Chessformer - Leveraging Transformer-based neural networks to predict results and ratings in chess games

Luiz do Valle

December 2023

Abstract

Transformer-based architectures have been applied to a variety of sequence related tasks. Chess games can be naturally represented as a sequence of moves. This project shows the potential of using Transformer-based neural networks to model chess games. In predicting which player won a game given only the moves, the model achieved an overall accuracy of 86.2%. Excluding games that ended in a tie, the model had an even higher accuracy of 89.5%. In the Elo rating prediction task, the model learned the most common range of ratings, but did not manage to capture the full variability of the data. The best model had a Root Mean Square Error of 333.

1 Introduction

Transformer-based neural networks have been used successfully to model a wide variety of sequence-related tasks since their introduction in [4]. This architecture has significant advantages over previous seq-to-seq models, like LSTMs [1]. For example, the attention mechanism, which is at the heart of the Transformer, allows the model to capture long range dependencies better than LSTMs, which suffer significantly more from vanishing and exploding gradients during training.

Chess is an extremely popular strategy board game. Given that a game can be naturally represented as a sequence of moves, it is well suited to be modeled by a Transformer-based architecture. The goal of this project is to investigate whether a Transformer-based neural network can learn to identify who won a chess game and how skilled the players involved are based solely on the sequence of moves in the game.

2 Background

Chess is a two-player strategy board game played on an 8x8 grid. Each player starts with 16 pieces arranged on opposite sides of the board. There are 6 types of pieces (pawns, knights, bishops, rooks, queens, and kings), each with a different set of allowed moves. Players take turns moving one of their pieces and a player is not allowed to skip their turn. If a player moves a piece to a square that has an opponent's piece, the opponent's piece is "captured", meaning it is removed from play. The most important piece is the king. If the king is under threat of being captured, it is said to be in check. A player cannot make a move that would result in their own king being left in check. The goal is to trap the opponent's king such that it is in check and would remain in check no matter the opponent's next move, a position known as checkmate. The game can also end if one of the players resigns. However, it is also possible for a game to result in a tie. This can happen if, for instance, the two players agree to a tie or if a player's king is trapped but not in check and the player has no other legal moves.

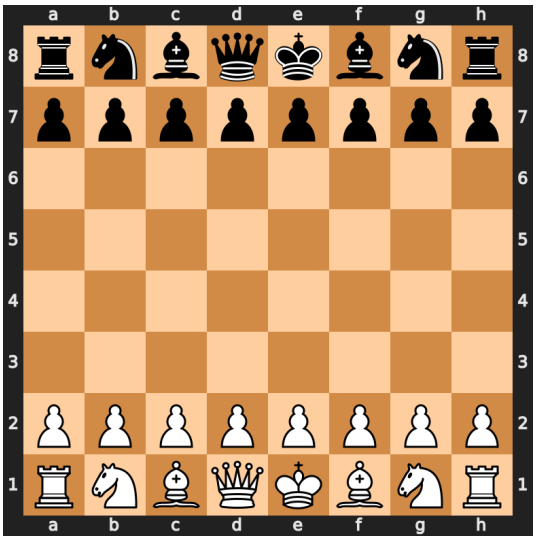

Figure 1: Initial position of the pieces in a chess game.

In order to match players of similar ability, players are given a numerical rating. The higher the rating, the more skilled a player is. Wins, losses, and ties affect a player's rating. The magnitude of the rating increase or decrease caused by these results also depends on the rating difference between the two players. For example, if a higher rated player ties with a lower rated one, the higher rated player loses points while the lower rated one gains points. Rating systems are prevalent in organized chess events and online platforms to maintain competitive balance. There are different schema for calculating a player's rating,

the most popular of which is the Elo rating system [7].

Given the sequence of moves in a game, this project aims to predict who won (or whether there was a tie) and the rating of the two players.

3 Data

3.1 Data source

There are several online chess platforms where people can play games against each other. One of the most popular is Lichess, a free and open-source chess server run by a non-profit [8]. Lichess makes available several of the games played on the platform through their open database. As of the moment of this writing, there over 5 billion standard rated games available, going from November 2023 all the way back to January 2013. After every month, the games played during that month are added to the database.

Each game is in the Portable Game Notation (PGN). This format is used to represent a chess game in plain text and stores the sequence of moves along with some metadata, like the players' ratings, the game's result, the time control, etc. [9]. Below is an example of a game in PGN from the Lichess database:

```
[Event "Rated Blitz game"]
[Site "https://lichess.org/9opx3qh7"]
[White "adamsrj"]
[Black "hamiakaz"]
[Result "0-1"]
[UTCDate "2012.12.31"]
[UTCTime "23:02:48"]
[WhiteElo "1522"]
[BlackElo "1428"]
[WhiteRatingDiff "-14"]
[BlackRatingDiff "+14"]
[ECO "A40"]
[Opening "Englund Gambit Complex: Hartlaub-Charlick Gambit"]
[TimeControl "180+5"]
[Termination "Normal"]

1. d4 e5 2. dxe5 d6 3. exd6 Bxd6 4. Nf3 Nf6 5. Nc3 O-O 6. a3
Nc6 7. e3 a6 8. Be2 h6 9. O-O Ne5 10. Bd2 Nxf3+ 11. Bxf3 Be5 12.
Rc1 c6 13. Qe2 Qd6 14. Rfd1 Bxh2+ 15. Kh1 Be5 16. e4 Bxc3 17. Bxc3
Qe6 18. Rd3 Bd7 19. Rcd1 Rad8 20. Bxf6 gxf6 21. Rd6 Qe7 22. Rd1d2
Be6 23. Rxd8 Rxd8 24. Rxd8+ Qxd8 25. c4 Qd4 26. c5 Qxc5 27. Qd2 f5
28. exf5 Bxf5 29. Qxh6 Bg6 30. Be4 Bxe4 31. Qh4 Bg6 32. Qd8+ Kg7
33. Qc7 b5 34. b4 Qc1+ 35. Kh2 Qxa3 36. Qe5+ Kg8 37. Qe8+ Kg7 38.
Qxc6 Qxb4 39. Qxa6 Qh4+ 40. Kg1 b4 41. Qa1+ Qf6 42. Qa4 Qc3 43.
f3 b3 44. Qa3 Qc2 45. Kh2 b2 0-1
```

The moves are in Standard Algebraic Notation (SAN) [5]. In this notation, a move is represented by the following components:

1. The piece's upper case letter representation: **N** (knight), **B** (bishop), **R** (rook), **Q** (queen), and **K** (king). The pawn does not have its own symbol and it is ommitted.

2. The destination square: Each square of the board is identified by a unique coordinate from the white piece's perspective. The columns are labeled from a to h. The rows are labeled from 1 to 8. A coordinate is the column followed by the row.

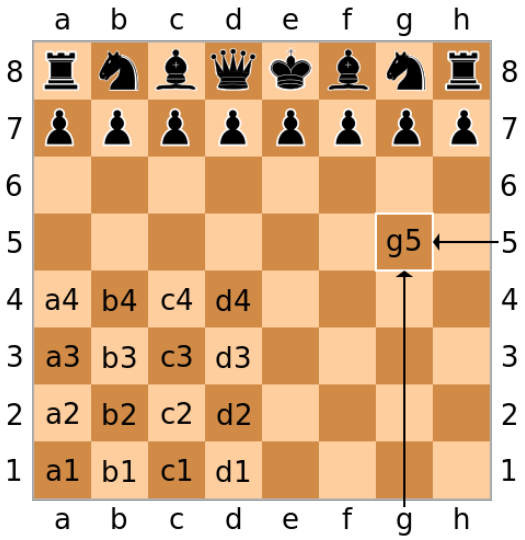

Figure 2: Coordinates in Standard Algebraic Notation (SAN) [5].

For example, **Qf7** (queen to f7) and **e5** (pawn to e5). The starting square is not specified unless the move is ambiguous. There are some other additional pieces of notation, like how to represent captures, but they will not be covered here as they are not important for the purposes of this project (for more details see [5]).

3.2 Metadata download

The original goal was to download as much of the dataset as possible in order to have enough training data. This posed a significant engineering challenge given there is a total of **1.58 TB of compressed data** (uncompressed, the size is about 7 times larger) and the workstation the project was developed on had less than 500 GB of available disk space. The dataset is sharded into 131 files but even these individual files are of a significant size, with the most recent ones (which are the largest) having compressed sizes around 30 GB.

The moves themselves are the most storage consuming portion of the PGN files, so only the metadata was downloaded at first so that a reasonable subset of the games could be selected. To do so, a custom Python script was developed. It uses the third-party *python-chess* library to parse the PGN files and extract only the games' metadata, ignoring the moves. The script is engineered so that it streams files from the Lichess database instead of downloading them at once. This prevents the computer from running out of memory when processing the files. As each file is being processed, the relevant information is saved to a Parquet file [6] on disk. To further speed up processing, the script spawns a separate process per file (up to a pre-determined number of processes to prevent the system from being overwhelmed). Each of these processes then starts two

threads: a producer thread, which downloads the data from the database and sends it to the consumer thread, which saves the records to disk. The producer-consumer paradigm [10] is used because writing data to disk is slow and as such the producer can be downloading data while the writing to disk takes place. In addition, the script backs up the files in Google Drive.

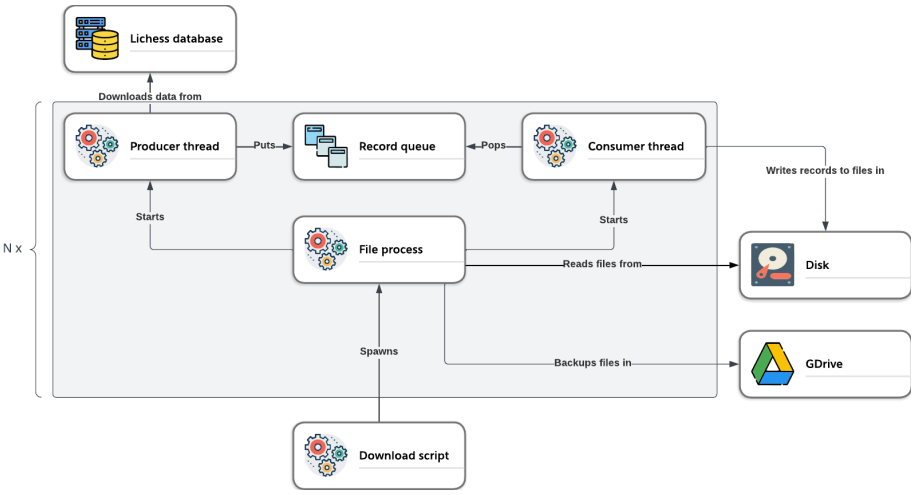

Figure 3: Download script diagram.

Using this script, the data from January 2013 to September 2023 was downloaded, a total of about 4.3 billion games. To analyze this data, parallel computing Python libraries called Dask and DuckDB were used.

3.3 Metadata analysis and game filtering

Figure 4 shows the frequency of different event types in the dataset. Event types represent similar time controls, which is the total time a player has. For example, a time control of 180+2 means each player starts with 3 minutes (180 seconds) and gets an additional 2 seconds per move. When a player makes a move, their clock is paused and the opponent's resumes. If a player runs out of time and the opponent has enough pieces to checkmate, the player loses. Because players receive different Elo ratings for each event type, a single event type needs to be chosen for the rest of the project. Since it is the the most frequent event type, Blitz was selected. Blitz games generally last from 3 to about 7 minutes.

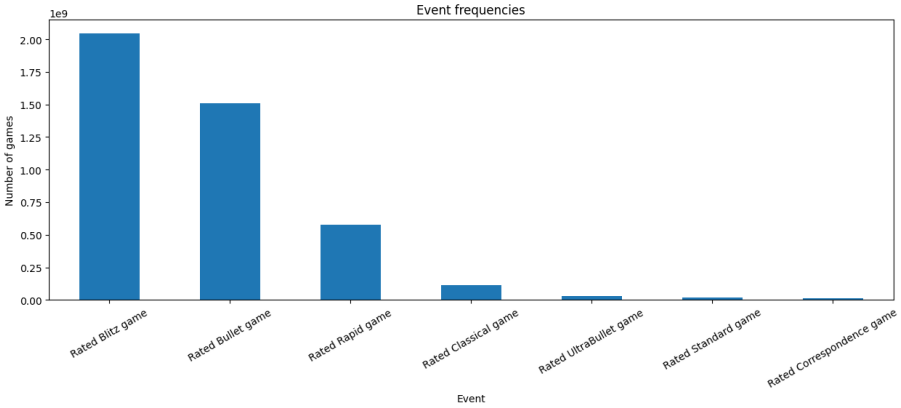

Figure 4: Event type frequencies.

Because there are more than 2 billion Blitz games, it would be infeasible to train the models on all of them. Figure 5 shows the time control distribution within Blitz games. Once again, the most frequent category was chosen: 180+0 (3 minute games with no increment).

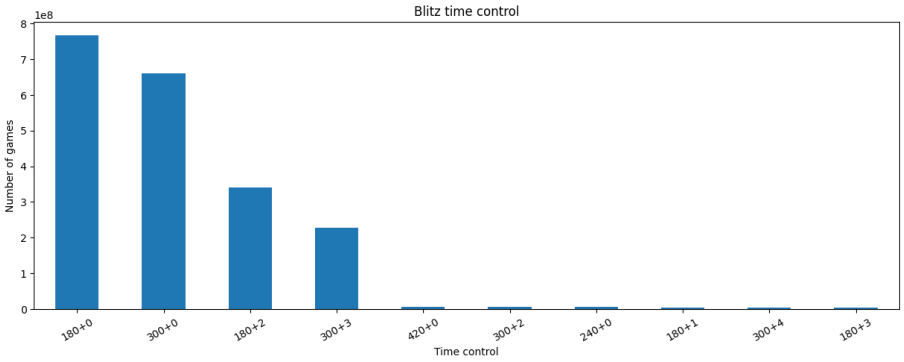

Figure 5: Blitz time control frequencies.

Finally, Figure 6 shows the distribution of how the games ended. Time forfeit means a player ran out of time on their clock. Normal means a game ended in a checkmate, a tie, or a player resigned. Since we would like the model to learn how to determine the result of a game from the moves only, using games that ended due to time forfeit would not provide a complete signal to the model. Hence, only games that ended normally were chosen.

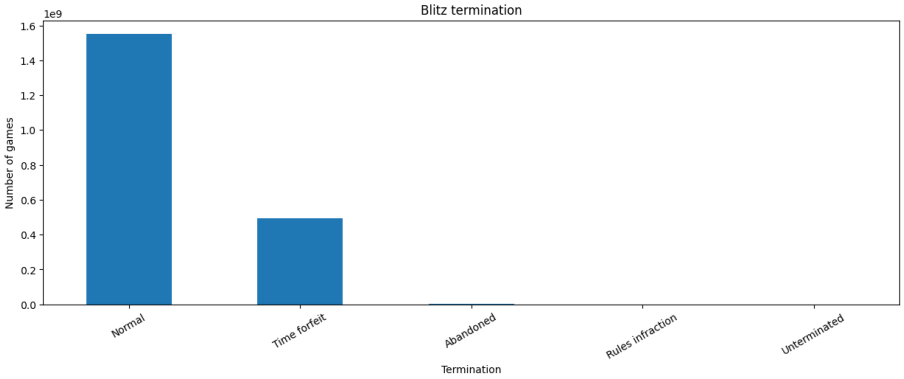

Figure 6: Blitz termination frequencies.

3.4 Final dataset

All games in the dataset match the following criteria:

- Time control = 180+0
- Termination = Normal

Originally, games from January 2013 all the way to August 2019 were downloaded, a total of about 100 million games. However, training and evaluating the models on this large dataset took too long with the hardware available. So, a smaller dataset of **around 5 million games** was sampled from the June to October 2023 period (about 1 million games from each month).

Figure 7 shows the proportion of result types in the dataset. The games are reasonably balanced between those won by the player with the white pieces and those won by the player with the black pieces. However, there is very small proportion of games that resulted in a tie.

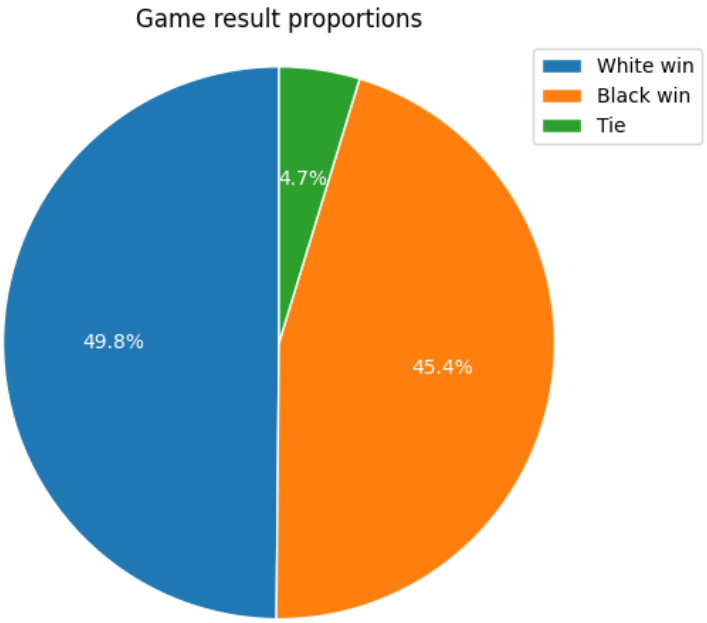

Figure 7: Result proportions.

Figure 8 shows the distributions of the Elo ratings. Most values are concentrated around a relatively narrow range from 1500 to 2000, with a few outliers on either direction.

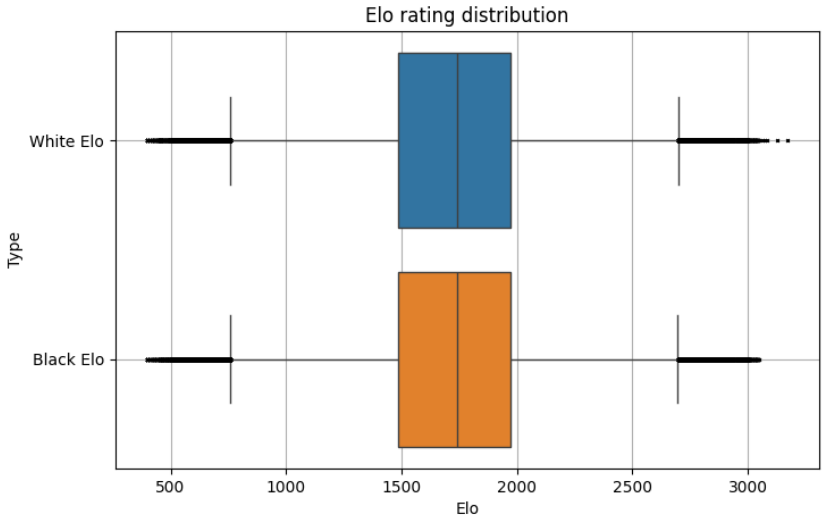

Figure 8: Elo rating distribution.

Figure 9 shows the distribution of game lengths in terms of the number of moves. Most games are relatively small, with less than 60 moves. However, there are some unusually large games, with the longest one being 264 moves long.

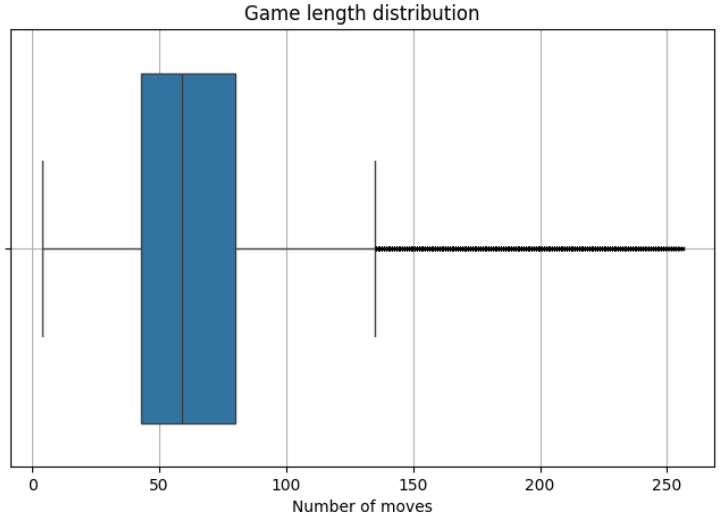

Figure 9: Game length distribution.

3.5 Game tokenization

Tokenization is the process of breaking down the sequence of moves in a game into smaller units. How the game is tokenized is a foundational pre-processing step as it affects how the model represents the data and how fast the model will be. A tokenization schema that results in sequences being broken down into many tokens will cause the model to run slower because processing time grows quadratically with the number of tokens in a sequence. In addition, the more tokens there are, the more memory the model will use. This will reduce the size of the model that is possible or the batch size that can be used in training. Since the model has to learn a representation for each token type, the more token types there are, the more latent representations the model has to learn. On the other hand, having too few tokens can result in poorer representations of the data, loss of contextual information, and poor generalization.

An option to tokenize the sequence of moves is to just have a token for each possible move. However, a back-of-the-envelope calculation of the number of tokens types this would result in shows this is infeasible. Let's say each move is represented by the piece that is being moved, the starting square, and the destination square. With 6 piece types, 64 possible starting squares, and 63 possible ending squares, there could be $6 * 64 * 63 = 24{,}192$ token types. The actual number would be smaller since the pieces cannot be moved to any square, but would still be large.

The tokenization scheme that was chosen was to further break down each move into four components:

1. Piece being moved: Each piece is assigned a lower-case letter to represent it: **p** (pawn), **n** (knight), **b** (bishop), **r** (rook), **q** (queen), **k** (king).

2. Starting square: Represented by the coordinate of the square, as discussed in Section 3.1. A total of 64 options.

3. Destination square: Same options as for the starting square.

4. Promotion type: Whether the piece was promoted to another: **-** (no promotion), **=n** (promotion to knight), **=b** (promotion to bishop), **=r** (promotion to rook), **=q** (promotion to queen).

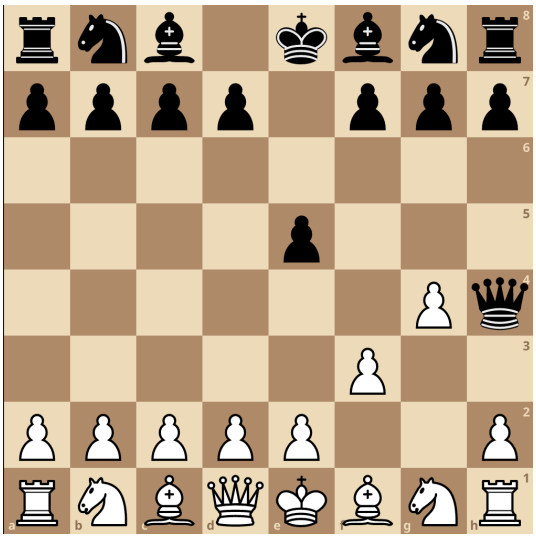

Figure 10: Example tokenization: **p f2 f3 - p e7 e5 - p g2 g4 - q d8 h4 -**

Under this schema, there only 6 (piece types) + 64 (squares) + 5 (promotion types) = **75 total tokens types**. There are no tokens representing captures or checks as this information can be gleaned from the game position and the goal is that the model learns to identify this. The square coordinates have the same embedding no matter whether they are the start or destination square. However, this tokenization means that the length of each input sequence to the model is going to be 4 times the total number of moves in the game.

4 Model architecture

Figure 11 shows the general architecture used for both the result classification and Elo regression models. They share the same encoder architecture described in the original Transformer paper [4] but with the addition of a classification head, in the case of the result classifier, or a regression head, in the case of the Elo regressor.

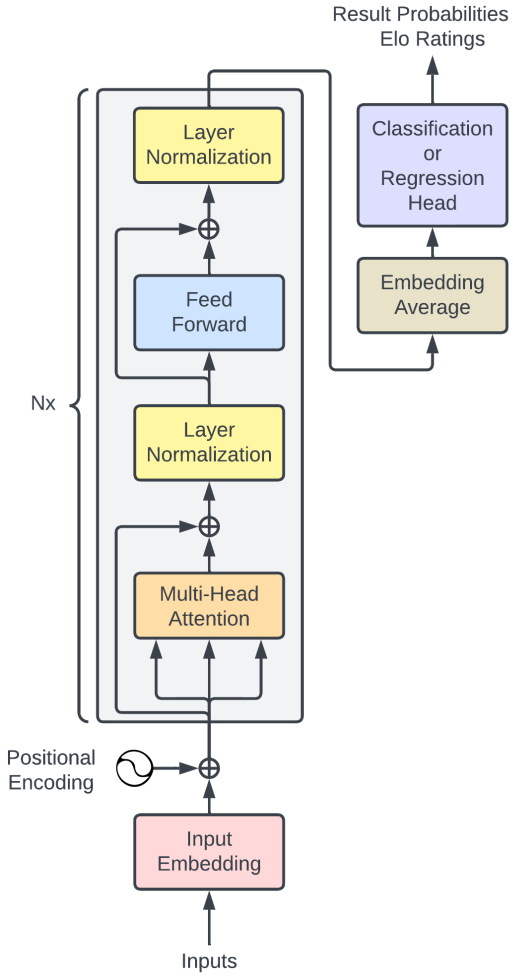

Figure 11: The general model architecture.

Table 1 describes the hyper-parameters used to configure the model.

Parameter	**Description**
d_{model}	Dimension of the token embeddings
N	Number of encoder layers
h	Number of attention heads per encoder layer
$e_{d_{\text{ff}}}$	Dimension of the hidden-layers in the encoder's feed-forward layer
$h_{d_{\text{ff}}}$	Dimension of the hidden-layers in the classification/regression head

Table 1: Model parameters

4.1 Input embedding

The model learns the embeddings to convert the input tokens to vectors of dimension $d_{\text{model}} \in \mathbb{Z}^+$.

4.2 Positional encoding

The positional encoding used is the same **Absolute Position Encoding (APE)** proposed in the original Transformer paper [4].

$$PE(pos, 2i) = \sin\left(\frac{pos}{1000^{2i/d_{\text{model}}}}\right) \tag{1}$$

$$PE(pos, 2i+1) = \sin\left(\frac{pos}{1000^{2i/d_{\text{model}}}}\right) \tag{2}$$

Here, *pos* if the position of the token in the sequence and i is the dimension within the vector embedding of the token at position *pos*.

The only difference from the original implementation in [4] is that instead of interleaving the sines and cosines, the vectors for the sines and cosines are just concatenated. Proposed in [3], this is functionally equivalent to the original scheme but simpler to implement.

4.3 Encoder

The encoder is implemented exactly as described in the original Transformer paper [4].

4.4 Embedding average

The output of the encoder layer is of shape $(\text{seq\_length}, d_{\text{model}})$, where seq_length is the number of tokens in the input sequence. The embeddings for all tokens are averaged element-wise in order to get a single-vector representation of the entire game. The output of this layer is a vector of shape d_{model}.

4.5 Result classifier head

Figure 12 shows the architecture for the result classification head. It takes the vector representation of the entire game and outputs the probabilities of the game being a win for the player with the white pieces, a win for the player with the black pieces, or a tie. The first two linear layers have a dimension of $h_{d_{\text{ff}}}$ and a ReLu activation function. The last linear layer has a hard-coded dimension of 3, as there are three possible results for a chess game, and a softmax activation function to produce the probabilities.

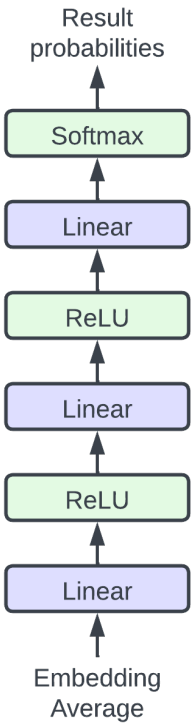

Figure 12: Classification head architecture.

4.6 Elo regressor head

Figure 13 shows the architecture for the Elo regression head. Like the classification head, it has two hidden linear layers with dimension $h_{d_{\text{ff}}}$. The difference is the output linear layer has dimension 2 (for the two Elo scores) and no activation function. In addition, there are layer normalization layers after each ReLU activation layer.

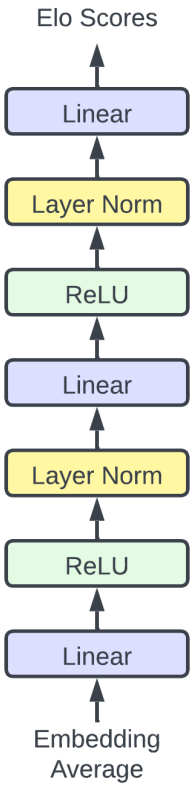

Figure 13: Elo regression head architecture.

5 Model training

5.1 Training data and batching

Both models were trained and validated on the dataset with 4,999,955 games mentioned in Section 3.4. The training set contains 3,990,276 games (80% of the dataset) and the validation set contains 1,009,679 (20% of the dataset). A batch size of 128 and maximum sequence length of 512 tokens were used to train both models. All models were trained for 1 epoch.

5.2 Hardware and schedule

The models were trained on a Google Cloud machine with 1 NVIDIA Tesla T4 GPU. For the hyper-parameters mentioned in Section 6, each training step took on average around 1.5 seconds. The models were trained for 31,174 steps or 13 hours. The validation round took an additional 7888 steps or 1.85 hours.

5.3 Optimizer

The same optimizer described in [4] was used: the Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 10^{-9}$, and a learning rate defined by the formula below.

$$lrate = d_{\text{model}^{-0.5}} * \min(step\_num^{-0.5}, step\_num * warmup\_step^{-1.5}) \quad (3)$$

Figure 14 shows how the learning rate changes during training for $warmup\_steps = 4000$. The linear increase during the warm-up prevents the model from overfitting to the initial examples it sees, while the inverse square root decrease after the warm-up allows the model to settle into the optimum it found.

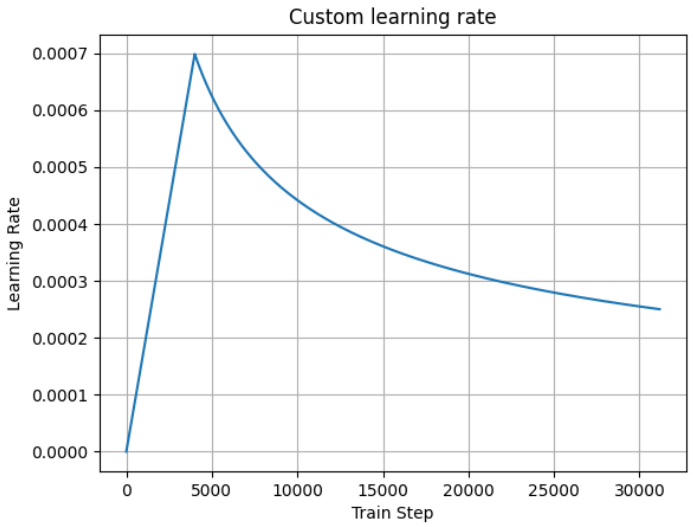

Figure 14: Learning rate during training.

5.4 Regularization

Dropout [2] was applied to the output of each sub-layer in the encoder before it was added to the sub-layer's input and normalized. Dropout was also used in the sums of the embeddings in the attention module. In all instances, a dropout rate of 10% was used.

6 Results

6.1 Result classification

Table 2 shows the setting of the hyper-parameters used for result classification. The total number of parameters is 7,433,027. The model is relatively

small due to hardware limitations.

Parameter	Value
d_{model}	512
N	2
h	3
$e_{d_{\text{ff}}}$	512
$h_{d_{\text{ff}}}$	64
$warmup\_steps$	4000

Table 2: Result classification model parameters.

Figures 15 and 16 show the instantaneous and cumulative batch accuracy for the training and validation rounds, respectively. The instantaneous batch accuracy is computed over a single batch while the cumulative batch accuracy is updated as the training or validation goes on.

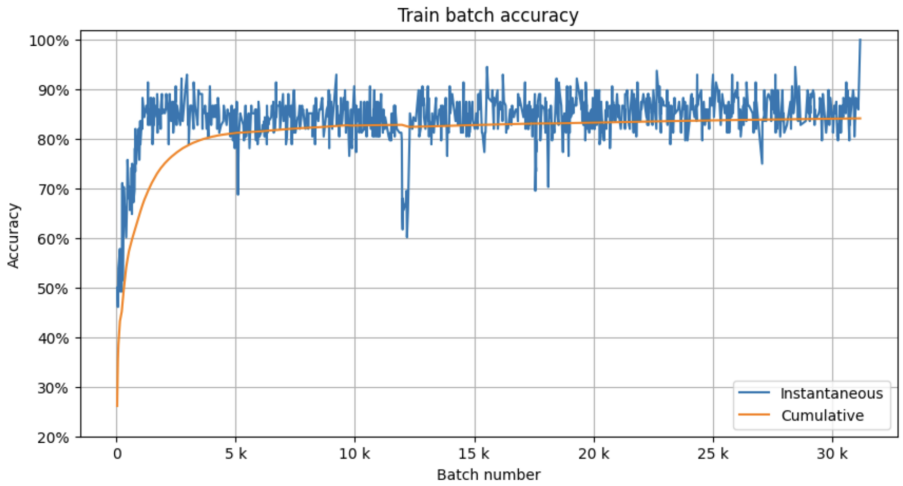

Figure 15: Result classifier training batch accuracy.

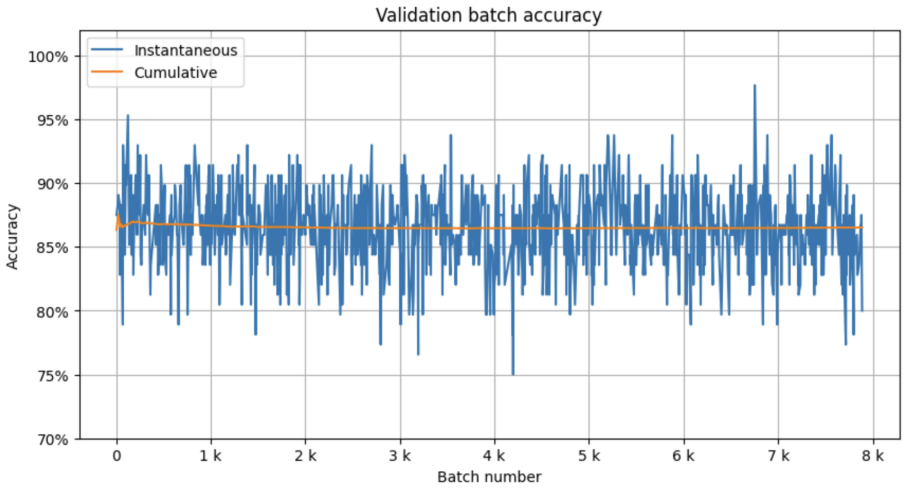

Figure 16: Result classifier validation batch accuracy.

Figures17 and 18 show the instantaneous and cumulative batch cross entropy loss.

Figure 17: Result classifier training batch loss.

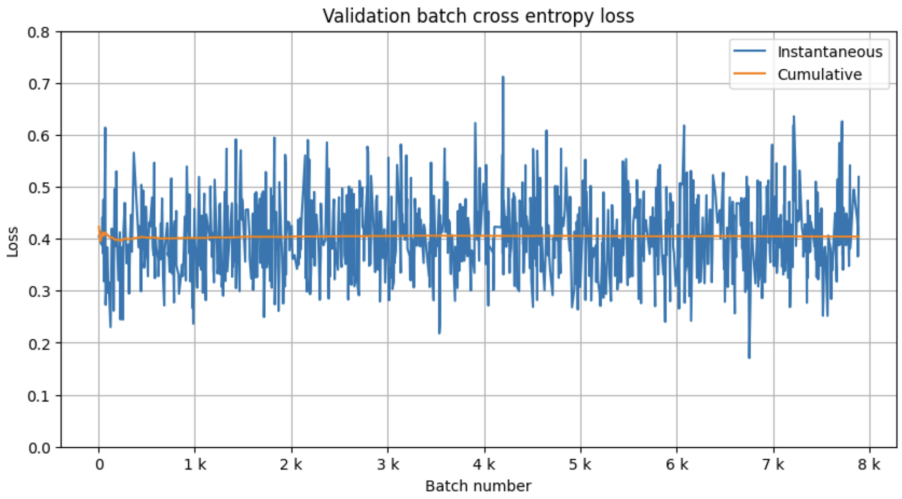

Figure 18: Result classifier validation batch loss.

Table 3 shows the final cumulative metrics for the training and validation rounds.

Metric	**Training**	**Validation**
Accuracy	82.8%	86.2%
Cross-entropy loss	0.437	0.404

Table 3: Result classification cumulative accuracy and loss.

Figure 19 shows the confusion matrix on the validation set.

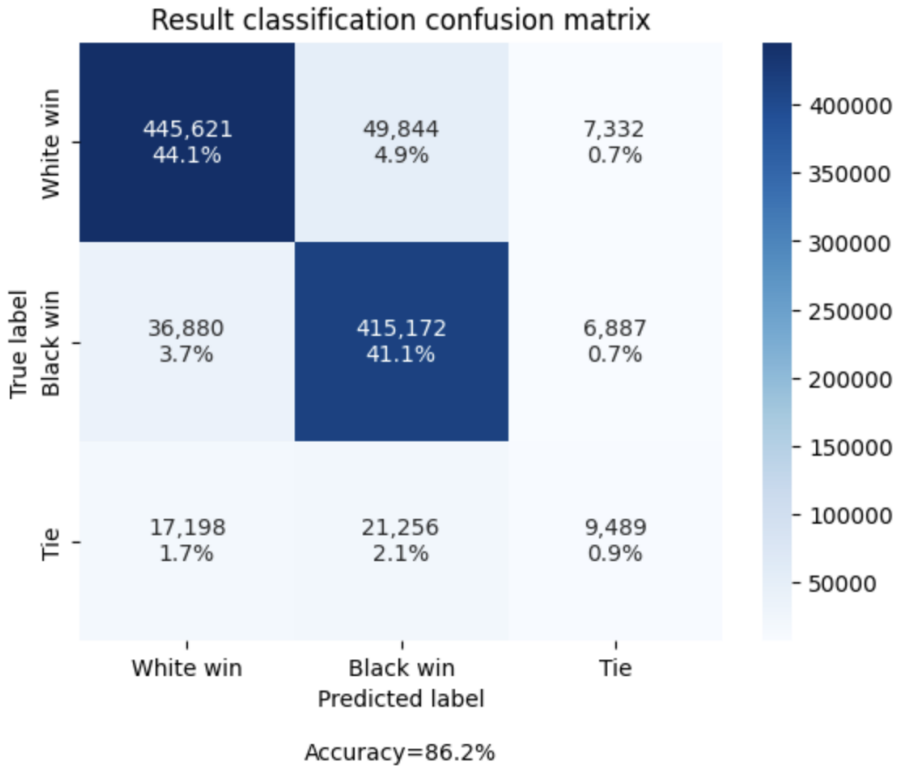

Figure 19: Result classification confusion matrix.

As Figure 7 shows, the dataset is well balanced with respect to games won by players with either piece color. Therefore, a naive classifier that just outputs the most frequent result (White win) would have an accuracy of only 49.8%. The classifier presented here has an accuracy almost twice as large. As Figure 19 shows, the model does very well on the two largest categories, but does not perform as well on the smaller Tie category (4.7% of the dataset). This may be because there are relatively few Tie examples to learn from or it is difficult to predict whether a game resulted in a tie from the moves alone since players can agree to a tie even when the position is not balanced.

Interestingly, the model had better metrics in the validation round. This is likely because of the model's poor performance at the beginning of the training round (since the metrics are cumulative) and the fact that during validation the model is not hampered by the dropout layers and can use all the learned parameters.

6.2 Elo regression

6.2.1 Training from scratch

Table 4 shows the setting of the hyper-parameters used for the Elo regression task. The total number of parameters is 7,433,218.

Parameter	Value
d_{model}	512
N	2
h	3
$e_{d_{\text{ff}}}$	512
$h_{d_{\text{ff}}}$	64
warmup_steps	9000

Table 4: Elo regression model parameters.

Figures20 and 21 show the instantaneous and cumulative batch mean square error loss.

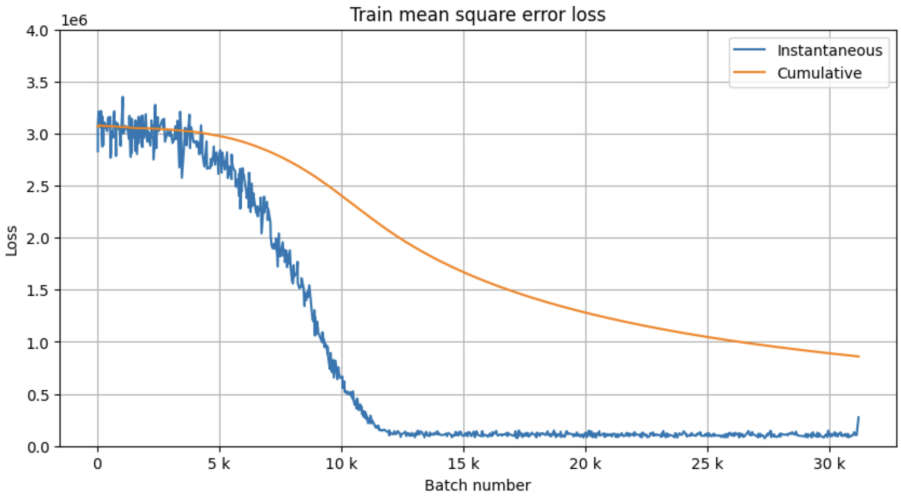

Figure 20: Elo regressor training batch mean square error loss.

Figure 21: Elo regressor validation batch mean square error loss.

Table 5 shows the final cumulative metrics for the training and validation steps.

Metric	**Training**	**Validation**
Mean Square Error (MSE)	860,306	110,718
Root Mean Square Error (RMSE)	928	333

Table 5: Elo regression cumulative loss.

Figures 22 and 23 show the actual vs. the predicted Elo ratings for 2000 examples from the validation set.

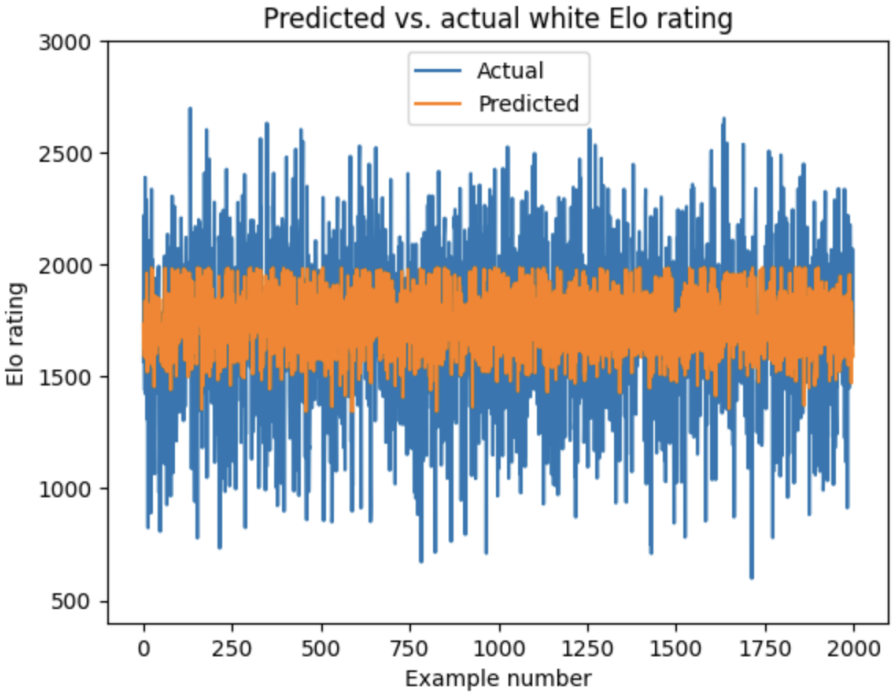

Figure 22: White Elo prediction spot-check.

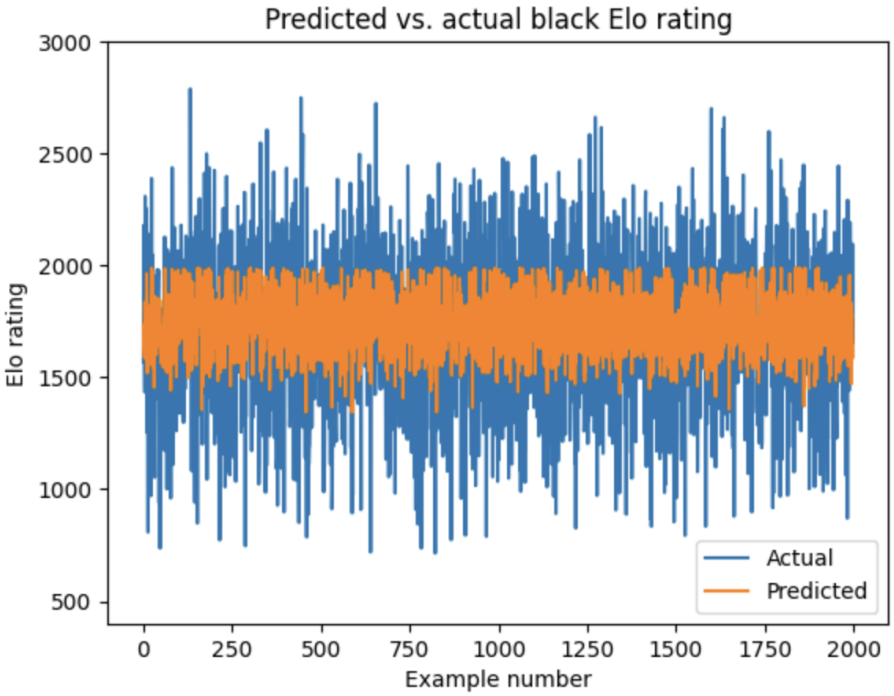

Figure 23: Black Elo prediction spot-check.

Figure 24 shows the difference between the actual Elo ratings of players for 2000 games from the validation set. Figure 25 shows the same information for the corresponding predicted Elo ratings.

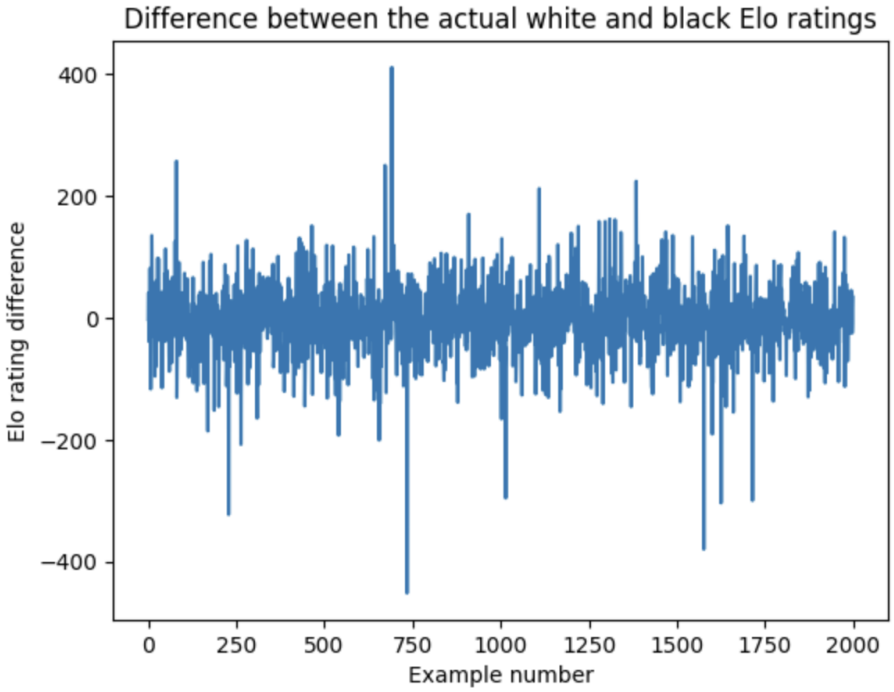

Figure 24: Difference between actual Elo ratings (white - black).

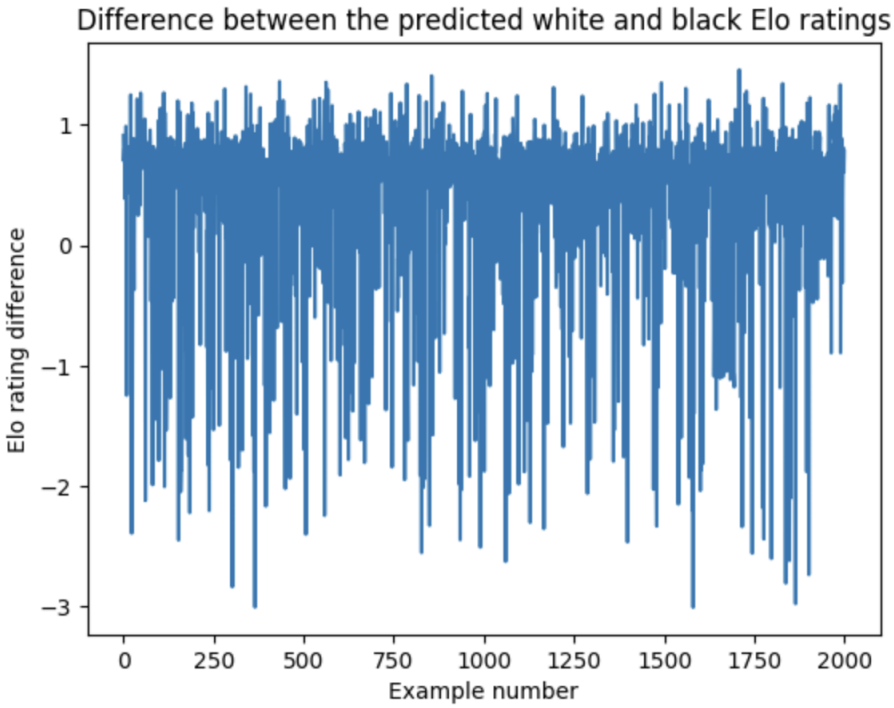

Figure 25: Difference between predicted Elo ratings (white - black).

As Table 5 shows, the RMSE for the validation set is 333 Elo points, which is not high considering the Elo scores range from 400 to around 3000. However, as Figure 8 shows, most Elo ratings are between 1500 and 2000, which is exactly the range of ratings the model predicts most often, as Figures 22 and 23 show. This indicates that the model was not able to capture the full variability of the data, instead predicting values close to the mean. Additionally, Figures 24 and 25 show that the model predicts the ratings of the two players to be much closer than they actually are.

6.2.2 Fine-tuning from the trained result classifier encoder

A model with the encoder initialized with the weights of the trained result classifier from Section 6.1 was also fine-tuned. The hypothesis was that it would converge faster or perform better by leveraging the representation learned by the other model. The parameters are the same as those shown in Table 4.

Figures 26 and 27 show the instantaneous and cumulative batch mean square error loss.

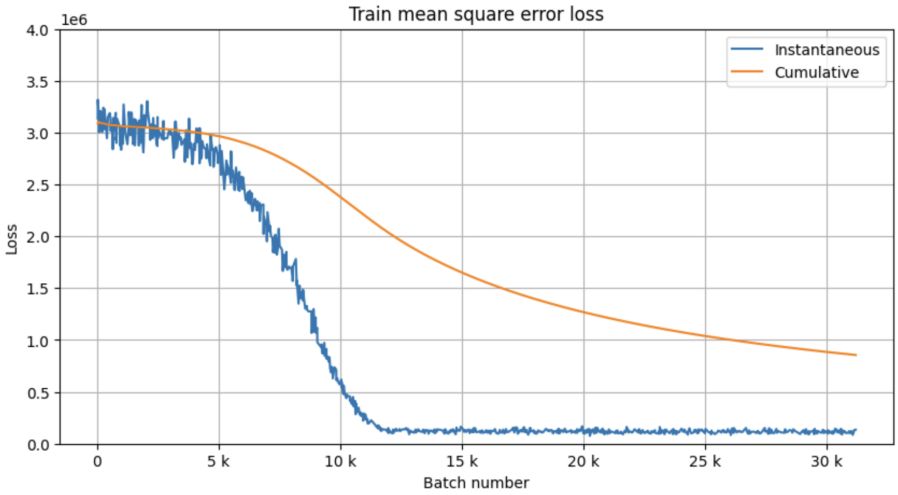

Figure 26: Fine-tuned Elo regressor training batch mean square error loss.

Figure 27: Fine-tuned Elo regressor validation batch mean square error loss.

Table 6 shows the final cumulative metrics for the training and validation rounds.

Metric	**Training**	**Validation**
Mean Square Error (MSE)	855,656	117,232
Root Mean Square Error (RMSE)	925	342

Table 6: Fine-tuned Elo regression cumulative loss.

As Figure 26 shows, the model did not converge faster than that in Figure 20. Table 6 shows the final validation loss is not much different either.

6.2.3 Larger model

A third attempt was made to improve the model performance by increasing the number of trainable parameters. The hypothesis was that a model with more capacity may be able to learn more complex representations.

Table 7 shows the setting of the hyper-parameters used. The batch size had to be reduced to 64 to accommodate the larger model.

Parameter	Value
d_{model}	512
N	5
h	3
$e_{d_{\text{ff}}}$	512
$h_{d_{\text{ff}}}$	256
$warmup\_steps$	9000
$batch\_size$	64

Table 7: Large Elo regression model parameters.

As the Figures and Table below show, the larger model converged much faster but did not perform any better than the smaller models.

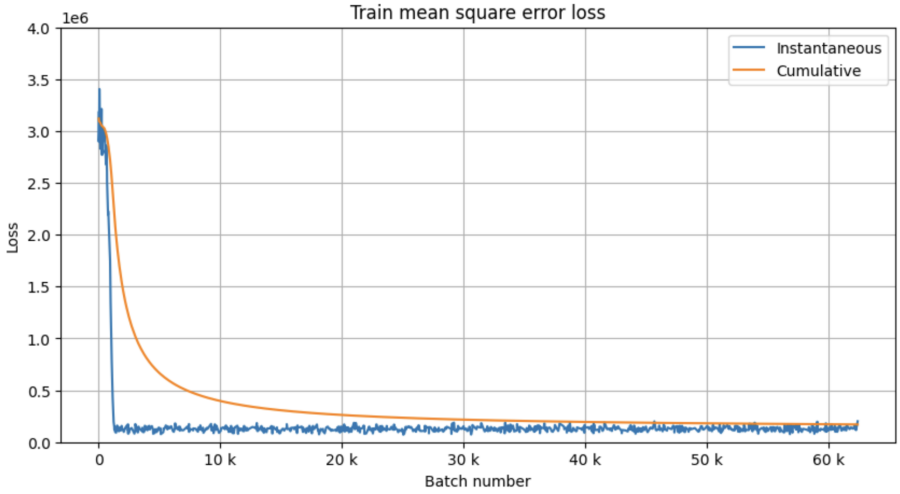

Figure 28: Large Elo regressor training batch mean square error loss.

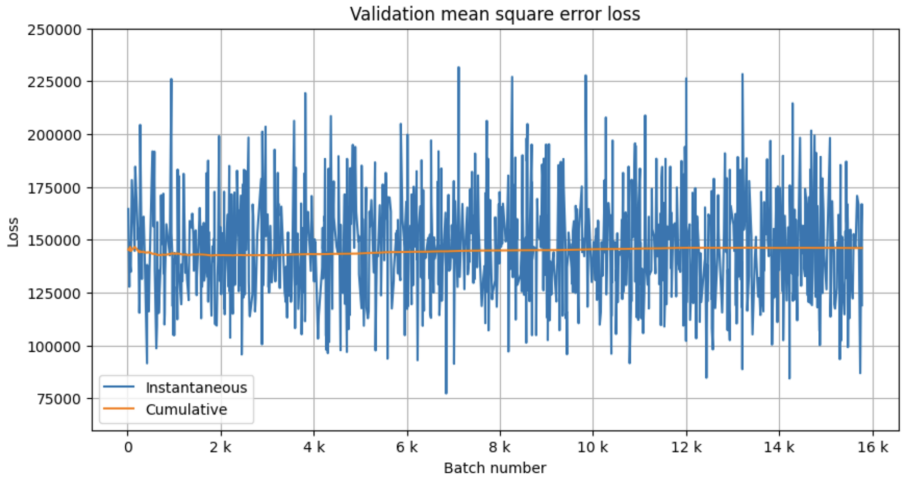

Figure 29: Large Elo regressor validation batch mean square error loss.

Metric	Training	Validation
Mean Square Error (MSE)	170,657	146,104
Root Mean Square Error (RMSE)	413	382

Table 8: Large Elo regression cumulative loss.

7 Conclusion

This work showed the potential of using Transformer-based neural networks to model chess games.

For result classification, the model achieved a respectable 86.2% accuracy, performing even better (89.5% accuracy) when only considering games that did not end in a tie. Given that the model only has information about the moves played in the game, it seems the model was able to learn concepts like checkmate on its own. The model did not perform well on games that ended in a tie, which may be because not enough training examples that have this label were provided or because the moves alone may not provide enough information to decide whether the game was a tie (i.e. players may have agreed to a tie even when the position was not balanced).

For Elo regression, the model learned to predict values closer to the mean but the predicted ratings did not match the actual ratings very closely. This may indicate that the model did not have enough capacity, the training data was insufficient, or the game moves alone do not provide enough signal to predict a player's rating.

An interesting direction to pursue in the future would be to increase the size of the training dataset (only 5 million out of 5 billion games available in

the original database were used) and make the model larger. Another potential change would be to use games whose number of moves is at most the maximum the model can take (in this case, 512 tokens). The dataset used here had a small set of games that exceeded the 512-token limit and these were truncated to 512 tokens.

The code used to download the data, and train and evaluate the models is available at https://github.com/luizvalle/chessformer.

References

[1] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, nov 1997.

[2] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, jan 2014.

[3] TensorFlow. Neural machine translation with a transformer and keras. `https://www.tensorflow.org/text/tutorials/transformer#the_embedding_and_positional_encoding_layer`, 2023. [Online; accessed 08-December-2023].

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[5] Wikipedia. Algebraic notation (chess) — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Algebraic%20notation%20(chess)&oldid=1184027217`, 2023. [Online; accessed 08-December-2023].

[6] Wikipedia. Apache Parquet — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Apache%20Parquet&oldid=1173786069`, 2023. [Online; accessed 08-December-2023].

[7] Wikipedia. Elo rating system — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Elo%20rating%20system&oldid=1185663969`, 2023. [Online; accessed 08-December-2023].

[8] Wikipedia. Lichess — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Lichess&oldid=1187623082`, 2023. [Online; accessed 08-December-2023].

[9] Wikipedia. Portable Game Notation — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Portable%20Game%20Notation&oldid=1178763636`, 2023. [Online; accessed 08-December-2023].

[10] Wikipedia. Producer–consumer problem — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Producer%E2%80%93consumer%20problem&oldid=1166652765, 2023. [Online; accessed 08-December-2023].