RESEARCH ARTICLE OPEN ACCESS

Predicting Chess Moves with Multilayer Perceptron and Limited Lookahead

Fenil Mehta[1], Hrishikesh Raipure[2], Shubham Shirsat[3], Shashank Bhatnagar[4], Prof. Bailappa Bhovi[5]
[1,2,3,4,5]*(Department of Computer Engineering, International Institute of Information Technology, Pune, India)*
[1]*(Email id: fenilgmehta@gmail.com)*
[2]*(Email id:hrishikesh.raipure64@gmail.com)*
[3]*(Email id:shubhamshirsat00@gmail.com)*
[4]*(Email id: shashank.bhatnagar27@gmail.com)*
[5]*(Email id: bailappab@isquareit.edu.in)*

ABSTRACT
The game of chess has been a testbed for the application of Artificial Intelligence for a long time. It has been a tough task for an engine of any board game to make an optimal move with limited computational power and time constraints. However, there are numerous chess engines that have been developed by combined efforts of the best developers and chess grand masters which were designed by considering various starting books, ending books, game specific techniques and hard coded algorithms to gain strategic advantage over the opponent. Even after all these efforts, humans have defeated chess engines time and again with just a few steps lookahead, while on the other hand the computer is unable to win even after having high computational power and using expensive lookahead algorithms. In this project, we train a neural network to learn a chess board evaluation function which can be used to evaluate the board without deep lookahead search algorithms. From the Shannon number, we can infer that it is not possible to train the model for each possible state. Hence, we need a computational technique which can approximately predict the score of unseen boards based upon training on other board states.

***Keywords* -** Artificial Intelligence, Chess Engines, Lookahead, Neural Network, Shannon Number.

Date of Submission: 20-03-2020 Date of Acceptance: 06-04-2020

I. INTRODUCTION

Despite what most people think, highly rated chess players do not differ from the lower rated ones in their ability to calculate a lot of moves ahead. On the contrary, what makes chess grandmasters so strong is their ability to understand which kind of board situation they are facing very quickly. According to these evaluations, they decide which chess lines to calculate and how many positions ahead they need to check, before committing to an actual move [1]. Considering the size of any board game, we can say that every board game like Chess, Shogi, Go, Checkers, Tic Tac Toe, and many more have at least one optimal for each position which is probabilistic considering the various game parameters. This probability, also called the winning probability of any player, is the deciding factor in choosing the optimal move. As these games have finite board/game states, we can conclude that they are computable provided we have infinite time, space and computational power. It is the time and space complexity of the algorithm to find the winning probability of these game states that define their effectiveness and usefulness in other domains. Considering this, the human brain has been able to solve these puzzles to a large extent though not 100% accurately. A problem mapping strategy which can mimic the human brain and take advantage of the computational power of computers will be a big step towards game strategy solving algorithms. We have used Multilayer Perceptron (MLP) to learn the chess board evaluation function as they have given best results for chess as compared to other architectures and this helps achieve a constant time complexity as well as limited space complexity.

II. PREVIOUS WORK

There has been extensive research in the field of machine learning to develop an architecture that suits well to the rules of a particular or multiple [2] board games, such as Chess, Go, Shogi, Checkers, Backgammon, etc. All of the research suggests using various machine learning techniques like supervised or reinforcement learning depending on the dataset quality available. For example, the CrazyAra engine makes use of supervised learning as only data of lower quality was available [3]. However, implemented using Monte Carlo Tree Search (MCTS), it achieved an accuracy of 60.4% [3]. This paper demonstrates using an architecture that will only need the game states as inputs and their respective centipawn scores. Thus, the base will be supervised learning. Prior work however, has shown that reinforcement learning can overtake supervised learning given an efficient searching technique [2]. The AlphaZero engine uses self-play to improve itself and only requires the games' rules. It works on symmetric as well as asymmetric board games and shows that Alpha-Beta search is not necessarily superior to MCTS [2]. The Giraffe chess engine [4] uses the TDLeaf(λ) algorithm. The drawback of this chess engine is the search speed. This search speed is low because of the low hit rate of the cache. In the Giraffe thesis, the implementation uses a neural network that takes the positions of chess pieces as inputs and for each position, a sequence of numbers is given as output that can work as a signature [4]. The similarity in positions is the reason for humans' high search efficiency. In this, unwanted searching can be avoided if humans can make out the equally efficient moves. This will dramatically reduce the average branching factor of search trees. The Giraffe chess engine shows all the probabilities for a move by a particular chess piece. This significantly reduces the search space. Similar to this, our model also reduces the search space by computing the centipawn score for all possible moves from the current board state, looking only one depth ahead in the search space [4].

The choice of the basic model in the architecture used in this paper is MLP. As proven previously, MLPs have a much better performance as compared to Convolutional Neural Networks, and this is using both the Algebraic and the Bitmap notations [1]. Again, the input data representation can be in the form of Algebraic notation where each board is represented using a string of the positions of its pieces, or the representation can be in Bitmap format, where the entire board is represented as binary string. Algebraic notation gives more information about a board than Bitmap, but this is counter-productive, thus the concise Bitmap format is chosen here [1].

Board games have been known to have a very large game-tree complexity, with chess having one with a lower bound of $10^{120,}$ as given by the Shannon number [5]. Even though it has already been established that MCTS will be a good choice for tree searching [1], we will not be using any search algorithm here. This is due to the below described architecture where the model only generates the possible board states after one move and chooses the most optimal one among those.

III. METHODOLOGY

A. Dataset, centipawn score generation and normalization, and board representation

The proposed model makes use of supervised learning. Hence, there is a need for labeled data. For this purpose, we have used a large dataset containing thousands of games. All games have been taken from the KingBase dataset [6]. Bitmap notation for representing games makes it possible to keep track of piece position and helps to determine whether a piece is present on the board or not [7]. The data requires preprocessing. Preprocessing involves parsing and creating board representations suitable for MLP model [7].

First, each board state is given a centipawn score generated using the Stockfish 10 engine, which is one of the strongest existing chess engines. Each board was evaluated with the following Stockfish configuration: Hash table size=16MB, Depth=20, Max evaluation time=1 second, and all centipawn scores were converted such that they are from the perspective of white side, i.e. positive centipawn score means white side is at an advantage and negative centipawn score means black side is at an advantage, where each centipawn corresponds to 1/100th of a pawn.

There is no exact range of centipawn score. However, after evaluating a subset of the training dataset, we concluded that the centipawn score is generally between -7000 to 7000 or it is a checkmate in some 'n' steps. Hence we decided to set the centipawn range as -10000 to 10000 for our evaluated boards. If the Stockfish engine is not able to do a checkmate in 'n' steps, then it returns a centipawn score, otherwise it returns the number of moves in which it can do a checkmate. Hence, this checkmate move count is converted to centipawn score using the following formula:

$$cp = (max\_score - mate\_steps * min\_dif) * sign \quad (1)$$

where,

cp is the Centipawn score,
max_score = 10000,
min_dif (minimum difference) = 50,
sign = +1, if black is getting checkmated,
-1, if white is getting checkmated,
mate_steps = it's the absolute value of the number of steps in which one of the sides can be checkmated.

After the score generation, the centipawn score is normalized from [-10000, 10000] to [-1, 1].

After analyzing the centipawn score distribution and the importance of each centipawn value in winning and losing of the game, we have normalized the scores in the following manner:

Table 1: Normalized Centipawn scores

Input range	**Output range**
[-10000, -2000]	[-1.0, -0.95]
[-2000, -50]	[-0.95, -0.3]
[-50, 50]	[-0.3, 0.3]
[50, 2000]	[0.3, 0.95]
[2000, 10000]	[0.95, 1.0]

Secondly, the input data is converted into a concise numerical representation i.e. the Bitmap format. Here, we have used 778 bits to represent each board state. The DeepChess engine also used a similar form of 773 bits representation for the input [8]. In Bitmap representation, each of the 64 squares of the chess board are given 12 features (for the 12 types of pieces), and the rest are additional bits added to get more information about advanced chess moves [1]. Each bit can either mark the presence or absence of a piece on that square, or of a situation being present (such as the En-passant move).

Table 2: Calculation for bits in 778 bits Bitmap representation

Feature	**Bits**	**Type**	**Comment**
P1 piece	6*64	bool	order: {KING, QUEEN, BISHOP, KNIGHT, ROOK, PAWN}
P2 piece	6*64	bool	order: {KING, QUEEN, BISHOP, KNIGHT, ROOK, PAWN}
Turn	1	bool	Which player to play next, 1=> White, 0=>Black
Checkmate	1	bool	Indicate whether it is a checkmate
King side castling	2	bool	One bit for each player, 1=> allowed
Queen side castling	2	bool	One bit for each player, 1=> allowed
Check	2	bool	Whether it is a check or not, 1 bit for each player
Queen	2	bool	Whether queen is alive or not, 1 bit for each player queen
Total bits	778		

B. Multilayer Perceptron Architecture

This subsection gives the details of the MLP architecture that has been used to achieve the results presented in the next section. Mean Squared Error (MSE) loss function is used for the regression, and considering the large number of board positions of the chess board, we decided to use six hidden layers where the first two hidden layers have 2048 neurons and the later four layers have 1024 neurons each. In order to prevent overfitting, a Dropout value of 10% is used for each hidden layer. Each layer is connected with a nonlinear activation function: the six hidden layers use Rectified Linear Unit (ReLU) activation function, while the final output layer uses hyperbolic tangent as the activation function. Adam optimizer is used for the stochastic optimization problem with its parameters initialized to: learning rate(η)=0.001, β_1=0.9 and β_2=0.999. The network has been trained with Mini-batches of 16384 samples.

IV. RESULTS

This section presents the results that have been obtained in the MLP architecture discussed previously. All the training data is from white's perspective hence the model tends to play better as white (1st player) as compared to black (2nd player). After training the model for 128 epochs, the resultant MSE was 0.0264.

Following is the game played by the model with itself:

['e2e4', 'd7d5', 'e4d5', 'e7e5', 'b1c3', 'f8a3', 'f1b5', 'c7c6', 'b5c6', 'b8c6', 'd5c6', 'd8d2', 'c1d2', 'a3b2', 'c6b7', 'g8f6', 'b7a8r', 'b2a1', 'a8a7', 'c8h3', 'a7f7', 'h3g2', 'f7g7', 'g2c6', 'g7g3', 'h8g8', 'g3g8', 'e8d7', 'd1a1', 'c6h1', 'a1d1', 'f6g8', 'a2a4', 'g8f6', 'f2f4', 'd7c6', 'd1g4', 'e5f4', 'g4g8', 'f6g8', 'd2f4', 'h1f3', 'g1f3', 'c6b7', 'f3e5', 'g8e7', 'e1f1', 'e7c8', 'a4a5', 'c8b6', 'a5b6', 'b7b6', 'f4g3', 'b6a6', 'f1g2', 'a6b7', 'g2f1', 'b7b8', 'f1g2', 'h7h6', 'c3a4', 'b8b7', 'h2h3', 'b7b8', 'c2c4', 'b8b7', 'g2f2', 'h6h5', 'f2g2', 'h5h4', 'g3h4', 'b7c7', 'h4g3', 'c7d8', 'g2h1', 'd8c8', 'c4c5', 'c8c7', 'c5c6', 'c7d6', 'h1g2', 'd6c7', 'h3h4', 'c7d6', 'h4h5', 'd6e7', 'g3f2', 'e7f6', 'f2b6', 'f6e5', 'b6c7', 'e5f6', 'g2h3', 'f6g5', 'c7e5', 'g5f5', 'e5h8', 'f5e4', 'h5h6', 'e4e3', 'c6c7', 'e3e2', 'c7c8n', 'e2f3', 'h8g7', 'f3e2', 'g7h8', 'e2f3', 'h8g7', 'f3e2'].

Following is the game played by the model with stockfish 10 configuration (Hash size=16MB, Threads=1, analyse time=0.01 seconds):
['e2e4', 'd7d5', 'e4d5', 'd8d5', 'g1f3', 'g8f6', 'd2d4', 'd5e4', 'f1e2', 'c8f5', 'e1g1', 'e4c2', 'b1c3', 'c2d1', 'g2g4', 'd1c2', 'c3d5', 'f5g4', 'd5f6', 'g7f6', 'a2a4', 'c2e2', 'c1e3', 'g4f3', 'f1b1', 'h8g8', 'e3g5', 'g8g5']

When the game starts, the model is able to play just decently. However, considering the sparseness when chess boards are converted to binary form and the humongous possibilities of the different board positions for input to the MLP, exploratory search is necessary as the game proceeds.

In order to evaluate the performance of the MLP, we used the model to predict the chess moves on the Kaufman special dataset of 25 complicated board positions, and the model was unable to predict the expected move for any of the board positions.

V. CONCLUSION AND FUTURE WORK

In this work, we discussed a chess engine, which avoids the use of state space search to find the next optimal move. It only relies on the trained network to select the most favourable move after being trained on millions of games played between players. This work tries to show that for board games like chess, there should be an optimal move for each board state. However, we cannot say that there is no need to perform exploratory search for every request to the model or the engine to play or predict an optimal move.

The system is developed by using only the Portable Game Notation files for the chess game and its board states evaluated scores generated using Stockfish 10. Thus, the model can be ported to other board games as well, requiring only their PGN files and an evaluation method.

As the system is partially aware of good moves and bad moves. Hence, using MCTS and self-play can help enhance and fine tune the board evaluations. The system can also be extended to learn from games played against human players, or against any other game engines.

REFERENCES

[1] M. Sabatelli, F. Bidoia, V. Codreanu, and M. Wiering, Learning to Evaluate Chess Positions with Deep Neural Networks and Limited Lookahead, *Proc. 7th International Conf. on Pattern Recognition Applications and Methods,* Funchal, Madeira-Portugal, 2018.

[2] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play, *Science, vol. 362, no. 6419*, 2018, 1140–1144.

[3] J. Czech, M. Willig, A. Beyer, K. Kersting, and J. Fürnkranz, (Aug. 2019). Learning to Play the Chess Variant CrazyHouse Above World Champion Level with Deep Neural Networks and Human Data. [Online]. Available: https://arxiv.org/abs/1908.06660

[4] M. Lai, *Giraffe: Using Deep Reinforcement Learning to Play Chess*, M.Sc. thesis, Imperial College London, London, 2015.

[5] C. Shannon, Programming a Computer for Playing Chess, *Philosophical Magazine, vol. 41, no. 314,* 1950.

[6] KingBase2019 database, KingBase2019, Oct. 2019. [Online]. Available: https://www.kingbase-chess.net/

[7] M. Sabatelli, *Learning to Play Chess with Minimal Lookahead and Deep Value Neural Networks*, M.Sc. thesis, University of Groningen, Amsterdam, 2017.

[8] O. E. David, N. S. Netanyahu, and L. Wolf, DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess, Artificial Neural Networks and Machine Learning – ICANN 2016, in Lecture Notes in Computer Science, (Barcelona: Springer Science+Business Media, 2016) 88–96.