BoardMind:
A Deep learning Chess Model

Philip Virdo and Matthew Borkowski

3/6/2023

Table of contents

Preface

Chess has long been considered a benchmark for artificial intelligence (AI) research, which challenges the limits of computational power and machine learning algorithms. Researchers strived to create chess engines, such as IBM's famous *Deep Blue* model, that could finally rival human grandmasters and push the boundaries of what is possible with AI.

This research project aims to explore new approaches to the models' understanding of game analysis and move selection while leveraging cutting-edge techniques, such as deep learning. Our goal is to create a chess engine that can play at a relatively high level of comprehension while making human-like decisions and developing its own style of natural play. In essence, the human playing the engine would think it is playing another person as opposed to some machine.

To achieve this goal, we used Python to develop our own engine from scratch that utilizes a convolutional neural network and deep learning training techniques. Alongside this, we used a move analysis helper method, such as, Monte Carlo Tree Search to assist in the engine's evaluation. Our engine is designed to be fast, efficient, and scalable, allowing us to analyze large volumes of game data and generate sophisticated move sequences in real time. All of this is done within a framework that can continually be expanded on in terms of data size and the model training depth.

Abstract

Our machine learning final project aims to design and develop a cutting-edge chess engine, BoardMind, which leverages machine learning techniques to play chess effectively. The primary objectives are to teach the engine chess intricacies and train it for high-level comprehension and skill. The methodology will involve deep learning and decision tree methods.

The project includes data collection, feature engineering, model development, training, evaluation, and iterative improvement. We will gather historical chess games, preprocess the data to extract relevant features, and explore deep learning architectures such as convolutional neural networks, alongside decision tree methods such as the Monte Carlo Tree Search method. The model will be trained using supervised learning techniques and possibly reinforcement learning through self-play.

BoardMind's performance will be evaluated against existing chess engines and human players, using metrics like the ELO rating system, win-loss ratios, and game quality. Evaluation results will inform iterative refinements to improve the model's architecture, hyper parameters, and training data.

Upon completion, BoardMind is expected to be a sophisticated chess engine that demonstrates high comprehension and skill in playing chess. This project will contribute to AI research in strategy games, and provide valuable insights into machine learning algorithms' capabilities.

1 - Introduction to Project

1.1 Overview

The objective of this project is to develop a chess AI in Python that uses deep learning to review large volumes of chess games and make calculated moves in real time against a user. Traditional chess engines rely on complex algorithms that limit their ability to learn and adapt to new situations. By using deep learning, and more specifically, convolutional neural networks, we can create an AI that learns from experience. The use of deep learning will allow our chess AI to adapt to live games against opponents and make more accurate moves.

1.2 Existing System

Many current powerful chess AIs utilize deep learning techniques such as AlphaZero and Leela Chess Zero. This is because deep learning techniques, such as neural networks, are highly effective at recognizing patterns and making decisions based on large sets of data. Commonly, convolutional neural networks are used to identify spatial patterns in images. This ability to recognize patterns makes its an ideal candidate for developing a chess AI that can identify patterns in chess boards. We aimed to use BoardMind as an opportunity to gain a clearer understanding of deep learning, neural networks, and the practical process of developing a machine learning model.

1.3 Objectives of Project

1. Develop a chess AI in Python that uses deep learning techniques

2. Optimize the performance of the AI using various decision models such as Monte Carlo Tree Search or Alpha Beta Pruning

3. Create a framework to allow the model to continually improve by expanding the data sets when required

4. Gain a foundational understanding of convolutional neural networks and how they work in a practical setting, such as chess

2 - Pre-Processing and Exploratory Data Analysis

2.1 Dataset Collection

The dataset for this project was collected from chess databases such as, the lichess.org open database (https://database.lichess.org/), and the pgnmentor database (https://www.pgnmentor.com/files.html). Both sites had data that could be downloaded or easily formatted into a pgn file format, allowing for easy training for the chess AI. The data contains games recorded from a wide spectrum of players, ranging from beginner to professional levels of play. The AI was also trained on pgns that focused on specific chess "opening moves", teaching the AI to maintain control of the chess board and perform specific "castling" formations. As of April, 3rd, 2023, the dataset actively used by BoardMind amounts to a combined total of 253,624 games (approximately 192 MB of data).

2.2 Data Pre-processing

After downloading a large general dataset, we parsed through the data using some helper functions to organize the games into smaller sub-files based on the opening theory used in the games. This allowed us to train the model with a batch of games that use the same opening theory, but have varying middle and late game moves. This gave us the ability to teach the model the correct opening moves and responses that are used in both casual and professional level games. This helped the model gain a foundational understanding of crucial movements of the beginning of a chess match, and the variability to transition seamlessly into the middle game.

2.3 Exploratory Data Analysis and Visualisations

After our first few trial runs of our model, we recognized that the AI rarely made moves that lead to checkmate. This was because the initial training set for the AI was purely professional player games. If a professional player notices that they will be forced into checkmate in *n* number of moves, they typically forfeit the game. This left the initial models with a fundamental lack of understanding for one of the key moves in chess. Following this realization, we had adapted our data processing to purely consist of chess games ending in checkmate and to expand our chess data set to include more than just professional chess games. This would also allow our neural network to expand its knowledge by evaluating casual and intermediate chess player's games, which it would eventually be tested on.

3 - Methodology

3.1 Introduction to Python for Machine Learning

Python is the most commonly used language for machine learning for many reasons. It has simple syntax making it easier to read, it has a variety of powerful libraries to be used for machine learning, and it can be used on a variety of operating systems. These libraries, such as scikit-learn, TensorFlow, and Keras, provide a high-level interface for building and training machine learning models, which saves time and effort compared to implementing algorithms from scratch.

In the case of BoardMind and creating a powerful chess AI, Python gave us access to PyTorch, which is a popular library for using deep learning techniques. PyTorch is one of the default libraries for neural networks because it contains a dynamic computational graph feature. This means that the structure of the neural network used can be changed during runtime, allowing for easier experimentation.

3.2 Platform and Machine Configurations Used

The platforms used to collect the data were the lichess database, and the pgnmentor database, which are both popular collections of pgn files of chess games. The data was then formatted and ran through the code which was built on Python with machine learning libraries included. We used PyTorch to set up and train the convolutional neural network and the Python Chess library to give context to the pieces, actions, and rules of chess to the neural network. Finally, the AI was trained on a desktop running Windows 11 with an AMD Ryzen 5 5600 6-core processor. On this hardware, the average training time with approximately 6 epochs was anywhere between 4 and 5.5 hours.

3.3 Data Split

The training set for the AI were all of the pgn's from the previously mentioned databases. This allowed for the AI to build a general understanding of movements during various stages of a game of chess(early, middle and end game). To validate the model, the AI played itself on both sides of the board and then we determined its accuracy. Using existing chess analysis programs, we analyzed the moves that the AI made for each piece and looked for inaccuracies and patterns that needed to be optimized. From there we would go back to the model and tune the hyperparameters to improve the AI. Finally, the model was tested by playing against human players to see how it matches up against the natural move selection of a human and to see how accurate its responses were to player's of varying degree.

3.4 Model Planning

From the beginning of the assignment, we were set on using a neural network as the foundation of our AI. The planning began when we had to decide which algorithm to implement on top of the neural network to help make decisions during a chess game. The initial algorithms we looked at were:

1. Minimax Algorithms: Which use recursive techniques to systematically explore a decision tree to determine the optimal move for the engine at each turn, operating on the principle of minimizing the maximum possible loss.
2. Alpha-Beta Pruning: An optimization of the Minimax algorithm that reduces the number of nodes to evaluate by using pruning, allowing the engine to make moves faster using less time traversing nodes that will lead to lower value outputs.
3. Monte Carlo Tree Search (MCTS): A search-based algorithm that uses probability to traverse through a decision space of a problem by performing random simulations, then building a search tree, ultimately selecting the best moves based on the best values returned by the algorithm.

Through the trial and error of each algorithm, we ended up choosing the MCTS because it can be engineered to be more efficient than Minimax and Alpha-Beta, especially when using a CNN. Our model makes random moves and sorts each move based on the expected outcome, then iterates through the sorted moves and selects the most promising one. This may seem like an implementation that can be done with any of the models listed above, but MCTS is particularly good at this because it mitigates the Horizon Effect. The Horizon Effect is a common problem with chess models where the algorithm cannot see far enough into the future, leading to a sequence of inaccurate moves or failure to respond to a player's attacking moves. With optimization of our search depth, we were able to get the best results by implementing the MCTS algorithm.

3.5 Model Training:

Due to the large amounts of data used to train the model, our first issue was to find a way to train the model efficiently and effectively without overloading our computer's resources such as avoiding RAM overload or extremely high CPU usage as these can lead to performance issues, system or program crashes, and overheating. Our solution to this was to use a combination of chunk loading and batch sizing to manage the data more efficiently and train the model more accurately. This works by training the model on a batch of games for a given number of epochs, saving the weights from those epochs and then moving onto the next batch until it completes the dataset. We also utilize random shuffling when creating batches to prevent the model from overfitting and increase its adaptability as that is a necessary skill in the game of chess.

3.6 Model Evaluation

The model was evaluated during the validation stage by using publicly accessible chess analysis programs. Chess.com and lichess.org's board analysis programs feature a large number of metrics to analyze each chess move and the board state to determine which side has an advantage. Using the visualizers in the program, we were able to determine which moves the AI made that resulted in weak play and which patterns of moves may be due to bad habits learned during training. In the early stages of training, our model was opening the game by pushing the 'a' or 'h' pawns one or two squares forward. This is an incredibly weak opening move as it does not follow the chess opening theory principles of developing your pieces towards the center of the board. To fix this bad habit, we increased the number of epochs per batch during training, allowing the model to go deeper into its analysis of each game and update its weights more accurately. The results of this training update allowed for the model to develop its own opening style where it tends to lean towards the use of the King's Indian Defence, Sicilian Defense, and French Defence while still being able to play variations based on the user's response.

3.7 Model Optimization

One of the first changes we made was to the architecture of the mode by removing an extra convolutional layer from our neural network. We began with three layers but noticed that during the initial validation sets, the model was overfitted to the training data. Many of the moves the model made during the opening stages of the game had no relation to what its opponent was doing. By removing the third layer, the neural network was less complex and was able to make more accurate decisions to chess moves it had not encountered before.

Hyperparameter tuning also came in many stages, figuring out how many epochs and how big our data chunk sizes should be during training was a learning curve for both of us.

Initially our epoch count was extremely low and our chunk size was large. Our model took far too long to train due to the chunk size and was not able to gain enough context from the board with such a low amount of epochs. Turning down our chunk size and increasing our epoch size allowed our model to focus on a smaller set of games and review it more to better understand the purpose of each move on the board.

4 - Results

4.1 Description of the Models

BoardMind is a convolutional neural network that utilizes Monte Carlo tree searching to evaluate chess board positions, calculate, and output the best move at any given turn. BoardMind's purpose is to help explore the use of neural networks in the realm of chess with regards to position and move evaluation, but also in general how we can apply to neural networks used in decision models that require human-like responses.

4.2 Performance Metrics

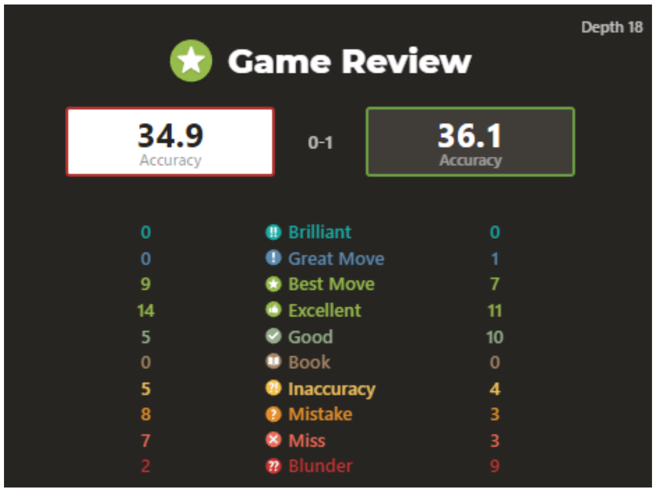
We utilize mean squared error (MSE) as a performance metric to measure the difference between the predicted outputs and the actual target values during the training process. In our training method, for each batch of input data and corresponding target values, the model generates predictions. The MSE loss is then calculated between these predicted outputs and the target values by calling a criterion function to assist with calculation. This calculation computes the average squared difference between the predicted and target values, resulting in a scalar value that represents the model's performance for the current batch. The computed MSE loss is then used to update the model's weights through backpropagation and the optimizer's step function. The training process goes on for the specified number of epochs, with the goal of minimizing the MSE loss to improve the model's performance over time. The running loss is updated and displayed in the progress bar, allowing for the monitoring of the model's training progress.

4.3 Results Tables and Visualization

Below we will show a progression in BoardMind's understanding of chess and its growing rate of play. In figure 1.1 below is the 'pgn' for one of the first self play games BoardMind conducted after training on a small set of data. In figures 1.2 and 1.3, we can see chess.com's and lichess.orgs analysis of the game.

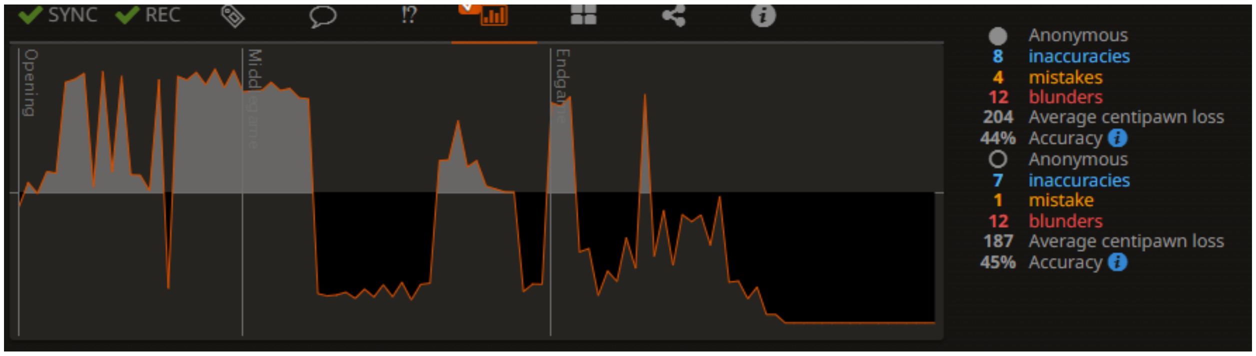
1. Nh3 b5 2. e3 d6 3. Bxb5+ Nc6 4. Bxc6+ Qd7 5. f4 f6 6. Rg1 e5 7. Nc3 Qxc6 8. Rb1 Qxg2 9. Nb5 Qf2+ 10. Kxf2 c6 11. Nxd6+ Kd7 12. Kg3 g5 13. c3 Bxd6 14. d4 exd4 15. exd4 h6 16. Qg4+ Kc7 17. Qe6 Bxe6 18. Bd2 Bxa2 19. Rbf1 h5 20. Rc1 Kb7 21. Rcf1 f5 22. Bc1 Rf8 23. Nxg5 Bxf4+ 24. Bxf4 Rc8 25. Nf3 Ne7 26. Rf2 Rhg8+ 27. Kh3 Rxg1 28. b4 Rd1 29. Rxa2 Rxd4 30. cxd4 Ka8 31. Rxa7+ Kxa7 32. Bc7 Ka6 33. Bf4 Ng8 34. Ng5 Rb8 35. Kg3 Rb6 36. Nf7 Kb5 37. Nd6+ Kxb4 38. Bg5 c5 39. Bf4 cxd4 40. Kh3 Kb3 41. Be3 dxe3 42. Nb7 Rxb7 43. Kh4 Kc2 44. Kh3 Rb8 45. Kg3 e2 46. Kh3 Rb3+ 47. Kg2 e1=Q 48. h4 Qe7 49. Kg1 Qe1+ 50. Kh2 *

Figure 1.1



34.9 Accuracy	0-1	36.1 Accuracy
0	Brilliant	0
0	Great Move	1
9	Best Move	7
14	Excellent	11
5	Good	10
0	Book	0
5	Inaccuracy	4
8	Mistake	3
7	Miss	3
2	Blunder	9

Figure 1.2



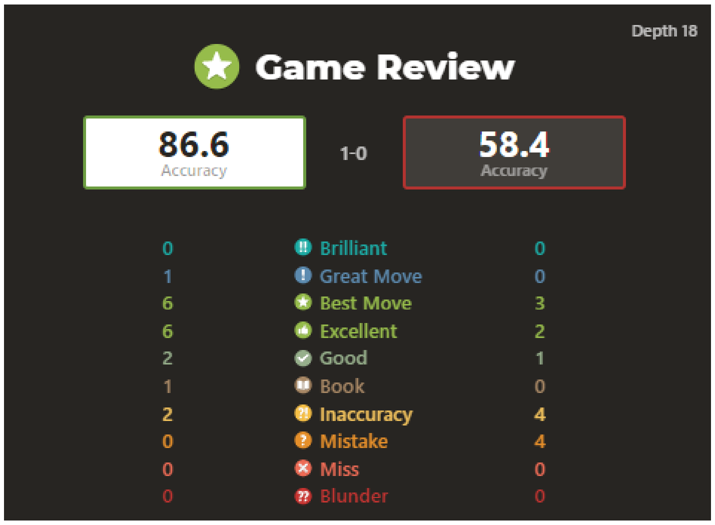
● Anonymous
8 inaccuracies
4 mistakes
12 blunders
204 Average centipawn loss
44% Accuracy
○ Anonymous
7 inaccuracies
1 mistake
12 blunders
187 Average centipawn loss
45% Accuracy

Figure 1.3

In the following diagrams we can see some improvements to the model's decision making after some more training sessions. Figure 2.1 is the pgn and 2.2 and 2.3 are the analysis of a human playing as white and BoardMind playing as black.

```
1. e4 h5 2. d4 a5 3. Nf3 d6 4. d5 Bg4 5. Bb5+ Nd7 6. O-O Bxf3 7. Qxf3 c6 8. dxc6
bxc6 9. Bxc6 a4 10. Bxa8 Qxa8 11. b3 Qxe4 12. Qxe4 axb3 13. Qa8+ Nb8 14. Qxb8+
Kd7 15. axb3 e6 16. Ra7+ Kc6 17. Rd1 h4 18. Rc7# 1-0
```

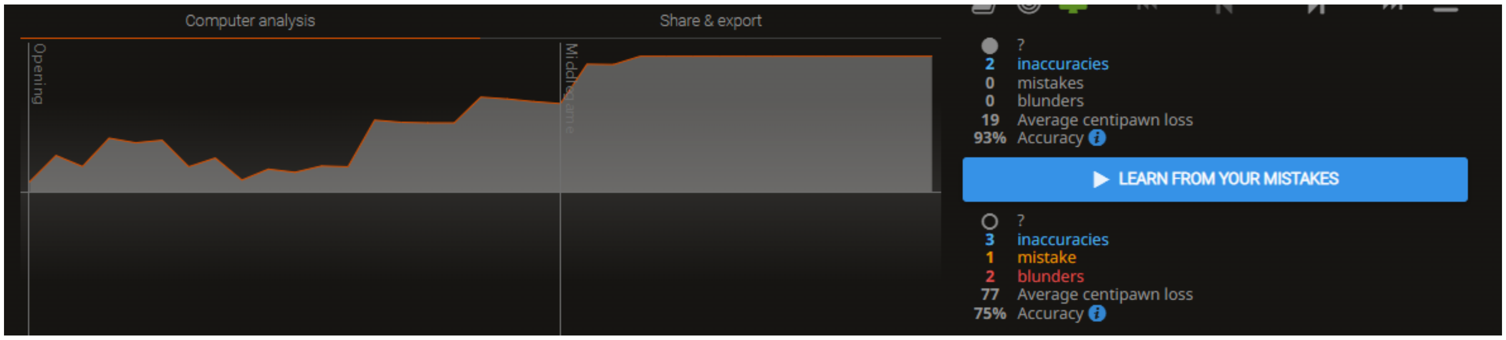
Figure 2.1


Depth 18

Game Review

86.6 Accuracy	1-0	58.4 Accuracy
0	Brilliant	0
1	Great Move	0
6	Best Move	3
6	Excellent	2
2	Good	1
1	Book	0
2	Inaccuracy	4
0	Mistake	4
0	Miss	0
0	Blunder	0

Figure 2.2



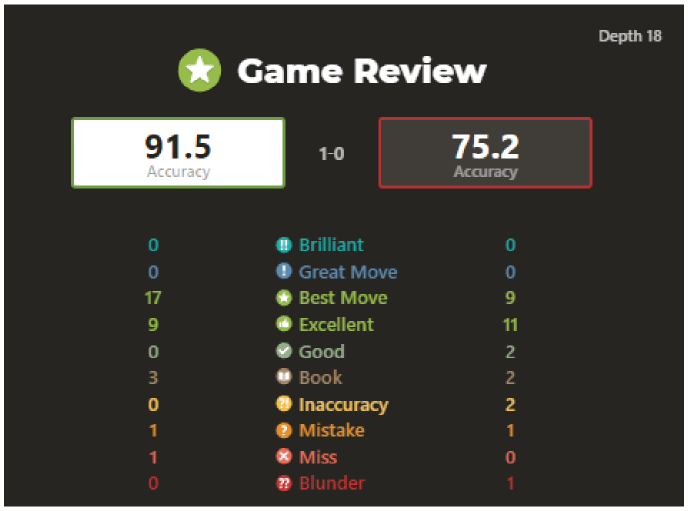
```
●  ?
2   inaccuracies
0   mistakes
0   blunders
19  Average centipawn loss
93% Accuracy

○  ?
3   inaccuracies
1   mistake
2   blunders
77  Average centipawn loss
75% Accuracy
```

Figure 2.3

Finally, below in figures 3.1, 3.2, and 3.3 we can see BoardMind's current level of play prior to submission.

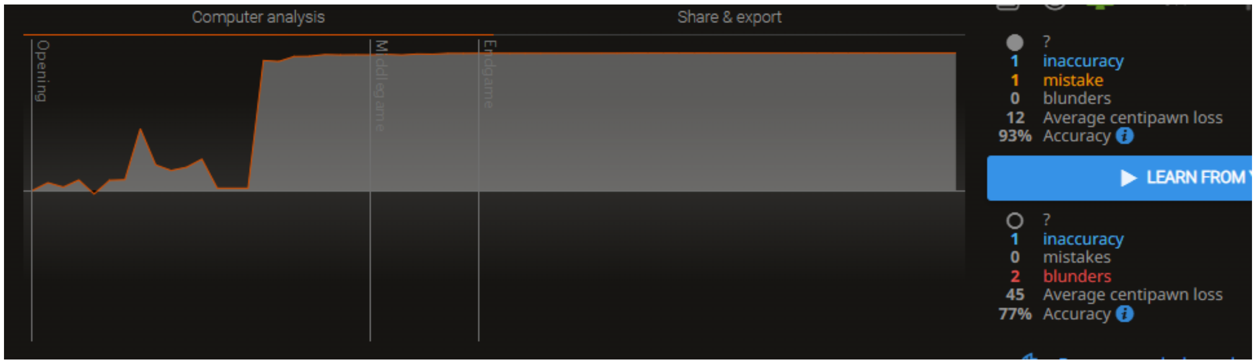
```
1. d4 Nf6 2. Nf3 e6 3. Nc3 h6 $6 4. e4 c5 $2 5. Be3 $9 cxd4 6. Bxd4 Be7 $6 7. Bxf6 $2
Bxf6 8. e5 a5 $4 9. exf6 b5 10. fxg7 Qf6 11. gxh8=Q+ Qxh8 12. Bxb5 Bb7 13. O-O
Bxf3 14. Qxf3 Qxc3 15. Qxc3 Na6 16. Qh8+ Ke7 17. Qxa8 Nc7 18. Qb8 Nxb5 19. Qxb5
d6 20. Qxa5 Kf6 21. Rad1 h5 22. Rxd6 h4 23. Rd7 h3 24. gxh3 e5 25. Re1 Ke6 26.
Qd5+ Kf6 27. Qxe5+ Kg6 28. Rd6+ Kh7 29. Qh5+ Kg7 30. Re7 Kf8 31. Qxf7# 1-0
```

Figure 3.1



91.5 Accuracy	1-0	75.2 Accuracy
0	Brilliant	0
0	Great Move	0
17	Best Move	9
9	Excellent	11
0	Good	2
3	Book	2
0	Inaccuracy	2
1	Mistake	1
1	Miss	0
0	Blunder	1

Figure 3.2



```
●  ?
1   inaccuracy
1   mistake
0   blunders
12  Average centipawn loss
93% Accuracy

○  ?
1   inaccuracy
0   mistakes
2   blunders
45  Average centipawn loss
77% Accuracy
```

Figure 3.3

4.4 Interpretation of the Results

In this early stage of training we can see that BoardMind has an extremely low range of accuracy and it is evident that the model requires more time training and a more intricate dataset. We have also decided to test the interaction of the model based on human inputs so we made changes to the code to allow a user to play as white and have BoardMind play as black.

In this second round of testing we can see that the model is making significant improvements playing at a range of accuracy somewhere between 58% and 75% which is a great improvement from the earlier self games where it was playing at a 34 % and 45%. As we continued to scale the model we included a large dataset of games that ended in checkmate as we noticed the model did not go for moves that would lead to it winning by checkmate, therefore, we needed to adjust those weights as it is a fundamental part of the game.

After concluding these results we can see that BoardMind now plays at a confident range within 70%-80% which is a drastic improvement from its earliest stage and is not at a level that can compete and challenge a human player.

4.5 Sensitivity Analysis

The parameters we tested for sensitivity analysis were the depth search value and the number of simulations value. Both of these are used by the MCTS and evaluation functions to calculate the optimal move by allowing the model to look further into the future of a position. When the search depth was set to a value lower than 4, we noticed that the model was just reacting to singular positions with no follow up plans, resulting in weak positions. On the other hand, when this value was set to above 8, the model began taking too much time to make moves that had little to no improvement as the depth increased. The number of simulations is important as well as its size helps the model by presenting numerous situations. When this number is too low, the move accuracy weakens but if the number was too high it would demand too many resources and take too long to output a value. As a result we settled on a search depth value of 6 and a number of simulations value of 200.

5 - Conclusion

In conclusion, the primary aim of this project was to develop BoardMind using deep learning techniques to analyze a large set of chess games and make calculated moves in real-time against a user. By employing convolutional neural networks, the AI was able to learn from experience and adapt to various game scenarios, a feat that conventional chess engines with their complex algorithms do not necessarily utilize.

The data pre-processing and exploratory data analysis phases led to valuable insights and adjustments to the training dataset, ensuring the AI had a comprehensive understanding of crucial chess positions and understanding of its pieces. We utilized a variety of tools like Python, PyTorch, and the Python Chess library to develop the chess AI on a platform compatible with various operating systems.

Through a series of trials and optimizations, we selected the Monte Carlo Tree Search (MCTS) algorithm, which outperformed Minimax and Alpha-Beta pruning algorithms when combined with a CNN. The MCTS algorithm enabled BoardMind to mitigate the Horizon Effect and make better decisions throughout the game.

Careful consideration was given to the model's training and evaluation, with the use of chunk loading, batch sizing, and hyperparameter tuning to efficiently train the model without overloading computer resources. Model optimization involved adjusting the architecture and training parameters, ultimately improving the AI's performance.

We used the mean squared error (MSE) as a performance metric to measure the difference between predicted outputs and target values during training. The model's performance improved significantly throughout the training process, as evidenced by the increase in its accuracy range.

By combining cutting-edge deep learning techniques, meticulous data pre-processing, and thoughtful model optimization, BoardMind has evolved into a powerful chess AI capable of challenging human players. This project demonstrates the potential of neural networks in decision-making models and serves as a foundation for further exploration and development in the field of chess AI and beyond.

References

1. *lichess.org open database*. (n.d.). https://database.lichess.org/
2. *Download PGN Files*. (n.d.). https://www.pgnmentor.com/files.html
3. *Python-chess* (n.d.). https://python-chess.readthedocs.io/en/latest/#