MTL782: Data Mining

Semester II, 2024-25

Assignment 1: Track 8: The King, the Queen, and the all-seeing Fish

Naman Goel 2022MT11272

Agrim Chandra 2022MT11952

Team Name: Noobs

1 Problem Statement

This assignment aims to develop a system for evaluating chess positions using modern machine learning techniques, specifically convolutional neural networks (CNNs). The primary objective is to create a model that can:

1. Take a chess position represented in Forsyth-Edwards Notation (FEN) as input.
2. Analyze the board state and assess the relative strength of each side.
3. Output a numerical score that indicates which side has an advantage and to what degree.

The score should correlate with the likelihood of winning from the given position, providing a quantitative measure of the board state's favorability for each player.

2 Data Pre-Processing

Our data preprocessing journey involved several iterations to find the most efficient and effective representation of chess positions:

1. Initially, we converted FEN notations to images for manual analysis of positions and their corresponding evaluations. This approach provided visual insights but was not suitable for large-scale processing.
2. We then explored a bitmap representation of chess positions, converting FEN notations to a binary format and saving them in a CSV file. The bitmap representation uses a 64×12 binary matrix, where each of the 64

squares is represented by 12 bits (one for each piece type and color). This matrix was then flattened into a one-dimensional array of length 768 for easier processing. However, this method presented significant challenges:

- The resulting dataset consumed excessive storage space.
- We encountered memory constraints when working with the large CSV file.
- Loading and processing times were prohibitively long.

3. To address these issues, we attempted to use the feather file format, which is optimized for fast reading and writing. Unfortunately, this approach also proved time-consuming, both in the conversion process and subsequent data usage.

4. We also experimented with algebraic notation, where the chessboard is represented by 8*8*1 tensor where corresponding to each piece their material value is given, positive for white, negative for black and zero if empty. While concise, this notation did not yield satisfactory results when fed into our neural network.

5. Finally, we settled on an efficient solution: converting FEN notations to bitmap representations on-the-fly within our code. This method eliminates the need for intermediate file storage and allows for faster processing and reduced memory usage.

This iterative process of refining our data preprocessing approach allowed us to overcome initial challenges and arrive at a solution that balances efficiency and effectiveness for our chess position evaluation task.

3 Exploratory Data Analysis (EDA)

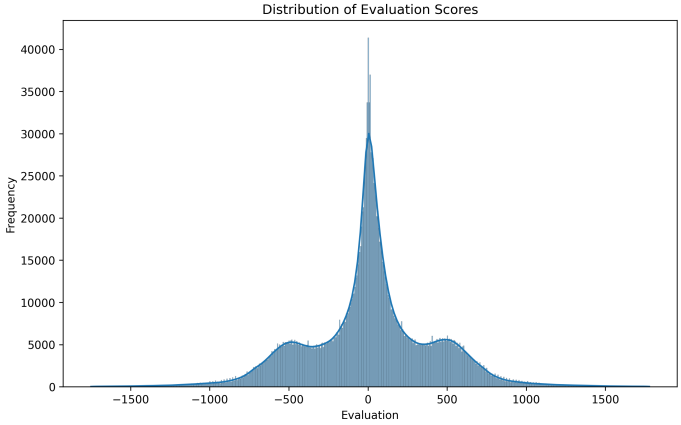
In this section, we analyze the dataset to understand the distribution of evaluated scores and explore relationships between various features of the chessboard and the evaluation scores. The analysis was conducted on three datasets: `Chess_1.csv`, `Chess_2.csv`, and `Chess_3.csv`.

3.1 Outlier Removal

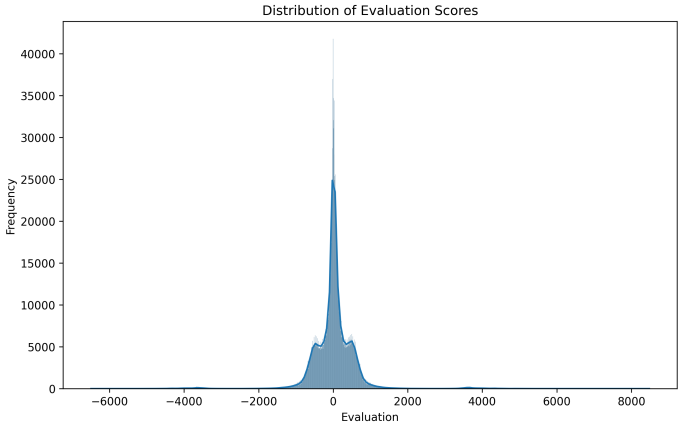
To identify and remove outliers in the evaluated scores, we used the Z-score method. Scores with a Z-score greater than 3 or less than -3 were considered outliers and removed from the dataset. This helped in ensuring that extreme values did not distort our analysis.

3.2 Distribution of Evaluated Scores

To examine the distribution of evaluated scores, histograms were plotted for each dataset both before and after removing outliers. These histograms provide insights into how evaluation scores are distributed across different chess positions.

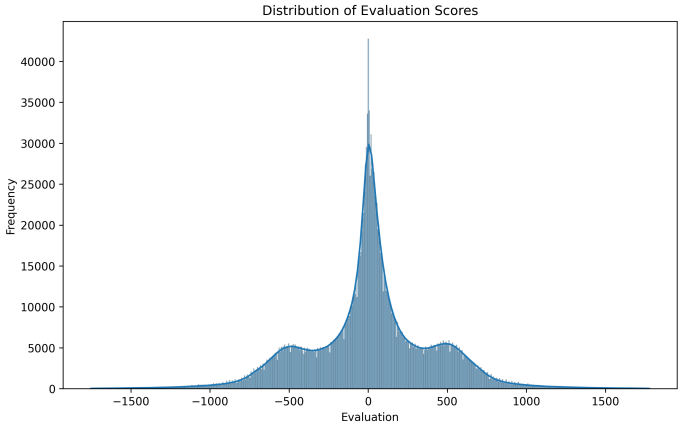


(a) Chess 1 (without outliers).

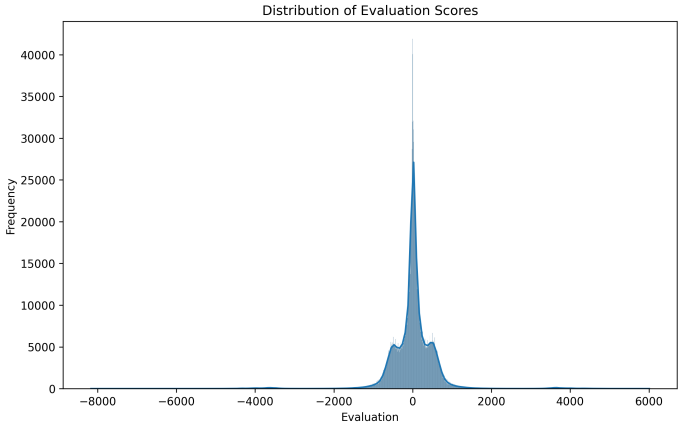


(b) Chess 1 (with outliers).

Figure 1: Histogram of Evaluated Scores (with and without outliers) for `Chess_1.csv`.

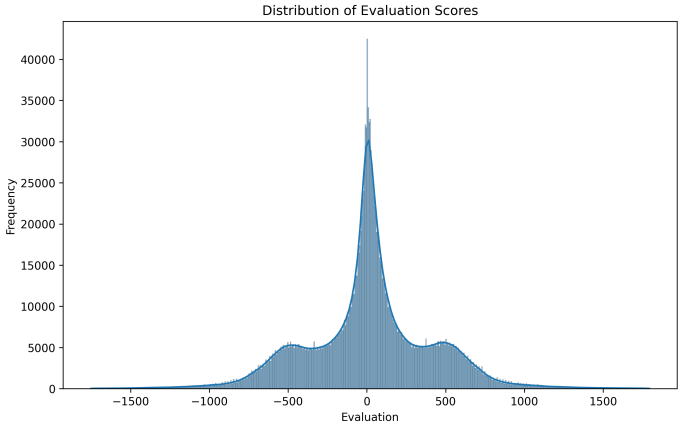


(a) Chess 2 (without outliers).

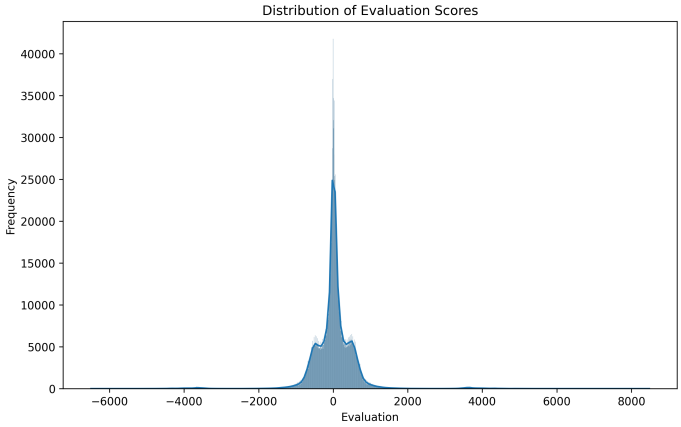


(b) Chess 2 (with outliers).

Figure 2: Histogram of Evaluated Scores (with and without outliers) for `Chess_2.csv`.



(a) Chess 3 (without outliers).



(b) Chess 3 (with outliers).

Figure 3: Histogram of Evaluated Scores (with and without outliers) for `Chess_3.csv`.

3.3 Feature Analysis

We explored several features of the chessboard to understand their relationship with evaluation scores, and scatter plots were created for each dataset- one with outliers included and one without. The features analyzed include:

3.3.1 Material Difference vs Evaluation

Material difference is calculated as the difference in material value between White and Black. The material value is assigned as follows: 9 points for a queen, 5 points for a rook, 3 points each for a bishop or knight, and 1 point for a pawn. .From the plots, we can observe that as the material difference increases,

the evaluation score also increases in favor of the side with the higher material advantage. This indicates that material difference is a significant feature in determining the evaluation of a chess position.

3.3.2 Mobility Difference vs Evaluation

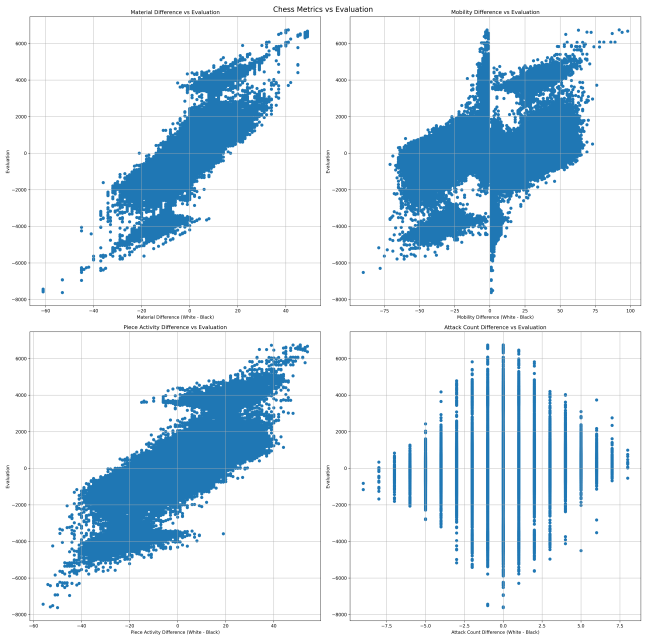
Mobility difference is defined as the difference in the number of legal moves available to White and Black. From the plots, we can observe that as the mobility difference increases, the evaluation score also increases in favor of the side with greater mobility. This highlights that mobility difference is an important feature, as having more legal moves provides greater flexibility and control over the chessboard. In practical chess play, having space and options on the board is crucial for strategic advantage.

3.3.3 Piece Activity Difference vs Evaluation

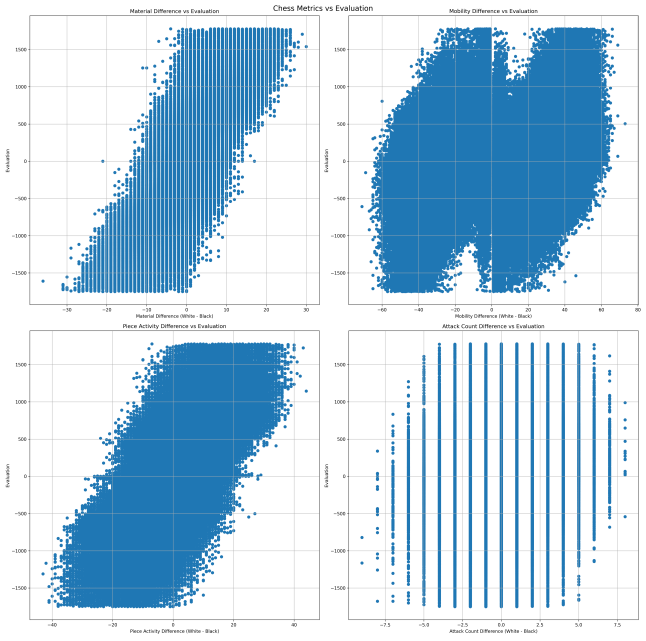
Piece activity difference measures the difference in the number of squares attacked by pieces controlled by White versus Black. From the plots, we can observe that as the piece activity difference increases, the evaluation score also increases in favor of the side with more active pieces. This demonstrates that piece activity difference is a significant feature, as having developed and active pieces on the chessboard is crucial for maintaining control and creating threats during a game of chess.

3.3.4 Attack Count Difference vs Evaluation

Attack count difference is defined as the difference between the number of Black pieces attacked by White and the number of White pieces attacked by Black. From the plots, we can observe that even though an increase in attack count difference does not lead to a significant change in evaluation scores, this feature remains important for training the model. Attacking pieces plays a critical role in practical chess, as it creates threats, forces defensive moves, and can lead to tactical opportunities.

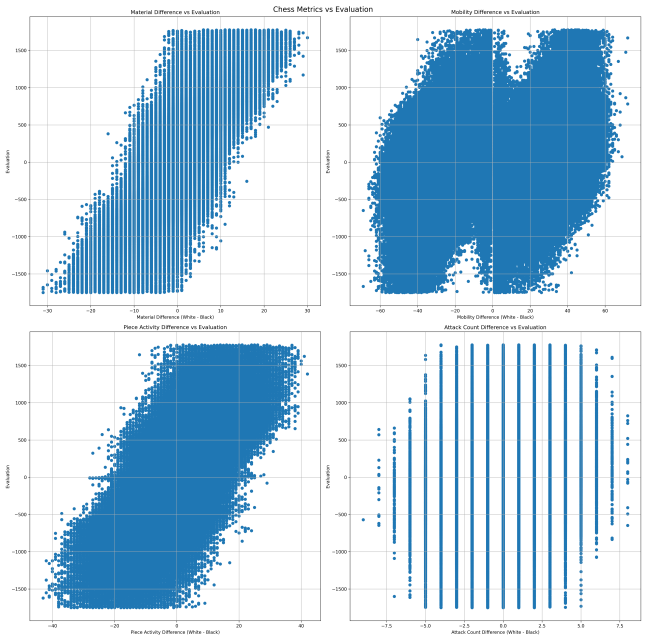


(a) Chess Metrics vs Evaluation (with outliers) for `Chess_1.csv`.

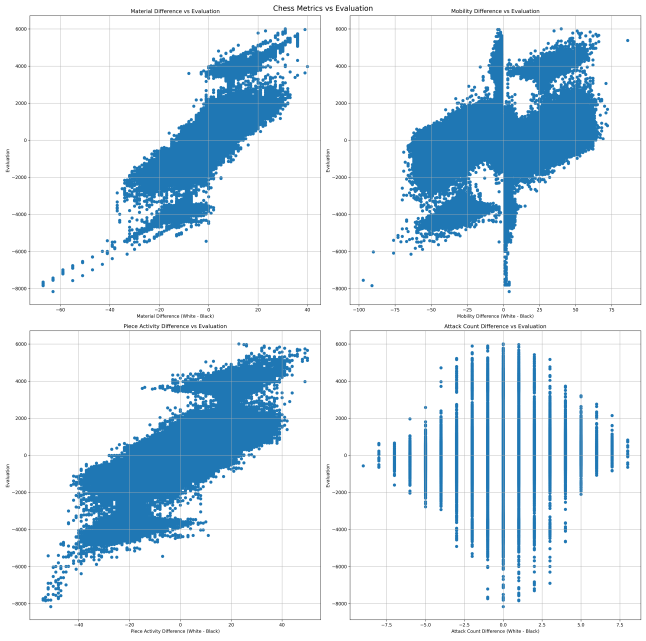


(b) Chess Metrics vs Evaluation (without outliers) for `Chess_1.csv`.

Figure 4: Comparison of Chess Metrics vs Evaluation (with and without outliers) for `Chess_1.csv`.

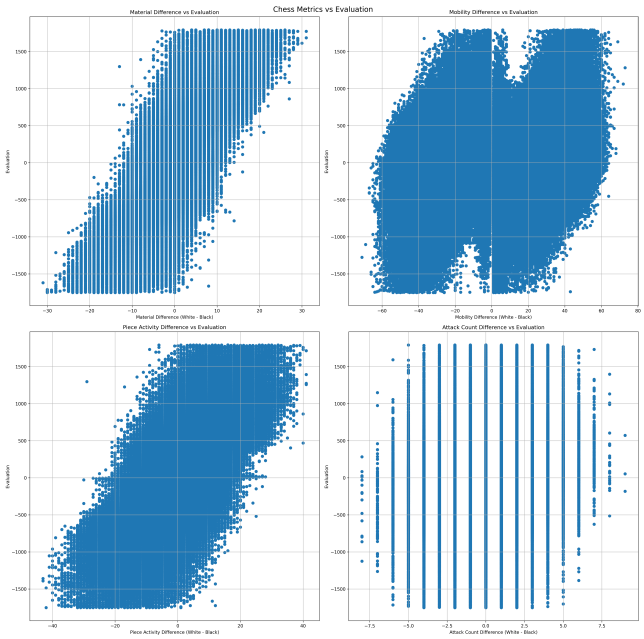


(a) Chess Metrics vs Evaluation (with outliers) for `Chess_2.csv`.

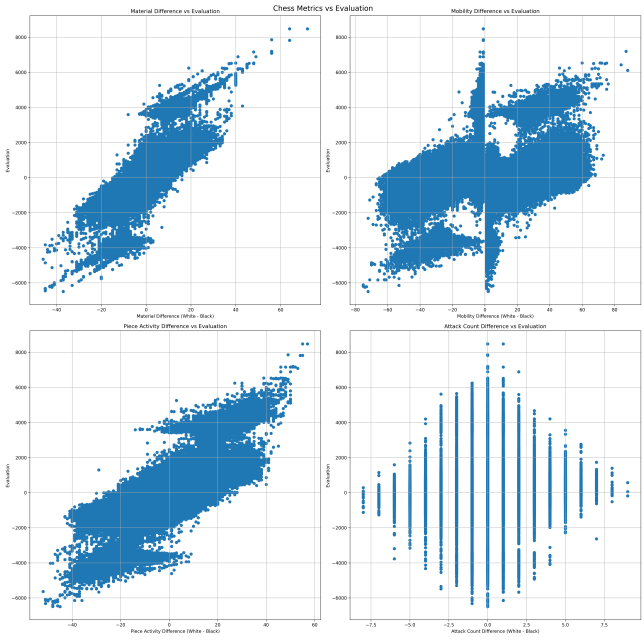


(b) Chess Metrics vs Evaluation (without outliers) for `Chess_2.csv`.

Figure 5: Comparison of Chess Metrics vs Evaluation (with and without outliers) for `Chess_2.csv`.



(a) Chess Metrics vs Evaluation (with outliers) for `Chess_3.csv`.



(b) Chess Metrics vs Evaluation (without outliers) for `Chess_3.csv`.

Figure 6: Comparison of Chess Metrics vs Evaluation (with and without outliers) for `Chess_3.csv`.

This analysis provides valuable insights into how different features influence evaluation scores, helping us better understand chess positions.

4 Feature Engineering

The different encoding methods that we tried have been discussed above in the Data Pre-Processing section.

4.1 Compare the results for different formats

We trained a Convolutional Neural Network (CNN) for 25 epochs on the Chess_1 dataset (with 2 million data points) to assess the effectiveness of CNNs for this task. The feature representation used was 8×8×18, where 12 features represented the bitmap notation of pieces, and 6 features indicated the current player's color, castling rights, and en passant availability. The model architecture consisted of 4 convolutional layers with batch normalization followed by 2 fully connected layers.

The performance of the CNN was compared with a linear neural network consisting of 4 fully connected layers with ReLU activation functions using the same features as above but flattened to a 774 feature vector. This linear network was trained for 10 epochs on a dataset of 4 million samples.

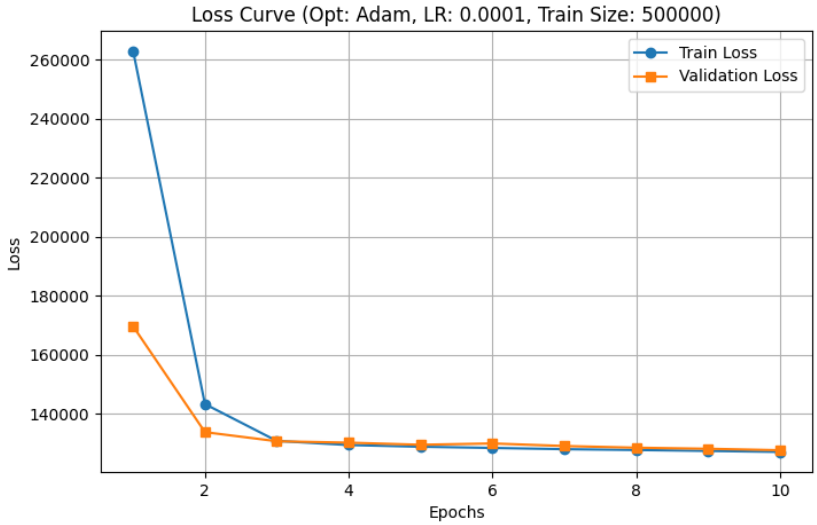
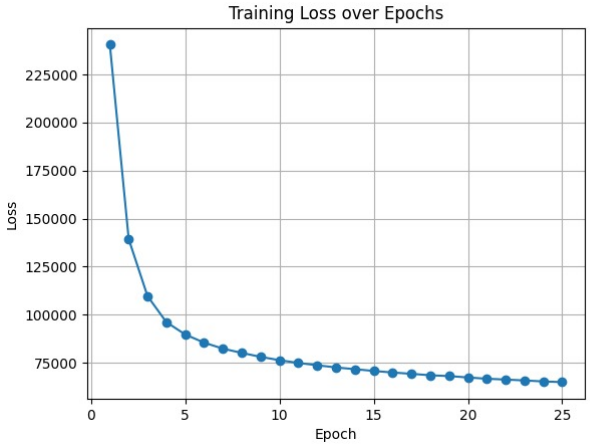


Figure 7: Feed Forward Network Training Loss



(a) CNN Training Loss

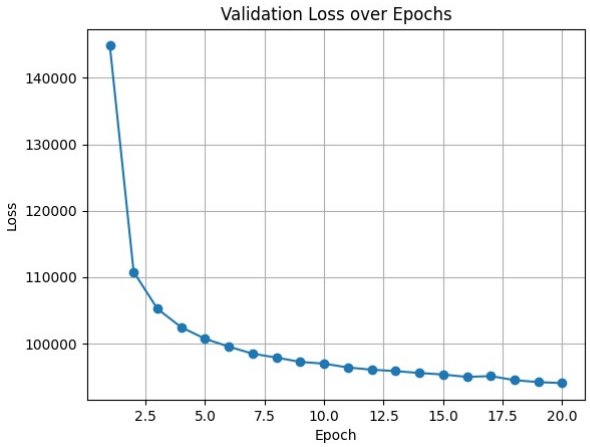
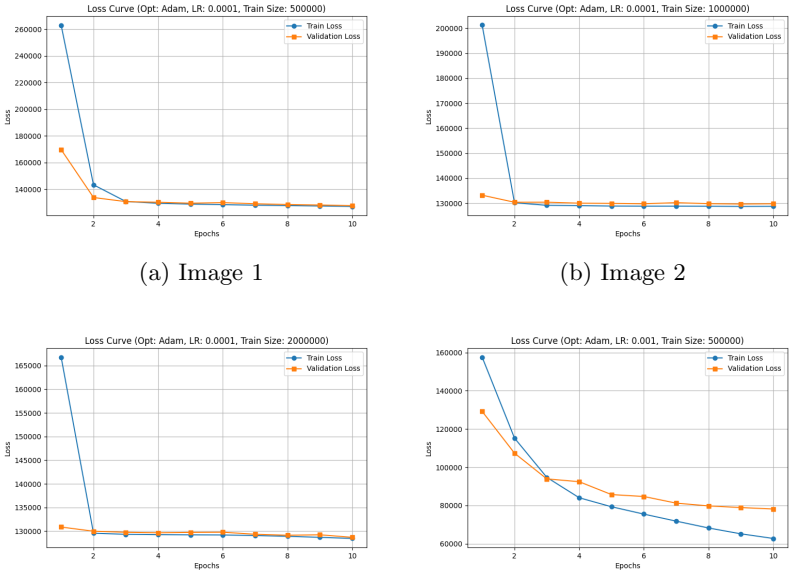


Figure 8: CNN Validation Loss

5 Neural Network Training and Analysis

5.1 Hyperparameter-Tuning

5.1.1 ADAM



(a) Image 1

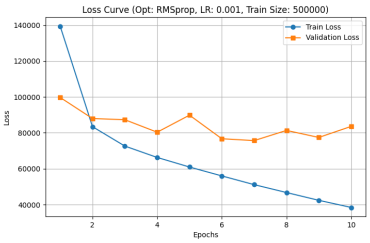
(b) Image 2

(c) Image 3

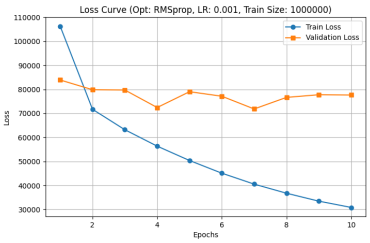
(d) Image 4

Figure 9: Hyperparameter tuning results for ADAM.

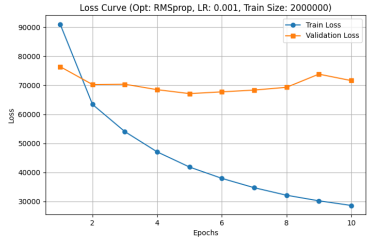
5.1.2 RMSprop



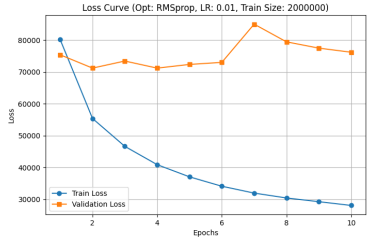
(a) Image 5



(b) Image 6



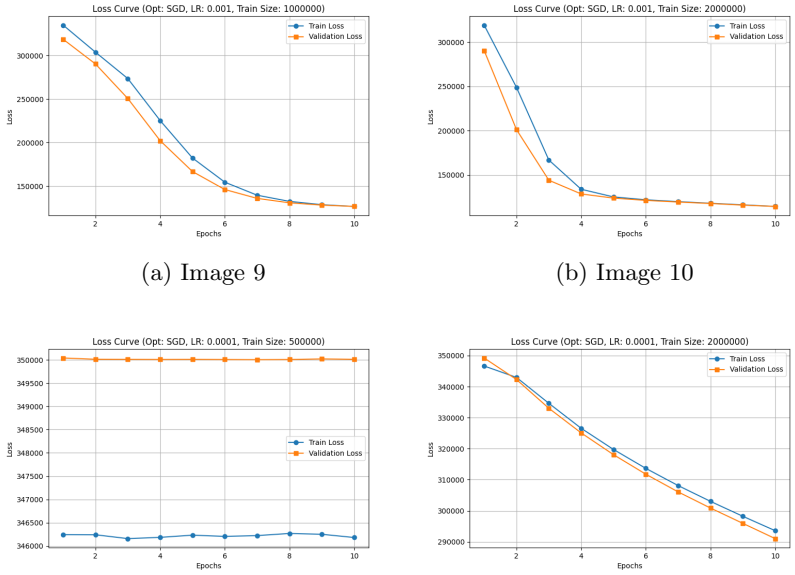
(c) Image 7



(d) Image 8

Figure 10: Hyperparameter tuning results for RMSprop.

5.1.3 SGD + Momentum



(a) Image 9

(b) Image 10

(c) Image 11

(d) Image 12

Figure 11: Hyperparameter tuning results for SGD + Momentum.

To mitigate exploding gradients and NaN losses observed during training with SGD + Momentum, **gradient clipping** was applied.

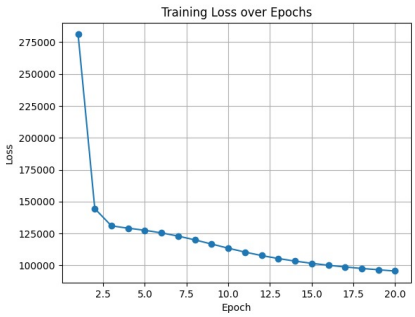
Training with limited data for multiple epochs allows the model to learn deeper patterns but increases the risk of overfitting, as it may memorize the training set rather than generalize well. On the other hand, training with more data for fewer epochs typically leads to better generalization since the model is exposed to a wider variety of examples, reducing overfitting. While more data is usually preferable, if only limited data is available, techniques like regularization, data augmentation, and early stopping can help improve performance. In general, training with more data for fewer epochs is the better approach, as it enables the model to learn more robust and transferable features which is the approach that we have adopted.

5.2 Baseline Neural Network

In the baseline neural network models, the input representation of the chessboard is designed to capture both board state and strategic features. Each board position is encoded using **768 bits**, corresponding to an $8 \times 8 \times 12$ bit representation, where 12 channels denote the presence of different piece types for both players. Additionally, some **extra features** are appended:

- **6 features** representing *colour, en passant availability, and castling rights.*
- **6 features** for strategic evaluation from White's perspective, including *material balance, mobility, piece activity, attack count, doubled pawns, and isolated pawns.*
- **6 features** for the same strategic factors from Black's perspective.

This enriched input encoding enhances the network's ability to model positional and tactical aspects of the game.



(a) Loss vs Epoch graph

Modules	Parameters
fc1.weight	823728
fc1.bias	1048
fc2.weight	536576
fc2.bias	512
fc3.weight	65536
fc3.bias	128
fc4.weight	128
fc4.bias	1

Total Trainable Params: 1427657

(b) Parameters

Figure 12: Baseline Neural Network

```
class FeedForwardNN(nn.Module):
    def _init_(self, input_size, hidden_size_1,
        hidden_size_2, hidden_size_3, output_size):
        super(FeedForwardNN, self)._init_()
        self.fc1 = nn.Linear(input_size, hidden_size_1)
        self.elu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size_1, hidden_size_2)
```

```
        self.fc3 = nn.Linear(hidden_size_2, hidden_size_3)
        self.fc4 = nn.Linear(hidden_size_3, output_size)

    def forward(self, x):
        out = self.fc4(self.elu(self.fc3(self.elu(self.fc2(
            self.elu(self.fc1(x)))))))
        return out
```

6 Competitive

6.1 Explanation of Features Used

We primarily used 18 features for representing a chessboard state. Out of these, 12 features represent the bitmap notation of each square on the chessboard, indicating which piece is present at each position. Each square is encoded with a unique value corresponding to one of the 12 types of pieces (6 for white pieces and 6 for black pieces), and a value of 0 is assigned for an empty square.

Additionally, we used 6 features to capture critical game-related information:

- **Turn indicator**: To specify whose turn it is, which plays a crucial role in evaluating the current board. This can be easily understood by noting that two successive moves for one player can provide a significant advantage in chess.
- **Castling rights**: A feature to indicate the castling rights for both the white and black kings, whether they can castle kingside or queenside.
- **En passant availability**: A single bit indicating whether en passant capturing is available or not.

All this information is encoded using the FEN, which succinctly represents the state of a chess game.

In certain models, following our EDA, we incorporated additional features to enhance the model's understanding of the position:

- **Mobility difference**: The difference in the number of legal moves available for each player's pieces.
- **Material difference**: The difference in the material (value of the pieces) between the two players.
- **Piece activity**: How actively the pieces are positioned on the board.
- **Pawn structure**: Features like the count of doubled pawns and isolated pawns, which are important structural aspects of the game.
- **Empty squares adjacent to pieces**: The number of empty squares next to each piece, which can influence piece mobility and positioning.

These additional features can be easily computed using the **python-chess** library and are widely regarded as significant by chess players, as inferred from the EDA plots.

Some of these features, such as the **attack_count difference** feature, may not show significant differences in the evaluation when considered individually. However, they can provide valuable information to the model, as it cannot inherently understand how chess pieces move and interact with each other without some form of external guidance.

While we considered these handcrafted features, we deliberately chose not to rely heavily on them due to our limited expertise in chess. Instead, our focus was on developing a neural network capable of evaluating chessboard positions without embedding explicit domain knowledge of the game.

NOTE: Unless stated otherwise, all the following models are trained on 4 million, i.e. , `Chess_1.csv` and `Chess_2.csv` combined. `Chess_3.csv` has been used for hyperparameter-tuning. In all subsequent models, gradient clipping has been implemented as a preventive measure against exploding gradients.

The final checkpoint of each of these models has been submitted in the final folder. Parameter count analysis for all models done and presented to ensure that number of parameters do not exceed no of datapoints in the dataset thus preventing overfitting.

NOTE: We have finally trained all of our models on the entire dataset along with the outliers because removing them was leading to a substanial decrease in performance. We hypothesise that this is because removing them makes it difficult for the model to genearlise for those extreme values when later seen during validation or testing leading to a decrease in performance.

6.2 CNN Squeeze and Excitation Network

The Convolutional Neural Network (CNN) with a Squeeze-and-Excitation (SE) architecture was trained for **30 epochs**. During evaluation, signs of **overfitting** were observed, with a training loss of approximately **50k** and a test loss of approximately **70k** on the Kaggle.

The input representation of the chessboard is structured as an $8 \times 8 \times 18$ tensor, where 18 channels encode various board-related features. The model was trained using the **Feather file format**, ensuring efficient storage and fast data loading. Additionally, the architecture incorporates **depthwise convolution**, improving computational efficiency while preserving spatial information.

This design allows the network to effectively capture board dynamics while optimizing model complexity and performance.

Modules	Parameters
conv1.weight	16200
conv1.bias	36
conv2.weight	16200
conv2.bias	50
se1.fc1.weight	150
se1.fc1.bias	3
se1.fc2.weight	150
se1.fc2.bias	50
depthwise.weight	450
depthwise.bias	50
pointwise.weight	3200
pointwise.bias	64
se2.fc1.weight	256
se2.fc1.bias	4
se2.fc2.weight	256
se2.fc2.bias	64
conv3.weight	16384
conv3.bias	64
fc1.weight	663552
fc1.bias	128
fc2.weight	128
fc2.bias	1

Total Trainable Params: 717440

Figure 13: Parameters

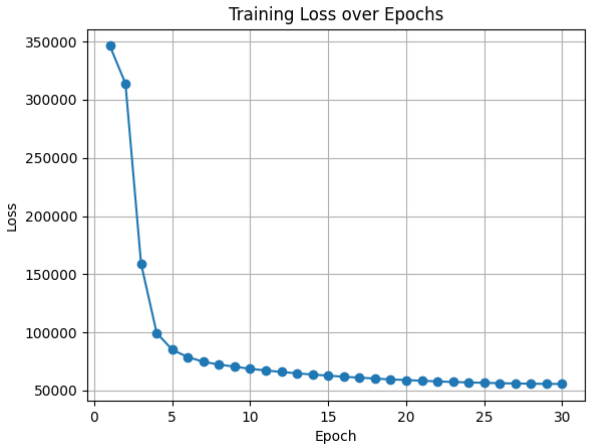


Figure 14: Loss vs Epoch graph

```python
# CNN Model
class ChessCNN(nn.Module):
    def _init_(self):
        super(ChessCNN, self)._init_()
        print("Initializing CNN model...")
        self.conv1 = nn.Conv2d(in_channels=18, out_channels=20, kernel_size=5, stride=1, padding=2)
        self.bn1 = nn.BatchNorm2d(20)
        self.conv2 = nn.Conv2d(in_channels=20, out_channels=50, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(50)
        self.conv3 = nn.Conv2d(in_channels=50, out_channels=64, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(64)
        self.conv4 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=2, stride=1, padding=1)
        self.bn4 = nn.BatchNorm2d(64)
        self.fc1 = nn.Linear(64 * 9 * 9, 128)
        self.fc2 = nn.Linear(128, 1)  # Regression output
        self.relu = nn.ReLU()
        print("CNN model initialized!")

    def forward(self, x):
        # residual1 = x
        x = self.relu(self.bn1(self.conv1(x)))
        # residual2 = x
        x = self.relu(self.bn2(self.conv2(x)))
        # x += residual2
        # residual3 = x
        x = self.relu(self.bn3(self.conv3(x)))
        # x += residual3
```

```
        x = self.relu(self.bn4(self.conv4(x)))
        x = torch.flatten(x, start_dim=1)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

class SqueezeExcitation(nn.Module):
    def _init_(self, in_channels, reduction=16):
        super(SqueezeExcitation, self)._init_()
        self.global_pool = nn.AdaptiveAvgPool2d(1)
        self.fc1 = nn.Linear(in_channels, in_channels //
            reduction)
        self.fc2 = nn.Linear(in_channels // reduction,
            in_channels)

    def forward(self, x):
        b, c, _, _ = x.size()
        squeeze = self.global_pool(x).view(b, c)
        excitation = torch.sigmoid(self.fc2(F.relu(self.fc1(
            squeeze))))
        excitation = excitation.view(b, c, 1, 1)
        return x * excitation  # Scale channels dynamically

class ChessCNN2(nn.Module):
    def _init_(self):
        super(ChessCNN2, self)._init_()
        self.conv1 = nn.Conv2d(in_channels=18, out_channels
            =36, kernel_size=5, stride=1, padding=2)
        self.conv2 = nn.Conv2d(in_channels=36, out_channels
            =50, kernel_size=3, stride=1, padding=2, dilation
            =2)
        self.se1 = SqueezeExcitation(50)
        # self.residual1 = nn.Conv2d(36, 50, kernel_size=1)
        self.depthwise = nn.Conv2d(in_channels=50,
            out_channels=50, kernel_size=3, groups=50, stride
            =1, padding=1)
        self.pointwise = nn.Conv2d(in_channels=50,
            out_channels=64, kernel_size=1)
        self.se2 = SqueezeExcitation(64)
        # self.residual2 = nn.Conv2d(50, 64, kernel_size=1)
        self.conv3 = nn.Conv2d(in_channels=64, out_channels
            =64, kernel_size=2, stride=1, padding=1)
        self.fc1 = nn.Linear(64 * 9 * 9, 128)
        self.fc2 = nn.Linear(128, 1)

    def forward(self, x):
        x = F.leaky_relu(self.conv1(x))
        x = F.leaky_relu(self.conv2(x))
        x = self.se1(x) + x
        x = self.depthwise(x)
```

```
        x = self.pointwise(x)
        x = F.leaky_relu(x)
        x = self.se2(x) + x
        x = F.leaky_relu(self.conv3(x))
        x = x.view(x.size(0), -1)
        x = F.leaky_relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

6.3 ParNet

The **ParNet** model was trained for **20 epochs**, with experiments conducted using **Huber loss** to assess its effectiveness in stabilizing training against outliers. However, the impact of Huber loss was found to be limited, with no significant improvement in training stability.

The input representation consists of an $8 \times 8 \times 18$ feature map, capturing various board-related attributes essential for position evaluation.

Despite the introduction of Huber loss, further tuning and architectural modifications may be required to mitigate the effect of outliers and enhance model robustness.

The total number of trainable parameters in the model is **4,811,329**. Due to the extensive size of the parameter list, it is not displayed here for brevity.

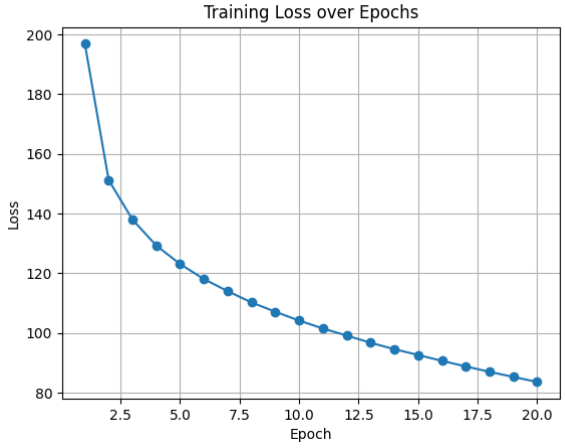


Figure 15: Loss vs Epoch graph

```
class SSEBlock(nn.Module):
    def _init_(self, in_channels, out_channels):
        super(SSEBlock, self)._init_()
        self.bn = nn.BatchNorm2d(in_channels)
        self.global_pool = nn.AdaptiveAvgPool2d(1)
```

```
        self.conv = nn.Conv2d(in_channels, out_channels,
            kernel_size=1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.bn(x)
        scale = self.global_pool(x)
        scale = self.conv(scale)
        scale = self.sigmoid(scale)
        return x * scale

class BaseConv(nn.Module):
    def _init_(self, in_channels, out_channels, ksize,
        stride, groups=1):
        super(BaseConv, self)._init_()
        # same output size
        pad = (ksize - 1) // 2
        self.conv = nn.Conv2d(
            in_channels,
            out_channels,
            kernel_size=ksize,
            stride=stride,
            padding=pad,
            groups=groups
        )
        self.bn = nn.BatchNorm2d(out_channels)
    def forward(self, x):
        return self.bn(self.conv(x))

class Down_sampling(nn.Module):
    def _init_(self, in_channels, out_channels):
        super(Down_sampling,self)._init_()
        # self.avg_pool = nn.AvgPool2d(kernel_size=2)
        self.Conv_1_1 = BaseConv(in_channels=in_channels,
            out_channels=out_channels, ksize=1, stride=1)
        self.Conv_3 = BaseConv(in_channels=in_channels,
            out_channels=out_channels, ksize=3, stride=1)
        self.global_avg_pool = nn.AdaptiveAvgPool2d((None,
            None))
        self.Conv_1_2 = nn.Conv2d(in_channels=in_channels,
            out_channels=out_channels, kernel_size=1, stride
            =1)
        self.act_1 = nn.Sigmoid()
        self.act_2 = nn.SiLU()
    def forward(self, x):
        # x_1 = self.avg_pool(x)
        x_1 = self.Conv_1_1(x)
        x_3 = self.Conv_3(x)
        x_13 = x_1 + x_3
        x_se = self.global_avg_pool(x)
```

```
        x_se = self.Conv_1_2(x_se)
        x_se = self.act_1(x_se)
        out = torch.mul(x_13, x_se)
        out = self.act_2(out)
        return out

class Fusion(nn.Module):
    def _init_(self, in_channels, out_channels):
        super(Fusion,self)._init_()
        self.bn1 = nn.BatchNorm2d(in_channels)
        self.bn2 = nn.BatchNorm2d(in_channels)
        # self.avg_pool = nn.AvgPool2d(kernel_size=1)
        self.Conv_1_1 = BaseConv(in_channels=in_channels*2,
            out_channels=out_channels, ksize=1, stride=1,
            groups=2)
        self.Conv_3 = BaseConv(in_channels=in_channels*2,
            out_channels=out_channels, ksize=3, stride=1,
            groups=2)
        self.global_avg_pool = nn.AdaptiveAvgPool2d((None,
            None))
        self.Conv_1_2 = nn.Conv2d(in_channels=in_channels*2,
             out_channels=out_channels, kernel_size=1, stride
            =1)
        self.act_1 = nn.Sigmoid()
        self.act_2 = nn.SiLU()

    def forward(self, x1, x2):
        x1 = self.bn1(x1)
        x2 = self.bn2(x2)
        x = torch.cat((x1,x2), dim=1)
        # idx = torch.randperm(x.nelement())
        # x = x.view(-1)[x].view(x.size())
        # x_1 = self.avg_pool(x)
        x_1 = self.Conv_1_1(x)
        x_3 = self.Conv_3(x)
        x_13 = x_1 + x_3
        x_se = self.global_avg_pool(x)
        x_se = self.Conv_1_2(x_se)
        x_se = self.act_1(x_se)
        out = torch.mul(x_13, x_se)
        out = self.act_2(out)
        return out

class Par_block(nn.Module):
    def _init_(self, in_channels, out_channels):
        super(Par_block,self)._init_()
        self.Conv_1 = BaseConv(in_channels=in_channels,
            out_channels=out_channels, ksize=1, stride=1)
        self.Conv_2 = BaseConv(in_channels=in_channels,
            out_channels=out_channels, ksize=3, stride=1)
```

```
90          self.SSE = SSEBlock(in_channels=in_channels,
                out_channels=out_channels)
91          self.act = nn.SiLU()
92      def forward(self, x):
93          x_1 = self.Conv_1(x)
94          x_3 = self.Conv_2(x)
95          x_se = self.SSE(x)
96          out = x_1 + x_3 + x_se
97          out = self.act(out)
98          return out
99
100 # 18 -> 64 -> 128 -> 256
101 class Parnet(nn.Module):
102     def _init_(self, input_channels, hidden_1, hidden_2,
            hidden_3):
103         super(Parnet,self)._init_()
104         self.downsamping_2 = Down_sampling(in_channels=
                input_channels, out_channels=hidden_1)
105         self.downsamping_3 = Down_sampling(in_channels=
                hidden_1, out_channels=hidden_2)
106         self.downsamping_4 = Down_sampling(in_channels=
                hidden_2, out_channels=hidden_3)
107
108         self.Stream_1 = nn.Sequential(
109                                     Par_block(in_channels=
                                        hidden_1,out_channels
                                        =hidden_1),
110                                     Par_block(in_channels=
                                        hidden_1,
                                        out_channels=hidden_1
                                        ),
111         )
112         self.S1_downsamping = Down_sampling(in_channels=
                hidden_1,out_channels=hidden_2)
113
114         self.Stream_2 = nn.Sequential(
115                                     Par_block(in_channels=
                                        hidden_2,
                                        out_channels=hidden_2
                                        ),
116                                     Par_block(in_channels=
                                        hidden_2,
                                        out_channels=hidden_2
                                        ),
117                                     Par_block(in_channels=
                                        hidden_2,
                                        out_channels=hidden_2
                                        ),
118         )
119         self.S2_Fusion = Fusion(in_channels=hidden_2,
```

```
            out_channels=hidden_3)

        self.Stream_3 = nn.Sequential(
                                    Par_block(in_channels=
                                        hidden_3,
                                        out_channels=hidden_3
                                        ),
                                    Par_block(in_channels=
                                        hidden_3,
                                        out_channels=hidden_3
                                        ),
                                    Par_block(in_channels=
                                        hidden_3,
                                        out_channels=hidden_3
                                        ),
        )
        self.S3_Fusion = Fusion(in_channels=hidden_3,
            out_channels=hidden_3)

        self.global_pool = nn.AdaptiveAvgPool2d(1)
        self.fc1 = nn.Linear(hidden_3, 1024)
        self.fc2 = nn.Linear(1024, 1)
        self.silu = nn.SiLU()

    def forward(self,x):
        x = self.downsamping_2(x)

        x_s1 = self.S1_downsamping(self.Stream_1(x))

        x = self.downsamping_3(x)
        x_s2 = self.Stream_2(x)
        x_s12 = self.S2_Fusion(x_s1, x_s2)

        x = self.downsamping_4(x)
        x_s3 = self.Stream_3(x)
        x_s123 = self.S3_Fusion(x_s12,x_s3)

        x_out = torch.flatten(self.global_pool(x_s123),
            start_dim=1)
        x_out = self.silu(self.fc1(x_out))
        x_out = self.fc2(x_out)
        return x_out
```

6.4 CNN CBAM 3 layers

In the Convolutional Neural Network (CNN) with Convolutional Block Attention Module (CBAM) architecture consisting of three layers, the input representation of the chessboard is structured as follows: Each board position is encoded using an $8 \times 8 \times 12$ bit representation, where 12 channels correspond to

the presence of different piece types for both players. Additionally, **six extra features**—representing *colour, en passant availability, and castling rights*—are appended to each cell, enriching the input representation for enhanced model performance.

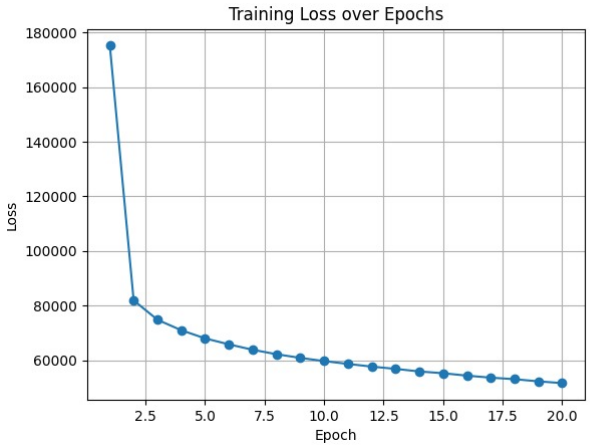


Figure 16: Loss vs Epoch graph

Modules	Parameters
conv1.weight	14400
conv1.bias	32
CBAM1.channel_attention.fc.0.weight	256
CBAM1.channel_attention.fc.2.weight	256
CBAM1.spatial_attention.conv.weight	98
bn1.weight	32
bn1.bias	32
conv2.weight	18432
conv2.bias	64
CBAM2.channel_attention.fc.0.weight	512
CBAM2.channel_attention.fc.2.weight	512
CBAM2.spatial_attention.conv.weight	98
bn2.weight	64
bn2.bias	64
conv3.weight	73728
conv3.bias	128
CBAM3.channel_attention.fc.0.weight	1024
CBAM3.channel_attention.fc.2.weight	1024
CBAM3.spatial_attention.conv.weight	98
bn3.weight	128
bn3.bias	128
fc1.weight	8388608
fc1.bias	1024
fc2.weight	524288
fc2.bias	512
fc3.weight	131072
fc3.bias	256
fc4.weight	256
fc4.bias	1

Total Trainable Params: 9157127

Figure 17: Parameters

```
class ChessCNN(nn.Module):
    def _init_(self):
        super(ChessCNN, self)._init_()
        print("Initializing␣CNN␣model...")
        self.conv1 = nn.Conv2d(in_channels=18, out_channels
            =32, kernel_size=5, stride=1, padding=2)
        self.CBAM1 = CBAM(in_channels=32, reduction=4)
        self.bn1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels
            =64, kernel_size=3, stride=1, padding=1)
        self.CBAM2 = CBAM(in_channels=64, reduction=8)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(in_channels=64, out_channels
```

```
            =128, kernel_size=3, stride=1, padding=1)
        self.CBAM3 = CBAM(in_channels=128, reduction=16)
        self.bn3 = nn.BatchNorm2d(128)
        self.fc1 = nn.Linear(128 * 8 * 8, 1024)
        self.fc2 = nn.Linear(1024, 512)
        self.fc3 = nn.Linear(512, 256)
        self.fc4 = nn.Linear(256, 1)  # Regression output
        self.relu = nn.ReLU()
        print("CNN model initialized!")

    def forward(self, x):
        x = self.relu(self.bn1(self.CBAM1(self.conv1(x))))
        x = self.relu(self.bn2(self.CBAM2(self.conv2(x))))
        x = self.relu(self.bn3(self.CBAM3(self.conv3(x))))
        x = x.view(x.size(0), -1)  # Flatten
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        x = self.fc4(x)
        return x
```

6.5 CNN CBAM 5 layers (Best Performance Model)

The Convolutional Neural Network (CNN) with Convolutional Block Attention Module (CBAM) architecture, consisting of five layers, was initially trained on `Chess_1` and `Chess_2` for **18 epochs**. Following this, the first three convolutional blocks were **frozen**, and the model was fine-tuned on the remaining dataset, `Chess_3`, for **3 epochs**.

The input representation of the chessboard is structured as an $8 \times 8 \times 17$ tensor, where:

- **12 channels** represent the presence of different piece types for both players in a bitmap format.
- **1 channel** encodes the number of attackers for each square.
- **1 channel** encodes the number of defenders for each square.
- **1 channel** indicates whether a neighboring square is empty.
- **2 channels** represent the material weight, normalized by 39.

This enriched input encoding enhances the model's ability to capture both tactical and strategic aspects of board evaluation.

6.5.1 Justification of architecture

Drawing from results of CNN CBAM 3 layer network, we improve upon by manually adding layers to increase the modelling power of the network and place

more parameters in the convolutional layers as compared to the fully connected layers.

The architecture is well-suited for evaluating chessboard positions as it effectively captures spatial relationships and piece interactions, which are critical in chess. The five-layer CNN extracts hierarchical features, while CBAM enhances feature representation by focusing on important board regions. Freezing initial convolutional layers during fine-tuning helps retain learned patterns while adapting to new data. The input encoding ensures a rich representation, incorporating piece placement, attack/defense dynamics, and positional factors. This structured input allows the network to learn both tactical motifs and strategic principles, making it a robust model for chess evaluation.

The training for this auto terminated on Kaggle after 18 epochs (originally scheduled for 20 epochs) and so the training loss plot could not be produced. The final submitted entry on Kaggle corresponds to 36th epoch of this model (without any fine tuning),

Modules	Parameters
preprocess.conv.weight	289
preprocess.bn.weight	17
preprocess.bn.bias	17
conv1.weight	13600
conv1.bias	32
bn1.weight	32
bn1.bias	32
CBAM1.channel_attention.fc.0.weight	256
CBAM1.channel_attention.fc.2.weight	256
CBAM1.spatial_attention.conv.weight	98
conv2.weight	18432
conv2.bias	64
bn2.weight	64
bn2.bias	64
CBAM2.channel_attention.fc.0.weight	1024
CBAM2.channel_attention.fc.2.weight	1024
CBAM2.spatial_attention.conv.weight	98
conv3.weight	73728
conv3.bias	128
bn3.weight	128
bn3.bias	128
CBAM3.channel_attention.fc.0.weight	1024
CBAM3.channel_attention.fc.2.weight	1024
CBAM3.spatial_attention.conv.weight	98
conv4.weight	294912
conv4.bias	256
bn4.weight	256
bn4.bias	256
CBAM4.channel_attention.fc.0.weight	4096
CBAM4.channel_attention.fc.2.weight	4096
CBAM4.spatial_attention.conv.weight	98
conv5.weight	32768
conv5.bias	128
bn5.weight	128
bn5.bias	128
fc1.weight	524288
fc1.bias	256
fc2.weight	32768
fc2.bias	128
fc3.weight	128
fc3.bias	1

Total Trainable Params: 1006348

(a) Parameters

Figure 18: CNN CBAM 5 layers

```
# Channel Attention Module
class ChannelAttention(nn.Module):
    def _init_(self, in_channels, reduction=16):
        super(ChannelAttention, self)._init_()
        self.global_avg_pool = nn.AdaptiveAvgPool2d(1)
        self.global_max_pool = nn.AdaptiveMaxPool2d(1)
        self.fc = nn.Sequential(
            nn.Conv2d(in_channels, in_channels // reduction,
                kernel_size=1, bias=False),
            nn.ReLU(),
            nn.Conv2d(in_channels // reduction, in_channels,
                kernel_size=1, bias=False),
        )
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        avg_out = self.fc(self.global_avg_pool(x))
```

```
        max_out = self.fc(self.global_max_pool(x))
        out = self.sigmoid(avg_out + max_out)
        return x * out

# Spatial Attention Module
class SpatialAttention(nn.Module):
    def _init_(self):
        super(SpatialAttention, self)._init_()
        self.conv = nn.Conv2d(2, 1, kernel_size=7, padding
            =3, bias=False)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        avg_out = torch.mean(x, dim=1, keepdim=True)
        max_out, _ = torch.max(x, dim=1, keepdim=True)
        combined = torch.cat([avg_out, max_out], dim=1)
        out = self.sigmoid(self.conv(combined))
        return x * out

# CBAM Block
class CBAM(nn.Module):
    def _init_(self, in_channels, reduction=16):
        super(CBAM, self)._init_()
        self.channel_attention = ChannelAttention(
            in_channels, reduction)
        self.spatial_attention = SpatialAttention()

    def forward(self, x):
        x = self.channel_attention(x)
        x = self.spatial_attention(x)
        return x

# Preprocessing Block to handle different types of input
    features
class PreprocessCNN(nn.Module):
    def _init_(self, in_channels):
        super(PreprocessCNN, self)._init_()
        self.conv = nn.Conv2d(in_channels, in_channels,
            kernel_size=1, bias=False)  # Lightweight
            transformation
        self.bn = nn.BatchNorm2d(in_channels)
        self.relu = nn.Mish()

    def forward(self, x):
        return self.relu(self.bn(self.conv(x)))

# Main ChessCNN Model with Dropout
class ChessCNN(nn.Module):
    def _init_(self):
        super(ChessCNN, self)._init_()
```

```
        print("Initializing␣CNN␣model...")

        self.preprocess = PreprocessCNN(in_channels=17)  # Preprocessing input

        self.conv1 = nn.Conv2d(in_channels=17, out_channels=32, kernel_size=5, stride=1, padding=2)
        self.bn1 = nn.BatchNorm2d(32)
        self.dropout1 = nn.Dropout2d(p=0.2)  # Dropout after first conv block
        self.CBAM1 = CBAM(in_channels=32, reduction=4)

        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.dropout2 = nn.Dropout2d(p=0.2)  # Dropout after second conv block
        self.CBAM2 = CBAM(in_channels=64, reduction=4)

        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.dropout3 = nn.Dropout2d(p=0.2)  # Dropout after third conv block
        self.CBAM3 = CBAM(in_channels=128, reduction=16)  # CBAM on deeper layers

        self.conv4 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1, padding=1)
        self.bn4 = nn.BatchNorm2d(256)
        self.dropout4 = nn.Dropout2d(p=0.2)  # Dropout after fourth conv block
        self.CBAM4 = CBAM(in_channels=256, reduction=16)

        # Reduce feature size before FC
        self.conv5 = nn.Conv2d(in_channels=256, out_channels=128, kernel_size=1)  # Reduce channels
        self.bn5 = nn.BatchNorm2d(128)

        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout5 = nn.Dropout2d(p=0.2)  # Dropout after final pooling

        # Fully Connected Layers
        self.fc1 = nn.Linear(128 * 4 * 4, 256)  # Reduced FC layer
        self.dropout_fc1 = nn.Dropout(p=0.5)  # Dropout in FC
        self.fc2 = nn.Linear(256, 128)
        self.dropout_fc2 = nn.Dropout(p=0.5)  # Dropout in
```

```
            FC
        self.fc3 = nn.Linear(128, 1)  # Regression output

        self.relu = nn.Mish()
        print("CNN model initialized!")

    def forward(self, x):
        x = self.preprocess(x)  # Preprocess input features

        x = self.relu(self.bn1(self.conv1(x)))
        x = self.dropout1(x)
        x = self.CBAM1(x)

        x = self.relu(self.bn2(self.conv2(x)))
        x = self.dropout2(x)
        x = self.CBAM2(x)

        x = self.relu(self.bn3(self.conv3(x)))
        x = self.dropout3(x)
        x = self.CBAM3(x)

        x = self.relu(self.bn4(self.conv4(x)))
        x = self.dropout4(x)
        x = self.CBAM4(x)

        x = self.relu(self.bn5(self.conv5(x)))
        x = self.maxpool(x)  # Max pooling (2*2)
        x = self.dropout5(x)

        x = torch.flatten(x, start_dim=1)  # Flatten before
            FC layers

        x = self.relu(self.fc1(x))
        x = self.dropout_fc1(x)

        x = self.relu(self.fc2(x))
        x = self.dropout_fc2(x)

        x = self.fc3(x)  # Output regression value

        return x
```

We also experimented with transformer and Vision transformers as model considering their attention architecure to be able to model the global context of the board and be better representers of the chess board position but their training timed out on Kaggle after running for about 12 hours each (almost 3 epochs). The implementation for these is available in the folder on google drive link.

We also experimented on training a linear layer connected to the output of pre-trained ResNet model for comparison purposes. This was not submited on Kaggle. Please find the implementations in google drive link.

NOTE: It can be observed that our introduced features were able to increase accuracy of the models.