

Math 156 Report

Anton Saleem Almgren, Srishti Ganu, Hyunchae Kim, Henry Lindhurst

December 6th, 2024

1 Abstract

Artificial intelligence, particularly through artificial neural networks, has emerged as a popular trend in modern technology. Companies fiercely compete to develop the most advanced AI systems, igniting the AI race with billions of dollars invested around the world. AI uses neural networks, structures inspired by the human brain, utilizing nodes to represent neurons and connections to resemble neural pathways. This report primarily focuses on convolutional neural networks (CNNs), which are particularly successful when applied to tasks like image recognition. However, the focus here is shifted to Chess, investigating whether a CNN can identify patterns in Chess positions and evaluate them effectively. This approach only covers a portion of the functionality of traditional Chess engines, yet an important portion nonetheless. This report outlines the process of implementing a CNN for Chess evaluation and the different components of the process.

Keywords: neural network, convolutional neural network, machine learning, Chess, Chess engine

2 Introduction

Chess is one of the world's most widely recognized and influential board games. Achieving mastery in Chess requires a tremendous amount of time, dedication, and skill, and throughout history, being a Chess master has been associated with high intellect and a strategic mind. In the 20th century, computers entered the competition, and in 1996 a computer was crowned world champion in Chess for the first time. Ever since then, Chess engines have become increasingly stronger and more powerful. Today, modern computers can perform billions of computations per second but despite this, Chess remains unsolved, meaning there is no known strategy that guarantees victory, or even perfect play, in a game. The vast number of possible positions quickly escalates toward infinity, rendering even billions of computations insufficient for a solution. To address this complexity, computers traditionally rely on algorithms and decision trees to prioritize certain moves and disregard others. However, with advancements in artificial intelligence, modern engines have become significantly more sophisticated by implementing neural networks. The goal of this project is to develop a Chess engine entirely based on a convolutional neural network and to evaluate its performance. Convolutional neural networks seek to find patterns by looking at the chessboard, but in our case the chessboard is represented as a three-dimensional vector (8x8x12). The first two dimensions illustrates the regular Chess board, but the third layer separates each piece into its own layer, making it easier for the model to distinguish the pieces as it does not know how pieces move or how valuable they are to begin with.

3 Background

A Convolutional Neural Network (CNN) is a type of deep learning model for processing data structures formatted as grids or lattices, such as images. The convolutional neural network consists of multiple layers, including input layer, convolutional layer, and sub-sampling layer. It uses three mechanisms: (i) local receptive fields, (ii) weight sharing, and (iii) subsampling (pooling) (Bishop 268). When constructing a model using fully connected layers in three dimensional space, the data was converted into one-dimensional data then applied into input layer. The conversion of data creates a major disadvantage of losing the geometrical shape of the data. The data with three-dimensional shape (height, weight, channel) has a spatial structure. For example, the Chess positions contains its information in three dimensional shape (two dimensions of chessboard + Chess piece type). In the convolutional layer the output maintains the shape of the input data. The convolution operator on real valued functions maps x and w to s which is defined as

$$s(t) = (x * w)(t) = \int_{-\infty}^{\infty} x(a)w(t-a)da$$

(Goodfellow Ch 9.1)[2]

In convolution neural network, the convolution is referred as the input, function g as kernel or filter, and the output as featured map. If we apply this to three-dimensional input I , using three-dimensional kernel K :

$$S(i, j) = \sum_m \sum_n \sum_c I(i+m, j+n, c)K(m, n, c)$$

where S is the output feature map.

The convolution layer is followed by the nonlinear activation function using ReLU which is applied to featured map:

$$f(x) = \max(0, x)$$

Lastly, we use a pooling function in the pooling layer to reduce the size of the feature map. For example, the max pooling operation reports the maximum output within a rectangular neighborhood. (Goodfellow)[2]

$$S(i, j) = \max_{m=1, \dots, l} \max_{n=1, \dots, w} I(hi+m, hi+n)$$

where l, w are the kernel dimensions and h is the stride (step size of the convolution).

4 Dataset

We are using a Chess Evaluations [1] dataset, found on Kaggle, which includes 12,954,834 positions with a score tied to each position. The evaluations come from Stockfish, one of the most popular and powerful Chess engines. Some of the scores indicate when a forced checkmate is possible, which does not have a clear numeric value. We handled these by replacing them with a very high score for the player forcing the checkmate. Since the dataset was very large, we also had to reduce the batch sizes in order to run our model. Some runs of the model were even done on just a subset of the dataset, in order to more clearly demonstrate the model's effectiveness in a reasonable amount of time. Other data preprocessing and cleaning we had to perform was transforming all of our input data from a string to a vector (technically a tensor) that our model could accept. The input data

used a FEN-string-format, which is a compact way of describing a Chess position. We created our own module to help with this transformation, going from a FEN string to a 8x8x12 vector. The dataset was already clean and no values were missing. We also chose not to normalize the data as the dataset would have too many extreme values. We did a standard train-test-validation split with our training data being 70% of the dataset and the validation set and testing set each being 15%.

In terms of exploratory data analysis, we looked at a few rows of the data, examined the max and min values, and generated a histogram of the dependent variable values.

5 Model

We experimented with feed forward neural networks at first before moving on to our convolutional neural network instead. We did not implement it fully from scratch because we used premade functions from PyTorch, but we still made a custom model using a sequential container and did not rely on a pretrained model. We implemented a convolutional 2D layer that takes the 12 channel input and applies 16 3x3 filters, producing an output with 16 channels that are 8x8. Then we put it through a ReLU activation function to introduce additional variance. Then we use MaxPool2d to downsample to a 4x4. Finally, we apply fully connected layers, first flattening and then putting it through the linear layer and outputting one value as a position score. We used the PyTorch Mean Squared Error loss function and used AdamW for the optimizer. Mean squared error is good for regression tasks and predicting continuous values. It penalizes larger errors more than smaller errors, which is helpful for our goal of minimizing how far off the predicted score is from the actual target score. It is better to have many scores be off by a little bit than one score be off by a lot, since that one incorrect classification is an easily exploitable weakness, whereas many small errors are hard to exploit. We chose AdamW which is Adam with weight decay because it penalizes large parameter values. This helps prevent overfitting, so that the model generalizes better to new Chess positions. Our dataset was very large, so we only used about 40% of the total data so that it could run on our laptops in a timely manner.

6 Results

6.1 Few Big Epochs

On our initial run with 10 epochs and only 10,000 batches of randomly selected 35% subset of the total data, we observed an average improvement in the loss. Our model ran for 3 minutes despite a significant reduction in how much of the dataset we used to train which shows how much data was in the dataset we chose.

We spent a lot of time fine-tuning the model to achieve a training loss graph that did not oscillate as heavily. We set the initial learning rate much lower from 0.001 to 0.000001 to 0.0000001. This didn't visibly improve our results so we set it back to 0.001. Finally, after adding a small change to the training loop, our model's performance improved and gave the loss a more downward trend as pictured in Figure 1.

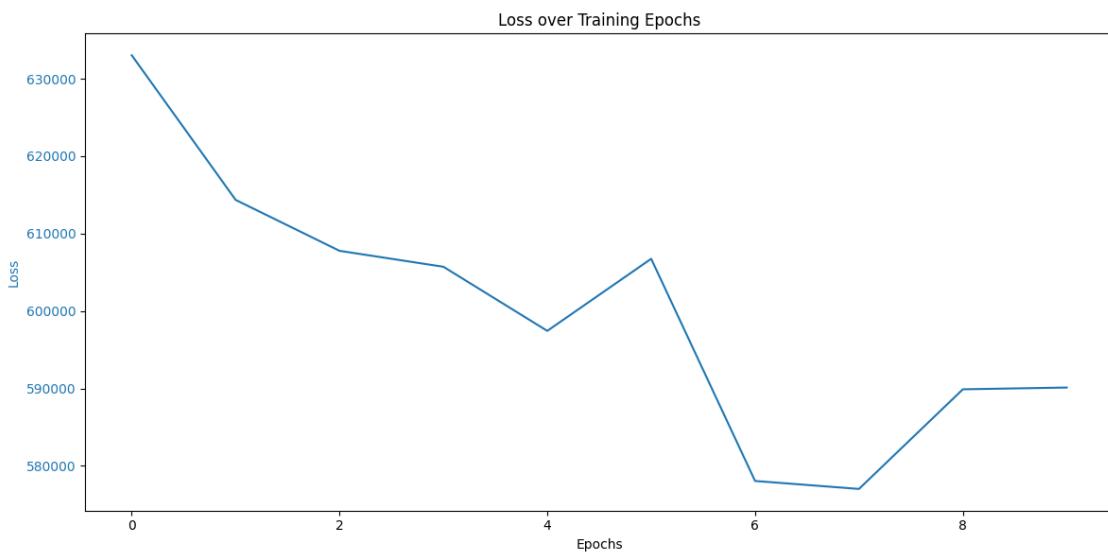


Figure 1: Training Loss over 10 Epochs

→	Epoch [1/10], Loss: 633032.4468
	Epoch [2/10], Loss: 614353.6434
	Epoch [3/10], Loss: 607775.1899
	Epoch [4/10], Loss: 605716.2845
	Epoch [5/10], Loss: 597436.3437
	Epoch [6/10], Loss: 606745.7912
	Epoch [7/10], Loss: 578050.6737
	Epoch [8/10], Loss: 577023.0202
	Epoch [9/10], Loss: 589889.4497
	Epoch [10/10], Loss: 590119.1958

Figure 2: Training Loss over 10 Epochs Table

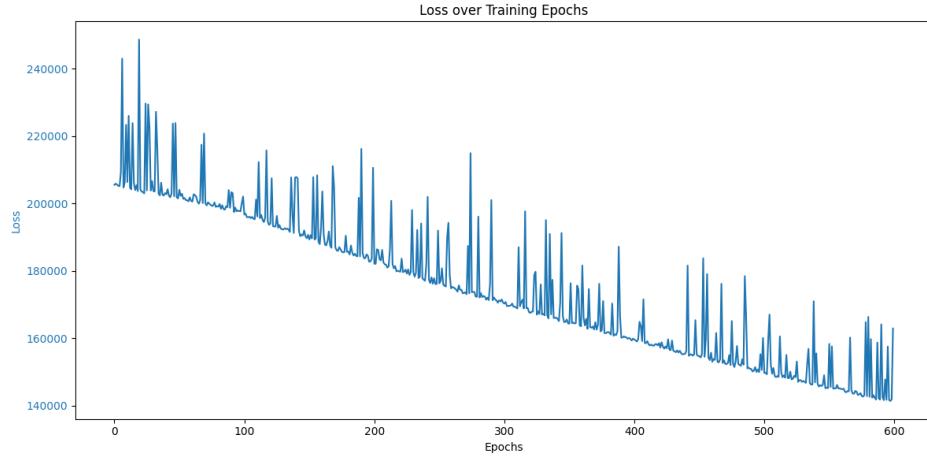


Figure 3: Short Epoch Training Loss

6.2 Many Small Epochs

Later on, we tried training the model with a much smaller subset of the dataset, only 3000 data points with many more epochs with a smaller to get a high resolution graph of the training and see if training for longer would improve results. The batch limit didn't seem to have that much of an effect on the training loss, so it was reduced to improve runtime. The loss graph seems to show significant improvement, however the validation MSE was only slightly lower than the training MSE was before training.

```
MSE before training: 692788.0625
Validation MSE: 666561.5000
```

This would suggest that the model, when trained in this way, generalizes poorly to new data and is overfitting to the training set. Also, the loss seems to randomly spike constantly in the training process, despite an overall downward trend. We believe the spikes were caused by outliers in the dataset, if an outlier is present in the batch, the loss is high, if not it is low.

7 Conclusion

The overall goal of the project was to train a convolution neural network that evaluates a Chess position. This was done by converting the position into a three-dimensional vector and applying a convolutional neural network. We achieved a gradual decrease(improvement) in the training loss, but the validation loss did not improve as much, suggesting that the model overfit to the data. The model was not very successful overall.

8 Author Contributions

Anton contributed by helping with the project proposal by finding applications for the model and how to convert input data into data the the CNN could learn from. Anton also contributed

by working on the data-cleaning part and doing parameter tuning for the model. He also wrote "abstract", "introduction", and contributed to the "dataset" section.

Srishti contributed by writing most of the code for the neural network and model training. She also wrote the "Model", "Missing Requirements", and "Results" section of the report, and assisted with the "Dataset" section.

Hyunchae contributed on mathematical and scientific description of CNN model. He also helped applying it to the structure of neural network and the conversion of input to three dimensional in the project. He wrote background, and conclusion.

Henry performed the small epoch evaluations of the model, and provided consultation on the network structure and theory. He edited all sections of the report and wrote up the results on the shorter epochs. He also provided an interface through which a player can actually play against the network in a game if provided with an evaluation function that maps FEN strings to numbers.

9 Acknowledgments

We acknowledge that Ronak Badhe contributed with the data set for the model.

In addition sample neural network code was referenced from a few sources to get the syntax right, including https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html and some sample code from Sandra Batista's Fall 2024 Computer Science M148 course.

Here is our github link: https://github.com/Srishtiganu/Math156_Project

10 Missing Requirements

If we had a bit more time to work on this project, we would have liked to implement a more complicated convolutional neural network, with more, larger layers. We tried to make a simpler one that would run a bit faster since we had so much data. We wanted to spend a lot more time fine-tuning our model for the Chess dataset. If we had better resources and access to better and faster GPUs to run our model with, we may have been able to properly learn all the Chess positions in the dataset. Another improvement we did not have time for was to ensure we had a downward trend in our Loss vs Epochs graph for the few large epochs. the one shown in the figure may have simply been the result of luck. It should have been at least 100 large epochs to establish a trend. With more time, we would have also tried to address the overfitting that the short epochs presented, perhaps by simply using larger sample sizes and using the extra time to get around the performance barrier this posed.

Finally, we would have liked to play against the network to see if it plays with any intelligence or not and discussed the results.

References

- [1] Ronak Badhe. Chess evaluations, 2024. Kaggle.
- [2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.