KNIGHTS AND KERNELS: CONVOLUTION IN CHESS AI

Kyle Sung
Undergraduate Student
Department of Mathematics and Statistics
McMaster University
Hamilton, Ontario
sungk5@mcmaster.ca

April 6, 2024

ABSTRACT

We present a novel approach to train a neural network to play chess. Using machine learning techniques from Datasci 3ML3 and those inspired by previous chess endeavours, we aim to develop a model that not only understands the rules of chess but is able to make moves which rival the average player. Unlike the conventional reinforcement learning approach, we implement a convolutional neural network trained on pre-evaluated positions to produce a Jack-of-all-trades model.

Before continuing to read, please consider playing a game against the chess model!

Step 1: Create an account on lichess.org .

Step 2: In the top left corner, click **PLAY**. On the right side, click **PLAY WITH A FRIEND**.

Step 3: Keep variant as **Standard**. Choose a **Real-Time** game with a finite length in minutes and seconds. Choose your starting side.

Step 4: In the section titled **Or invite a Lichess user**, enter `KS_ChessAI`

Step 5: The game should begin. Enjoy!

1 Introduction

Chess, a game with a rich history spanning centuries, has long been a benchmark for artificial intelligence research. The game's complexity and need for strategic foresight makes it an ideal domain for exploring the capabilities of neural networks.

The chessboard consists of 64 squares arranged in an 8x8 grid. Each player controls sixteen pieces, including a king, queen, rooks, knights, bishops, and eight pawns. The primary objective for each player is to checkmate their opponent, which involves using their pieces to trap their opponent's king in a position where it cannot escape capture. Players take turns making moves, attacking and capturing their opponents pieces, often involving tactics and in-depth thinking to plan a winning strategy.

In Section 2, we outline the objective for the project including relevant background from similar works. In Section 3, we investigate the dataset and the preprocessing used. In Section 4, we describe the model architecture and training. Additional information about the process used to develop this project may be found in Appendix A. We provide more information about the code in Appendix B. More information about the rules of chess can be found in Appendix C. Finally, Appendix D outlines why this final project deserves a grade of 100%!

2 Objective

Historically, leading chess engines such as Stockfish have used traditional programming using predefined heuristic hard-coded evaluation functions [2]. In recent years, machine learning has revolutionized AI chess, where algorithms are trained using reinforcement learning to learn the rules and strategy of the game to play the best moves.

The primary objective of this project is to train a convolutional neural network on positions pre-evaluated by the Stockfish engine to predict the evaluation of a given position. Other projects have used similar approaches: Oshri and Khandwala [3] developed a convolutional neural network, aiming to predict a move based on a multiclass classification problem in which a piece is first selected, and then a move is selected. We streamline this process by instead predicting the evalution given by Stockfish, then selecting a move to play by checking each legal move's evaluation.

3 Data

3.1 Dataset

Lichess is an open-source chess platform which allows chess enthusiasts to play each other online. Chess games from Lichess are publicly available via their database.

We make use of a cleaned and preprocessed dataset of Lichess games released by the University of Toronto's Computational Social Science Lab [1]. In particular, information about games including the board, stored in Forsyth-Edwards Notation (FEN), and centipawn score, a measure of the light player's advantage measured in 1/100th pawns, are stored; these two variables act as our primary feature and predictor, respectively.

3.2 Preprocessing

Due to the large size of the files, preprocessing is necessary. First, we utilize multiple bash scripts to split the csv files into smaller subsets.

```
cut -d ',' -f 6,7,15,16,17,18,19,25,29,35,\
41 lichess_db_standard_rated_2019-01.csv \
> JanDataParsed.csv

tail -n +2 JanDataParsed.csv | \
split -l 100000 - DataTrain/Chess_Jan_
for file in DataTrain/Chess_Jan_*
do
    head -n 1 JanDataParsed.csv > tmp_file
    cat $file >> tmp_file
    mv -f tmp_file $file
done
```

We also implement data loaders to extract information from the csv files.

A FEN string is difficult for machine learners to parse. We preprocess further by converting the FEN into a tensor using one-hot-encoding. The first axis represents a given piece (both players). The second axis represents the rank (row) on the board and the third axis represents the file (column) on the board. Since there are six types of pieces and the chess board is a square of length 8, each tensor has dimension (6x8x8).

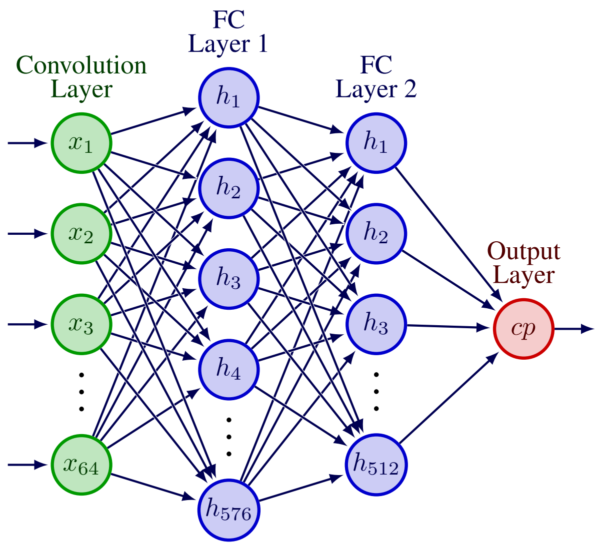
```
def fen_str_to_3d_tensor(fen):
    # See Appendix 3 for implementation
```

We extract additional inputs about the state of the board, including a bool representing whether it is the player with the light pieces' moves and whether the previous move was a capture. In some previous iterations, these were fed into the model in the linear layers to help encourage capture, although later testing revealed that the engine would be too susceptible to making these types of moves, thus we omit them in the final model (see Appendix A).

4 Training

4.1 Architecture

We utilize a convolution neural network with two fully connected layers. The convolution layer has a kernel size of 5 with 6 input channels and 16 output channels. Then, we use two fully connected layers. We use `LeakyReLU` activation to avoid the dying neuron problem.



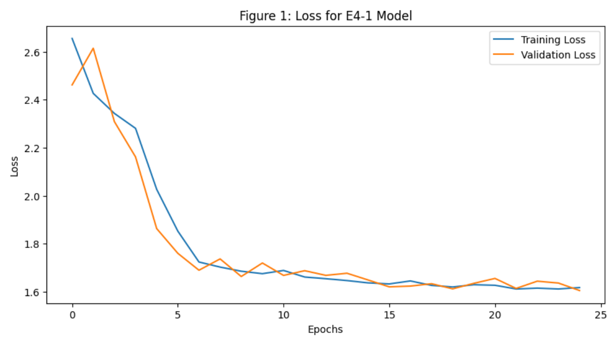

```
class EvalNet(nn.Module):
    def __init__(self):
        super(EvalNet, self).__init__()
        self.conv1 = nn.Conv2d(6, 16,
            kernel_size = 5,
            stride = 1,
            padding = 1,
        )
        self.fc1 = nn.Linear(576, 512)
        self.fc2 = nn.Linear(512, 1)

    def forward(self, x, scalars):
        x = F.leaky_relu(self.conv1(x))
        x = x.view(x.size(0), -1)
        x = torch.cat((x, scalars), dim=1)
        x = F.leaky_relu(self.fc1(x))
        x = F.leaky_relu(self.fc2(x))
        return x
```

To avoid losing intricacies in piece movement and interaction, we opt not to use pooling or dropout layers as suggested by [3].

4.2 Training

We use a fairly standard training loop implemented in `PyTorch`. Due to the large size of dataset we use batch training with a batch size of 25000. We train for 25 epochs using the Adam optimizer with a learning rate of 0.006. We choose to use L1 Loss as it aid interpretation: the loss is the average error in evaluation measured in 100 centipawns. We also make use of cross-validation. Figure 1 presents the loss history for the model.



We train using the Nvidia Cuda Toolkit on an RTX GeForce 3070 GPU.

4.3 Prediction

The engine suggests moves by first parsing a given board state, converting its FEN to a tensor and extracting the necessary predictor variables. For each legal move, it predicts the evaluation for the position after the legal move has been made. Based on the active player's turn, we return the move with the greatest advantage for the active player.

As sometimes players will opt to make a second or third best move for strategical or gameplay reasons, we include a stochastic mode: the engine can choose to play the first through fourth best moves with certain probabilities. This helps keep games against the bot interesting and unrepetitive for players who repeat opening moves, as without this implementation, it would predict the same move for the same position every time.

Move (according to engine)	Chance Played
1st Best	65
2nd Best	20
3rd Best	10
4th Best	5

For implementation, please see Appendix B.

5 Playing and Interface

5.1 Lichess API

We use the `Python-Lichess` API to allow players to directly interact with the board on `lichess.org`. If you haven't tried challenging the bot yet, please consider having a game!

6 Reproducability

The source code for the entire project can be found at my Datasci 3ML3 Final Project Github Repository. Please navigate to the repository and follow the instructions in the README.

A Decision Process

Note: in the Appendix I switch to a first-person description as we move to a slightly less formal setting.

This project took many twists and turns. Here, are some of the decisions I made at each point in the project and how I used techniques and strategies from the class.

In my free time, I love playing chess (as many do!). I stumbled across Lichess's open database at database.lichess.org and wondered if making a chess bot would be a feasible challenge for a final project.

The dataset directly available on Lichess is massive and includes games without direct evaluation by Stockfish. In addition, each game was stored as a string of moves which may be difficult to preprocess and feed into a model. Trimming this dataset down to one which would be easy to train a model would be a difficult endeavour.

The University of Toronto's modified Lichess dataset presented a solution. Each csv is given such that each row represents the state of a game at a given move. Thus, a game may make up dozens of rows in the csv. This organization can help us train a model that is able to predict a move at a given point.

It was difficult to determine the structure for the model. Initially, I thought of a structure in which the model could take in the input position, given in a single two-dimensional tensor, and return 2 two-dimensional tensors representing the position of a starting square and a position of an ending square. To try to ensure a given prediction is legal, a huge penalty is added during training for any move made illegally. My hope was that the convolutional layer would naturally learn the rules of the game of chess including the ways in which pieces would move.

Since we are taking in an input of a two-dimensional board, not unlike image recognition problems we have done previously in lecture and homeworks, I decided to use a similar approach with a convolutional neural network.

After a handful of implementations of this technique including training for many epochs with various parameters, it quickly became clear that this strategy would be ineffective. The loss did not decrease during training and the model would predict illegal moves. The model tried to interpolate between previous moves it had seen, attempting to make moves such as making knights move diagonally like bishops. To attempt to fix this we add small changes: additional parameters were added to the fully connected layers telling the model whether or not it was in check and whether or not the previous move was a capture. I even added functions which would modify the predicted output tensors by adding small amounts of normally-distributed noise to illegal moves until they became legal; it became apparent that the moves it would predict were entirely comprised on random noise.

It was clear that it would be difficult for a neural network of this scale to be able to learn the rules of a game so complicated. As such, I decided to modify the prediction process. The following goal was paramount: the end result should be playable, regardless of the skill level attained. This would mean I would need to find infrastructure which would make it possible for a neural network to reliably make legal moves. We modify the model to instead predict the position evaluation not unlike conventional hard-coded chess engines, however, using machine learning on the previously-evaluated positions. Being able to predict the position evaluation would then allow it to brute-force all the legal moves and play the move which would yield the largest evaluation. Note that one limitation of this is that the engine is unable to make moves that may result in slight evaluation losses but lead to a stronger future position.

Initial testing of some models trained with this new paradigm show positive results. The engine is able to play moves! We notice though that in the initial models, the engine tends to sacrifice valuable pieces to capture smaller pieces or place the opponent king in check. We suspect that this may be because typically positions where an opponent's piece is captured or where the opponent's king is in check would be favourable. This causes the engine to learn this pattern and strive to capture pieces and put the opponent in check more than they should. This leads us to abandon the addition of scalar inputs in the fifth iteration of the model. In addition, we abandon the input of the current move as this is already baked in to the legal move selection with this infrastructure.

B Code

To reproduce the results in the code, please navigate to the GitHub repository and follow the instructions in the README.

C Rules

D Why This Project Deserves 100%!

According to the final project rubric, the project is graded on the following criterion: Quality and Complexity of Project (25%), Creativity and Innovation (25%), Quality of Coding (20%) and Quality of Writeup (30%). Here are some reasons I believe this project satisfies each of these!

D.1 Quality and Complexity

Our neural network aims to tackle a intricate challenge actively researched in machine learning. Throughout the project, we used complex techniques in deep learning learned both in and out of the classroom. We made use of convolutional layers and needed a thorough understanding of kernels, padding, stride, pooling and dropouts (or in the case of the latter two, when not to use them).

An intricate mesh of data-proprocessing, both beforehand (for training) and real-time (for prediction), deep learning (to train the models), API skills (to ship the model in a Lichess playable format) and knowledge of chess were intertwined to make this project successful!

D.2 Creativity and Innovation

Machine learning is a wonderful tool, but many times it ships in a format which cannot be directly interacted with. This project aims to add a twist to the models developed in class by producing a result which can be evaluated by more than just a loss function - it can be played!

We also create an interdisciplinary model which innovates upon actively researched models: we fuse convolutional deep layers, pre-evaluated positions by one of the world's leading chess engines, and machine learning techniques from class to create an ideal Frankenstein's monster in chess.

D.3 Quality of Coding

The code in this project speaks for itself. Using Python code with a number of libraries (including those for gameplay and Lichess implementation), Bash scripts to parse and preprocess CSV files, strong programming practices for speed like vectorization and batch training, and good documentation, are evidence of a strong quality of coding.

D.4 Quality of Writeup

I hope this writeup speaks for itself! Thoroughness in motivation and objective, arguments for implementation choices, and information about the model (plus, the great LaTeX figures!) show the quality of the writeup.

E Limitations and Future Work

References

[1] Computational Social Science Lab. "Lichess Monthly Chess CSV Games". URL: https://csslab.cs.toronto.edu/datasets/#monthly_chess_csv.

[2] Shiva Maharaj, Nick Polson, and Alex Turk. "Chess AI: Competing Paradigms for Machine Intelligence". In: *Entropy* 24.4 (Apr. 2022), p. 550. ISSN: 1099-4300. DOI: 10.3390/e24040550. URL: http://dx.doi.org/10.3390/e24040550.

[3] Barak Oshri and Nishith Khandwala. *Predicting Moves in Chess using Convolutional Neural Networks*. 2015.