ChessSL: Supervised Learning Chess engine with Monte Carlo Tree Search

Ching-Yin Ng

Department of Physics, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong, China

(Dated: November 28, 2024)

INTRODUCTION

Since AlphaGo defeated the 9-dan professional Go player Lee Sedol in 2016 and the development of AlphaZero algorithm, a new era of reinforcement learning has begun for superhuman game engines. In this project, we attempt to build a chess engine with monte carlo tree search (MCTS) from scratch. However, since reinforcement learning requires a vast amount of computational resources, we will use supervised learning to train the model instead. The source codes are available at the GitHub repository: https://github.com/alvinng4/ChessSL.

METHODS

Training data

Leela Chess Zero (Lc0) is an open source chess engine that attempts to reproduce and improve AlphaZero's algorithm for chess. In the Top Chess Engine Championship (TCEC) Leagues Season 26, Lc0 has attained the runner-up position, with Stockfish being the champion.

Using a reinforcement learning algorithm, they have produced a large amount of training data in the process. As they made the data publicly available, we chose this as our source of training data. About 3 millions games (Lc0 T80 data in v6 format) were downloaded from their data server, which can be readily extracted and processed into our training data.

Model architecture

For this project, we used a ResNetCBAM model, which consists of a residual network architecture with the addition of Convolutional Block Attention Module (CBAM) [1]. The model is based on the ResNetSE model of Lc0 with modifications.

Residual networks [2] are a type of convolutional neural networks that uses skip connections to mitigate the degradation problem in very deep networks. On top of that, CBAM improves the model performance by introducing attention mechanisms, which allows the model to focus on important features in both spatial and channel-wise dimensions.

In our model, we have used 20 residual blocks with 256 filters. The model architecture is as follows:

```
Input -> Conv2D -> BatchNorm -> ResCBAMBlock x 20 -> Output heads
```

And each ResCBAMBlock has the following structure:

```
Input -> Conv2D -> BatchNorm -> ReLU -> Conv2D -> BatchNorm
-> CBAM -> (Input + Output) -> ReLU
```

Model Input

The model input follows a similar format to Leela Chess Zero's input representation in order to match the training data. It consists of 112 planes of 8×8 boards, which includes the current board, seven previous boards and the board metadata. The boards are encoded into bitboards with one-hot encoding (See Fig. 1), which consists of 12 boards (6 type of pieces $\times 2$). The 13th board is used to record repetition since multiple repetition could leads to a draw. For the metadata, we includes the castling rights, the current player (white or black) and number of half moves for the 50 moves rule. The final plane is used to mark the edge of the input, which is always ones.

Model Output

The model has three output heads: policy, wdl and moves left. Again, it is based on Lc0's output heads with modifications.

Policy : A 1858 dimensional vector that represents the probabilities of all possible moves. Illegal moves will be masked before softmax normalization. Structure:

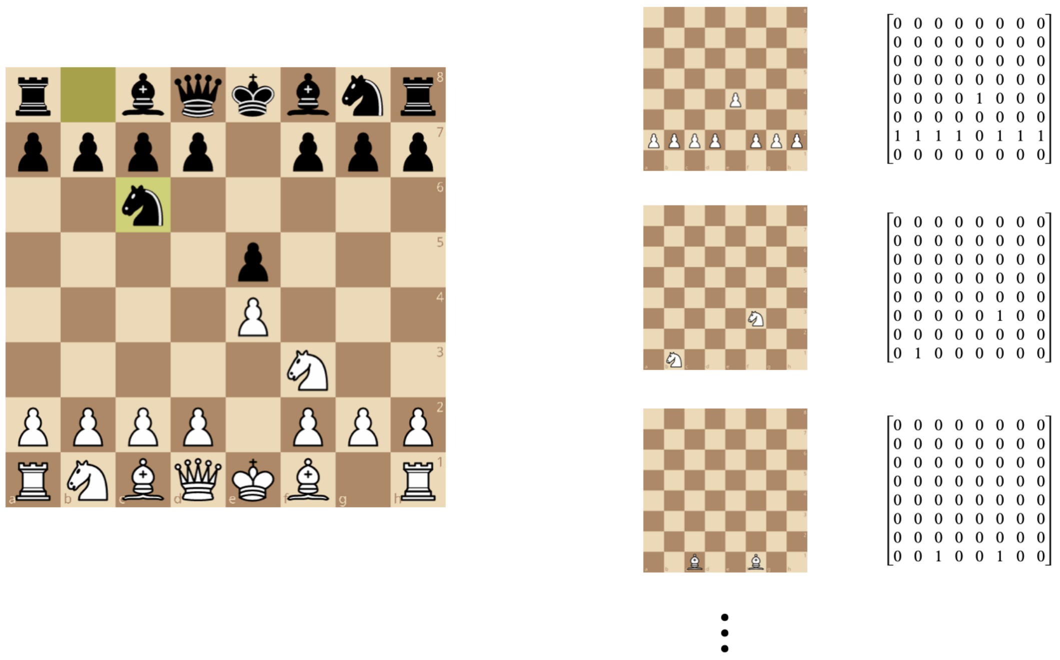
```
Conv2d -> BatchNorm -> ReLU -> Conv2D -> BatchNorm
-> ReLU -> Dense -> masking -> Softmax
```

WDL : A 3 dimensional vector that represents the probability of win, draw and loss. Structure:

```
Conv2D -> BatchNorm -> Conv2D -> ReLU -> Dense -> Softmax
```

Moves left : A scalar that represents the estimated number of moves left in the game. Although we have trained our model with this output head, the output value is not used in this project. Structure:

```
Conv2D -> BatchNorm -> Dense -> ReLU -> Dense -> ReLU
```



$$\begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&0&0&0\\1&1&1&1&0&1&1&1\\0&0&0&0&0&0&0&0\end{bmatrix}$$

$$\begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\end{bmatrix}$$

$$\begin{bmatrix}0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0\\0&0&1&0&0&1&0&0\end{bmatrix}$$

$$\vdots$$

FIG. 1. Encoding the chess board into bitboards

Training Details

The model is trained with the Stochastic Gradient Descent (SGD) optimizer in the PyTorch framework using a momentum of 0.9. When training, we randomly pick one position from each game as the training data. The loss history and learning rate is shown in Fig. 2. The weight of the policy, wdl and moves left loss is 1, 1 and 0.0005 respectively.

It is trained for 40 epochs with a batch size of 512. With each epoch taking around 2500 seconds, the whole training took about 28 hours on a single RTX2070 GPU.

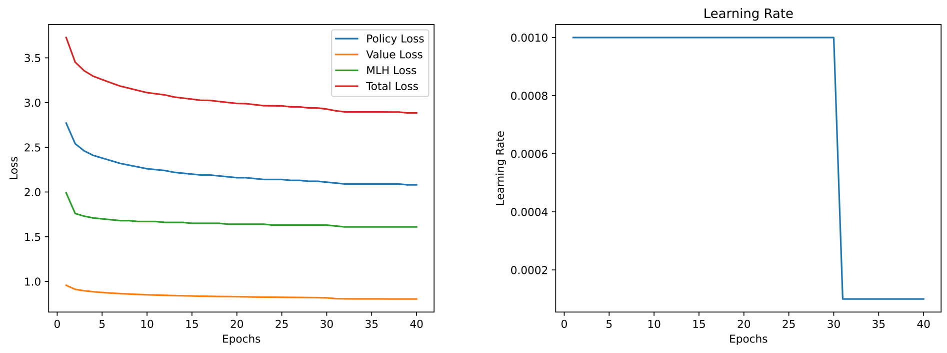


FIG. 2. Training loss and learning rate history

Monte Carlo Tree Search (MCTS)

There are four steps in the MCTS algorithm: selection, expansion, evaluation and back-propagation. For the selection, we use the predictor upper confidence bound applied to trees (PUCT) algorithm to balance the exploration and exploitation:

$$\text{PUCT} = Q(s,a) + c_{\text{puct}} \times \text{prior}(s,a) \times \frac{\sqrt{N(s)}}{1+N(s,a)}, \quad (1)$$

where $Q(s,a)$ is the total averaged value of the node. The node with the highest PUCT value will be selected. To promote computational efficiency, we implemented the Batch MCTS algorithm by T. Cazenave [3], which collects a batch of positions before passing it to the neural networks for evaluation.

Performance consideration

In chess, each player are given a limited time to make a move, and a faster engine are able to search for more nodes and produce a better move. Therefore, to improve the performance of the chess engine, functions that are computational expensive are written in C. It is compiled to a dynamic-link library (DLL) and can be called in Python using the ctypes library. The high-level user interface and neural network evaluation (which uses PyTorch) are done in Python.

RESULTS AND DISCUSSION

In this paper, we have used a MCTS batch size of 32, $c_{\text{puct}} = 1.1$ and maximum number of nodes to be 7168. We have tested our engine in both puzzles and games.

Puzzles

For puzzles, we simply use the checkmate in x puzzles we found online: `https://lichess.org/forum/general-chess-discussion/hundreds-of-mate-in-x-puzzles`. We tested our engine with checkmate in 1 and 2 puzzles. The result is as follows:

TABLE I. caption

Engine	Checkmate in 1	Checkmate in 2
Only network	$\frac{5}{40}$(12.5%)	$\frac{4}{40}$(10%)
Network + MCTS	$\frac{39}{40}$(97.5%)	$\frac{27}{40}$(67.5%)

With only the neural network, the engine has a very low success rate around 10 %. This shows that our neural network is not very strong (In fact, it is extremely weak compared to the Lc0 neural network). However, with the addition of MCTS, the engine has a much higher success rate in both checkmate in 1 and 2 puzzles. Therefore, we can conclude that searching is essential to improve the engine's performance.

Games on Lichess

Lichess (`https://lichess.org/`) is a popular online chess platform that supports bot accounts. Using their API, we have connected our chess engine to the Lichess server and played against other bots. For bullet mode (1 minute time control) and blitz mode (3 minutes time control), we have achieved a rating of 2154 and 2113 respectively as of 23/Nov/2024, which is better than 90.3% and 94.7% of all players.

However, we also observed a weakness of the engine. When the win rate is very high, it has difficulty searching good moves since all moves have similar value (e.g. sacrificing a pawn for no reason still gives 99.9% winrate). In these situations, the final decision is mostly dependent on the policy network output.

Due to the same reason, when there are only a few pieces left on the board with a very high winrate, the engine fails to find the moves to checkmate. As a compromise, we simply switch to a endgame tablebase when there are less than 8 pieces on the board.

Conclusion

In this project, we have built a supervised learning chess engine with MCTS from scratch, using 3 million games from the T80 training data from Leela Chess Zero. Despite the weakness of the policy network, combining it with MCTS has significantly improved the engine's performance. Putting the engine to play on Lichess, we have achieved a rating of 2154 and 2113 in bullet and blitz mode respectively, which is better than 90.3% and 94.7% of all players. This demonstrates the potential of neural networks, as we are able to build a competitive chess engine with only supervised learning on a single RTX2070 GPU.

[1] S. Woo, J. Park, J.-Y. Lee, and I. S. Kweon, Cbam: Convolutional block attention module, in *Proceedings of the European Conference on Computer Vision (ECCV)* (2018).

[2] K. He, X. Zhang, S. Ren, and J. Sun, Deep residual learning for image recognition (2015), arXiv:1512.03385 [cs.CV].

[3] T. Cazenave, Batch monte carlo tree search (2021), arXiv:2104.04278 [cs.AI].