Prediction of Chess Board State Using Vision Based Models

Lorenzo Lucena Maguire
University of Pennsylvania
Philadelphia, USA
llucena@seas.upenn.edu

Ruturaj A. Nanoti
University of Pennsylvania
Philadelphia, USA
ruturajn@seas.upenn.edu

Siddhant Mathur
University of Pennsylvania
Philadelphia, USA
siddm14@seas.upenn.edu

***Abstract*—Identifying chess positions can be used as a pre-processing step for chess engines to predict or recommend the next best move in that particular position. The recommendation relies heavily on the position of the black and white pieces on the board. Hence, creating a model that can accurately predict the current position on the board which enables a chess engine to perform better estimates of the upcoming moves. This project aims to predict the chess board state, given its image as the input. The labels of the images are represented in FEN notation.**

***Index Terms*—Chess, CNN, ConvMixer, MLPMixer, Deep Learning, FEN**

I. Introduction

Chess Engines rely heavily on identifying current chess-board states to effectively recommend the next move and predict the way in which the game might pan out up to a few moves in the future. Therefore, we aim to accurately predict the current chess board state based on the FEN notation. To achieve this we have trained our dataset, on multiple models and performed a comparative analysis. We have used deep learning for this problem since they efficiently capture the non-linearity in chess board state representations. Models like CNN perform dimensionality reduction on images and at the same time preserve important features and information.

II. Dataset

Our dataset contains chess board images and has been taken from Kaggle. It contains 100,000 images of randomly generated chess positions of 5-15 pieces (2 kings and 3-13 pawns/pieces). The chess boards and pieces are of different styles. The images were generated using a custom tool that generated random chess board positions. The labels for our dataset are *FEN* Notations that encode the current state of the chess board as a string. All the images were 400×400 pixels. The dataset consists of 80,000 training samples and 20,000 testing samples. However, we have used 8,000 images as our training set and 2,000 images as our testing set to make it computationally feasible.

III. Pre-Processing Data

The labels for our dataset are in the form of strings and need to be encoded in a way that would allow us to capture all the information in the form of numbers. Hence, we chose to *One-Hot Encode* the labels for the same.

A. FEN Notation

FEN stands for *Forsyth–Edwards Notation*, which is used to represent a particular state of a chess board. It is highly useful when a game needs to restart from a particular position while preserving all the relevant information to do so. The FEN strings in our dataset captures the piece placement and active color data. There are 6 unique pieces namely ***b**ishop, **k**ing, k**n**ight, **p**awn, **q**ueen*, and the ***r**ook* each for the black and the white camp. Every piece in white camp is represented using upper-case letters and the black camp is represented using lowercase letters. Each rank (row) of the chessboard is separated using a ”–” as the delimiter.

Fig. 1. Sample Chess Board Image

Following is a brief explanation of the FEN notation:

- This is the label/position for the state of the chess board.
- Here the **K** represents a king (white) in the first cell of the first row.
- The number **5**, represents 5 *empty* cells after the King (white).
- The **k** represents the king (black) in the seventh cell of the first row.
- Finally, the number **1**, represents an *empty* cell at the end of the first row.
- The other rows of the chessboard are encoded similarly.

B. One-Hot Encoding the FEN Strings

To use these labels for training our models, we will need to One-Hot encode them. Therefore, we created a 64 x 13 matrix that encoded this FEN string. The number of columns is 13, because there are 6 unique pieces in chess from both the white and the black camp. Finally, an extra column denotes whether the cell is empty. Each cell on the chessboard is represented by a 13-element long vector, where each entry signifies whether that piece is present in the cell. Whereas each row is the state of a single cell on the chess board.

IV. Loss Function and Optimizer

The loss function that we have used for our models is the Cross Entropy loss. It is used to adjust the model weights during training and the aim is to minimize the loss. Each predicted class probability is compared to the actual class desired output 0 or 1 and a score/loss is calculated that penalizes the probability based on how far it is from the actual expected value. In a sense, our problem statement is also a classification task where each cell is classified into one of the 13 classes (6 unique pieces in chess from both the white and the black camp and the last class denoting whether a certain cell is empty).

We used Adam Optimizer for training our models. It is an optimization algorithm for stochastic gradient descent that is used for training deep learning models. Adam Optimizer combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems. Accuracy is measured at an atomic level, where if at least one cell is mislabeled, the prediction is tallied as incorrect.

V. MLP-Mixer

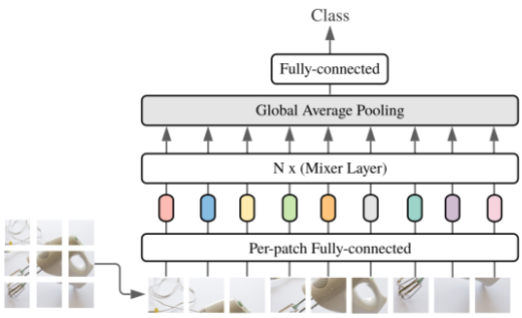

Fig. 2. General MLP-Mixer Model

The MLP-Mixer model is our baseline for predicting the position of a board from its image. It is a vision transformer model which first segments the input image into patches and linearly embeds them using a convolutional layer. The kernel size of the convolutional layer is set equal to the stride to map the input image onto a collection of linear embeddings for each patch of pixels. The embedding of each patch has dimensions of $(, h)$ where h is the number of channels specified in the convolutional layer. The vision transformer then applies a series of sequential *Mixer Layers*. Each layer first transposes the patch embeddings and applies an MLP model over the channels (each input vector consists of the different patch embeddings entry for the same channel). Then, the output is transposed and another MLP is applied over the patch embeddings.

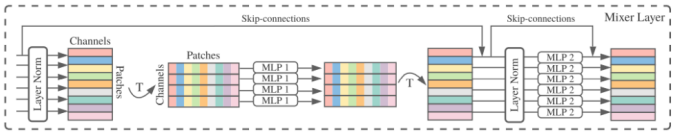

Fig. 3. Mixer Layers in MLP-Mixer

Each *Mixer Layers* uses two MLP models which share their weights among the segmented input, and employs residual connections after each MLP model. Given the hidden dimensions for the MLP applied to the patches and the channels, each mixer layer has $2 * (h_c p + h_p c)$ where c is the number of channels, p is the number of resulting patch embeddings, h_c is the hidden dimension in the MLP applied over the channels and h_p is the hidden dimension in the MLP applied over the patches.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
MLPMixer                                 [128, 832]                --
├─Conv2d: 1-1                            [128, 900, 8, 8]          810,900
├─Sequential: 1-2                        [128, 64, 900]            --
│    └─MLPMixerLayer: 2-1                [128, 64, 900]            --
│    │    └─LayerNorm: 3-1               [128, 64, 900]            1,800
│    │    └─MLPBlock: 3-2                [128, 900, 64]            --
│    │    └─LayerNorm: 3-3               [128, 64, 900]            1,800
│    │    └─MLPBlock: 3-4                [128, 64, 900]            --
│    └─MLPMixerLayer: 2-2                [128, 64, 900]            --
│    │    └─LayerNorm: 3-5               [128, 64, 900]            1,800
│    │    └─MLPBlock: 3-6                [128, 900, 64]            --
│    │    └─LayerNorm: 3-7               [128, 64, 900]            1,800
│    │    └─MLPBlock: 3-8                [128, 64, 900]            --
│    └─MLPMixerLayer: 2-3                [128, 64, 900]            --
│    │    └─LayerNorm: 3-9               [128, 64, 900]            1,800
│    │    └─MLPBlock: 3-10               [128, 900, 64]            --
│    │    └─LayerNorm: 3-11              [128, 64, 900]            1,800
│    │    └─MLPBlock: 3-12               [128, 64, 900]            --
│    └─MLPMixerLayer: 2-4                [128, 64, 900]            --
│    │    └─LayerNorm: 3-13              [128, 64, 900]            1,800
│    │    └─MLPBlock: 3-14               [128, 900, 64]            --
│    │    └─LayerNorm: 3-15              [128, 64, 900]            1,800
│    │    └─MLPBlock: 3-16               [128, 64, 900]            --
├─Linear: 1-3                            [128, 832]                749,632
==========================================================================================
Total params: 1,574,932
Trainable params: 1,574,932
Non-trainable params: 0
Total mult-adds (G): 6.74
==========================================================================================
Input size (MB): 29.49
Forward/backward pass size (MB): 531.69
Params size (MB): 6.30
Estimated Total Size (MB): 567.48
==========================================================================================
```

Fig. 4. MLP-Mixer Architecture

We hypothesized that given the nature of our problem, the formatted cell structure of the board could be leveraged by the model such that each individual patch corresponded to a single cell of the chess board. Thus, we set the hyperparameter of the patch size to be equal to 30, such that our initial 240 x 240 image was broken down into 64 patches.

VI. CNN

A convolutional neural network is a Deep Learning algorithm that takes images as input and reserves significance to certain features in the images and is able to differentiate one

image from another. A benefit of the CNN architecture is that they are easier to train and have a considerably lower number of parameters than fully connected networks despite having the same number of hidden units.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Net                                      [128, 832]                --
├─Conv2d: 1-1                            [128, 16, 78, 78]         1,952
├─ReLU: 1-2                              [128, 16, 78, 78]         --
├─MaxPool2d: 1-3                         [128, 16, 39, 39]         --
├─Conv2d: 1-4                            [128, 32, 19, 19]         25,120
├─ReLU: 1-5                              [128, 32, 19, 19]         --
├─MaxPool2d: 1-6                         [128, 32, 9, 9]           --
├─Dropout: 1-7                           [128, 2592]               --
├─Linear: 1-8                            [128, 2048]               5,310,464
├─ReLU: 1-9                              [128, 2048]               --
├─Linear: 1-10                           [128, 832]                1,704,768
==========================================================================================
Total params: 7,042,304
Trainable params: 7,042,304
Non-trainable params: 0
Total mult-adds (G): 3.58
==========================================================================================
Input size (MB): 29.49
Forward/backward pass size (MB): 114.46
Params size (MB): 28.17
Estimated Total Size (MB): 172.12
==========================================================================================
```

Fig. 5. Architecture for the Convolutional Neural Network

Fig. 5 represents the summary of the CNN architecture that was used for training the model. The architecture consists of two convolutional layers, and two max-pooling layers. Along with this, a dropout of 0.1 is added to introduce some regularization and reduce overfitting. Finally, there are two dense layers at the end, to obtain the prediction matrix for the chessboard state.

During the training phase, we experimented with the architecture to arrive at this model. When the number of convolutional layers was increased, we noticed that the model began to overfit the data. Also as the layers increase, the number of trainable parameters starts to increase, which bloats up the model and is not preferable in terms of computational feasibility. The dropout was also varied between values from 0.5 to 0.1. We observed that at a dropout of 0.5, the model did not learn the parameters well enough to accurately predict a chess board state. Upon testing our model with a dropout of 0.5, the testing accuracy came out to be around 13%. This can possibly be attributed to the fact that adding a substantial amount of dropout would make the model drop certain features in an attempt to add some regularization. However, this would not work well for our problem statement since even if the model gets one of the chess piece positions incorrect, it would imply that it has gotten the entire chess board state incorrect. For a chess board state to be accurately predicted, the model would need to get the positions of every single chess piece right.

VII. ConvMixer

The ConvMixer is another vision transformer model which first divides the image into patches and linearly embeds them using a convolutional layer. Unlike the MLP-Mixer, which is an all-MLP model, a sequential series of *ConvMix Layers* are applied onto each patch embedding, which operates on the patches independently, sharing weights among patches. Each *ConvMix Layer* consists of a depthwise convolution followed by a pointwise convolution with a residual connection between the two convolutions.

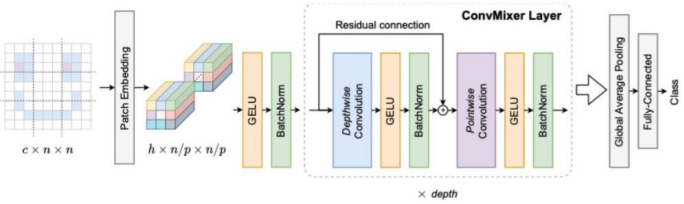

Fig. 6. ConvMixer Model

We observed that specifying the number of *ConvMix Layers* to two yields the optimal tradeoff between model size and performance. Each fully convolutional block, using a kernel size of 8 for the depthwise convolution, required an additional 50 million parameters. Limited by the hardware at our disposal, two ConvMix Layers would result in both training and testing accuracies over 98% after training for 50 epochs, using a 4:1 train to test the split of the dataset, and mantaining the patch size represent a chess cell.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
ConvMixer                                [128, 832]                --
├─Conv2d: 1-1                            [128, 900, 8, 8]          810,900
├─GELU: 1-2                              [128, 900, 8, 8]          --
├─BatchNorm2d: 1-3                       [128, 900, 8, 8]          1,800
├─Sequential: 1-4                        [128, 900, 8, 8]          --
│    └─Sequential: 2-1                   [128, 900, 8, 8]          --
│    │    └─Residual: 3-1                [128, 900, 8, 8]          51,842,700
│    │    └─Conv2d: 3-2                  [128, 900, 8, 8]          810,900
│    │    └─GELU: 3-3                    [128, 900, 8, 8]          --
│    │    └─BatchNorm2d: 3-4             [128, 900, 8, 8]          1,800
│    └─Sequential: 2-2                   [128, 900, 8, 8]          --
│    │    └─Residual: 3-5                [128, 900, 8, 8]          51,842,700
│    │    └─Conv2d: 3-6                  [128, 900, 8, 8]          810,900
│    │    └─GELU: 3-7                    [128, 900, 8, 8]          --
│    │    └─BatchNorm2d: 3-8             [128, 900, 8, 8]          1,800
├─AdaptiveAvgPool2d: 1-5                 [128, 900, 1, 1]          --
├─Flatten: 1-6                           [128, 900]                --
├─Linear: 1-7                            [128, 832]                749,632
==========================================================================================
Total params: 106,873,132
Trainable params: 106,873,132
Non-trainable params: 0
Total mult-adds (G): 33.30
==========================================================================================
Input size (MB): 29.49
Forward/backward pass size (MB): 358.43
Params size (MB): 427.49
Estimated Total Size (MB): 815.42
==========================================================================================
```

Fig. 7. Architecture for the ConvMixer

VIII. RESULTS

After training each model for 50 epochs, we obtained the following accuracy levels:

Model	Training Accuracy	Testing Accuracy
MLP-Mixer	55.57%	50.46%
CNN	90.40%	92.90%
ConvMixer	98.86%	98.85%

Following are the accuracy plots for the 3 models that we have implemented. These plots indicate that we achieved a highest accuracy of 98.86% using ConvMixer which can be explained by the fact that the ConvMixer model has the highest number of trainable parameters out of the 3 models.

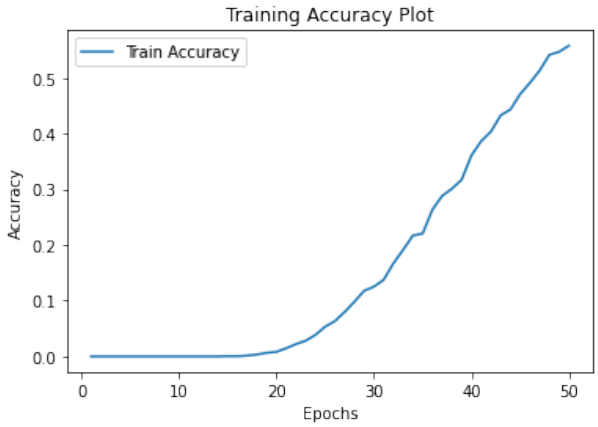

Fig. 8. Training accuracy for MLP-Mixer over 50 epochs

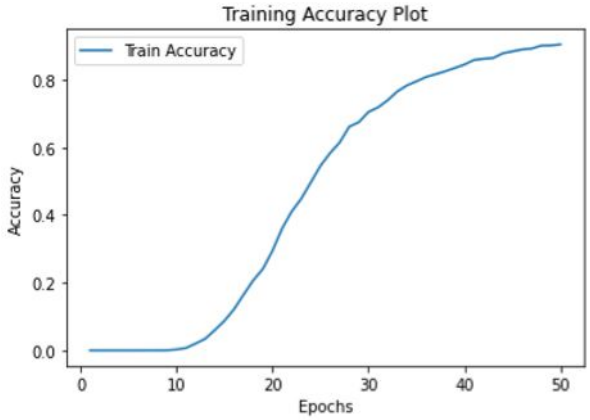

Fig. 9. Training accuracy for CNN over 50 epochs

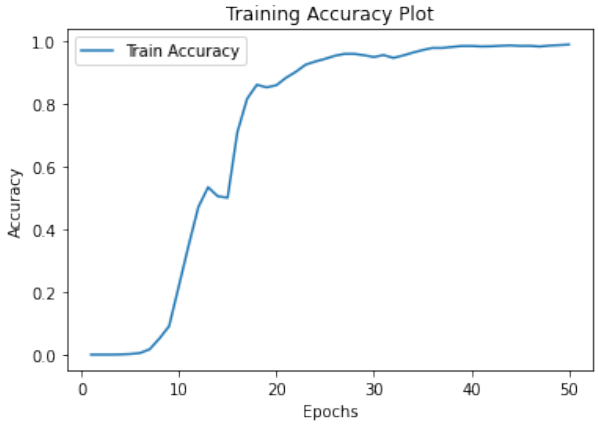

Fig. 10. Training accuracy for ConvMixer over 50 epochs

Following are some incorrect predictions from our models that were visualized and compared with the true labels from the dataset. Fig 11 and 12 show the chess board state represented by the FEN string *N1q5-8-5B2-2N1p2P-n5b1-6k1-2KP4-2r5* and *N1qq4-8-5B2-2N1p2P-n5b1-6k1-2KP4-2r5* respectively. It can be observed that the *black queen* on **d8** in the true label is not picked up by the model. Note that this completely changes the state of the board from an analytical perspective, since having an extra queen (obtained through a pawn promotion) on the black camp gives black an advantage (provided it's blacks' turn). Similarly from Fig. 13 and 14, we can observe that the *black bishop* on **a7** is not picked up by our model.

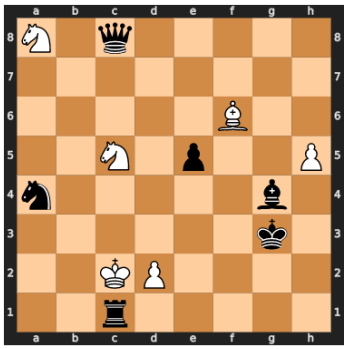

Fig. 11. Predicted FEN string converted to a Chess Board - 1

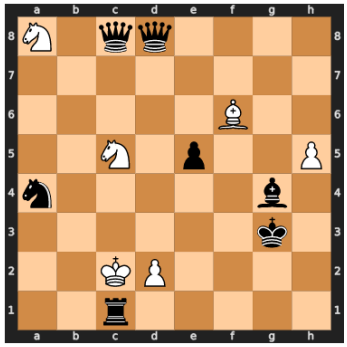

Fig. 12. True FEN string converted to a Chess Board - 1

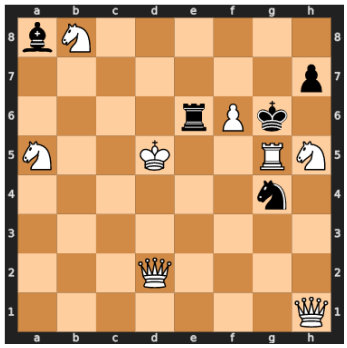

Fig. 13. Predicted FEN string converted to a Chess Board - 2

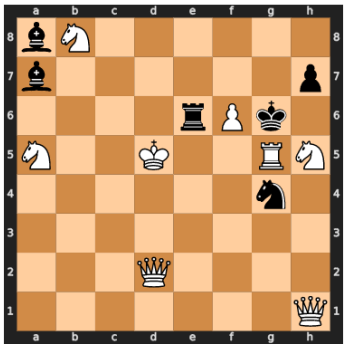

Fig. 14. True FEN string converted to a Chess Board - 2

IX. Conclusion

Our experiments demonstrate the different performances of different models with different costs, and highlight the importance of identifying the business needs in order to make the most rational model selection.

From the comparative study of our models, we observe that the ConvMixer model performs the best among all the three. However, such performance comes at a cost in the form of computational resources. It can be observed that the ConvMixer model has a higher number of trainable parameters as compared to the CNN and MLP-Mixer model. This can also be attributed to the high number of channels in the ConvMixer model, which was chosen to avoid dimensionality reduction and subsequent loss of information.

Setting the patch to of size 30×30 clearly improved the performance of the ConvMixer. This tackled the problem of predicting the piece present in an all-or-nothing fashion, as opposed to the CNN model (whose kernel size doesn't map to an individual cell). Moreover, the difference in the accuracies obtained by the models that perform the convolutional operation on images (CNN and ConvMixer) and the model that doesn't (MLPMixer) goes to show that convolutional filters extract great amounts of information from the image.

References

[1] A. Trockman and J. Z. Kolter, "Patches are all you need?", 2022.

[2] Tolstikhin, Ilya O., "Mlp-mixer: An all-mlp architecture for vision.", 2021

[3] Wolflein, G., & Arandjelovic, O. (2021). Determining Chess Game State from an Image. Journal of Imaging, 7.

[4] Sabatelli, Matthia & Bidoia, Francesco & Codreanu, Valeriu & Wiering, Marco. (2018). Learning to Evaluate Chess Positions with Deep Neural Networks and Limited Lookahead. 10.5220/0006535502760283.

[5] Mehta, Anav. (2020). Augmented Reality Chess Analyzer (ARChessAnalyzer): In-Device Inference of Physical Chess Game Positions through Board Segmentation and Piece Recognition using Convolutional Neural Network.

[6] Afonso de S $\tilde{A}$ Delgado Neto and Rafael Mendes Campello. Chess position identification using pieces classification based on synthetic images generation and deep neural network fine-tuning. In 2019 21st Symposium on Virtual and Augmented Reality (SVR), pages 152–160, 2019.

[7] Wu, Allen. "Efficient Chess Vision – A Computer Vision Application." (2022).

[8] Perez, Luis & Wang, Jason. (2017). The Effectiveness of Data Augmentation in Image Classification using Deep Learning.