Final Project Writeup
Classification of Chess Piece Images

Group 8

Orion Stavre
Thar Min Htet
Yared Taye

Introduction

For our final project we want to be able to predict the type of chess piece that is shown in the picture. For this project we will be using the chessman image dataset from kaggle.com. The dataset can be found here: https://www.kaggle.com/niteshfre/chessman-image-dataset/data. We want to use this data set to train and test different AI models and Techniques that result in good prediction accuracy, and are able to be used to predict future images and classify them into various chess piece categories.

Upon looking at the data, we discovered that the data is composed of 556 total files (or instances) and in most cases these were large size images. Based on the data type and size of data set we have decided to try the following models:

- Convolutional Neural Network (CNN)
- Principal Component Analysis (PCA) for dimensionality reduction, and then trying various methods like Random Forest, SVM etc. to use components to predict the images

CNN Transfer Learning Approach

As one approach to classifying chess images, we used Convolutional Neural Networks (CNN) with transfer learning. One of the major problems with image classification is the large number of features that come with it. Each pixel of an image represents a feature. For a small 600x600 pixel RGB image, there are 1,080,000 features. This makes feeding such features to neural networks impractical. CNNs are useful in extracting the useful parts of the image, effectively shrinking down the number of features to be fed into the neural network. We can also shrink down the size of the image before processing with CNNs. In the following architecture, we shrank down the size of all our images to 224x224 pixels. This reduces the feature size dramatically without losing much information as seen in Figure 1.

Figure 1: Demonstration of shrank down images accompanied by true labels

The architecture of our classifier consists of two main parts: (1) AlexNet and (2) custom feed-forward network. The AlexNet is an award-winning CNN architecture trained on ImageNet database, which contains more than 20,000 categories of real life entities. Since we are trying to classify chess-piece images of both real life photos and graphic renderings, it's justifiable to use transfer learning with AlexNet, which was trained on somewhat similar types of images. There are other architectures that did well on ImageNet. However, compared to the majority of those, AlexNet has a smaller architecture by sacrificing a small accuracy. This allows us to do operations more efficiently since we have limited computing resources.

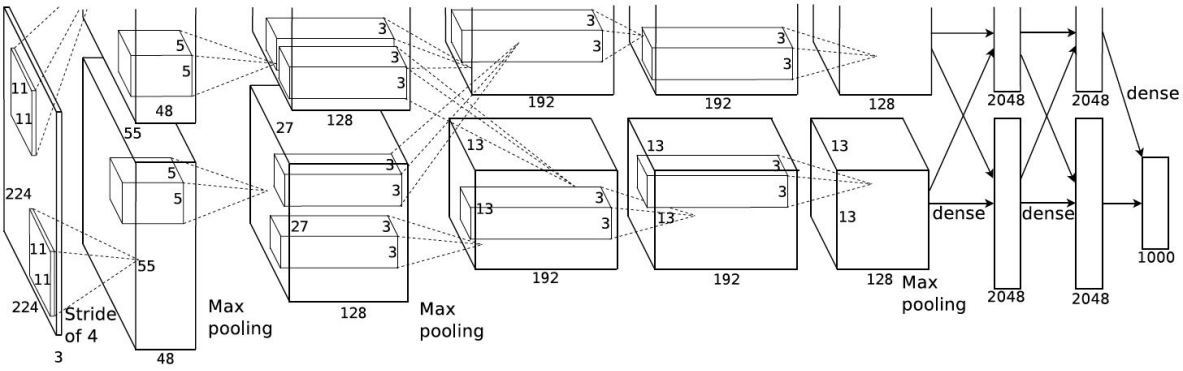

Figure 2: Architecture of AlexNet

As seen in the above figure 2, the architecture of AlexNet contains 8 total layers: 5 convolutional layers with or without max pooling and 3 fully connected layers. In our classifier we will be using transfer learning by using the pretrained weights of AlexNet provided by Pytorch's torchvision package. However, we made a few tweaks to the original pretrained weights in order to allow our model to learn on new dataset.

For large datasets, it's a common practice to fine-tune all the layers of the model. However, because we have a humble dataset of roughly 600 images, we decided to freeze the convolutional layers of AlexNet. If the new dataset is similar to the pretrained dataset, It's a common practice to replace the very last layer of the fully-connected network with an appropriate layer to accommodate new classes. Although chess and imagenet data have similarities, our dataset also contains graphic images. Therefore, we decided to slice off all the fully connected layers and replace with a custom one.

Our final architecture consists of the same convolutional layers from AlexNet and a fully-connected network made up of 2 hidden layers and 1 output layer. The first hidden layer contains 1024 neurons, the second hidden layer contains 128 neurons, and the output layer contains 6 neurons, corresponding to 6 classes: pawn, knight, bishop, rook, queen and king respectively. To prevent overfitting, dropout with 0.5 probability is applied to the output of the convolutional layer and the first fully-connected layer. The hidden layers of the fully-connected network are accompanied by ReLU activation functions.

Since this is a classification model, we used Cross Entropy Loss as our loss function. As for our optimizer, we used Adam Optimizer. The learning rate is 0.001.

Training the Model

To train our CNN model, the dataset is split into train, valid and test sets. Since we have a small dataset, it is hard to draw conclusions from a single model trained from 80:10 train vs valid dataset. Moreover, it is also a source of overfitting. To alleviate these problems, we were advised to perform k-fold cross-validation. Before we proceed to cross-validation, we set aside

10% of all data as the final test set in order to preserve the integrity of the model. The model will never see that subset of data during the training process. From the remaining 90%, we trained our models with 6-fold cross validation. This will allow both train and validation sets to contain a decent number of samples. Each fold is trained for 30 epochs, which is when the validation loss decreases diminishing manner.

Results

After training and testing we get the following results.

ith Fold	Train Loss	Test Loss	Train Accuracy	Test Accuracy
1	0.8782	0.9460	0.84	0.76
2	0.8431	0.8623	0.82	0.81
3	0.8932	0.8997	0.82	0.75
4	0.8793	0.8856	0.84	0.76
5	0.8743	0.9572	0.83	0.74
6	0.8393	0.9030	0.87	0.69
Average			0.837	0.752

Table 1: Comparison of Accuracies among different cross-validation folds

As seen in Table 1, the average accuracy on train data is 83.7% and that on test data is 75.2%. We didn't include the validation accuracy in the table above since we do not find it essential. As mentioned before, the model has never seen the test set and the model wasn't retrained to improve the test accuracy. Since there are 6 classes, the random guess is 16.7%. It can be concluded as a good result considering that we train on a small dataset and the accuracy is about 5 times better than random guessing.

Another thing we noticed is that the model with highest train accuracy has lowest test accuracy and the model with lowest train accuracy has highest test accuracy. This probably indicates the former model is more overfit than the latter.

To get an intuition of what our models did well and what not, we tested on a few test samples. Across different models and different combinations of datasets, the result looks similar to figure 3. The correct classes are in the parenthesis.

Figure 3: Demonstration of Prediction: Prediction(Ground Truth)

All the models tend to be confused between 'queens' and 'kings'. This is a foreseeable type of confusion. Not only queens and kings are very similar in shape but the images from the dataset contain different styles of chess pieces, making it harder for the models to distinguish. Figure 3 shows one of such examples.

Figure 3: Two different designs of kings

To formally prove the confusion among chess pieces, we made a confusion matrix out of the predictions of the best model we have as we have been suggested.

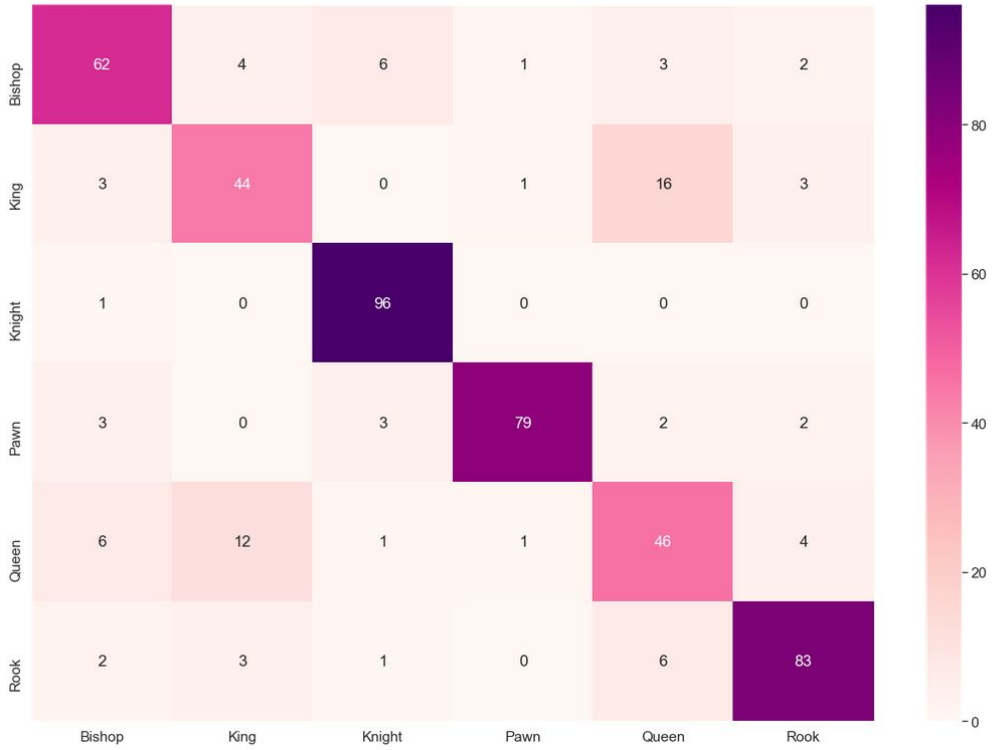

Figure 4: Confusion Matrix on noise-free Training Set (X= Predicted, Y=True)

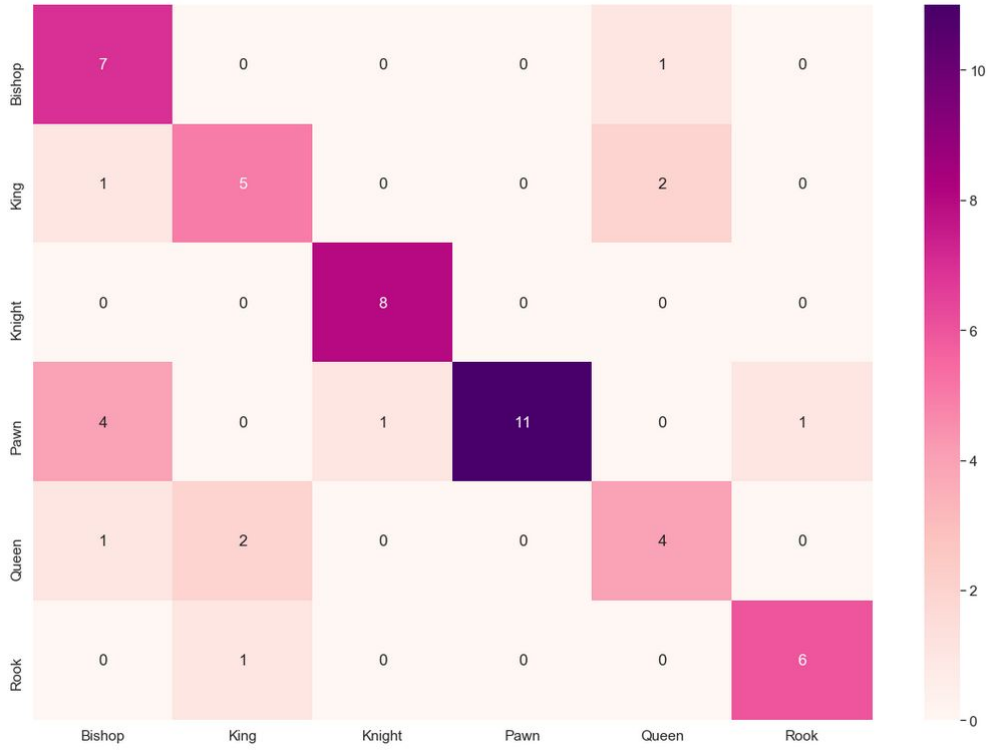

Figure 5: Confusion Matrix noise-free on Test Set (X=Predicted, Y=True)

Figure 4 and 5 plots the confusion matrix of the best model on both train and test sets. In the training set, as we suspected the model is mostly confused between queens and kings. There are 12 cases of mislabelling 'king' as 'queen' and 16 cases of mislabelling 'queen' as 'king'. These 28 cases make up roughly 6% of all training data. If we can improve this, the accuracy of the model will go up significantly.

In the test set though, the confusion matrix shows an unexpected source of error. There are 4 cases of mislabelling 'bishop' as 'pawn'. This is almost 1/3 of total test set error. Another 1/3 is made up of king and queen confusion.

Adding Noise to the Data

To hopefully make our model more robust, we added some static noise to the data and retrained the models from scratch using the same methods with identical parameters.

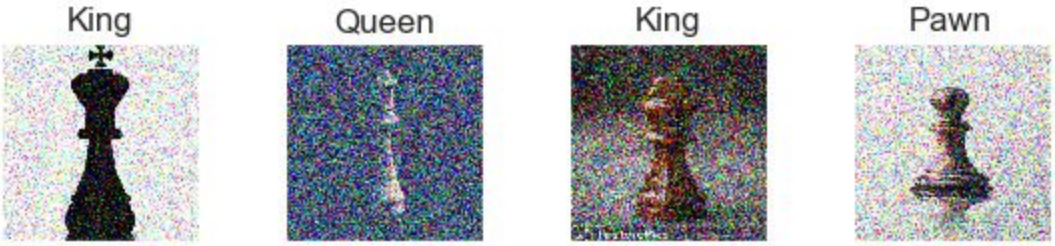

Figure 6: Sample images with 25% static noise

Figure 6 covers a few samples after adding 25% static noise. It's noticeably harder for even humans to recognize some photos that get deformed majorly from a few noises. The results of classification are as follows.

ith Fold	Train Accuracy	Test Accuracy (with Noise)	Test Accuracy (without Noise)
1	0.49	0.42	0.53
2	0.51	0.49	0.57
3	0.52	0.49	0.51
4	0.43	0.39	0.56
5	0.52	0.43	0.56
6	0.48	0.42	0.48
Average	0.492	0.44	0.535

Table 2: Results of training and testing on noisy images

As seen in Table 2, the models perform significantly worse on data with noise. The models trained on noise do about 10% better on classifying data without noise than those with noise. Once again, we took the model with best accuracy from above and formed a corresponding confusion matrix. The misclassifications are much more random than those without noise.

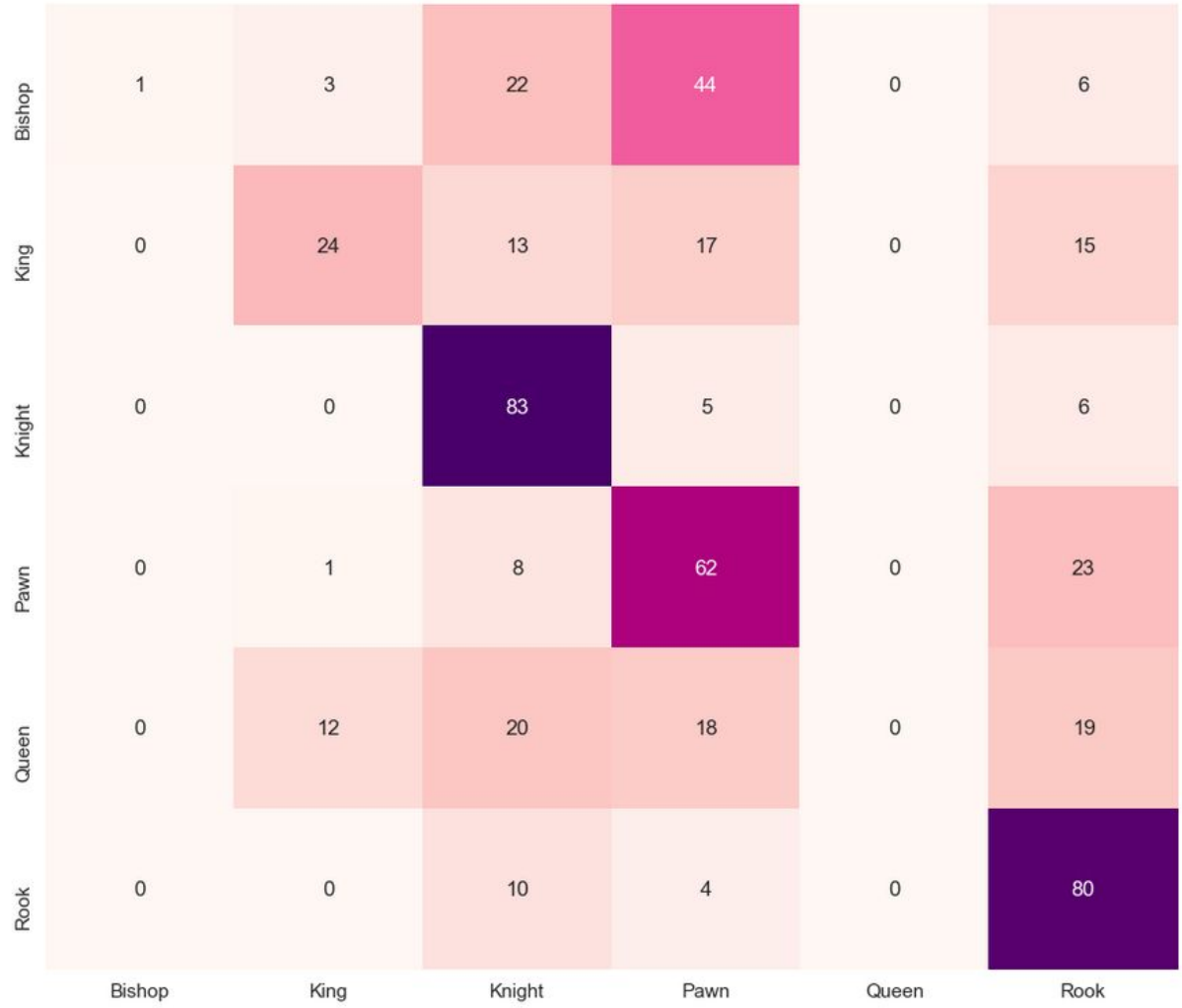

Figure 7: Confusion Matrix of Model on Training Data with Noise (X=Predicted, Y=True)

Principal Component Analysis (PCA) Approach

As our alternative approach, we implemented principal component analysis. Due to the high dimensionality nature of our data, we needed to try a dimension reduction approach and combine it with other prediction models. Based on our experience and research of multiple articles including:
https://www.analyticsvidhya.com/blog/2018/08/dimensionality-reduction-techniques-python/
https://www.analyticsvidhya.com/blog/2016/03/pca-practical-guide-principal-component-analysis-python/, we decided to use Principal Component Analysis (PCA) to reduce dimensionality of our data.

We were able to use PCA implementations within the sklearn library to reduce images in our dataset. Initially, we had to standardize the image aspect ratios since images in the dataset didn't have a standard size. Different standard image sizes were tested to find optimal ratio size for the dataset: PCA was done setting images to [100x100...600x600] sizes. As part of the standardization process we also used a built in function to normalize the data, which is usually recommended/required when running a PCA analysis: we used the normalize implementation within sklearn.preprocessing library In addition, we did our dimensionality reduction using both grayscale and original colors.
https://www.kaggle.com/hamishdickson/preprocessing-images-with-dimensionality-reduction was used as a reference point when devising our approach.

PCA was fit on the training data set which was predetermined using a 80:20 train/test ratio. We then attempted to find the principal components that captured 95% variance of the dataset. We then transformed our training feature dataset (train_X) and testing X feature dataset (test_X) using the determined components.

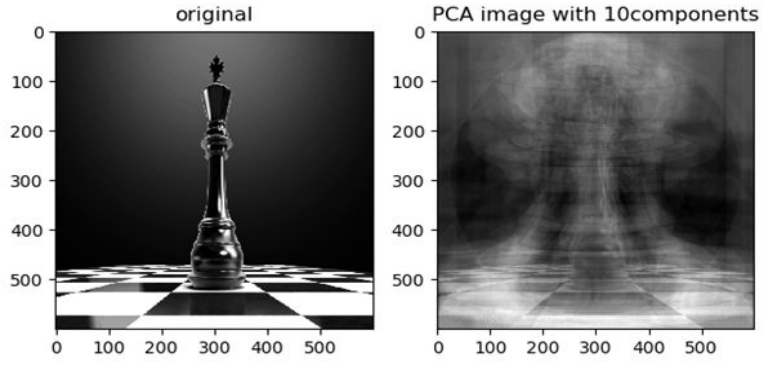

Figure 1: We used set no of components manually to 10, this approximately captured 76% of variance within features

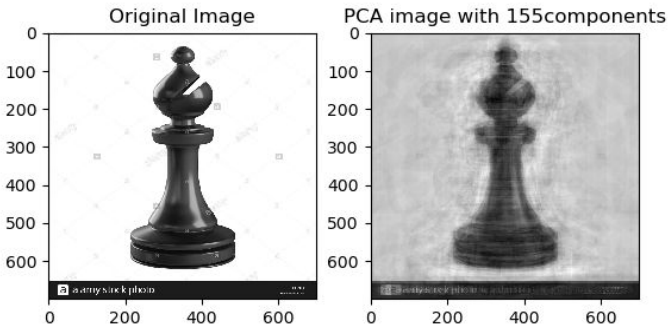

Figure 2: We used no of components that explained 95% variance within the features

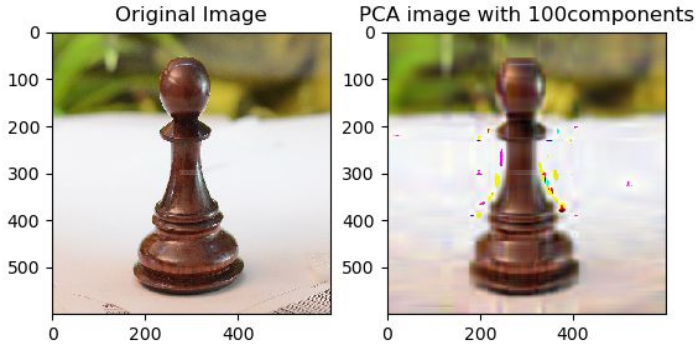

Figure 3: PCA reconstructed image using 100 Components with 95% explained variance

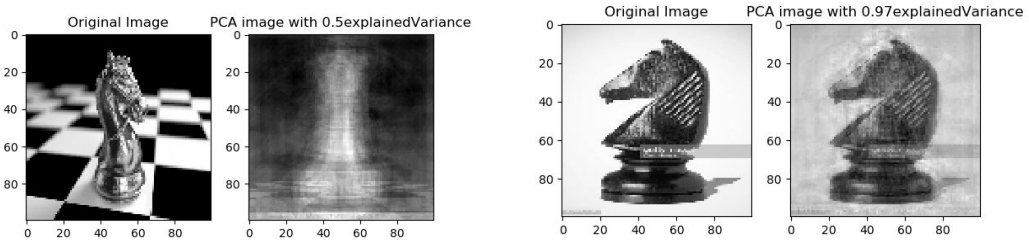

Figure for PCA reduced image comparison for components that explain variances of 50% and 97%

Model Background and Parameters

We used common classification models recommended for image classification problems. Although machine learning models have shown to perform less compared to deep learning ones, we wanted to examine how using PCA would improve their performance (https://medium.com/analytics-vidhya/cnn-convolutional-neural-network-8d0a292b4498).

After numerous experimental runs, we found that these 4 listed models performed better, hence were kept as our set of models.

- **Adaptive Boosting:** this model is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.
    - We set our base estimator(estimator from which the boosted ensemble is built) to a Random forest with 30 trees whereby each tree had a maximum depth of 5 leaf_nodes.
    - The maximum number of estimators where boosting is terminated was set to 30.
- **Random Forest:** an estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The whole dataset is used to build each tree in our version of the Random Forest model.
    - We used 30 trees as our estimators. We selected this value for computational complexity reasons since our Adaptive booster wouldn't compute in optimal time when values set higher than 100.
    - Like in the adaptive boosting, we used a maximum depth of 5 leaf_nodes.

- **Multilayer Perceptron:** this model is a type of feedforward artificial neural network (ANN). Multilayer Perceptron(MLP) utilizes a supervised learning technique called backpropagation for training. Its multiple layers and non-linear activation distinguish **MLP** from a linear **perceptron**. It can distinguish data that is not linearly separable.
    - We used a stochastic gradient-based optimizer known as adam
    - We set the number of epochs to 400.
    - All hidden layers were set to sizes 20.
- **Support Vector Machine:** a discriminative classifier formally defined by a separating hyperplane. Given labeled training data (supervised learning), the algorithm outputs an optimal hyperplane which categorizes new examples. In two dimensional space this hyperplane is a line dividing a plane in two parts where in each class lay in either side.
    - We implemented our model using a kernel that was gaussian. We also tested linear kernels before settling the gaussian kernel. Note: Polynomial kernel was tried but the kernel was omitted after the model wasn't able to compute in 10 minutes.
    - Kernel coefficient was set to 0.001

Program Instructions (Classifier.py)

- Program doesn't require additional inputs to run, might require some scikit-learn to be installed.
- Options for the test experiment can be determined by changing the size, ratio, color, comp, expVar, options within the program.
- Comp: can be used to override the number of components to be used. Leave as 0 to have a number of components that would explain 95% of the variance (or expVar if changed).
- Color: 'y' to read images with color, no for grayscale images.
- expVar: determines what percentage of variance components would explain
- Size: standard image aspect ratio size; width and height of images are set to be the same.
- Ratio: amount of data to be used for training use 0.8 instead of 80%.

Outputs: will have a random image from data that shows comparison between PCA compressed and Original image. In addition, there will be an excel file containing results from multiple iterations of the run using different models and options.

Results

As illustrated below, our ensemble methods Random Forest and Adaptive Boosting have shown to work better than Support Vector machines and Multilayer Perceptrons. In the case of ADA, highest accuracy was found to be 36%; this model also has a mean accuracy of 26% across the different experimental iterations. Although quite lower than what aimed for, this approach shows promise and evidence that it is learning as it's performing higher than random guess (~17%). Support Vector Machines seemed to be weaksest learner among candidates. This could be a result of kernel selection. Since computing resources have restricted us to linear and gaussian kernels. Polynomial and Sigmoid Function kernels could work better for this dataset.

Model Performance charts:

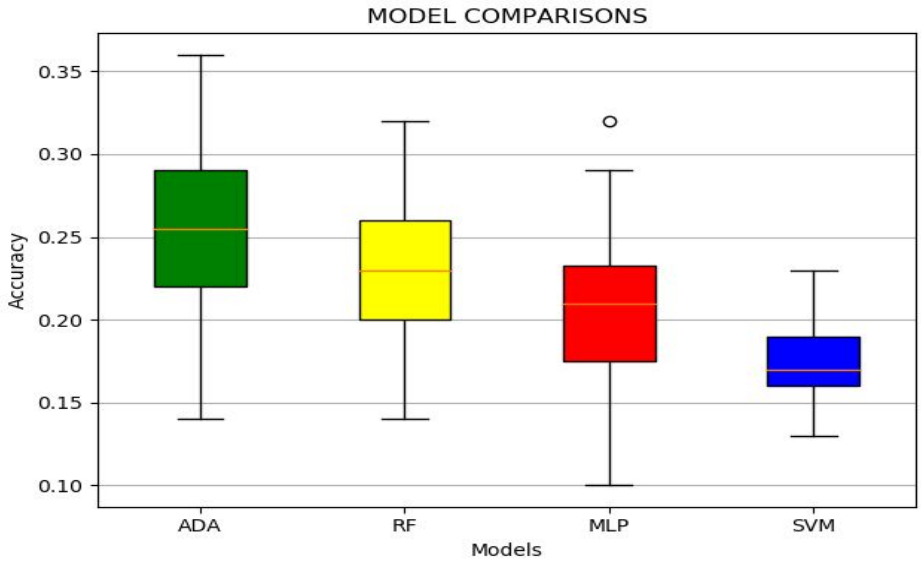

Figure represents model performance charts. *Orange line represents mean accuracy among numerous runs.*

PCA hasn't improved our overall predictions much. The 36% overall highest accuracy achieved with ADA model has been replicated with non-transformed dataset. We got 38% accuracy without using PCA with ADA models. However, PCA has helped with computing significantly. It has reduced running time by lowering the no_features from (width * height) to roughly 105 - 156 components: width * height usually coming to >10,000 features.

Below we have shown our PCA chart. Chart depicts where increasing the number of components doesn't add to variance retained in the dataset. This value seems to converge around 150 components.

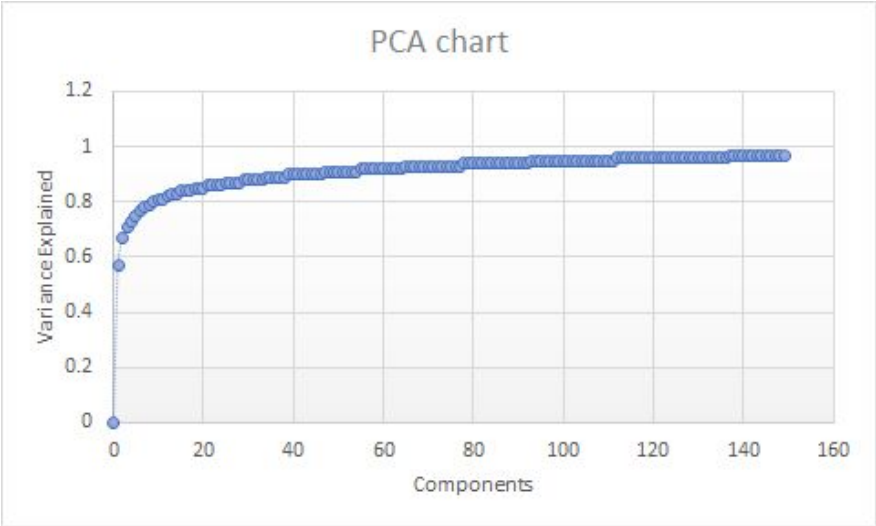

Figure shows Number of Components vs Explained Variance graph.

Figure below shows selecting a number of components that preserves variance of original features is instrumental for models to accurately predict. Models trained using data transformed with components that explained < 50% performed poorly. However, on the other hand, it also shows that components with explained variances greater than 75% don't show any performance patterns.

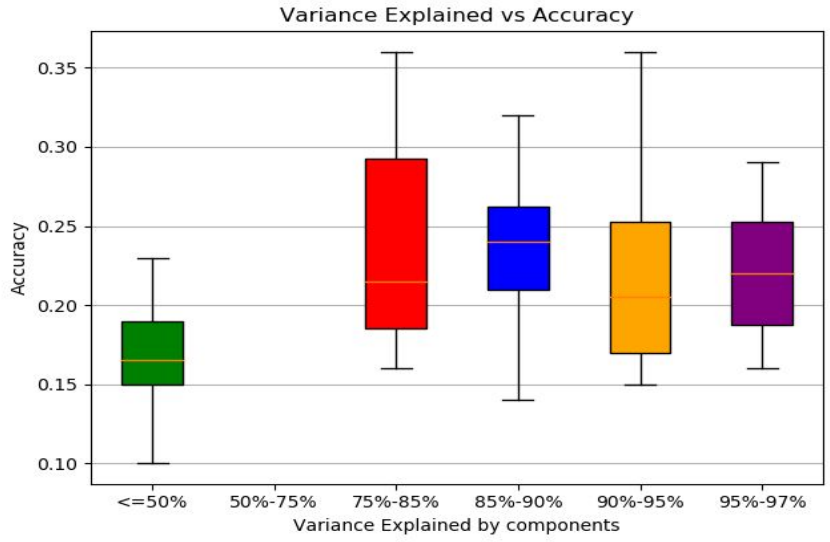

Figure represents the Accuracies achieved selecting components that explained the given variances.

Top 5 model results by performance are listed in the following table. As seen here it seems that the highest Accuracy is around 36%.

MODEL	**ACCURACY _PCA**	**ACCARACY _NoPCA**	**Components_ForPCA (explained variance)**	**Image Size**	**Train/Test Split**
ADA	0.36	0.35	100 (95%)	100	0.8
ADA	0.36	0.31	21 (85%)	200	0.8
ADA	0.32	0.37	20 (85%)	600	0.8
RF	0.32	0.31	20 (85%)	600	0.8
RF	0.32	0.26	100 (95%)	100	0.8

This table contains top 5 results from experimental runs

Overall based on these results we do see some improved performance vs random choice, but they may still need work to make them more accurate.

.

Conclusion

Based on the approaches we explore in this project, we think the one that would most accurately classify chess pieces would be CNN. Even when we used other proposed approaches, like PCA to reduce the image dimensionality combined with other AI models, the results were not able to match the prediction accuracy we experienced with CNN.

Other things we wanted to consider for the future to further test and/or improve performance would be:

- Expand dataset to include non chess images
- Train CNN model on images both with and without static noises
- Try other dimensionality techniques other than PCA. Some ideas would Non-negative Matrix Factorization and Chi^2 dimension reductionality methods
- In case of PCA approach, resizing the image shapes in a different way to feed PCA and machine learning methods. Currently converting images into numpy array which had initial shape of (samples, width, height, RGB) to (samples, width*height*RGB) array.
- Use a gridsearch to tune model hyper-parameters more effectively for our dataset.
- Experiment with XGBoost models
- Overall, the PCA approach didn’t result in favourable outcomes, but has room to be improved. Nevertheless, it is proven to be better than random prediction since for our case random would ~ 16.67%.