Knights and Kernels: Convolution in Chess AI

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

April 24, 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: a king, queen, two rooks, knights, bishops, and eight pawns. Each player's objective 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. We aim to use machine learning to learn and play such 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 preprocessing. In Section 4, we describe the model architecture and training. We discuss augmentation of the model using classical machine learning in Section 5 and discuss the interface of the model in Section 6. Finally, we discuss how to reproduce the project in Section 7

The story of the development of this project is Appendix A. More information about the code is in Appendix B. Future work can be found in Appendix C.

2 Objective

Historically, leading chess engines such as Stockfish have used traditional programming using predefined heuristic hard-coded evaluation functions [3]. 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.

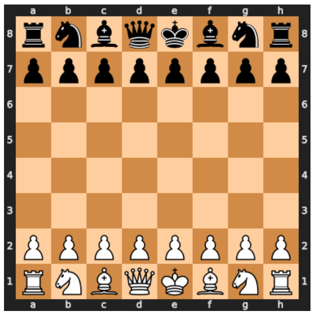
The primary objective of this project is to innovate and improve upon the current chess AI standards: instead of using reinforcement learning, we train a convolution neural network (CNN) on pre-evaluated positions by the Stockfish engine. Though CNNs have been used before, such as Oshri and Khandwala's [4] project, often these CNNs aim to predict a move based on a multiclass classification problem in which a piece is first selected, and then a move is selected. We transform this paradigm from a classification problem to a regression problem: our new goal is to instead predict the evaluation given by Stockfish. Then, the bot selects a move by checking the evaluation of each legal moves and playing the one which yields the best result.

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 partially preprocessed dataset of Lichess games released by the University of Toronto's Computational Social Science Lab [2]. In particular, information about games including a compressed string storing the current board state and all its pieces in Forsyth-Edwards Notation (FEN), which is shown below.



```
# The FEN for the opening position above
'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR'
```

The centipawn score (CP), a measure of the light player's advantage measured in 1/100th of a pawn, is stored also; the FEN and CP act as our primary feature and predictor, respectively.

3.2 Preprocessing

Due to the large size of the files, further preprocessing is necessary. Since loading csv files of this size into Python can take many minutes, we first utilize 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 the given piece: 1 for pawn, 2 for knight, etc., and both players are stored on the board with positive values for the light pieces and negative values for the dark peices. The second and third axes represent the rank (row) and file (column). Since there are six types of pieces and chess board is an 8x8 square, each tensor has dimension (6x8x8).

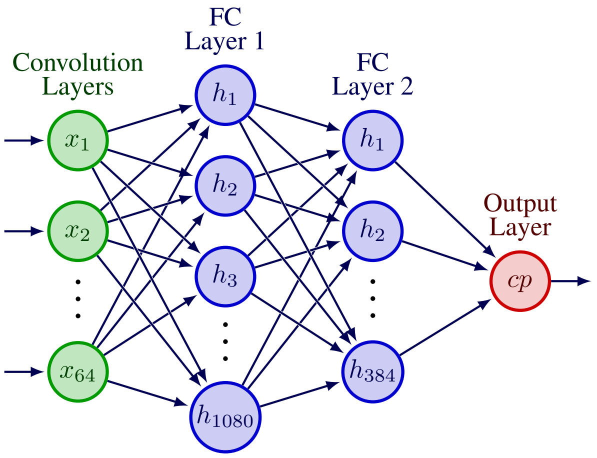
```
def fen_str_to_3d_tensor(fen):
    # See Appendix 3 for implementation
```

Additional inputs about the state of the board are extracting, including the number of each piece alive, current active player and whether the previous move was a capture/check. These were fed in the linear layers to encourage captures/checks, although play-testing revealed that the engine was too susceptible to making captures, thus we omit these in the final model (see Appendix A). The final model still uses the number of pieces.

4 Training

4.1 Architecture

We utilize a convolution neural network with two fully connected layers. The first convolution layer has a kernel size of 5 with 6 input and 16 output channels and the second convolution layer has a kernel size of 3 with 16 input and 30 output channels. The fully connected layers have dimensions (1080, 384) and (384, 1).



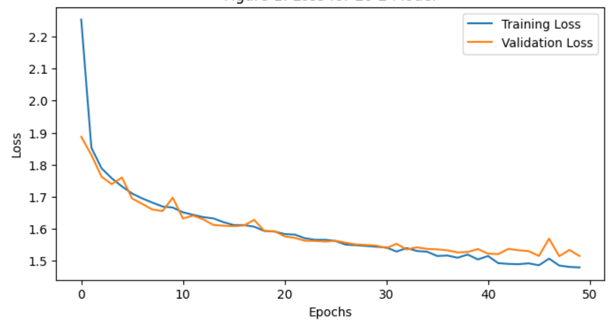
```
class EvalNet(nn.Module):
    def __init__(self):
        super(EvalNet, self).__init__()
        self.conv1 = nn.Conv2d(6, 16,
            kernel_size = 5, padding = 1)
        self.conv2 = nn.Conv2d(16, 30,
            kernel_size = 3, padding = 1)
        self.fc1 = nn.Linear(30 * 6 * 6, 384)
        self.fc2 = nn.Linear(384, 1)

    def forward(self, x):
        x = F.leaky_relu(self.conv1(x))
        x = F.leaky_relu(self.conv2(x))
        x = x.view(x.size(0), -1)
        x = F.leaky_relu(self.fc1(x))
        return self.fc2(x)
```

To avoid losing intricacies in piece movement and interaction, we opt not to use pooling or dropout layers as suggested by [4]. Subsampling may weaken the model's ability to learn. Similarly, we use `LeakyReLU` activation to avoid the dying neuron problem as the data is sparse and we must avoid losing the complex intricacies.

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 aids 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.

Figure 1: Loss for E6-2 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 strategic or gameplay reasons, we include a stochastic mode in the opening game: the engine chooses to play the first through fourth best moves with certain probabilities. This keeps games against the bot interesting and unrepetitive, preventing the bot from predicting the same move for the same position every time.

Move (according to engine)	Probability	Until
1st Best	65	All
2nd Best	20	7
3rd Best	10	5
4th Best	5	3

For implementation, please see Appendix B.

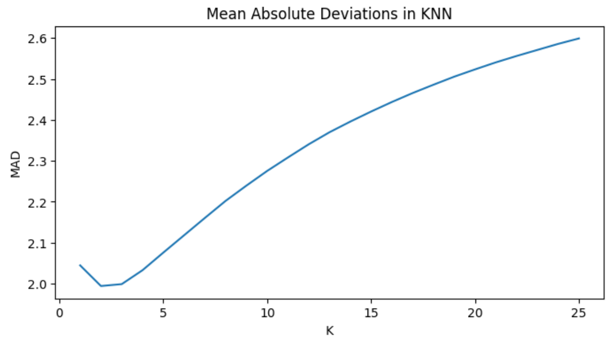
5 Classical Machine Learning

To improve the performance of the model, we make use of ensemble learning: training weak learners and combining their results through democracy vote or other selection methods. Since various machine learning algorithms are designed in different ways, our hope is that introducing classical machine learning algorithms can help detect features which are predictors of the chess board state. We implement each of these in `scikit-learn`. [5]

Since classical machine learning algorithms typically seek to learn a model $\hat{f} : \mathbb{R}^{n,P} \to \mathbb{R}^n$, we slightly modify our preprocessing, flattening the input tensor for each sample to a numpy array in a single dimension. We choose to still concatenate scalar inputs about the number of alive pieces as detailed in Section [3.2].

5.1 K-Nearest Neighbors (KNN) Regression

On a smaller subset of the data, we brute-force the optimal value of K. As chess positions often have many intricate details, we use a modified version of KNN which computes a weighted average depending on the distance to the nearest points. The mean absolute deviation score for each value of K is shown below.



Thus, we propose that $K = 2$ provides the best value of K.

5.2 Random Forest Regression

Random Forests can produce a robust model for predicting intricate details. We implement a Random Forest Regressor with 20 Decision Trees and use 20 Max Features.

```
model_RF = RandomForestRegressor(
    n_estimators = 20, # 20 DTs
    random_state = 0, # Deterministic
    # verbose = 1,
    n_jobs = 8, # run in parallel
    max_features = 20,
)
```

We train using multithreading on a 12th Gen Intel(R) Core(TM) i5-12400F CPU.

5.3 Support Vector Regression

Support Vector Regression (SVR) is another technique often used in classical machine learning which may help us learn more. We implement a standard SVR model.

5.4 Ensemble

To test how each of the classical machine learning algorithms mesh with each other, we implement some toy playtesting code. We implement an ensemble method to take the weighted average of each model.

```
predict_cp_democracy(inputs, models, weights)
```

The ensemble model seems to perform well - while testing individual models seems to make nonsensical moves, the ensemble model appears to play decently well.

More about the implementation may be found in Appendix B.

6 Playing and Interface

6.1 Lichess API

We use the `Python-Lichess` API [1] 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!

7 Reproducibility

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

Appendix B details more information about the code and how to interpret the files in the repository.

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 two 2-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 convolution 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 convolution 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.

I made further decisions to aim to improve the model. Using a second convolution layer with a different kernel size gave the hope that the model would be able to learn more about the intricacies in piece movement.

It quickly became clear that resources were a huge bottleneck in the training of this neural network as as epoch could take hours to train. As a result, going forward I tried to find additional solutions which could supplement the performance of the model.

Class discussions on ensemble methods led me to think about joining weak learners, perhaps those from classical supervised machine learning methods. I experimented with regression variants of K-Nearest Neighbors and Support Vector Machines. Also, I built upon class homework on Decision Trees by toying with Random Forest Regression. Each of these small learners were weak chess players individually but helped to increase the efficacy when combined with the deep learning model.

Throughout this decision process, we make a number of models each with different training data, preprocessing, parameters, and more. We give a description of each of the models here as well as links to their source files on GitHub.

SL V1 Library | Training. The first model trained.
Data: All csv files
Task: Predict move tensor (starting position, ending position) from board state, turn and check status
Model: FCNN. (66, 1024, 2048, 64*64).
Loss: Custom loss, MSE + Illegal Move Penalty of 5000
Optimizer: SGD(lr=0.001, momentum=0.9)
Epochs: 10

SL V2 Library | Training
Objective: Improve upon speed using batch training. Other optimizations.
Data: All csv files.
Task: Predict move tensor (starting position, ending position) from board state, turn and check status
Model: FCNN. (64*8*8+2, 1024, 2048, 64*64) with batch training
Loss: Custom loss, MSE + Illegal Move Penalty of 5000
Optimizer: SGD(lr=0.001, momentum=0.9)
Epochs: 10

SL V3 Library | Training
Objective: Speed up convergence by training with higher LR. Try tanh activation in one of the hidden layers.
Data: All csv files
Task: Predict move tensor (starting position, ending position) from board state, turn and check status
Model: FCNN. (66, 1024, 2048, 64*64)
Loss: Custom loss, MSE + Illegal Move Penalty of 5000
Optimizer: SGD(lr=0.003, momentum=0.9)
Epochs: 10

SL V4 Library | Training
Objective: Use batch selection and cross-validation inside training loop instead of in a dataloader to increase number of batches and speed up optimizer steps. Use convolutional neural network instead of flat input tensors. Now, inputs in the shape of (8,8).
Data: Uses csv file selection inside training loop
Task: Predict move tensor (starting position, ending position) from board state, turn and check status
Model: Conv2D(3, 64, ker=8), FCNN(64+2, 1024, 2048, 64*64)
Loss: Custom loss, MSE + Illegal Move Penalty of 250
Optimizer: SGD(lr=0.75, momentum=0.9)
Epochs: 8000

EL V1 Library | Training
Objective: Change paradigm to calculate positions rather than moves. Use a subset of data for faster training.
Training Data: All csv files starting with a or b
Validation Data: All csv files starting with c or d
Task: Predict Stockfish evaluation (CP) from board state (FEN), turn and check status
Model: Conv2D(1, 16, ker=6), FCNN(402,128,1)
Loss: L1Loss
Optimizer: SGD(lr=0.035, momentum=0.9)
Epochs: 200

EL V3 Library | Training
Objective: Use an odd, smaller kernel size. Use Adam optimizer. Update FCNN layers.
Training Data: All csv files starting with a, b, c or d
Validation Data: All csv files starting with d, e, f or g
Task: Predict Stockfish evaluation (CP) from board state (FEN), turn and check status
Model: Conv2D(1, 16, ker=5), FCNN(578, 512,1)
Loss: L1Loss
Optimizer: Adam(lr=0.006)
Epochs: 40

EL V4 Library | Training
Objective: Instead of a flat tensor for board position with different categorical data, use a separate channel for each

individual piece. Now, input shape is (6,8,8).
Training Data: All csv files starting with a, b, c or d
Validation Data: All csv files starting with d, e, f or g
Task: Predict Stockfish evaluation (CP) from board state (FEN), turn and capture status
Model: Conv2D(6, 16, ker=5), FCNN(578, 512, 1)
Loss: L1Loss
Optimizer: Adam(lr=0.006)
Epochs: 25

EL V5 Library | Training
Objective: Train faster without using as many validation csv files. Other optimizations.
Training Data: All csv files starting with a, b, c or d
Validation Data: All csv files starting with e or f
Task: Predict Stockfish evaluation (CP) from board state (FEN), turn and capture status
Model: Conv2D(6, 16, ker=5), FCNN(578, 512, 1)
Loss: L1Loss
Optimizer: Adam(lr=0.006)
Epochs: 50

EL V6 Library | Training
Objective: Add a second convolutional layer to increase extractable features.
Training Data: All csv files starting with a, b, c or d
Validation Data: All csv files starting with e or f
Task: Predict Stockfish evaluation (CP) from board state (FEN), turn and capture status
Model: Conv2D(6, 16, ker=5), Conv2D(16, 32, ker=3), FCNN(32*8*8, 512, 1)
Loss: L1Loss
Optimizer: Adam(lr=0.006)
Epochs: 50
Special: Reactivate V6 model to continue training over and over again instead of creating new model.

EL V7 Library | Training
Objective: Continue training models from V6 with more data.
Training Data: All csv files starting with a-p
Validation Data: Every fourth training file
Task: Predict Stockfish evaluation (CP) from board state (FEN), turn and capture status
Model: Conv2D(6, 16, ker=5), Conv2D(16, 32, ker=3), FCNN(32*8*8, 512, 1)
Loss: L1Loss
Optimizer: Adam(lr=0.006)
Epochs: 50
Special: Reactivate V6 model to continue training over and over again instead of creating new model.

EL V8 Library | Training
Objective: Add ensemble methods into prediction, reduce overhead, increase training
Training Data: All csv files.
Validation Data: Every fourth training file
Task: Predict Stockfish evaluation (CP) from board state (FEN), turn and capture status
Model: Conv2D(6, 16, ker=5), Conv2D(16, 32, ker=3), FCNN(32*8*8, 512, 1)
Loss: L1Loss
Optimizer: Adam(lr=0.0065)
Epochs: 30
Special: Ensemble methods (Random Forest, KNN, SVR).

B Code

The GitHub repository contains a number of folders and files. We detail some info about the files of interest to reproduce the results of the paper.

The `source_EL` folder contains source files for the newest models, in which the evaluation is the predictor variable. The `source_SL` folder contains source files for some older models in which the predictor variable is the move itself, in the form of a two-dimensional tensor representing starting and ending move. Each of these source file folders contain scripts of different versions: for each version, there is a library `.py` file containing the neural network architecture and any helper files and a Jupyter notebook in which the model is trained.

The `models_EL` and `models_SL` folders contain the torch model files which can be loaded and ran (so long as the neural network architecture is imported via the library file).

For the classical machine learning models, I implemented many individual model generation notebooks (these can be found in the main repository) and a library file. This library contains helpers for the ensemble methods as well as evaluating and using the classical machine learning models.

The `lichess_bot`[1] folder contains files in which the API may be initiated. In particular, `homemade.py` implements the interaction between the neural network and the Lichess API, and running the `lichess-bot.py` file initiates the connection.

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

C Future Work

C.1 Limitations

Through playtesting, we hypothesize some of these limitations with this model:

1. Since the model is only trained on pre-existing evaluations, it may incorrectly evaluate positions.
2. Computing resources and time were limited. Many advanced machine learning models were trained for longer on faster and higher capacity hardware.
3. Predicting only by considering the optimal evaluation in the next move misses out on strategies that involve strings of moves.

Future work could expand upon this model by meshing this neural network with more techniques used in hard-coded engines like the minimax algorithm and alpha-beta pruning to determine optimal positions to a further depth.

D Conclusion

Thank you for reading. Many, many hours of work and training time went into this project: I hope you had fun playing!

References

[1] MarkZH et al. “lichess-bot”. URL: https://github.com/lichess-bot-devs/lichess-bot.

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

[3] 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.

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

[5] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.