Using a neural network to estimate the winner of a chess game based on the opening moves

1 Introduction

Chess is a complex easy-to-learn but hard-to-master game that has been a focus for many mathematicians and chess theorists for centuries. The game can be roughly divided to three stages: opening, middlegame and endgame [1]. Every stage and every move of the game can make the difference, with one blunder in middle- or endgame possibly costing the whole match.

Still, as a hobbyist chess player, it is a intriguing question if the winner of the game can be estimated with some degree of accuracy already after the opening stage. The aim of this project was to build a machine learning model that tries to predict the winner of the game, given the board state after the opening. If proven to work, the model could offer great analysis tools for aspiring players.

Chapter 2 discusses the formal formulation of the machine learning problem. Chapter 3 then expands it by describing the used machine learning methods more thoroughly, and chapter 4 evaluates the created models. Finally, chapter 5 concludes this work.

2 Problem Formulation

Before defining the machine learning model, it is necessary to determine the meaning of "opening stage" in this context. The length of the opening stage depends on the strategy each player has. In this work, the opening phase of the chess game is defined to mean *all moves until the first capture of the game, that move included.* This might not be accurate from the chess theory standpoint. However, in average, when the first capture happens the game has progressed to such state that applying machine learning to it is interesting. Also, while not fully equivalent with the "real" opening stage, the first capture tends to occur when the game is only beginning.

I defined the machine learning problem as a classification problem. Based on the board state, the model tries to classify the game into one of three categories: white wins, black wins, or it ends in a draw. Data point is defined as *one chess match between two players*, from the very first to very last move. The data point contains all information of the game, most notably all moves made and the results.

As features I used the board state after the opening phase. The state can be formulated in multiple ways, but this project defined the game having 64 features – one for each square on the board, with each having one of 13 possible values (2 players * 6 different pieces per player + empty square). As labels I used the data of who won the game. Thus, there are three possible labels: white wins, black wins, or the game ends in a draw.

3 Methods

Data point was defined as one chess match between two player, from the very first to very last move. Such games can freely be downloaded by lichess.org database [2]. The database logs every game played on the site; currently, the site has data of around 1 973 860 947 chess matches to be freely downloaded. For my application, I downloaded games played on March 2017 and January 2019 and pre-processed them to extract the board state after the first capture. In total, the data I used had **21 824 356 data points**.

I tried four different neural network models to tackle the problem. First division was the way how the board state was fed into the network, which affects the network dimensions. First type of network had 64 inputs, one for each square on the board. For the board to be fed into this network, the board state had to be encoded. For this, a numerical value was chosen for each type of piece (and empty square) uniformly between -1 and 1. The output layer had 3 neurons, one for each possible label.

The second type encoded the board differently. One one-hot vector of length 13 was created for each square, and the piece standing on that square was encoded to that vector. Thus, the input had 13*64 = 832 dimensions. The output layer was identical to the first type: three neurons, one for each label.

The second division between the models was the amount of hidden layers in the network. The simpler type used one hidden layer of size 64, while the more complicated type used two hidden layers, both of size 64. All four models used sigmoid activation functions on each layer, except on the input layer, which had not an activation function. The different models are illustrated on table 1. The models were then created in Python with the help of a library called Tensorflow [3].

As a loss function I used categorical cross-entropy loss, which is often used for multi-label classification [4]. The labels (end results of the game) was encoded to network as one-hot vectors: [1,0,0] when white won, [0,0,1] when black won, and [0,1,0] for a draw.

	64 inputs	832 inputs
1 hidden layer	Model 1	Model 2
2 hidden layers	Model 3	Model 4

Table 1: The different models.

I used around 70% of the data into training the model. Rest of the data was divided evenly between validation and test sets, meaning that each of them had around 15% of the data. The division was static: the data was divided roughly evenly into 184 files, and I used first (around) 70% of those files for testing, next around 15% for validating, and the rest for testing. However, the games can be considered to be in random order in the files, and the games in general are uncorrelated; therefore, there was no need to shuffle them.

4 Results

Table 2 contains the results of testing and validation of the four models. From it is easy to see that while the results were close to each other, model 2 with 832 inputs and 1 hidden layer produced the best validation error. Thus, that model can be considered the best out of these four. The result was slightly surprising, considering that there was another, more complicated model among the models. Overfitting is not an probable answer, as model 2 achieved also lower training error than model 4.

	Number of inputs	Number of hidden layers	Train error	Validation error
Model 1	64	1	0.8274	0.8273
Model 2	832	1	0.8213	**0.8204**
Model 3	64	2	0.8270	0.8271
Model 4	832	2	0.8215	0.8211

Table 2: Training and validation errors of the two models.

One thing that might seem strange at first glance is the fact that training error was higher than validation error in almost all cases. This is probably due to the training error being calculated *as* the model is being trained, but the validation error is calculated *after* the validation is done. As such, the average train error is measured 1/2 epochs earlier than the validation error. More can be read from e.g. [5].

The best model was also tested with the not-yet-used test data. The results can be seen in table 3. Furthermore, the confusion matrix produced with the test data can be seen in figure 1. One interesting thing is that the model never estimated the game to end in a draw; this might be due to draws being underrepresented in the online chess world compared to professional games. The result might also suggest that draws are generally difficult to predict based on the early board state. Besides the draws, the model got the winner right most of the time. The accuracy of the model – percentage of times the model estimated the winner correctly – was 52.6%.

	Test error
Model 2	0.8198

Table 3: Test error of the best model.

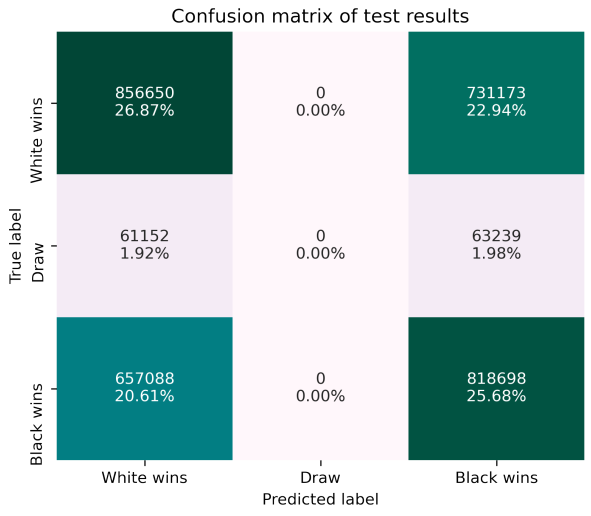

Figure 1: Confusion matrix of the test results. Percentages denote the portion of hits in that field compared to the whole matrix. Plotted with the help of [6].

5 Conclusions

The aim of this project was to train a machine learning model which estimates the winner of a chess match given the board state after the first capture of the game. For that aim, four different neural networks were created and tested.

The accuracy of the chosen model was 52.6%, which, while better than completely random, cannot be described as anything special. One fix would definitely be to design a better neural network and try to ensure that the data presents the population well. However, a more natural explanation also exists: generally, by the first capture, the game simply hasn't progressed so far that one can accurately deduce the winner-to-be by just looking at the board situation. So, while the results are not as accurate as one might hope, it might also mean that chess is such a complex game that one can't expect any certain predictions from such early state.

For future work, the network could be enhanced in various ways. The network could be made deeper, and also the encoding by which the board state is fed to the network should be examined more carefully. Since the board is two-dimensional by nature, a convolutional neural network (CNN) could also be tried.

References

[1] A. E. Soltis, “Chess: Development of theory.” https://www.britannica.com/topic/chess/Development-of-theory. Encyclopedia Britannica. Originally published 1999. Accessed 30.3.2021.

[2] lichess.org, “Lichess chess database.” https://database.lichess.org/. Accessed 28.3.2021.

[3] https://www.tensorflow.org/. Accessed 31.3.2021.

[4] R. Gómez, “Understanding categorical cross-entropy loss, binary cross-entropy loss, softmax loss, logistic loss, focal loss and all those confusing names.” https://gombru.github.io/2018/05/23/cross_entropy_loss/, 2018. Accessed 28.3.2021.

[5] A. Rosebrock, “Why is my validation loss lower than my training loss?.” https://www.pyimagesearch.com/2019/10/14/why-is-my-validation-loss-lower-than-my-training-loss/, 2019. Accessed 31.3.2021.

[6] D. Trimarchi, “cf_matrix.py.” https://github.com/DTrimarchi10/confusion_matrix. Python function to draw confusion matrices. Accessed 28.3.2021.