18 October 2022; available online: `https://database.lichess.org`.

[4] `Lichess.org`, "Weekly Rapid Rating Distribution" Accessed 08 December 2022; available online: `https://lichess.org/stat/rating/distribution/rapid`.

Appendix A: Estimation of Best Possible Human Error

Although it's difficult to determine Bayes error for the present task, having an idea of the lower limit of human error will still help us analyze the performance of our model. To establish this estimate of human error, I analyzed a series of videos, titled "Guess the Elo," by a popular chess YouTuber named GothamChess (note that "Elo" refers to player rating). GothamChess (real name Levy Rozman) is a high ranked chess player who holds the title of International Master. He has published over 50 YouTube videos in his "Guess the Elo" series where he manually accomplishes the present task of guessing chess players' ratings by analyzing games played on the popular chess website, `Chess.com`. Because GothamChess is a strong chess player who regularly *practices* the present task, he is an ideal individual to use to establish a lower limit for the human error of this work.

To estimate GothamChess's error on the present task, I analyzed four of his videos (videos 30, 31, 32, and 33 of his Guess the Elo series[3]) and computed his own RMS error by comparing his guesses to the actual ratings of the players. Occasionally, GothamChess attempts to guess the ratings of both players individually and/or gives an upper and lower bound for the ratings. In these cases, I simply averaged his guesses for both players and/or took the middle of the lower and upper bounds. From this analysis, GothamChess achieved an RMS error of 323 across these games, with the individual games summarized in Table 1.

Table 1: Breakdown of actual and guessed chess ratings from GothamChess's youtube series.

Video #	Game #	Guess	Actual – white	Actual – black	Error	Error squared
30	1	800	834	860	47	2209
30	2	1150	1144	1092	-32	1024
30	3	1650	1772	1828	150	22500
30	4	1650	1652	1618	-15	225
30	5	900	1281	1702	591.5	349872.25
31	1	650	685	591	-12	144
31	2	1100	849	881	-235	55225
31	3	900	1411	1438	524.5	275100.25
31	4	1250	1786	1674	480	230400
31	5	800	897	951	124	15376
32	1	1550	1115	2048	31.5	992.25
32	2	1050	1154	1132	93	8649
32	3	850	932	944	88	7744
32	4	600	1158	1111	534.5	285690.25
32	5	1100	1067	1900	383.5	147072.25
33	1	700	951	927	239	57121
33	2	1550	1115	1086	-449.5	202050.25
33	3	765	1323	1377	585	342225
33	4	200	318	373	145.5	21170.25
33	5	1150	1541	1268	254.5	64770.25

[3]Available online using the four URLs, each accessed on November 18 2022: `https://www.youtube.com/watch?v=X0J2Lq2z2Vk`, `https://www.youtube.com/watch?v=i__ZXVjCl_I`, `https://www.youtube.com/watch?v=0JhTJRl4uew`, `https://www.youtube.com/watch?v=X0xRJRF-g5w`.

Appendix B: Sample Move Vector

Here I show a sample move vector using the convention discussed in Section 4.1. Consider a game where Black won as a result of White running out of time. In the tenth move of the game, Black spent 22 seconds thinking before moving their knight to A3, capturing one of White's pieces and putting White's king in check. The engine evaluation of the board after playing the move is -0.53. Then, the vector corresponding to the tenth move is:

$$M^{[10]} = \Bigg[\underbrace{-0.53}_{E^{[10]}}, \overbrace{22}^{t^{[10]}}, \underbrace{1}_{Chk^{[10]}}, \overbrace{1}^{Cap^{[10]}}, \underbrace{0,0,}_{Cask^{[10]},Casq^{[10]}} \overbrace{0}^{Pr^{[10]}}, \underbrace{0,0,0,0,}_{K^{[10]},Q^{[10]},R^{[10]},B^{[10]}}, \overbrace{1}^{N^{[10]}}, \underbrace{0}_{P^{[10]}}, \overbrace{1,3,}^{c^{[10]},r^{[10]}} \underbrace{1,-1}_{Term,Res} \Bigg]$$

Then, the game matrix G is compiled by horizontally stacking all of the moves. That is, for a game with N moves, G is given by $G = \left[M_0^T, M_1^T ... M_N^T\right]$.