Chess AI Using Convolutional Neural Networks

Nidai Çağrı SAVRAN
Ali OKTAY

Abstract

The game chess is one of the most ancient board games in the world. Even if it is that old, it is still one of the most royal games because it is seen as an indicator of intelligence among society. The various strategies and infinite possibilities are making chess very complex, interesting and hard to play well. The basic intuition of playing chess well is calculating as many as the best future moves of the current state of the board. That means how many of the next moves you calculate, how much you can see the future is the decider of your power in chess. Therefore, computers are very useful when we are creating a chess AI that calculates best moves in the given state and also very good examples of them like Stockfish, which is commonly in use in chess but it does not use same method like we aim and there is very few neural network based artificial intelligences that introduced in past couple of year like AlphaZero. It is a game engine that understands the game and plays top level games in them. We attempt to create a chess AI that learns human intuition while playing chess. We introduce a convolutional neural network model that aims to find the best move in a current board state and solve checkmates, if any, trained on a data set of one million states of a chess board and their score as centipawn. The data set is found from an open-source platform named LiChess. Our model has already solved desired checkmate problems on the internet and several chess websites. The game power of it is also equivalent to intermediate amateurs. In addition to that, we implement a local web application to play against our model and also offline, it has several functionality like timers, undo move, new game, entering FEN which is a representation of a board as a string. Further training and improvement on the model will likely increase the power level of our chess AI.

I. Introduction

For any discrete game with perfect information, there is a policy that takes the current game state and returns the best move for the player at that moment that will maximize utility under perfect play. While it is possible in theory for neural networks to approximate any function, including this policy, in practice, this is practically not attainable. One example of a perfect information game is chess, an age-old and intensely researched board game that is famous for its complexity and challenge to play at a high level. To explain briefly, chess is a game in which two players take turns making moves on an 8 by 8 board, with white pieces making the first move. The goal of the game is to checkmate the opponent's king, which means restricting its movement in such a way that it cannot escape capture. Each type of piece has different movement types in the board and even if they are very straight forward, mastering the game requires deep understanding and strategic thinking.

Common used, popular commercial chess-playing programs often rely on complex computations of all possible moves of a board and after these computations, it reduces the obvious bad moves then focuses on other moves with deeper searches on them. However, these programs sometimes lack a deep understanding of the game mechanics, resorting to brute force search to make optimal moves. In recent years, various companies have endeavored to develop their own chess-playing programs, such as AlphaZero. These programs typically leverage a combination of convolutional neural networks and Monte Carlo Tree Search to achieve gameplay at or above the level of top human professionals. Nevertheless, these programs examine millions of places per second over several GPUs and processors, outperforming the computational power of their human opponents.

The goal of this project is to create a chess AI which has understanding for a chessboard. While we are using our tree search algorithm among possible moves in the current state of the board, it should give evaluation scores that describe which side is better off by giving values between -1 and 1 to the state of the board by using a convolutional neural network. Our expectation is that solving given mate problems and having the skill level of an intermediate amateur player. Finding a rough estimate of our model's rank can also be achieved by playing our model against other chess computers that have known ranks.

II. Dataset and Features

We collect the data from an open source web platform Lichess. We choose a .pgn file of games played in a month they shared [1]. This file contains a lot of games and each game has structure of moves, evaluation of each move, time spent while playing that moves. Our approach to create a dataset to train our model is creating a matrix representation of our board. The process begins with creating a beginning board with the python chess library and pushing each move in the data file to this board, in each push we convert the board to an 8x8x16 matrix. Our matrix represents the shape of the original chessboard which is 8x8 and each element of this matrix represents the square of the chessboard with a 16 length array which means each square has 16 different features represented. First twelve elements of our array represent piece types. Among these 12 indexes, the nonzero index gives us which piece stands on this square. Besides, the value of the nonzero index is determined with the weight of the piece in the game. The standard weights are as follows, white pawn has weight 1, white bishop and knight has 3, white rook has 5, white queen has 9, white king has 20 and the black equivalent of these pieces has weight their negatives. 13th and 14th indexes stand for whether that square is playable in the next move for each side respectively. If it is playable by the white 13th index becomes 1, if it is playable by the black 14th index becomes 1 and if the conditions do not hold, values of them stay 0. The 15th element represents whose turn in the board, if white’s turn, it is 1 otherwise it is 0. Lastly, the last element of the array stands for the result of the game, there are three possible states: white wins, black wins, draw or game is still going and respectively the value of the index is 1, -1 and 0.

Index	Possible Values	Index	Possible Values
0	0, 1 (White pawn)	8	0, -3 (Black knight)
1	0, 3 (White bishop)	9	0, -5 (Black rook)
2	0, 3 (White knight)	10	0, -9 (Black queen)
3	0, 5 (White rook)	11	0, -20 (Black king)
4	0, 9 (White queen)	12	0, 1 (Playable by white)
5	0, 20 (White king)	13	0, 1 (Playable by black)
6	0, -1 (Black pawn)	14	0, 1 (turn)
7	0, -3 (Black bishop)	15	-1, 0, 1 (result)

Figure 1: Data structure of the chessboard square

We have these features for every square on the board. In total we have 1 million board states with this representation and we decide to take the score of these states, that indicates which side is better off, from the evaluation score in the .pgn file. Normally, given evaluations are between -10000 and 10000 whereas minus values mean black side is in a better position, plus values mean white side is in a better position, and 0 valued score means neither side has advantage over the other side in the current state of the board. Then, we squeeze the scores to between -1 and 1 with dividing the all scores by the absolute value of the max score in the dataset and we use this normalized version in our training and validation phases.

III. CNN Model

Our model consists of an 11-layer convolutional neural network, using simpler architecture for our purpose. The main reason to use a simpler CNN model is the simplicity of our data structure compared to an image. We tried to create data that has a similar structure as an image which has 3 features (red, green and blue) in each pixel. Therefore, we thought that we can use a convolutional neural network to reach our purpose.

Each convolution layer consists of a 3x3 filter with stride 1, we choose the padding as it does not change the shape of the input and we use the Tanh activation function. Then, followed by batch normalization layers, Then, to reduce the dimensionality without losing a significant amount of data we use a max pooling layer with 2x2 filter after batch normalizations. After 3 blocks of this convolution, batch normalization, max pooling layers, the model flatten the data and dense to (1,) shape output. The output is again between -1 and 1, as an evaluation score of the current state of the board. The figure 2 is the plot of our CNN model.

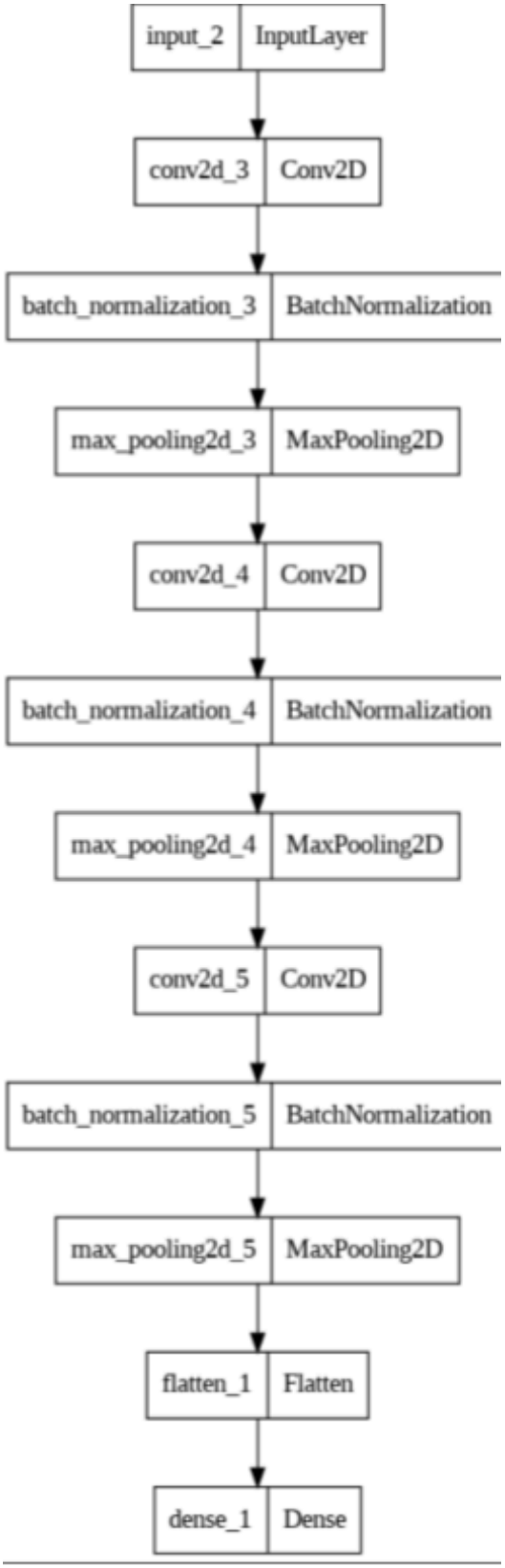

Figure 2: Plot of the model

For training, we use 1 million data as we mentioned in section 2: Dataset and Features, we split the dataset into two, ninety percent of the data for training and ten percent for validation set. The model uses Adam optimizer with learning_rate=1e-3, mean squared error and take as metric a custom accuracy. The reason for using custom accuracy is that it is impossible to get the exact value of the train scores. Therefore, we decided to split our range of output to 100 separate ranges which means if our model predicts in the same range of the train score, we take it as a successful prediction. In addition to these, we decide the batch size as 512 and we use early stopping on validation loss with patience three to prevent overfitting as much as we can. We train our model 25 epochs. Here are the plots of the validation loss versus train loss at figure 3 and validation accuracy versus train accuracy at figure 4. As we can see even though the loss gets relatively small the accuracy stayed at 50% level. Therefore, we need a further improvement that will make predictions of further state possibilities of the game and make calculations about the current evaluation of the board. As a result we decided to implement a minimax algorithm to.enhance the performance of our chessAI.

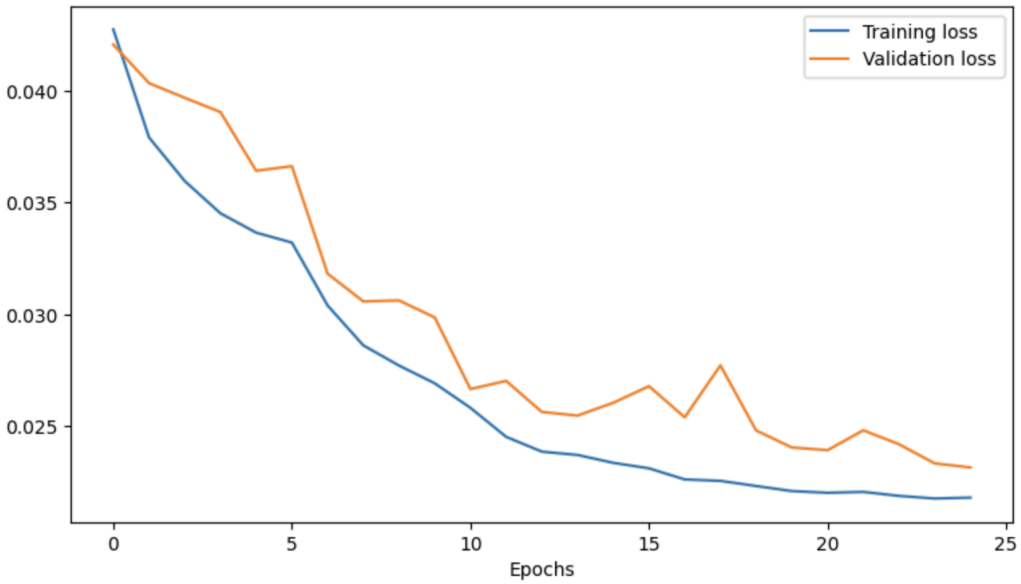

Figure 3: Training loss versus validation loss

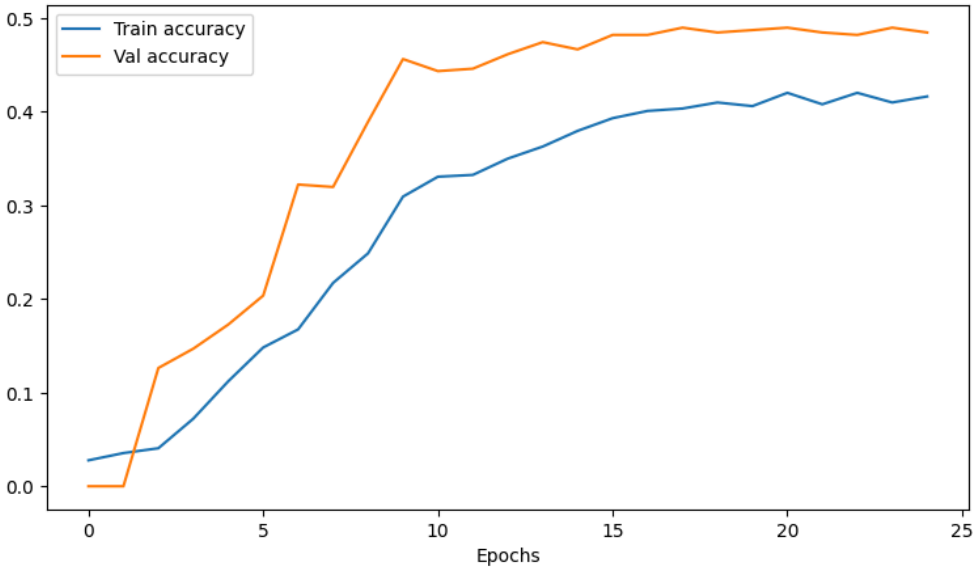

Figure 4: Training accuracy versus validation accuracy

IV. Minimax

Minimax is a commonly used algorithm in two player games. It takes the depth parameter and calculates every possible game state after that depth number of moves. For example if the depth is 5 it calculates all the possible states after 5 moves. We added alpha-beta pruning to enhance its performance.The goal of the minimax algorithm is to minimize the possible loss assuming the opponent playing optimally and the alpha-beta pruning is used to reduce the number of nodes evaluated by filtering out the nodes that have value bigger than given possible max value or smaller than given min value. It also adjusts the value of the possible min and max values as programs continue its iteration.[3] Here is how the algorithm works step by step:

1. Represent the states of the game as trees and each level of the tree represents a player's turn and each node represents a possible game state. The algorithm recursively explores the tree and finds the best move.
2. First provide an evaluation function to the minimax algorithm. In our case the evaluation algorithm is the prediction of our model.
3. Then the algorithm recursively explores the tree alternating between minimizing (black) and maximizing (white) players.
4. The alpha beta pruning sets the minimum score that maximizing player is assured of and maximum score that minimizing player is assured of. If there is a node that is out of that range it filters off that node and its children.
5. After finishing each node it returns the best move.[2]

And because this minimax function takes lots of time we added a cache to our system that stores predictions of the model and when the model makes a new prediction if it is available

in the cache it uses the prediction value in cache. That speeds up the thinking time of the AI significantly.

V. Results

For deciding the level of the model we tested the model using the puzzles of the Lichess. In lichess after solving some puzzles it tells you your puzzle solving rating which shows the strength of the player. 0-1000 shows the starter level, 1000-1500 shows the intermediate level, 1500-2000 shows advanced level and above 2000 shows the professional level. Figure 5 shows the performance of the model in each depth according the lichess puzzles after solving 10 puzzles:

Depth	Rating	Average Time per Puzzle
1	1246	30 seconds
2	1121	1 minute 25 seconds
3	1400	5 minutes
4	1810	1 hour 10 minutes
5	2076	5 hour 35 minutes

Figure 5: Rating of the chessAI according to Lichess puzzle rating

The surprising fact is depth 1 plays better then depth 2. The depths that are bigger than or equal to 5 plays like a professional player as we wanted but it takes hours to play a move.

VI. Conclusion

As a result, our model predicts well if there is an obvious good move in the current state of the board. Hence, it can solve any checkmate problems, if enough time is given. On the other hand, the state of the board has not any obvious good move, it gives a move among those good ones but we cannot say exactly the best move in each case. The model can be improved in various aspects. These aspects are model complexity, amount of data used, and using different techniques like unsupervised learning techniques. Hence, it can learn from its mistakes and improve its predictions.

References

[1] https://database.lichess.org/

[2] J. Rieffel, “AI applications - minimax algorithm,” *Parallel Computing for Beginners*, 2022. doi:10.55682/aopd4880

[3] J. S. Fulda, “Alpha-beta pruning,” *ACM SIGART Bulletin*, no. 94, pp. 26–26, Oct. 1985. doi:10.1145/1056313.1056315