The King, the Queen, and the all-seeing Fish

Submitted by

Rohin Garg (2023MAS7118)
Harshit Joshi (2023MAS7141)
Ananya Sharma (2023MAS7117)

Submitted to

Prof. Niladri Chatterjee

Department of Mathematics

INDIAN INSTITUTE OF TECHNOLOGY DELHI
HAUZ KHAS, NEW DELHI-110016 INDIA

February 9, 2025

1 Data Pre-processing

The 4 CSV data files are loaded into pandas dataframes:

```
csv1, csv2, csv3 = 'Chess_1.csv', 'Chess_2.csv', 'Chess_3.csv'
csvtest = 'test_track_8.csv'
df1, df2, df3 = pd.read_csv(csv1), pd.read_csv(csv2), pd.read_csv(csv3)
df4 = pd.read_csv(csvtest)
```

To ensure efficiency, we selected the first 1,00,000 rows from each of the dataframes and concatenated the training data:

```
    df1_s, df2_s, df3_s, df4_s = [df.iloc[:100000, :] for df in [df1, df2, df3, df4]]
    df = pd.concat([df1_s, df2_s, df3_s], axis=0).reset_index(drop=True)
```

To parse the FEN strings, we split it into separate columns. The 'next' column is encoded numerically, castling rights translate to binary indicators, and finally half and full moves also get encoded as integers.

```
    fen_cols = ['FEN_board', 'next', 'CA', '_', 'hm', 'fm']
    df[fen_cols] = df['FEN'].str.split(' ', expand=True)
    df.drop(['_', 'FEN'], axis=1, inplace=True)
    df['next'] = df['next'].map({'b': -1, 'w': 1}).astype(np.int8)
    castlingdict = {'K': 'CA_K', 'Q': 'CA_Q', 'k': 'CA_k', 'q': 'CA_q'}
    for char, col in castlingdict.items():
        df[col] = df['CA'].str.contains(char).astype(np.int8)
    df.drop('CA', axis=1, inplace=True)
    df['hm'] = df['hm'].astype(np.int8)
    df['fm'] = df['fm'].astype(np.int8)
```

After this, a function is defined to count the number of pieces for each colour, which is then applied to our data and added as new columns in our dataframe.

```
    def numbw(fen_str):
    blk= {
      'p': 0,
      'r': 0,
      'n': 0,
      'b': 0,
      'q': 0,
      'k': 0,
    }
    wht = {
      'P': 0,
      'R': 0,
      'N': 0,
      'B': 0,
      'Q': 0,
      'K': 0,
    }
    for c in fen_str:
        if c.islower():  # Check if piece is black
            blk[c] = blk.get(c, 0) + 1
        elif c.isupper():  # Check if piece is white
            wht[c] = wht.get(c, 0) + 1

    return blk, wht
```

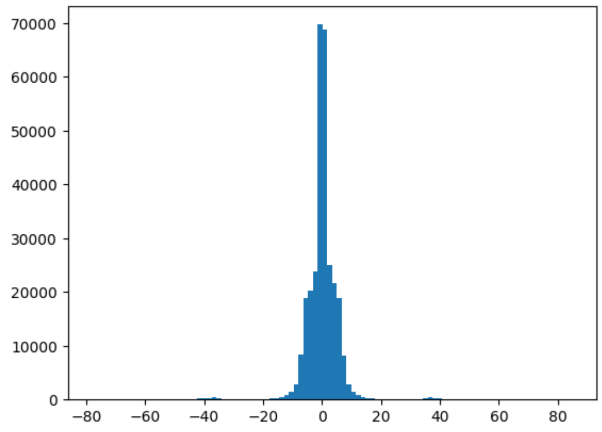
2 Exploratory Data Analysis (EDA)

Before we begin, we removed outliers from our data on the basis of their Z-score, removing all those data points for whom the absolute value of the Z-score is greater than 3 (i.e. all data that is more than 3 standard deviations away from the mean).

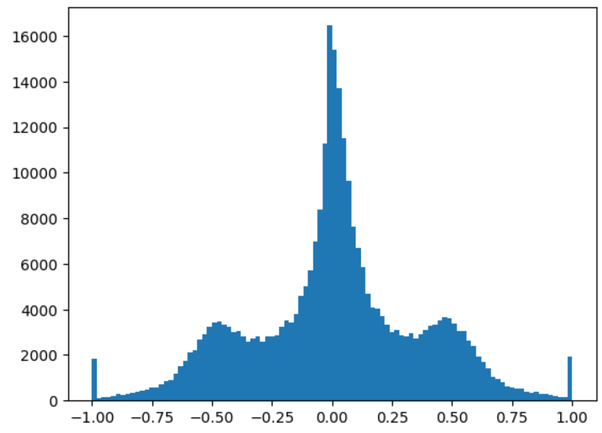
```
from scipy.stats import zscore

df['Z_score'] = zscore(df['Evaluation'])
```

```
print("Data before removing outliers:", len(df))
df4 = df[(df['Z_score'] <= 3) & (df['Z_score'] >= -3)]
print("Data after removing outliers:", len(df4))

Data before removing outliers: 300000
Data after removing outliers: 295749
```

First, we plotted the distribution of the evaluated scores (here we have divided by 100 since the evaluation scores are in centipawns). The evaluations display a strong central tendency as expected with a large cluster of observations around 0: a random sample of legal chess board configurations would not be expected to contain a lot of extremal evaluations. Furthermore, the data is largely symmetric about 0.



We also plotted the distribution of the evaluated scores after normalizing by dividing by 994.33977 and then applying the *tanh* function. The scaling factor of 994.33977 was chosen via optimization, as we explain below. The data retains both the clustering around 0 as well as the symmetry about 0.

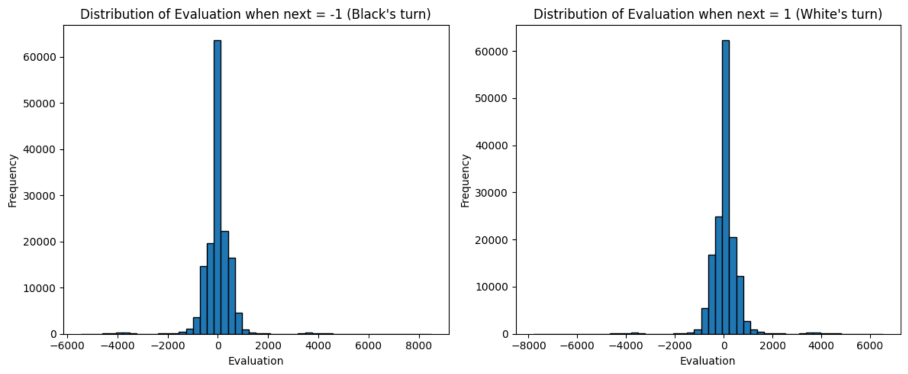


We arrived at the scaling value of 994.33977 by treating the scaling factor as a hyperparameter and applying hyper-parameter tuning for 20 trials over the value of k to obtain a value of k with for which a naive model gave us the least loss.

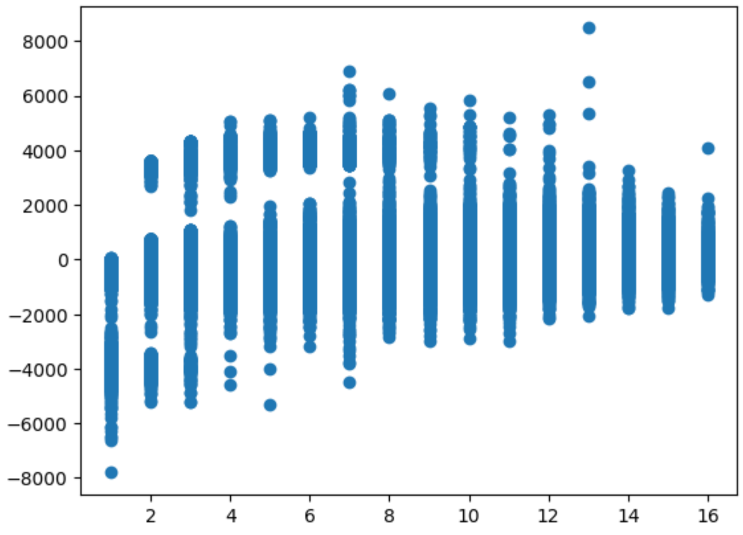
```
    def objective(k):
    normalizer = TanhNormalizer(k)
    y_train_normalized = normalizer.normalize(y_train.numpy())
    y_test_normalized = normalizer.normalize(y_test.numpy())

    y_train_normalized = torch.tensor(y_train_normalized, dtype=torch.float32)
    y_test_normalized = torch.tensor(y_test_normalized, dtype=torch.float32)
    model = ChessEvaluator()
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.0001)

    for epoch in range(20):
        optimizer.zero_grad()
        outputs = model(X_train)
        loss = criterion(outputs, y_train_normalized)
        loss.backward()
        optimizer.step()

    with torch.no_grad():
        predictions_normalized = model(X_test)
        test_loss = criterion(predictions_normalized, y_test_normalized)
    return test_loss.item()


    import optuna

def optuna_objective(trial):
    k = trial.suggest_float('k', -10, 10)
    return objective(k)

study = optuna.create_study(direction='minimize')
study.optimize(optuna_objective, n_trials=20)

print("Best k:", study.best_params['k'])
print("Best test loss:", study.best_value)
```

Running the optimisation provided us with a similar value of 994.9404061511041, indicating the robustness of our chosen value.

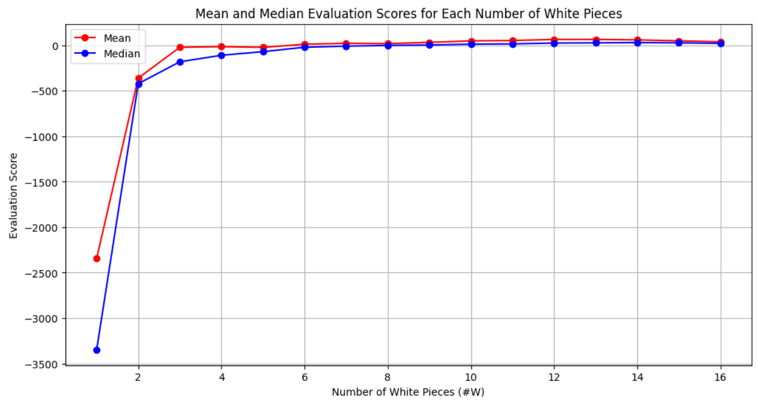
Next, plotted histograms for evaluations for which it was white (next = 1) and black's (next = -1) turn to move respectively. Both the distributions of evaluation scores retained a sharp central peak around 0 as well as the symmetricity of the previous graphs, with tails extending to extreme positive and negative values. As might be expected, for Black's turn we obtain a slightly broader right tail, indicating an advantage for white (which may have been accrued as a result of having played an extra move). Similarly, White's turn has a a slightly more pronounced left tail.



As one would expect, White's evaluation scores decrease as the number of White pieces decreases. Furthermore, the White has higher extreme evaluations in the 10-14 pieces range (compared to a full set of 16 pieces) - this too is in accordance with one's naive intuition, as the 16 piece situations likely correspond to Openings wherein there is not much advantage to be found, whereas the 10-14 piece range likely correspond to a middlegame wherein some advantage may have been obtained.

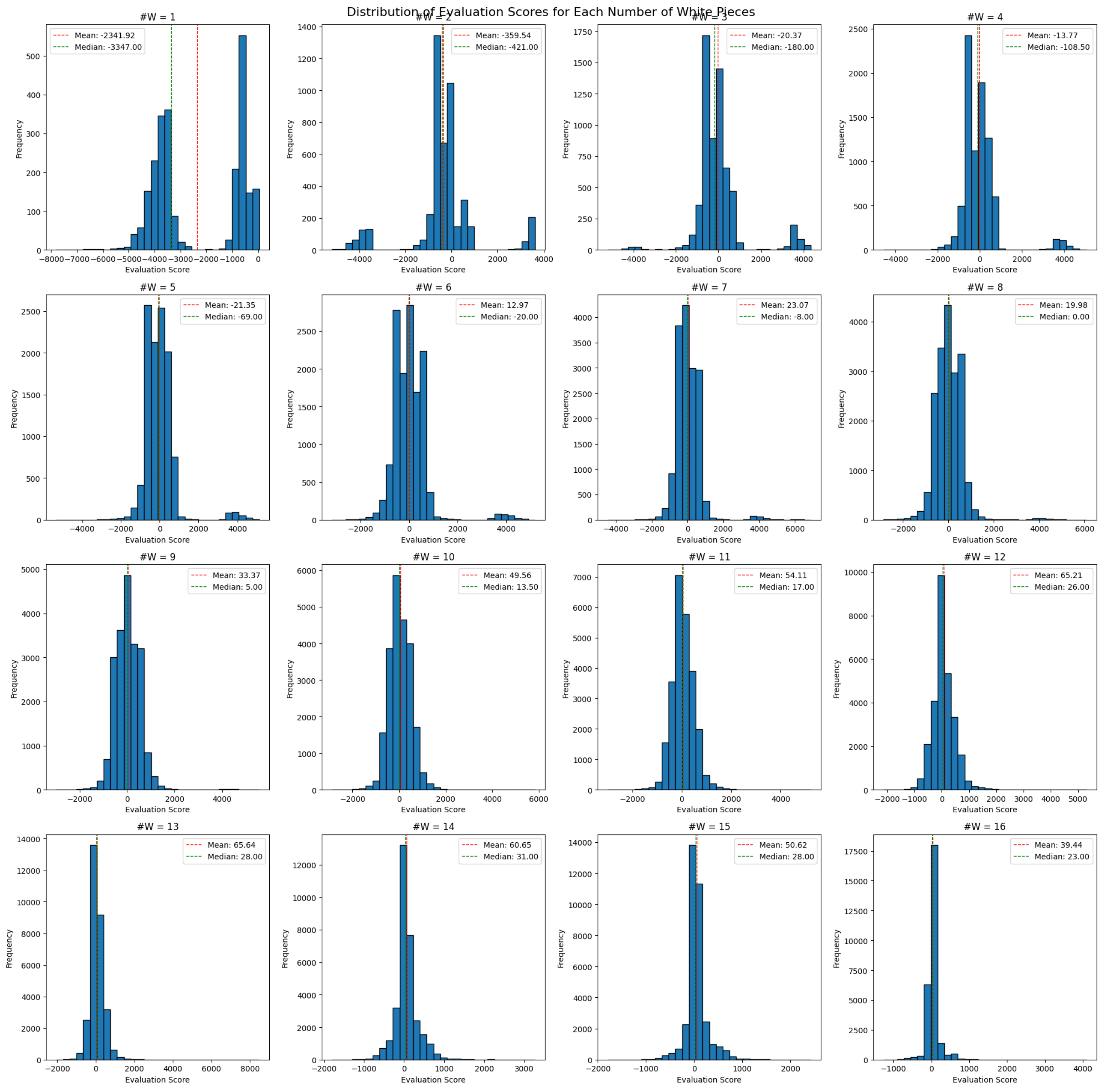


This slight bump is better illustrated by the following graph: white sees a slight spike in mean evaluation score in the 10-14 piece range.

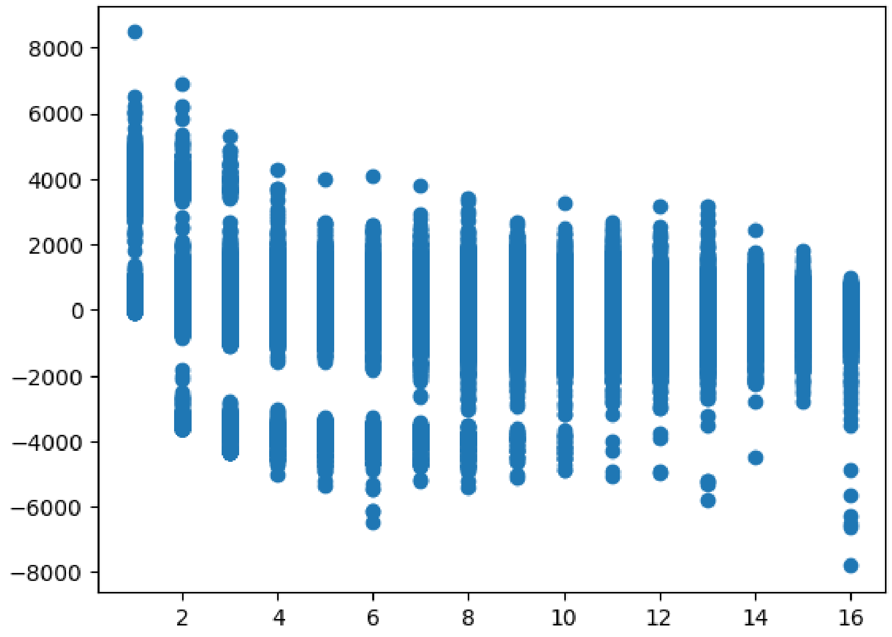


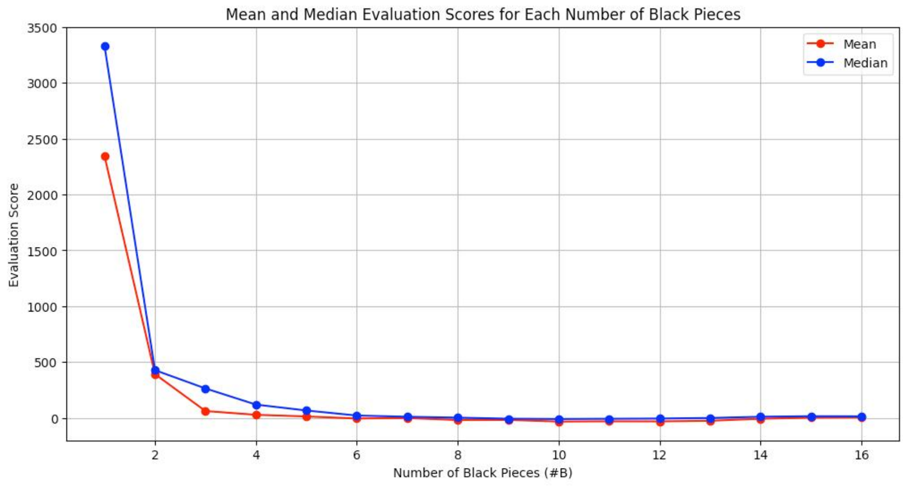
Plotting the individual histograms for the range of the number of white pieces, we see clearly that:

- When White has a low number of pieces (e.g., 1 to 4), the distributions are left-skewed, with a significant portion of the scores being negative.
- As the number of White pieces increases (5 to 12), the distributions become more symmetric, reflecting a balance between positive and negative evaluations.
- For a higher number of White pieces (13 to 16), the distributions remain symmetric but become slightly narrower, showing reduced variability.
- The highest mean evaluations occur in the 12-14 piece range, in concordance with the previous graphs.
- When White has 8 pieces, the most likely situation is a draw (with an effective evaluation of 19.98 i.e. an effective advantage of just 0.1998th of a pawn and a median evaluation of 0), after which it starts to turn negative.

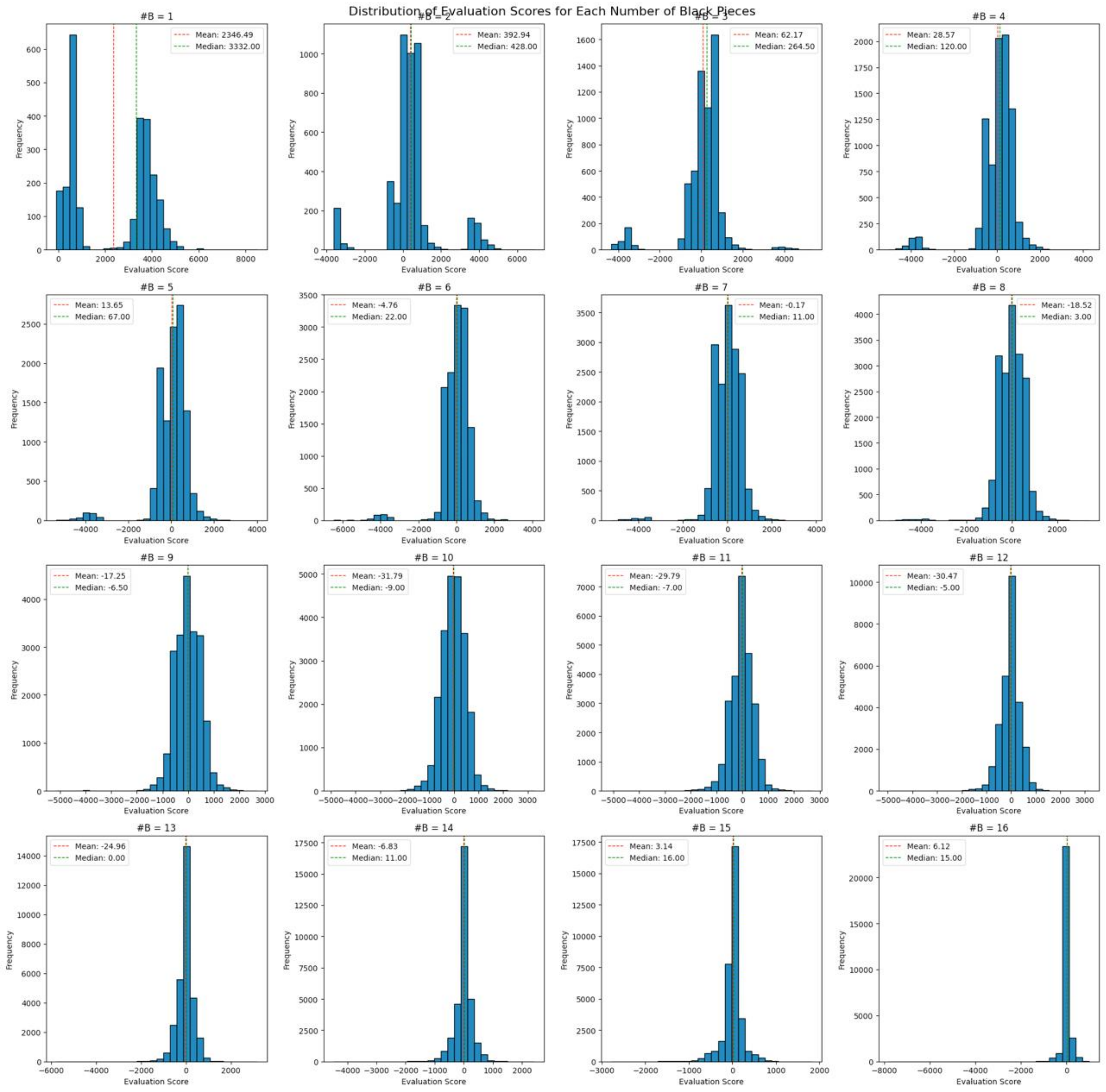

Distribution of Evaluation Scores for Each Number of White Pieces
#W = 1
Mean: -2341.92
Median: -3347.00
#W = 2
Mean: -359.54
Median: -421.00
#W = 3
Mean: -20.37
Median: -180.00
#W = 4
Mean: -13.77
Median: -108.50
#W = 5
Mean: -21.35
Median: -69.00
#W = 6
Mean: 12.97
Median: -20.00
#W = 7
Mean: 23.07
Median: -8.00
#W = 8
Mean: 19.98
Median: 0.00
#W = 9
Mean: 33.37
Median: 5.00
#W = 10
Mean: 49.56
Median: 13.50
#W = 11
Mean: 54.11
Median: 17.00
#W = 12
Mean: 65.21
Median: 26.00
#W = 13
Mean: 65.64
Median: 28.00
#W = 14
Mean: 60.65
Median: 31.00
#W = 15
Mean: 50.62
Median: 28.00
#W = 16
Mean: 39.44
Median: 23.00
Frequency
Evaluation Score

Black also see its evaluation scores decrease as the number of Black pieces decreases. In a similar manner to White, Black sees a small bump in evaluation around the 12 piece as mark. This makes sense for the same reason as it does for white: having a full set (or almost) a full set of pieces likely responds to openings in which a decisive advantage is unlikely.





The individual histograms for Black do show some variability. For instance, while for a higher number of pieces the evaluations cluster around zero, as the number of pieces reduces from 8 to 1 the data begins to develop an a symmetry: a large number of observations cluster around the mean followed by an island of evaluations at the extremes that slowly gro (in fact when oly 1 Black piece is left, the 'smaller island' of values has a higher mode than the cluster around zero!).

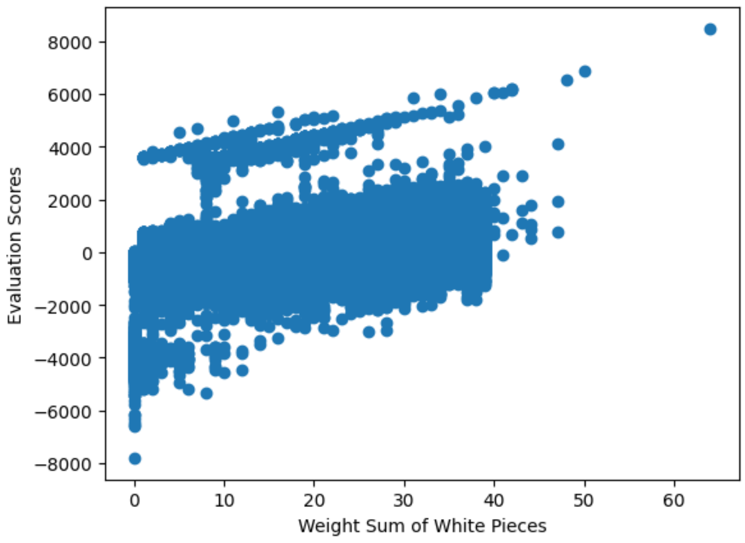


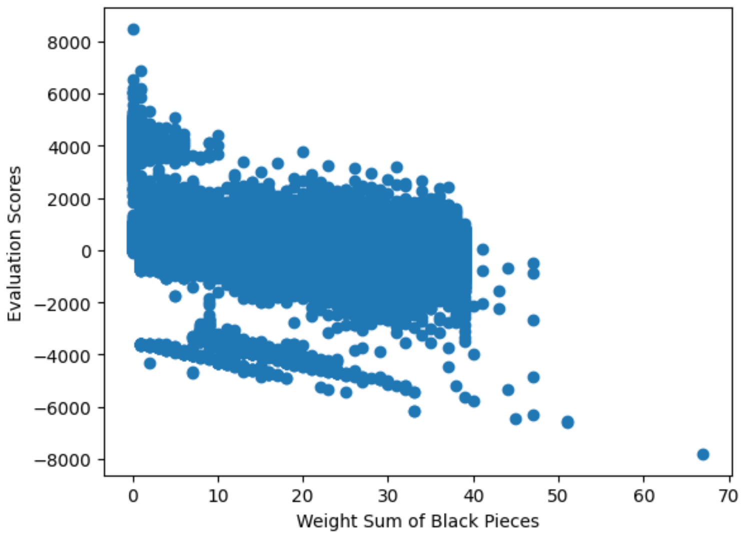
Finally, as expected as the weighted sums increase for White and Black respectively, the evaluation scores increase as well.

To determine optimal piece weights for evaluating chess positions, we employed linear regression. Using dictionaries representing the counts of black and white pieces, we trained separate linear regression models to predict evaluation scores. This approach allowed us to empirically derive weights that best correlate with the observed evaluations.

The resulting optimal weights, obtained through this linear regression process, are presented below:
White Piece Weights:

- P (Pawn): 0.5577
- R (Rook): -0.2238
- N (Knight): -0.6153





- B (Bishop): -0.6641
- Q (Queen): -1.9616
- K (King): 0.0

Black Piece Weights:

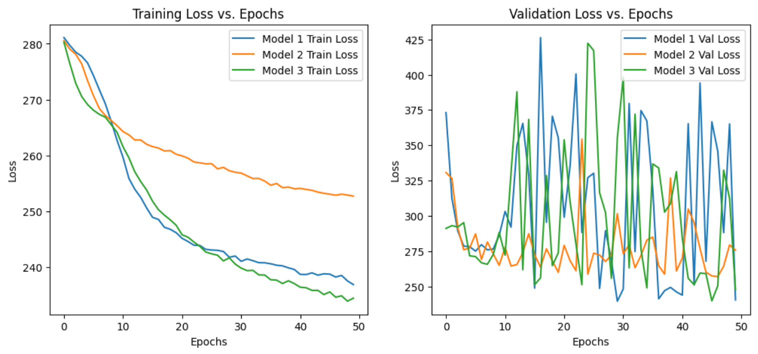
- p (pawn): 0.0447
- r (rook): 0.0723
- n (knight): -0.0573
- b (bishop): 0.1321
- q (queen): 0.7981
- k (king): -6.6613e-16 (approximately 0)

3 Feature Engineering

To explore different models for our CNN model, we experimented first with three different weighting schemes for chess pieces to convert FEN strings into images. In the first model, we assigned weights based on a hierarchical value system where the queen had the highest weight (15), followed by the king (10), rook (7), bishop (5), knight (2), and pawn (1), with negative values for black pieces to distinguish them from white. In the second model, we used a categorical encoding approach, assigning unique integer values from 0 to 11 to each piece, where white pieces ranged from 0 to 5 and black pieces from 6 to 11, ensuring a uniform differentiation between piece types. In the third model, we modified the first weighting scheme by significantly increasing the weight of the king (100) while keeping other values the same, emphasizing the king's strategic importance in chess.

```
    Model1 = {
    'P': 1, 'N': 2, 'B': 5, 'R': 7, 'Q': 15, 'K': 10,
    'p': -1, 'n': -2, 'b': -5, 'r': -7, 'q': -15, 'k': -10
}

Model2 = {
    'P':0, 'N':1, 'B':2, 'R':3, 'Q':4, 'K':5,
    'p':6, 'n':7, 'b':8, 'r':9, 'q':10, 'k':11
}

Model3 = {
    'P': 1, 'N': 2, 'B': 5, 'R': 7, 'Q': 15, 'K': 100,
    'p': -1, 'n': -2, 'b': -5, 'r': -7, 'q': -15, 'k': -100
}
```

On comparing training loss and validation loss for above models, we get:

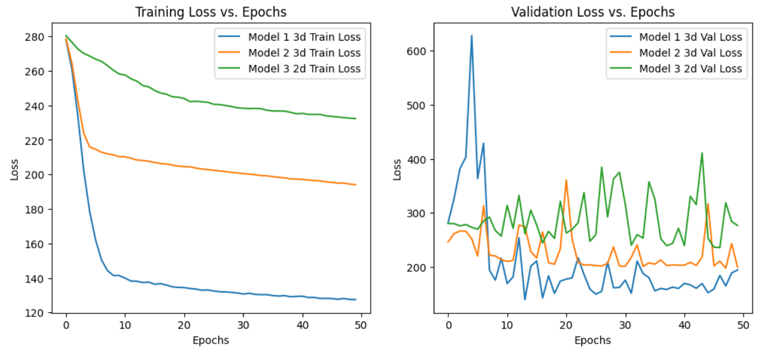


Model 1 performed well on the training data but its performance oscillated wildly on the validation data. On the other hand, Model 2 was slow to converge on the training data but performed much better and stabilized faster on the validation data. Model 3 was midway between Models 1 and 2 on both counts: on the training data Model 3 made rapid gains early on but was slow to converge, while on the validation data it oscillated but less than Model 1 while also being less stable than Model 2. However, this behaviour seems to strike a good balance between Models 1 & 2, which is why we find Model 3 is performing the best comparatively (perhaps emphasizing the king's importance compared to other pieces).

To further improve our FEN-to-image conversion approach, we explored encoding FEN strings into 3D arrays based on the maximum possible mobility of each chess piece on the board. In our first model, we assigned mobility-based vectors for each piece, with positive values for white pieces and negative values for black pieces. For example, rooks had high mobility in straight lines, knights had a uniform limited movement, and pawns had restricted forward mobility. Recognizing the importance of the king from our previous experiments, we refined this encoding by introducing weighted mobility values in our second model. Here, we scaled the mobility vectors by conventional chess piece values, assigning higher weights to more valuable pieces, with the king receiving the highest weight (10). This adjustment emphasized the strategic significance of piece mobility while preserving relative importance.

```
    Model3d1 = {
    'r': -np.array([0, 8, 0, 8, 0, 8, 0, 8], dtype=np.float32),
    'n': -np.array([2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5], dtype=np.float32),
    'b': -np.array([8, 0, 8, 0, 8, 0, 8, 0], dtype=np.float32),
    'q': -np.array([8, 8, 8, 8, 8, 8, 8, 8], dtype=np.float32),
    'k': -np.array([1, 1, 1, 1, 1, 1, 1, 1], dtype=np.float32),
    'p': -np.array([0, 0, 0, 0, 0, 1, 0, 0], dtype=np.float32),

    'R': np.array([0, 8, 0, 8, 0, 8, 0, 8], dtype=np.float32),
    'N': np.array([2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5], dtype=np.float32),
    'B': np.array([8, 0, 8, 0, 8, 0, 8, 0], dtype=np.float32),
    'Q': np.array([8, 8, 8, 8, 8, 8, 8, 8], dtype=np.float32),
    'K': np.array([1, 1, 1, 1, 1, 1, 1, 1], dtype=np.float32),
    'P': np.array([0, 0, 0, 0, 0, 1, 0, 0], dtype=np.float32),
}

Model3d2 = {
    'r': -5*np.array([0, 8, 0, 8, 0, 8, 0, 8], dtype=np.float32),
    'n': -3*np.array([2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5], dtype=np.float32),
    'b': -3*np.array([8, 0, 8, 0, 8, 0, 8, 0], dtype=np.float32),
    'q': -9*np.array([8, 8, 8, 8, 8, 8, 8, 8], dtype=np.float32),
    'k': -10*np.array([1, 1, 1, 1, 1, 1, 1, 1], dtype=np.float32),
    'p': -np.array([0, 0, 0, 0, 0, 1, 0, 0], dtype=np.float32),

    'R': 5*np.array([0, 8, 0, 8, 0, 8, 0, 8], dtype=np.float32),
    'N': 3*np.array([2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5], dtype=np.float32),
    'B': 3*np.array([8, 0, 8, 0, 8, 0, 8, 0], dtype=np.float32),
    'Q': 9*np.array([8, 8, 8, 8, 8, 8, 8, 8], dtype=np.float32),
    'K': 10*np.array([1, 1, 1, 1, 1, 1, 1, 1], dtype=np.float32),
    'P': np.array([0, 0, 0, 0, 0, 1, 0, 0], dtype=np.float32),
}
```

Model 2, which incorporated both mobility and positional advantage, stabilized early but at a high training loss, after which made slow gains, while Model 3 not only stabilized the slowest but also did so at the highest training loss. Both these models were comprehensively outperformed by model 1, which not only stabilized early like Model 2 but did so at a much lower training loss. On the validation data, after high loss initially (till about 10 epochs), Model 1 settled down (with few oscillations) at the lowest level of validation loss (potentially as a result of overfitting), while Model 2 displayed a stable validation loss consistently within a much narrow range and Model 3 was stabled but once again at a higher level of validation loss. As a result of this, we find that Model 1 performed the best comparatively.



4 Neural Network Training and Analysis

Our model architecture is designed to process the 8x8x8 input representing the chess board state and combine it with auxiliary information. Initially, the input channels are increased from 8 to 16 to facilitate richer encoding of board positions while maintaining the 8x8 spatial dimensions. A batch normalization layer is then applied to stabilize training and improve convergence. Subsequently, the spatial dimensions are reduced, and the number of channels is increased using a MaxPooling layer.

ReLU serves as the activation function for this convolutional block.

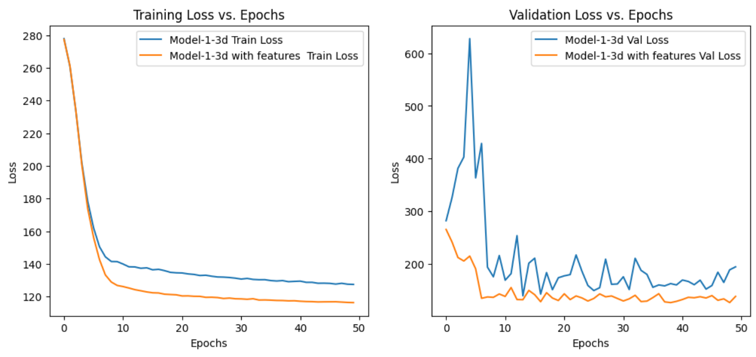
The resulting feature maps are then flattened into a 128-dimensional vector. This flattened vector is concatenated with two additional vector inputs: vector_input1 (representing castling availability) and vector_input2 (comprising half-move clock, full-move number, and the player whose turn it is).

Finally, fully connected layers are employed to reduce the output dimension to 1, representing the evaluation score. A dropout rate of 50% is applied within these fully connected layers to mitigate overfitting and enhance the model's generalization capabilities.
For our final model we added vector_input3 (containing the weighted sums of black and white pieces, wtdb and wtdw, respectively) to our basic CNN model.

```
def makemodel3d():
cube_input = Input(shape=(8, 8, 8), name='image_input')

casting = layers.Conv2D(16, (3, 3), activation=None, padding='same')(cube_input)
casting = layers.BatchNormalization()(casting)

x = layers.Conv2D(16, (3, 3), activation=None, padding='same')(casting)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)

x = layers.MaxPooling2D((2, 2))(x)

x = layers.Conv2D(32, (3, 3), activation=None, padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)

x = layers.MaxPooling2D((2, 2))(x)

x = layers.Conv2D(32, (2, 2), activation=None, padding='same')(x)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)

x = layers.Flatten()(x)
x = layers.Dropout(0.5)(x)

vector_input1 = Input(shape=(4,), name='vector_input1')
y1 = layers.Dense(16, activation='relu')(vector_input1)
y1 = layers.Dense(32, activation='relu')(y1)

vector_input2 = Input(shape=(3,), name='vector_input2')
y2 = layers.Dense(16, activation='relu')(vector_input2)
y2 = layers.Dense(32, activation='relu')(y2)

vector_input3 = Input(shape=(2,), name='vector_input3')
y3 = layers.Dense(16, activation='relu')(vector_input3)
y3 = layers.Dense(32, activation='relu')(y3)

combined = layers.Concatenate()([x, y1, y2, y3])

z = layers.Dense(128, activation=None)(combined)
z = layers.BatchNormalization()(z)
z = layers.ReLU()(z)

z = layers.Dense(64, activation=None)(z)
z = layers.BatchNormalization()(z)
z = layers.ReLU()(z)

z = layers.Dropout(0.5)(z)

z = layers.Dense(32, activation=None)(z)
z = layers.BatchNormalization()(z)
z = layers.ReLU()(z)

z = layers.Dense(16, activation=None)(z)
z = layers.BatchNormalization()(z)
z = layers.ReLU()(z)
```

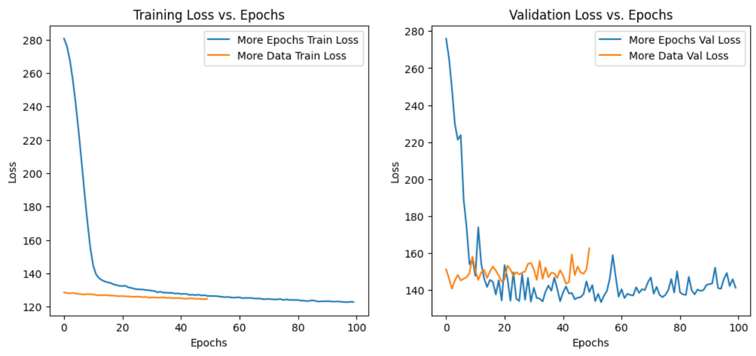
```
    z = layers.Dense(4, activation=None)(z)
    z = layers.BatchNormalization()(z)
    z = layers.ReLU()(z)

    output = layers.Dense(1, activation='linear', name='output')(z)

    model = Model(inputs=[cube_input, vector_input1, vector_input2, vector_input3],
                                            outputs=output)

    optimizer = tf.keras.optimizers.AdamW(
        learning_rate=1e-2,
        weight_decay=1e-3
    )

    model.compile(
        optimizer=optimizer,
        loss='mean_absolute_error',
        metrics=['mean_squared_error', 'mean_absolute_error']
    )

    return model
```

Comparing loss in featured model with basic model:



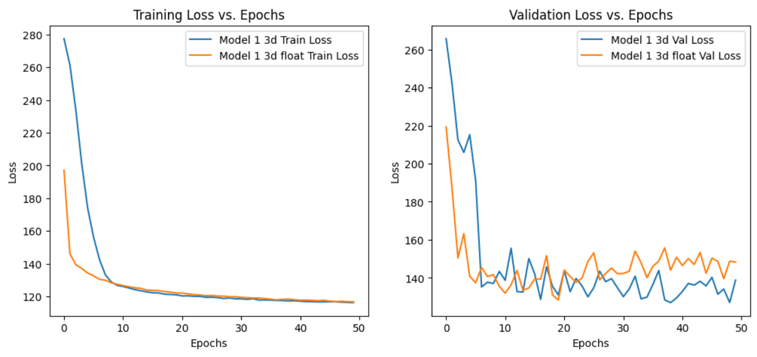
To answer the question of whether it is better to train with limited data on multiple epochs, or with more data on fewer epochs, we trained 2 models with an 80/20 training-validation split and a batch size of 2048:

- More Data: The model was on a larger dataset trained for 50 epochs.
- More Epochs: The model was trained on a smaller dataset for 100 epochs.



We found that in the More Data model, the training loss decreased steadily over 50 epochs, and converged to a stable loss early on, whereas in the More Epochs model the training loss continued to decrease over 100 epochs but showed diminishing returns after approximately 50 epochs. Furthermore, the 'More Data' model achieved a lower overall validation loss than the 'More Epochs' model and once again was much more stable (the 'More Epoch' model's lower validation loss at the end may in fact indicate overfitting). Thus, models trained on more data appear to be more effective than models trained for more epochs as they converge faster and display a reduced tendency for overfitting as they generalize better due to a larger amount of examples.

The following graph compares the training and validation loss over 50 epochs for two model variations: one using float32 precision (Model 1 3d) and another using float64 precision (Model 1 3d float). Both models show rapid initial decrease in training loss, indicating effective learning. However our original Model works better on Validation set.



As a comparison, we considered a naive model in which we assigned the weights queen = ± 9.75, rook = ± 5, bishop = knight = ± 3.25,pawn = ± 1. We then divided the scores by 994.339771480558 and applied the *tanh* function to obtain values between -1 and 1. We then defined our Neural Network, ChessEvaluator, four fully connected layers (64 256 128 64 1), added Rectified linear unit activations, and set a dropout rate of 50%. We then trained the model on the basis of Mean Square Error using the Adam Optimiser.

```
def fen_to_matrix(fen):
    board = chess.Board(fen)
    matrix = [[None for _ in range(8)] for _ in range(8)]
    for square in chess.SQUARES:
        piece = board.piece_at(square)
        if piece:
            matrix[7 - chess.square_rank(square)][chess.square_file(square)] = piece.
                                                                      symbol()
        else:
            matrix[7 - chess.square_rank(square)][chess.square_file(square)] = ' '
    return matrix

def matrix_to_numeric(matrix):
    piece_to_float = {
        'P': 1, 'N': 3.25, 'B': 3.25, 'R': 5, 'Q': 9.75, 'K': 1,
        'p': -1, 'n': -3.25, 'b': -3.25, 'r': -5, 'q': -9.75, 'k': -1,
        ' ': 0
    }
    return [[piece_to_float[piece] for piece in row] for row in matrix]

def flatten_matrix(matrix):
    return [item for sublist in matrix for item in sublist]

class TanhNormalizer:
    def __init__(self, k):
        self.k = k

    def normalize(self, scores):
        return np.tanh(scores / self.k)

    def denormalize(self, normalized_scores):
        return self.k * np.arctanh(normalized_scores)
```

```
k = 994.339771480558
normalizer = TanhNormalizer(k)

y_train_normalized = normalizer.normalize(y_train.numpy())
y_test_normalized = normalizer.normalize(y_test.numpy())

y_train_normalized = torch.tensor(y_train_normalized, dtype=torch.float32)
y_test_normalized = torch.tensor(y_test_normalized, dtype=torch.float32)

import torch.nn as nn
import torch.optim as optim

class ChessEvaluator(nn.Module):
    def __init__(self):
        super(ChessEvaluator, self).__init__()
        self.fc1 = nn.Linear(64, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 1)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = torch.relu(self.fc2(x))
        x = self.dropout(x)
        x = torch.relu(self.fc3(x))
        x = self.fc4(x)
        return x

model = ChessEvaluator()
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001)

for epoch in range(80):
    optimizer.zero_grad()
    outputs = model(X_train)
    loss = criterion(outputs, y_train_normalized)
    loss.backward()
    optimizer.step()
    print(f'Epoch {epoch+1}, Loss: {loss.item()}')
```

With this model, we obtained a final Denormalized Test Loss of 36486132.0, which clearly represents a more degree of learning.

5 Appendix

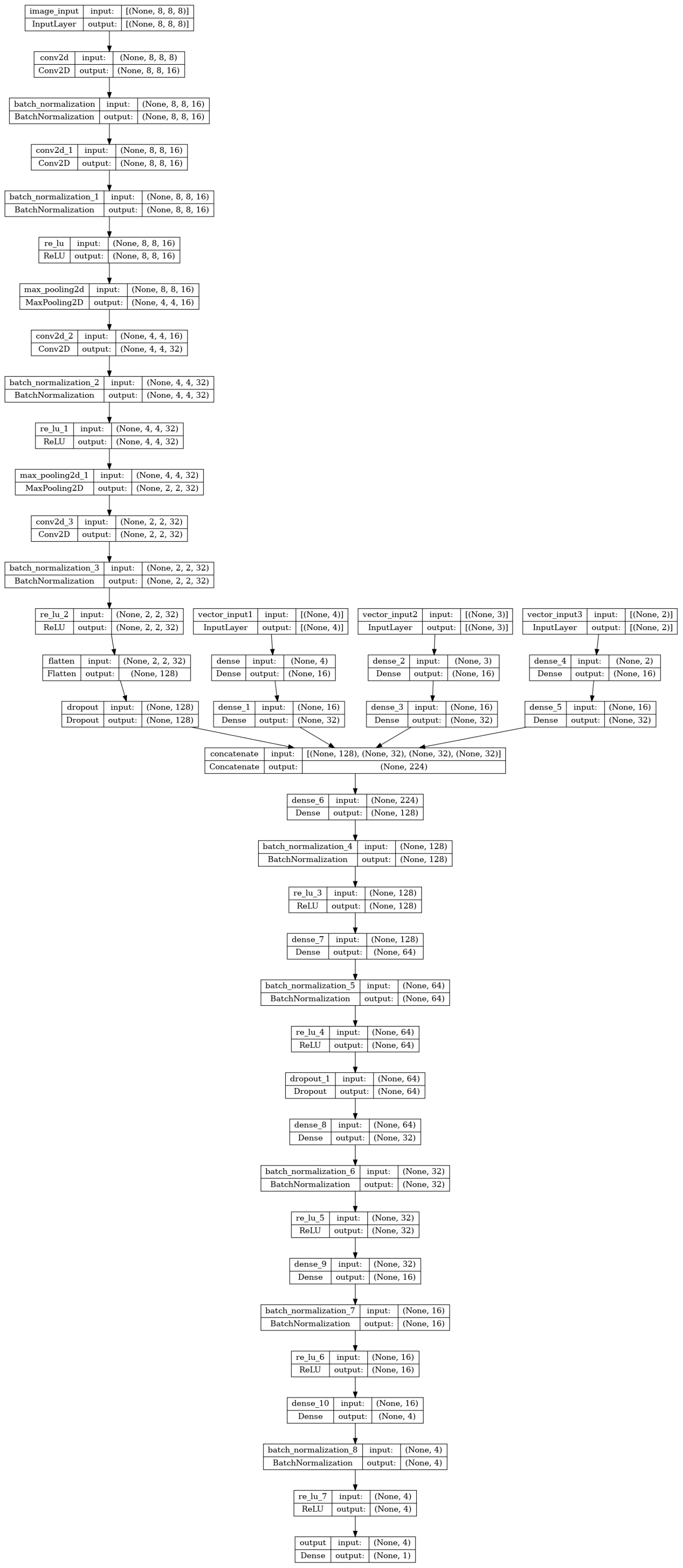

image_input | InputLayer | input: [(None, 8, 8, 8)] | output: [(None, 8, 8, 8)]
conv2d | Conv2D | input: (None, 8, 8, 8) | output: (None, 8, 8, 16)
batch_normalization | BatchNormalization | input: (None, 8, 8, 16) | output: (None, 8, 8, 16)
conv2d_1 | Conv2D | input: (None, 8, 8, 16) | output: (None, 8, 8, 16)
batch_normalization_1 | BatchNormalization | input: (None, 8, 8, 16) | output: (None, 8, 8, 16)
re_lu | ReLU | input: (None, 8, 8, 16) | output: (None, 8, 8, 16)
max_pooling2d | MaxPooling2D | input: (None, 8, 8, 16) | output: (None, 4, 4, 16)
conv2d_2 | Conv2D | input: (None, 4, 4, 16) | output: (None, 4, 4, 32)
batch_normalization_2 | BatchNormalization | input: (None, 4, 4, 32) | output: (None, 4, 4, 32)
re_lu_1 | ReLU | input: (None, 4, 4, 32) | output: (None, 4, 4, 32)
max_pooling2d_1 | MaxPooling2D | input: (None, 4, 4, 32) | output: (None, 2, 2, 32)
conv2d_3 | Conv2D | input: (None, 2, 2, 32) | output: (None, 2, 2, 32)
batch_normalization_3 | BatchNormalization | input: (None, 2, 2, 32) | output: (None, 2, 2, 32)
re_lu_2 | ReLU | input: (None, 2, 2, 32) | output: (None, 2, 2, 32)
flatten | Flatten | input: (None, 2, 2, 32) | output: (None, 128)
dropout | Dropout | input: (None, 128) | output: (None, 128)
vector_input1 | InputLayer | input: [(None, 4)] | output: [(None, 4)]
dense | Dense | input: (None, 4) | output: (None, 16)
dense_1 | Dense | input: (None, 16) | output: (None, 32)
vector_input2 | InputLayer | input: [(None, 3)] | output: [(None, 3)]
dense_2 | Dense | input: (None, 3) | output: (None, 16)
dense_3 | Dense | input: (None, 16) | output: (None, 32)
vector_input3 | InputLayer | input: [(None, 2)] | output: [(None, 2)]
dense_4 | Dense | input: (None, 2) | output: (None, 16)
dense_5 | Dense | input: (None, 16) | output: (None, 32)
concatenate | Concatenate | input: [(None, 128), (None, 32), (None, 32), (None, 32)] | output: (None, 224)
dense_6 | Dense | input: (None, 224) | output: (None, 128)
batch_normalization_4 | BatchNormalization | input: (None, 128) | output: (None, 128)
re_lu_3 | ReLU | input: (None, 128) | output: (None, 128)
dense_7 | Dense | input: (None, 128) | output: (None, 64)
batch_normalization_5 | BatchNormalization | input: (None, 64) | output: (None, 64)
re_lu_4 | ReLU | input: (None, 64) | output: (None, 64)
dropout_1 | Dropout | input: (None, 64) | output: (None, 64)
dense_8 | Dense | input: (None, 64) | output: (None, 32)
batch_normalization_6 | BatchNormalization | input: (None, 32) | output: (None, 32)
re_lu_5 | ReLU | input: (None, 32) | output: (None, 32)
dense_9 | Dense | input: (None, 32) | output: (None, 16)
batch_normalization_7 | BatchNormalization | input: (None, 16) | output: (None, 16)
re_lu_6 | ReLU | input: (None, 16) | output: (None, 16)
dense_10 | Dense | input: (None, 16) | output: (None, 4)
batch_normalization_8 | BatchNormalization | input: (None, 4) | output: (None, 4)
re_lu_7 | ReLU | input: (None, 4) | output: (None, 4)
output | Dense | input: (None, 4) | output: (None, 1)

Figure 1: Final Model Architecture