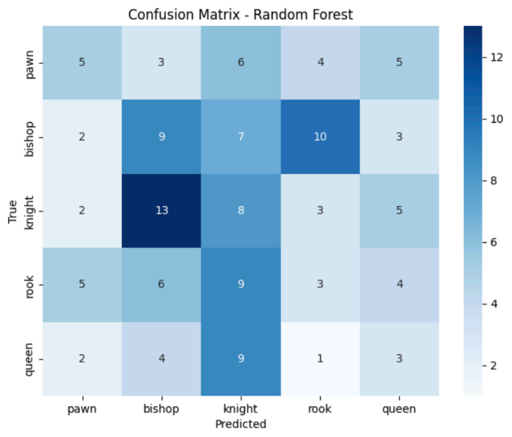
CPSC542 - Assignment 1

Chess Piece Detection: A Deep Neural Network Approach

DYLAN INAFUKU

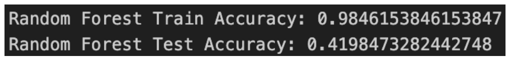
Problem Statement

For this technical report I decided to use the Chess Piece Detection Images Dataset as I personally enjoy the game of chess and also thought this would be an interesting computer visualization task. This project revolves around classifying between the various chess pieces including the pawn, rook, knight, bishop, and queen. Unfortunately, this data set does not have the king. But, this was by far the best dataset when compared to other chess datasets on kaggle in terms of image quality, variation, and number of images. This image classification task requires a deep learning solution because a simple random forest model isn't able to consider the spatial relationships within images.



True \ Predicted	pawn	bishop	knight	rook	queen
pawn	5	3	6	4	5
bishop	2	9	7	10	3
knight	2	13	8	3	5
rook	5	6	9	3	4
queen	2	4	9	1	3

Figure 1: RF Confusion Matrix



```
Random Forest Train Accuracy: 0.9846153846153847
Random Forest Test Accuracy: 0.4198473282442748
```

Figure 2: Random Forest Performance

While a random forest model performed very well on the training data it is clearly overfit in which there is a 0.57 discrepency between training and test performance. This was just an example of why a model such as random forest would perform terribly on image data and why a deep learning solution is needed for this task. Random forest models use individual pixels as features which does not help the model learn anything reakky useful about the different images. However, using filters from a CNN would help the model learn differentiating factors between classes.

Essentially, my plan to tackle this classification task is to implement a CNN using transfer learning and utilize techniques such as preprocessing, regularization, train/test/validation split, and learning rate adjustments to create a model that accurately classifies between chess pieces. The general procedure will be as follows, import data and preprocessing steps, split data into train/test/validation, create basic CNN model with transfer learning (VGG16), adjust hyperparameters of early stopping, learning rates, regularization (dropout), and epochs to get the best performance possible, display accuracy metrics for train/test/validation/split (test and plot), display a confusion matrix for CNN model predictions, and output images with corresponding true and predicted labels from the CNN model.

Data and EDA

The Chess Piece Detection Images Dataset consists of five different folders labeled as Queen-Resized, Rook-resize, bishop_resized, knight-resize, and pawn_resized. There are a total of 651 different images with a varying number of images per class.

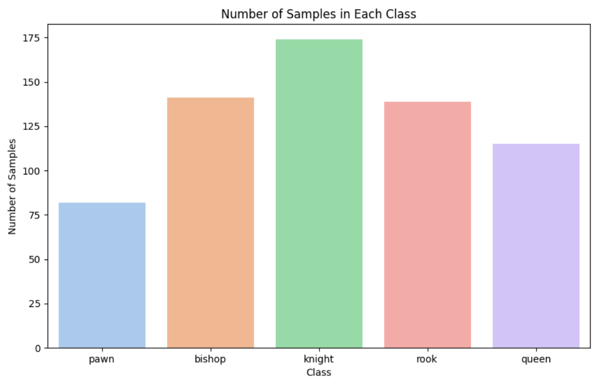
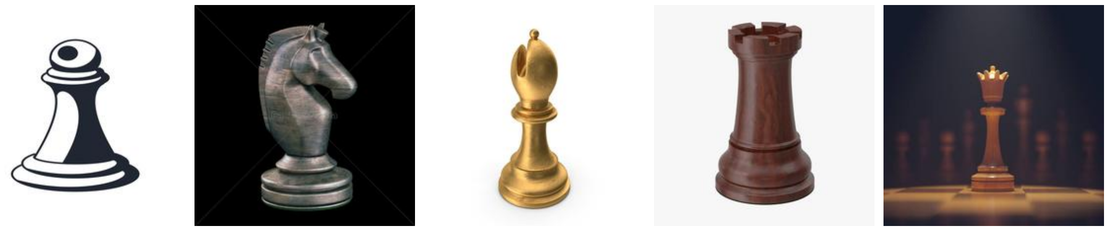


Figure 3: Class Distribution of Chess Pieces

The above image displays the distribution of images amongst the five different chess pieces. Knight is the majority class and pawn is the minority class in terms of number of images. I will elaborate more on the impact class imbalance has on my model's performance and my attempts to resolve the issue in the discussion section.



The five images above display the types of variation available in the chess dataset. There are clear differences in drawing style, realism, and dimensionality (2D vs 3D). The amount of variation present in the dataset for each class is very useful in preventing overfitting. Overall the dataset provides image variation and a decent amount of images (over 600). However, some negatives about the dataset is that images of the king piece are missing (for the sake of this assignment I am disregarding the king) and there is a class imbalance.

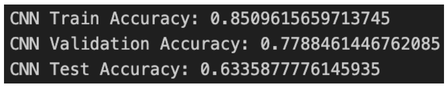
Methods (Preprocessing)

For the preprocessing step of my pipeline I decided to use keras's ImageDataGenerator function. This function allowed me to rescale the pixels in my images as well as augment the images through the use of shear, zoom, rotations, shifts. etc. Also, I resized each image to 224 by 224 since the transfer learning model VGG16 was trained on 224 by 224 sized images. While decent variation is already present in the data set, I implemented data augmentation through ImageDataGenerator to provide even more variation. The next step was to split my data up into train, validation, test sets in preparation for training my CNN model.

Methods (Model)

This CNN model uses the VGG16 model trained on the imagenet dataset. The fully connected layers at the top of the VGG16 model are disregarded and the weights are frozen since we don't want VGG16's weights being updated during training. I utilized global average pooling to prevent overfitting by reducing spacial dimensions of the tensor. Then I added a series of densely connected layers with a relu activation function to help the model train efficiently, introduce sparsity to prevent overfitting, and prevent the vanishing gradient problem. I also added dropout after each densely connected layer to regularize and further prevent overfitting. Furthermore, I utilized batch normalization to help stabilize and accelerate the training process. For the model's predictions I used the softmax activation function since this classification task deals with five different classes and not just two. I also adjusted the learning rate to 0.001 to avoid taking large steps and skipping over minima. Finally I compiled the model using the Adam optimizer to configure the learning rate, configure the loss function as sparse_categorical_entropy since I use numerical class labels, and track accuracy throughout training.

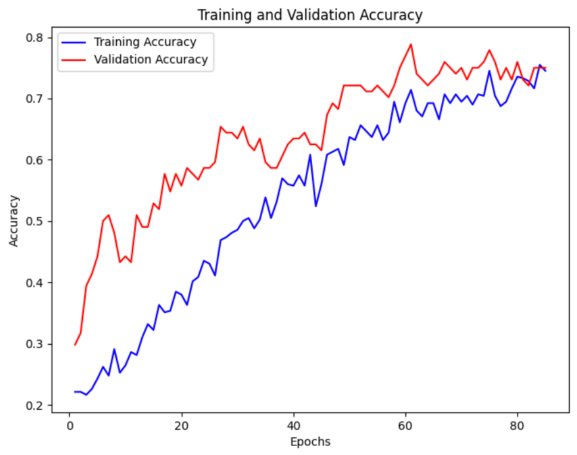
Results with Plots and Figures

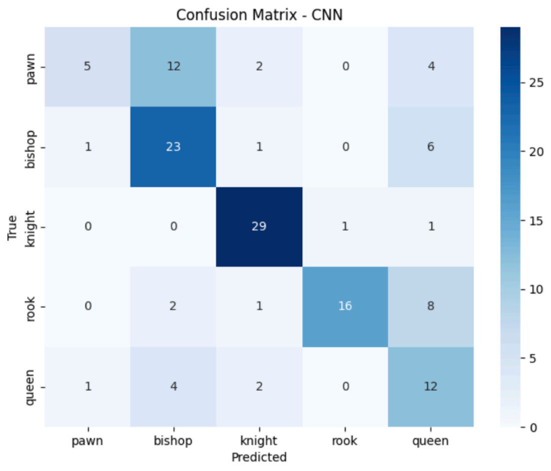


```
CNN Train Accuracy: 0.8509615659713745
CNN Validation Accuracy: 0.7788461446762085
CNN Test Accuracy: 0.6335877776145935
```

Figure 4: CNN Metrics

The image above displays the accuracy metrics from the CNN for the train, validation, and test sets. The model was able to achieve 85% accuracy on the train set, 78% accuracy on the validation set, and 63% accuracy on the test set. While the model is slightly overfit on the training data, the model still performed decently on the validation set and the difference in performance wasn't egregious. However, the model performed worse on the test set with a 63% accuracy even after thorough hyperparameter changes and adjustments.





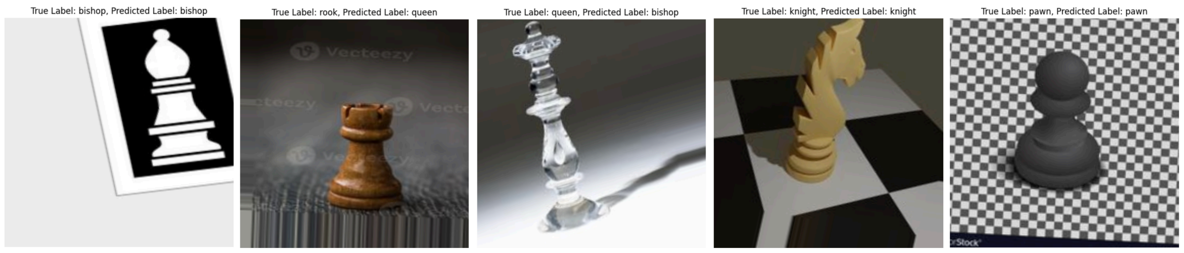
True \ Predicted	pawn	bishop	knight	rook	queen
pawn	5	12	2	0	4
bishop	1	23	1	0	6
knight	0	0	29	1	1
rook	0	2	1	16	8
queen	1	4	2	0	12

I used a large number of epochs because I set the learning rate to 0.001 for a slower but more optimal convergence and to prevent skipping over minima. Even though I implemented multiple regularization techniques the model ended up slightly overfit on the training data.

The confusion matrix displays the CNN model's predictions on the test data with true labels predicted labels. The ideal confusion matrix would have a high density of predictions on the main diagonal. In the case of this confusion matrix, most of the predictions were on the main diagonal. One thing that stands out from the confusion matrix is that the model was not accurate at differentating between a queen and the other chess pieces. Looking at the queen prediction column, it is clear that the queen is commonly mistaken to be in place of the rook, bishop, and pawn. From a human perspective, I can see why this could occur as a queen has a somewhat similar shape to a bishop and pawn. It also makes sense that the queen is rarely predicted when the true label is a knight since a knight has a distinguishing shape from all the other pieces. This is also evident when we look at the knight predicted/knight true label cell (middle cell in confusion matrix), as it has the highest density of predictions. Another small value that stood out to me was the predicted bishop/pawn true label cell. This cell had quite a few incorrect predictions but again this makes sense as bishops and pawns have a similar appearance.

Discussion/Results

With respect to the problem the model did alright in differentiating between the five different chess pieces. While the model had decent accuracy on the train and validation datasets, it did not perform as well on unseen data/the test set. There was quite a lot of variation within the dataset which could have impacted the model's performance on unseen data. Something to note is the evident class imbalance in the dataset. The minority class was the pawn chess piece by a wide margin. One idea I had to fix this issue would be to utilize oversampling of the pawn minority class. However, generally with oversampling I am used to applying this technique to numerical data and not with image data. After conducting some research, it seemed to be more effective to apply oversampling to feature extractions of the CNN model rather than directly oversampling the images themselves. I attempted to apply oversampling to extracted CNN features, however the performance was slightly worse and so I decided not to include it.


True Label: bishop, Predicted Label: bishop
True Label: rook, Predicted Label: queen
True Label: queen, Predicted Label: bishop
True Label: knight, Predicted Label: knight
True Label: pawn, Predicted Label: pawn

The 6 images above show the model's predicted label vs. true label.

Hyperparameter tuning and model adjustments were the bulk of this assignment for me. The initial CNN model performed extremely well on the training set but much worse on the validation set. Within a couple of epochs the model reached 99% accuracy on the training set with only about 30% accuracy on the validation set. In order to combat this overfitting issue of the model I added more complexity and regularization methods. I added multiple densely connnected layers with a dropout layer after each of those. By continuously adjusting the dropout rate I was able to minimize the discrepency in performance between train and validation sets. However, the model was still quite overfit and I added early stopping along with a small learning rate. Thus attempting to prevent overfitting on training data. The small learning rate required a larger number of epochs for the model, but this allowed for better performance. In the end, all of these adjustments resulted in a less overfit and better performing CNN model.

In the future, I would consider using a different base pre-trained model other than VGG16 just to see how the model's performance would differ. It would be an interesting comparison to see what types of model's are better for different applications. I would also like to implement a different type of class imbalance solution to see how it would either increase or decrease the model's classification accuracy. I wasn't able to achieve an increase in performance with an oversampling method, but I am curious to check out other solutions.

Overall, I learned how to create an efficient pipeline from start to finish of a CNN model utilizing transfer learning. While I have covered these topics through a different class before, this assignment helped me to refresh my knowledge of these concepts as well as get better with adjusting hyperparameters.