Project 2 Report

By: Harun Gopal

Table of Contents

Introduction

This project entailed implementing a linear Support Vector Machine (SVM) classifier and a convolutional neural network on a MNIST dataset. This included designing both classifiers and evaluating their performance with different parameters. In this report I will discuss a few of the experiments I conducted with both classifiers to better understand their operation. With respect to the linear SVM classifier, I adjusted the preprocessing parameters to view the effect of PCA and LDA would have on a SVM classifier. With respect to the convolutional neural network, I changed the optimizer used and the learning rate, as well as adjusted the number of convolutional features to view their effect on the accuracy of the system. I will not discuss my code directly, only the results I have obtained. All my code is contained within the code folder and is thoroughly commented if you are interested in how I achieved my results

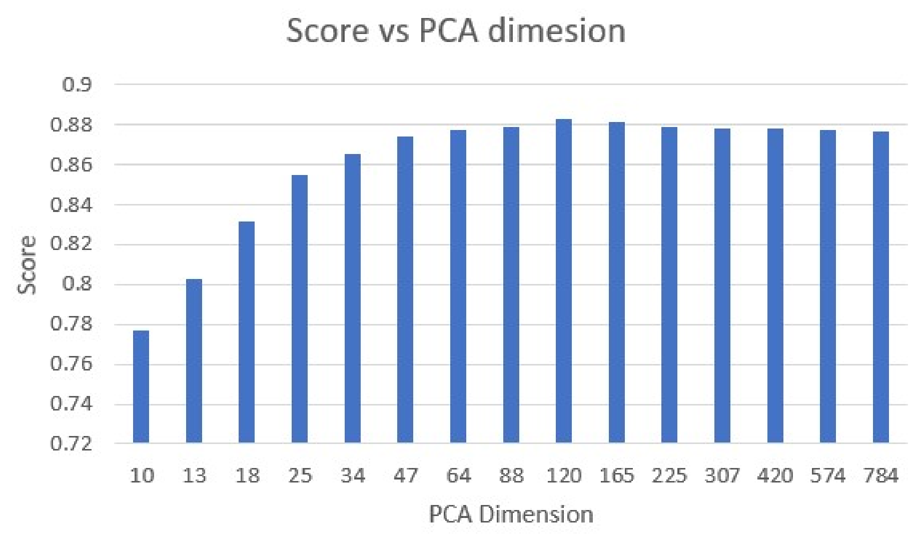
Classifier One: Linear Support Vector Machine

Implementation

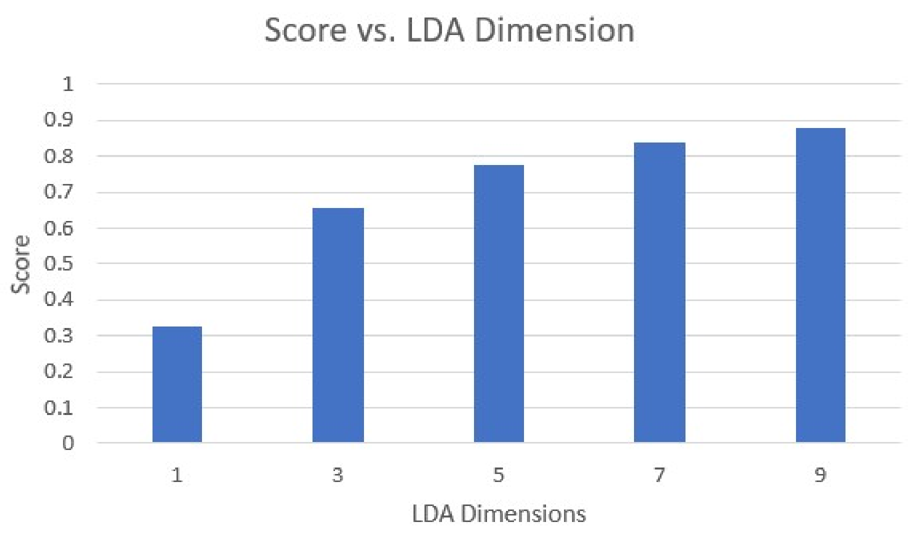
I implemented this classifier in python using libraries available through Scikit-learn. I used the SVM, PCA and LDA subcomponents part of sklearn library to implement and preprocess the classifier. The preprocessing pipeline included flattening each image to a 1 dimensional array and then performing PCA followed by LDA on the data before training the SVM algorithm.

PCA and LDA effect on SVM outcome

I experimented with the preprocessing pipeline by altering the number of dimensions I would reduce via PCA and LDA. After conducting these alterations, I would train and test the SVM algorithm to discover the effects the preprocessing steps would have on the overall accuracy of the data. I designed and ran a script that tested every combination of PCA and LDA dimension from a given set (this script can be found in my code folder).



When determining the effect of PCA on the SVM, I used the preprocessing pipeline described earlier and maintained the LDA dimension reduction to 9 dimensions while adjusting the number of PCA dimensions. The resultant graph is shown above. The final score ranges from .78 to a maximum of .88 due to the alteration of PCA dimensions. There is also a logarithmic relationship between the number of PCA dimensions and the score. After about 64 PCA dimensions there are diminishing returns to the score variable.

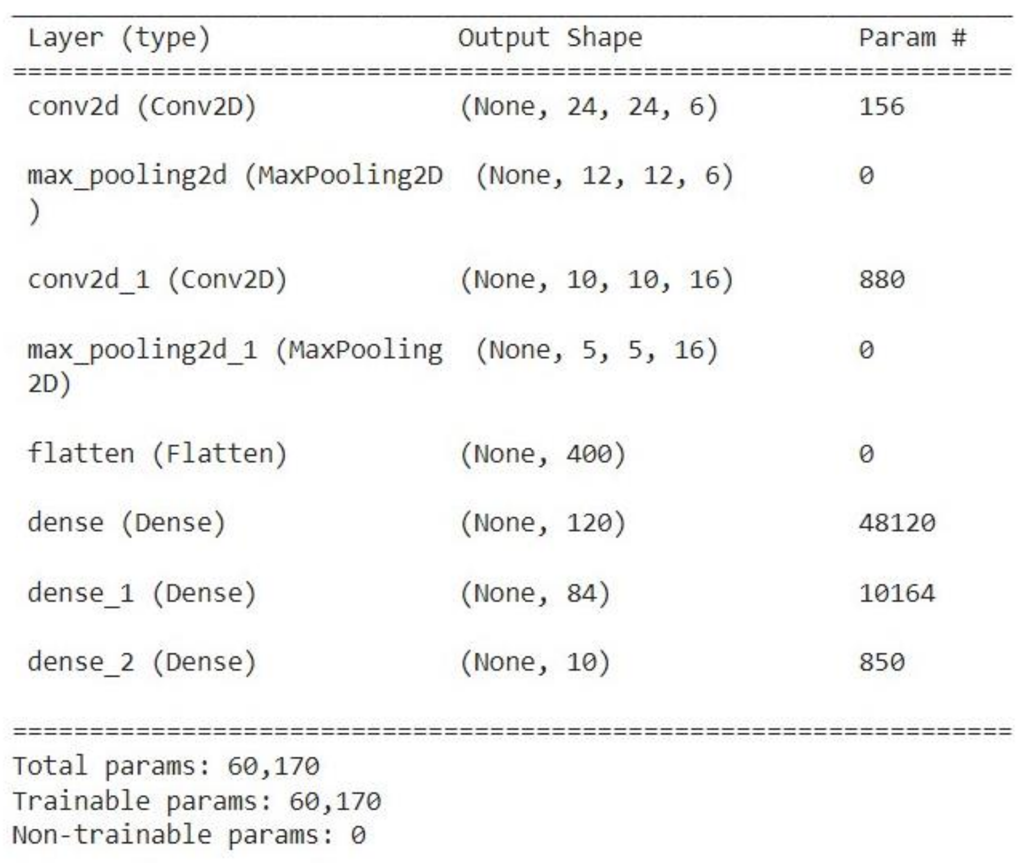


Similar to my PCA experiment, with LDA I held the PCA dimensions at a constant value as I changed my LDA dimensions and viewed the effect on the score of my SVM. The LDA has a much greater effect on SVM as the score ranged from .3 to .88. Although LDA also shows a logarithmic relationship, there is no point at which improvements become negligible. This leads me to conclude that the number of LDA dimensions has a greater impact than the number of PCA dimensions of a SVM classification system.

Classifier Two: Convolutional Neural Network (CNN)

Implementation

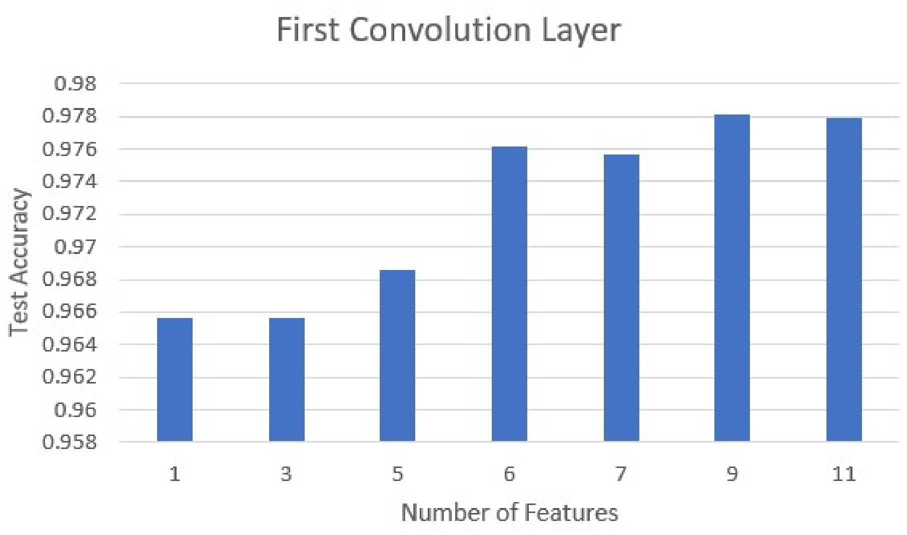
When creating my implementation of a CNN I started with the jupyter notebook provided which built a framework in keras with two convolutional layers fed into two full connection layers. With this as a starting point, I adjusted the layers and parameters in the following experiments. I did not preprocess the data prior to feeding it into the NN. Below is a printout of the baseline structure used in the NN:



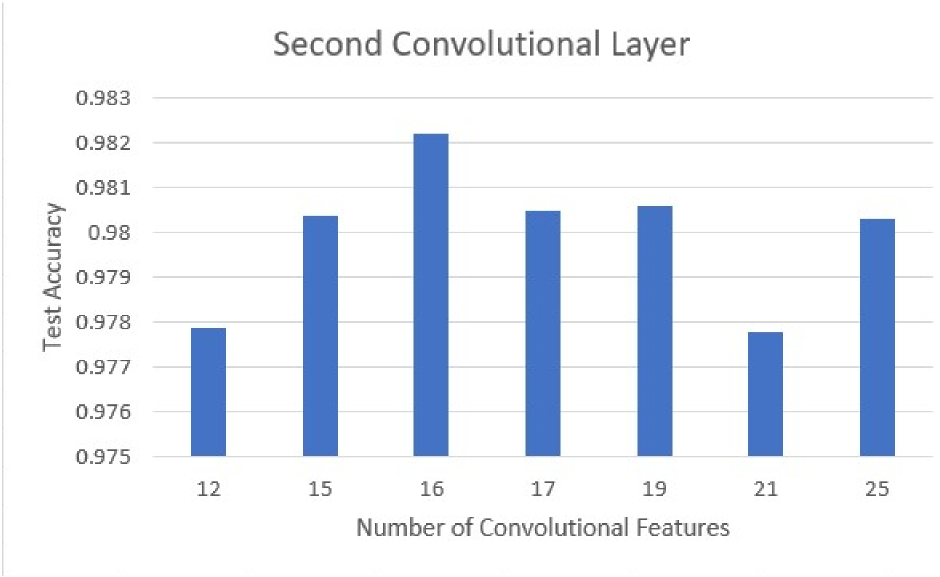
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 24, 24, 6)         156

 max_pooling2d (MaxPooling2D  (None, 12, 12, 6)        0
 )

 conv2d_1 (Conv2D)           (None, 10, 10, 16)        880

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 16)         0
 2D)

 flatten (Flatten)           (None, 400)               0

 dense (Dense)               (None, 120)               48120

 dense_1 (Dense)             (None, 84)                10164

 dense_2 (Dense)             (None, 10)                850

=================================================================
Total params: 60,170
Trainable params: 60,170
Non-trainable params: 0
```

Convolutional Layer Features

I experimented with the number of features in each convolutional layer to see what effect it would have on the overall accuracy of the model. This is useful because reducing the number of features in a convolutional layer decreases the number of parameters that need to be trained. This, in turn, reduces training time as well as the storage space a neural network will take up.



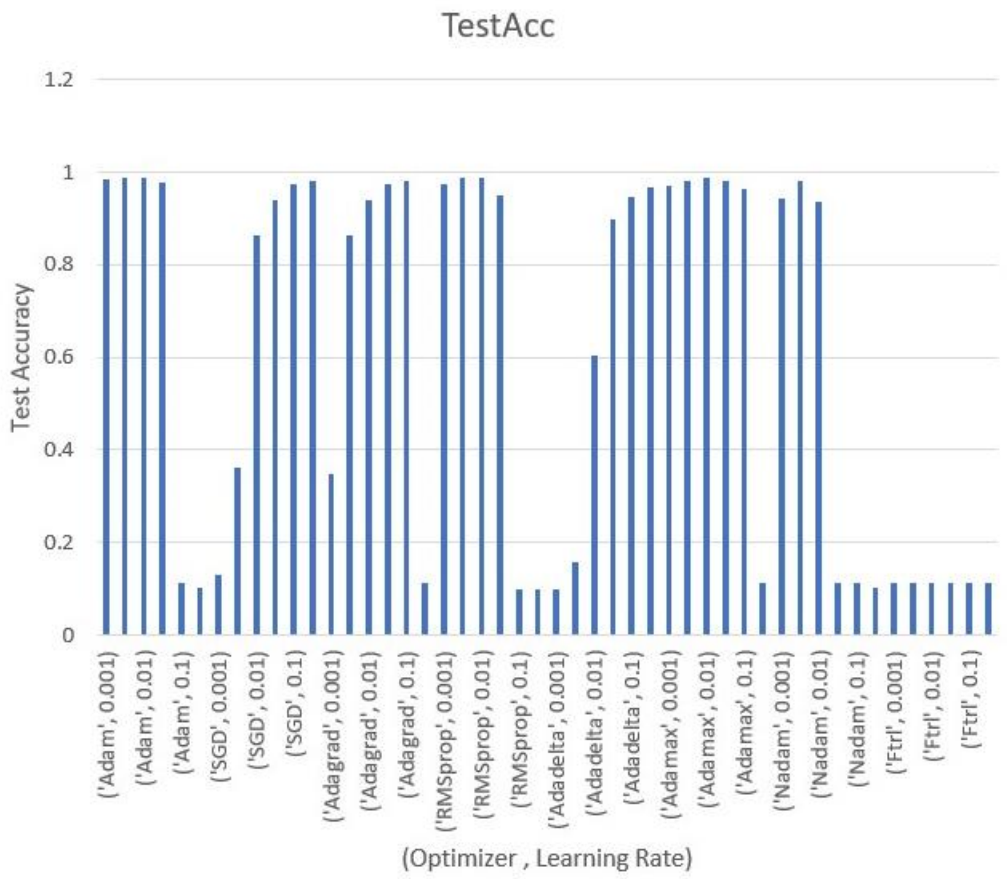
Since my baseline CNN had two convolutional layers, I adjusted the number of features in each independently. This would also let me see which layer had the greatest impact on output performance. Above is a graph relating test accuracy to the number of features present in the first convolutional layer, while holding the number of features in the second layer to be 12. There is a clear correlation between the number of features and better performance. However, one fact that surprised me was that with even just one feature, the CNN was able to score above 96% accuracy. When I increased the number of features I was only able to bring up that number to around 98%. This indicates that even just one feature in a convolutional layer can make a huge difference to a models performance but additional features will only marginally benefit the model.



Next I adjusted the number of features in the second convolutional layer and recorded the test accuracy shown in the graph above. I set the first convolutional layer to maintain 11 features as I adjusted the number of features in the second layer. This data shows no significant correlation between the number of features in the second convolutional layer and the test accuracy. This signifies that when layering multiple convolutional layers on top of one another in a NN, many features is not necessary as they don't yield a large increase in accuracy. This also suggests that there is diminishing returns to have more than two convolutional layers stacked on top of each other with large number of features.

Optimal Optimizers

To determine the optimal optimizer, I tested a variety of optimizers each with different learning rates and saw what the testing accuracy of each was after 5 epochs of training. I also recorded the training and testing accuracy after each epoch and stored that data in a csv. Below is a graph of every optimizer I tested at various learning rates and their testing accuracies at the end of 5 epochs.



I conducted this experiment to discover two things. First, I wanted to see which optimizers were the best for this problem. I also wanted to see the impact of learning rates on the testing accuracy for each of the optimizers. To begin, some optimizers are much better suited for this application for others. The best training accuracy occurred when using the ADAM optimizer at the proper learning rate. On the other hand, the FTRL optimizer didn’t produce an acceptable testing accuracy regardless of learning rate. This suggests that the FTRL optimizer isn’t suited to learn in this application.

Another interesting finding is the optimal learning rate for each of optimizers. For example, for the ADAM optimizer, the best learning rate is .01 and when it surpassed that, the testing accuracy would plummet. This trend is shared across most of the optimizers tested. The SGD optimizer learned very slowly so even when the learning rate was set to .1, the model performed better than before. SGD is more of an exception, as most other optimizers shown peak at a certain learning rate and then decay past that.