ENEE 633: Statistical Pattern Recognition

Project 02: Implementing a Digit Recognizer

Anirudh Nakra

University of Maryland,
College Park
Prof. Behtash Babadi

December 7, 2021

Abstract

In this project, I implement a digit recognizer on the standard MNIST dataset. Using techniques like Kernel SVM, Logistic Regression and Convolutional Neural Networks, I achieve accuracies of above 90 percent on the dataset. I also apply standard dimensionality reduction techniques to make the data easier to work with and implement on. In the second part of the project, I benchmark my CNN on the Monkey Dataset and apply transfer learning to make use of the power of pretrained models.

1 Data Preprocessing

1.1 Handling Data

Data handling is a major part of the project although the MNIST dataset can be easily imported from any of the standard toolboxes. Using keras, I imported the MNIST dataset using the load_data() function. I further go on to convert all the data into a single array using matrix operations so that I am able to manipulate it easily. I also create a complete matrix with all the dataset including the train and the test partitions in a large numpy array. As shown in the histogram in the experiments section, the MNIST dataset is very balanced with near equal data for all the 10 labels. The data is scaled and regularized before feeding it into any classification technique to make working on the data computationally efficient.

1.2 Dimensionality Reduction

Using PCA and MDA, I reduce the data size. The MNIST images are 28*28 in size and are encoded in a grayscale channel. This means that for every sample (*i.e image*) there are 704 pixels, each ranging from 1 to 255. Using the abovementioned techniques, I reduce the dimensions from 704*1 to something much more workable. The choice of the dimensions was a hyperparameter tuned during the creation of the program itself. Since LDA reduces the datasize to atmost 1 less than the total number of classes, the dimensionality can be reduced from 704*1 to 9*1 and lower.

For PCA, the things were a little more complicated. I underwent quite a lot of trial and error when understanding how to correctly implement it on the dataset. I undertook three methods. In the first method, I conjoined the training and testing dataset into a single large dataset and performed PCA reduction on it. This is clearly a wrong way of thinking about dimensionality reduction since we want to keep our training and testing set as distinct as possible. The correct way of approaching it was then to apply the PCA transform on the training dataset and use the eigenvalues calculated to perform the same PCA transform on the testing set without fitting to it. This would prevent the data leakage issue that arose from the previous issue. This method is also viable for the case when the data is already provided to us in a batch form and we have a readily available testing data rather than one sample incoming at a time.

However using this method, there might be issues with generalising to future testing sets. I think this is not a big issue since the top X eigenvectors can be easily stored for future use along with the model (*where X is the number of dimensions we are reducing the data to*). We can also choose to perform a separately perform PCA on every batch of testing data we get, but there is an inherent issue of the testing and training data being projected in different directions even though this might help with generalization issues. After evaluating the two logically sound ideas, I went with using the PCA transform of the training set to transform the test data. For most of my experiments, I have chosen the value of my reduced dimensions to be 20*1 which is much lower than 704 and storing 20 eigenvalues is not a huge concern.

1.3 Data for CNNs

The transformed data can be easily used to solve the SVM problem using the SVC() function which is based on libSVM. However for CNNs and especially when we are implementing standard architectures such as Lenet, the data needs to be in a very specific form. For example, for Lenet-5 the data needs to be padded from a 28*28 image to a 32*32 image so that the convolution gives us results similar to what Lecunn Et al. got when they were implementing the architecture. Furthermore there are some library based constraints. The tensorflow and keras models require the data to be explicitly in the form of a tensor with channels and other important parameters defines. Aside from the standard regularization and scaling, I also needed to expand the numpy array data into a tensor with the parameter *num_channel* as 1. After this, the data was ready to be put into the CNN architecture after the model was created.

1.4 Data for Transfer Learning

The data for the transfer learning task was perhaps the most complicated of the bunch to read. However, after many years having data in this format has become very standard and there are utilities in most DL packages to allow us to manipulate the data easily. Particularly, the keras library has an object called ImageDataGenerator which allows us to bypass the complicated procedure of accessing paths and appending and creates an object which can be directly passed into the CNN model to use. For each architecture, the data was converted into a form that was acceptable by the input layers as desired in the original research papers.

2 Classification Techniques

2.1 Kernel SVM

The kernel SVM is an interesting classifier to implement. In the project, I implemented the SVM classifier using linear, polynomial and RBF kernels. Rather than resorting to solving the primal SVM problem, I approached the dual problem instead. The dual problem can be neatly reconstructed into the following equation:

$$\min_{\alpha} \frac{1}{2}\alpha^T G\alpha - \alpha^T \mathbb{1}$$

This is now in the form of a standard quadratic programming optimization problem. There are several solvers available in the form of packages. In Python, the library sklearn has various efficient implementations of support vector machines and I have used the function SVC() based on libSVM which allows me to manipulate a lot of hyperparamaters such as the kernel and even allows in built cross validation using GridSearchCV. This however takes a lot of time and has been commented out.

2.2 Logistic Regression

For Logistic Regression the implementation was pretty straightforward. The sklearn library has many resources available to solve the regression fitting problem and optimise it. For the purpose of my experiments, I tuned the hyper-parameters of the Logistic Regression model and ran different experiments on the models. The major experimental setups that I created can be enaumerated as below:

- <u>Implementing Class Balance Weights</u>: Created a weighted Logistic Regression classifier for the MNIST database.
- <u>Performing Cross Validation</u>: Implemented Cross Validated Logistic Regression.
- <u>Using L1 penalty</u>: Used L1 loss instead of L2 loss.
- <u>Comparing Solvers</u>: Compared solvers such as SAGA, liblinear, Newton-CG, etc.

2.3 Convolutional Neural Network

2.3.1 Architecture

LeNet 1 is a Convolutional Neural Netwrok proposed by Lecun et al. in 1990 and was one of the greatest breakthroughs in the field of neural modeling at the time. The model consists of two convolutional layers with a subsampling layer attached to each. At that time, the pooling layer was designed to be trainable but over time, this practice has been stopped and this is something I noted in my architecture design.

Similarly the LeNet 5 is a deeper convolutional NN which came to be in 1998 and brought about a major chnage in the way people approached NNs. This architecture consists of three convolutional layers with a subsampling layer attached to each of them. It also has fully connected layers after the last convolutional layer and terminated the NN with a gaussian connection.

Both of the architectures use the now archaic activation and loss functions like tanh which have been dropped in favour of ReLU and leaky ReLU. However I have modeled the CNNs with tanh activations and have experimented with ReLU in a separate architecture that modifies both the LeNets.

2.3.2 Modifications/Experiments

- <u>Changing to ReLU</u>: I changed the activation functions in the design of the LeNet model to ReLU's and they performed better than tanh for both LeNet 1 and 5. The accuracy rose from about 98% to 99.14% which is a huge gap created just from the change to modern activation designs.
- <u>Changing Kernel Sizes</u>: It has been found that two stacked 3 X 3 kernels usually perform better than a single 5 X 5 kernel for convolution. This experiment was very insightful into modern architectures and the accuracy rose from 97.5 % to 99.24 %.

#	Layer	Size
0	Input	1 @ 28*28
1	Convolution	4 @ 24*24
2	Avg Pool	4 @ 12*12
3	Convolution	12 @ 8*8
4	Avg Pool	12 @ 4*4
5	Output	10 @ 1*1

Table 1: LeNet 1

#	Layer	Size
0	Input	1 @ 32*32
1	Convolution	6 @ 28*28
2	Avg Pool	6 @ 14*14
3	Convolution	16 @ 10*10
4	Avg Pool	16 @ 5*5
5	Convolution	120 @ 1*1
6	Fully Connected	84 @ 1*1
7	Output	10 @ 1*1

Table 2: LeNet 5

- Evaluating Losses and Dropout: I justified the use of Categorical Cross Entropy loss by showing that the MSE error performed very bad and the optimisation problem was non-convex. I also introduced dropout to the architecture and noticed a drop in validation accuract by around 4% seemingly improving future generalizations.

- Evaluating Optimisers: I evaluated the ADAM, SGD and AdaGrad optimisers for the CNNs I made.Through experimenting with them, I found that the ADAM optimiser is much much better than the SGD method with it taking only one iteration to reach atleast 50 % validation accuracy where the other optimiser takes atleast 5. ADA-Grad is even more volatile in training and keeps getting stuck at some points. It also never reaches an accuracy of above 90 % in 40 epochs which justifies the choice of ADAM in cutting edge NN training.

3 Transfer Learning

3.1 Benchmarking

After loading the data using ImageDatagenerator in keras, I created a simple CNN to benchmark the dataset and how it performs. The structure I used was very similar to LeNet 1 with some modification in the input image size and other hyperparamaters. After training the CNN, I found that the validation set accuracy stagnated at around 62 %. Two techniques were used to combat this issue. The first one was using pretrained models to apply transfer learning on the task and use the trained weights with some fully connected layers to solve the classification task. The other

Dim Redn	Linear SVM	RBF SVM	Polynomial SVM
N/A	92.8%	-	-
PCA	90.01%	96.18%	95.53%
MDA	89.3%	92.08%	90.88%

Table 3: Accuracy for SVM's

Preprocessing	Normal LR	L1 Loss	Cross Validation
Balancing	87.24%	-	-
PCA	87.26%	85.73%	87.26%
MDA	88.57%	87.92%	88.57%

Table 4: Accuracy for Logistic Regression

more sneaky technique was to *interpolate* the data. Using data augmentation techniques such as rotation and translation, I was able to generate a dataset with redundancies that would be able to train a CNN effectively.

3.2 Pretrained Models

Through the course of the project, I applied transfer learning on 2 different pretrained models: **Inception V3** and **Xception**. Both the models were imported from keras with the weights that were trained for the ImageNet contest. Using keras processing functions, I was able to chop off the fully connected layers of the models and create my own FC layers. After locking the part of the model with the pretrained weights, the model was trained on the generator object created and the system reached very high accuracies of more than 92 % which is about 30 % more than the accuracies I found during the benchmarking procedure. I was also able to experimentally see the gulf in accuracies between the Xception model and the Inception v3 model and analyse the effect of depthwise separable convolutions. The Xception CNN was on average 5-6 % more accurate than the Inception V3 model which is a big difference considering both of them are atleast 90 % accurate.

4 Experimental Observations

4.1 Evaluating Performance

- CNN Experiments Through the project, I evaluated a lot of different layers along with the general LeNet 1 and 5 architectures that enabled me to improve CNN performance while creating experiments that verified intuitive and theoretical knowledge.

- Logistic Regression manipulations Rather

SAGA	Newton-CG	LibLinear
87.25%	87.26%	85.72%

Table 5: Solver Accuracy for Logistic Regression

Experiment	Validation Accuracy
LeNet-1	97.97 %
LeNet-5	98.18 %
Kernel Change	99.24 %
ReLU	99.14 %
Dropout	95.55 %
MSE	9% (Doesn't Train)

Table 6: CNN Experiments and Accuracies

Inception V3	Xception
93.01%	99.23%

Table 7: Transfer Learning Accuracies

than only work on the the normal Logistic Regression function, I experimented with different hyperparameter manipulations such as losses and solvers and compared them.

- Transfer Learning Comparisons I was able to successfully implement Transfer Learning using both Inception v3 and Xception which in turn allowed me to investigate depth based convolutional paradigms.

- Data Augmentation methods I implemented a data pipeline that was very easy to work with and using standard data augmentation techniques, I was able to effectively load my data into classifiers such as SVM and CNNs while incorporating redundancies where needed to help the training process.

5 Key Conclusions

I implemented a fundamental digit recognizer based on the standard MNIST dataset. For Kernel SVMs with dimensionality reduction techniques, I reached 90 % accuracy with the Linear Kernel and improved it to 95 % and 96 % accuracy using Polynomial and RBF Kernels respectively. Empirically, it has been shown that Logistic Regression performs comparatively as well as Linear SVM methods and this can be verified in my project with logistic regression achieving accuracy rates of $\sim$ 85-90 %. Furthermore, the power of Neural Networks can be shown with the designed CNNs being as high as 99 % accurate.

Transfer Learning has been implemented on Inception v3 and Xception models. We observe that even without training the filters and the convolutional layers, we reach as high as 94 % accuracy on the Monkey Dataset with Inception v3 architecture and 97 % using the Xception architecture. This is all evaluated using pretrained weights! This makes it even more amazing.

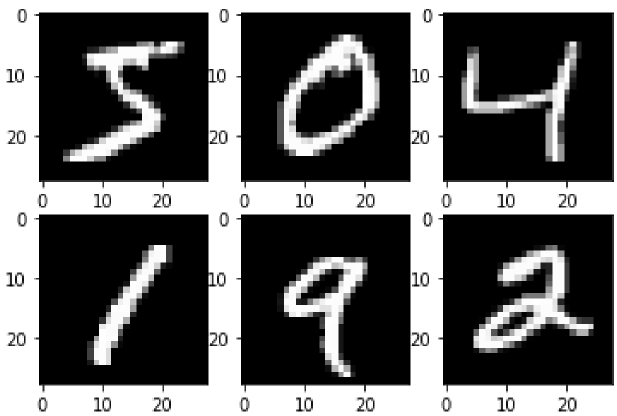

Figure 1: Visualising MNIST Train Splice

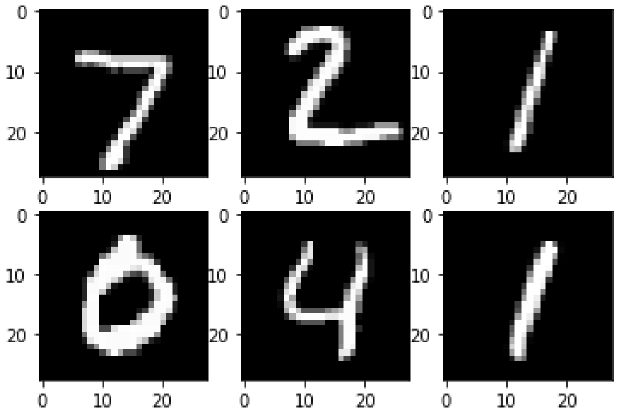

Figure 2: Visualising MNIST Test Splice

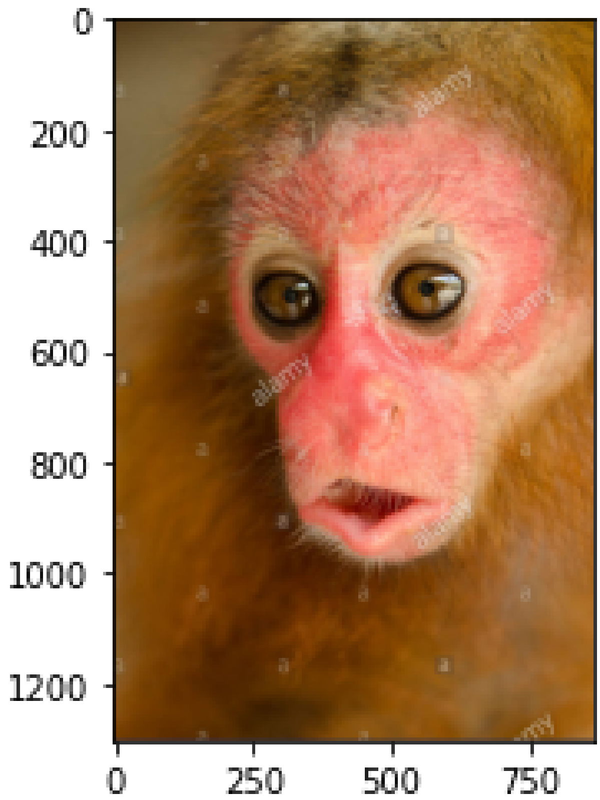

Figure 3: Visualising Monkey Images for Transfer Learning

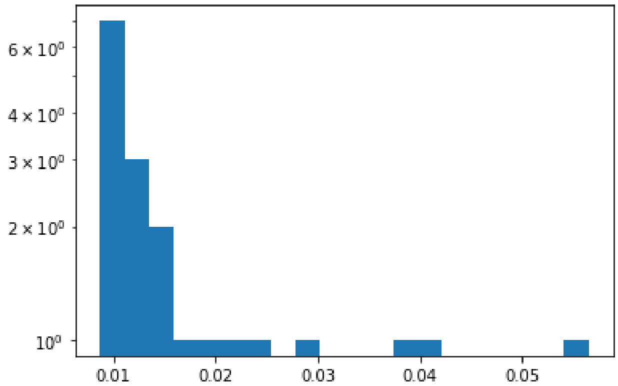

Figure 4: PCA Variance Histogram

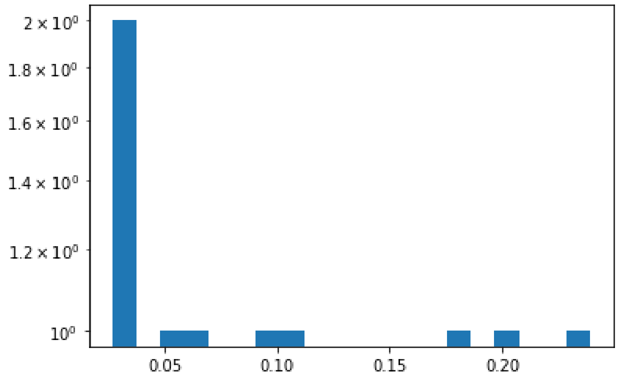

Figure 5: MDA Variance Histogram

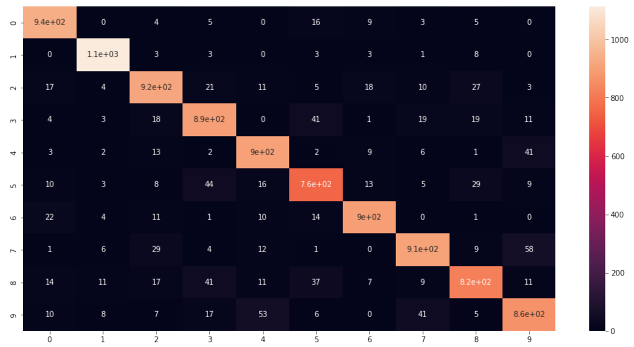

Figure 6: Confusion Matrix for Linear SVM with PCA

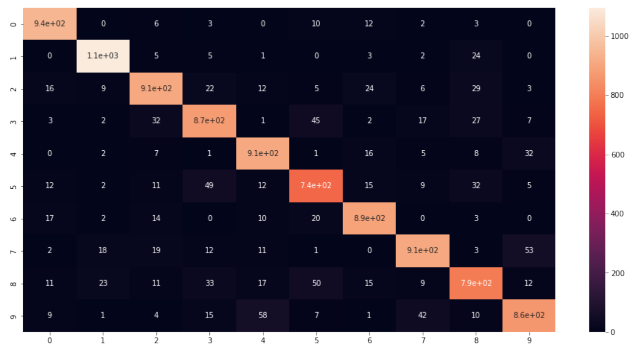

Figure 7: Confusion Matrix for Linear SVM with MDA

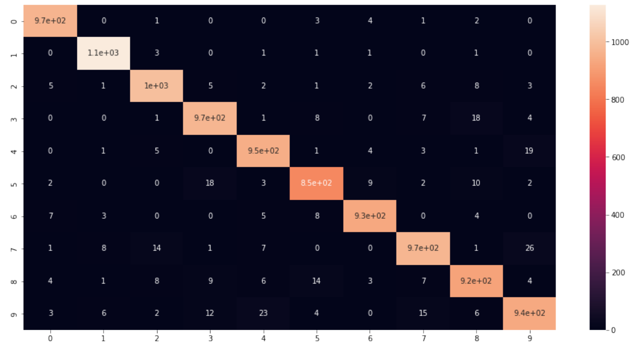

Figure 8: Confusion Matrix for RBF SVM with PCA

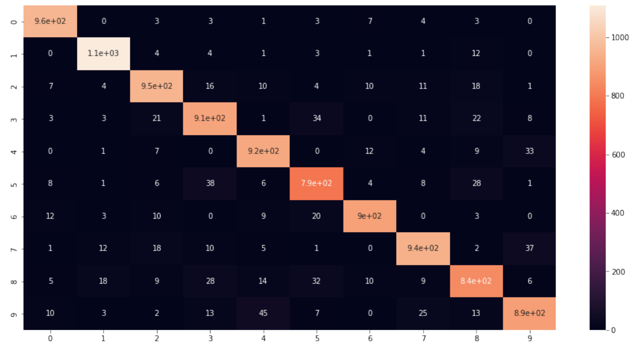

Figure 9: Confusion Matrix for RBF SVM with MDA

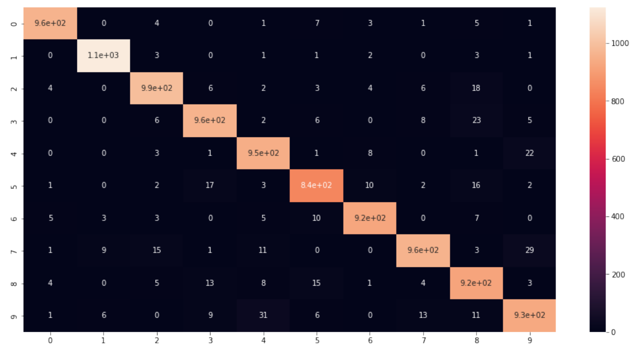

Figure 10: Confusion Matrix for Poly SVM with PCA

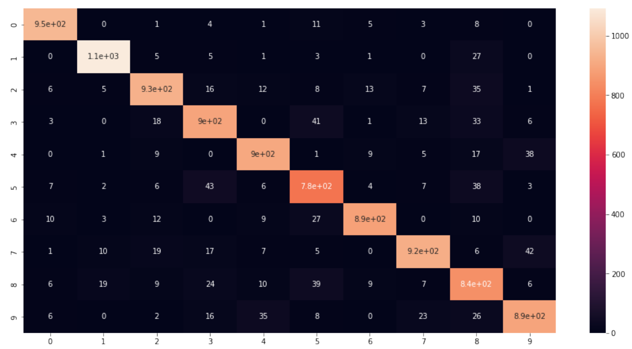

Figure 11: Confusion Matrix for Pol SVM with MDA

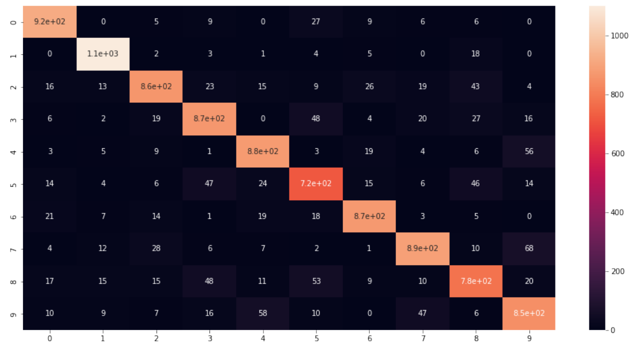

Figure 12: Confusion Matrix for Logistic Regression with PCA

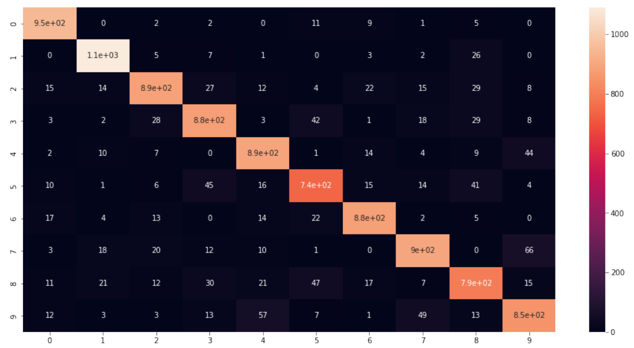

Figure 13: Confusion Matrix for Logistic Regression with MDA

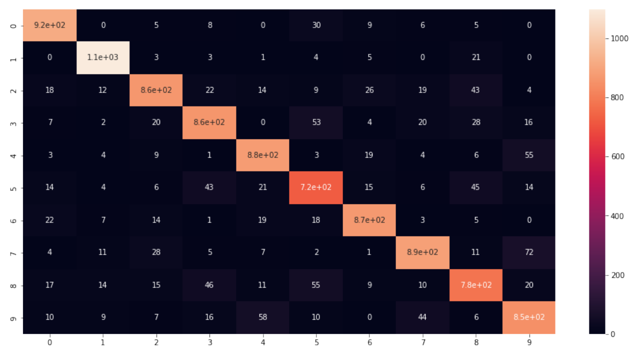

Figure 14: Confusion Matrix for Balanced Logistic Regression

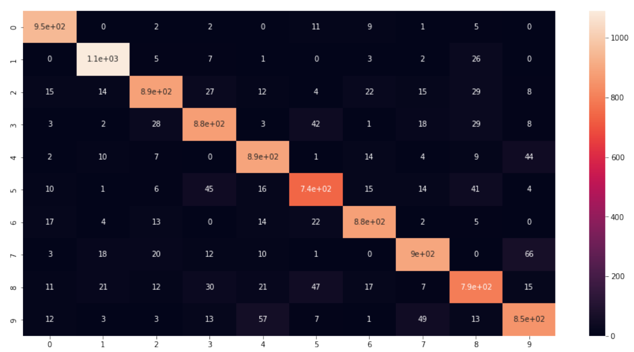

Figure 15: Confusion Matrix for Logistic Regression with Cross Validation MDA

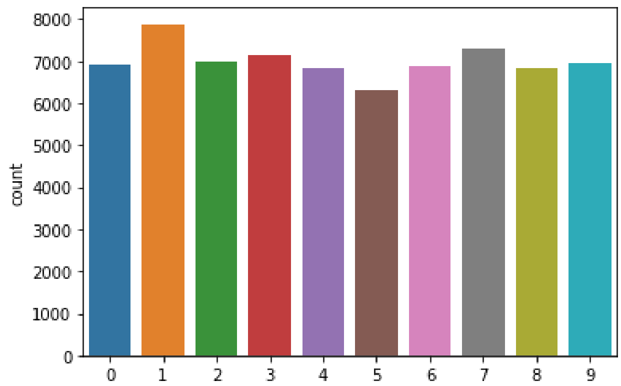

Figure 16: Label Balance of MNIST

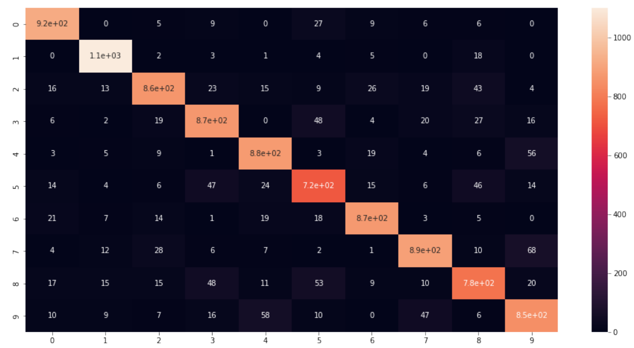

Figure 17: Confusion Matrix for Logistic Regression with Cross Validation PCA

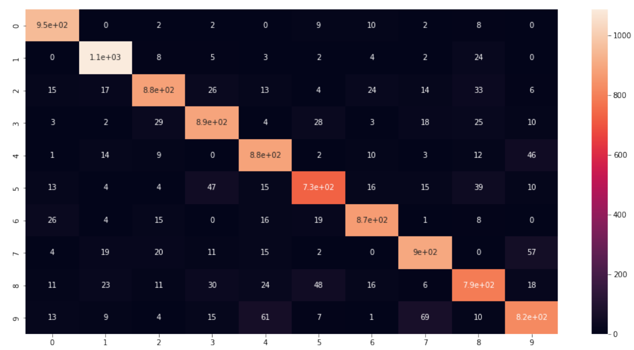

Figure 18: Confusion Matrix for Logistic Regression with l1 loss and MDA

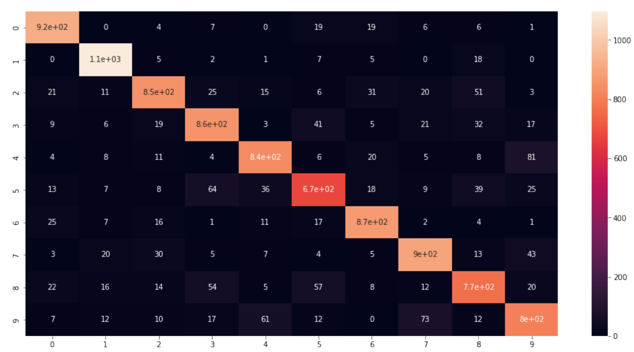

Figure 19: Confusion Matrix for Logistic Regression with l1 loss and PCA

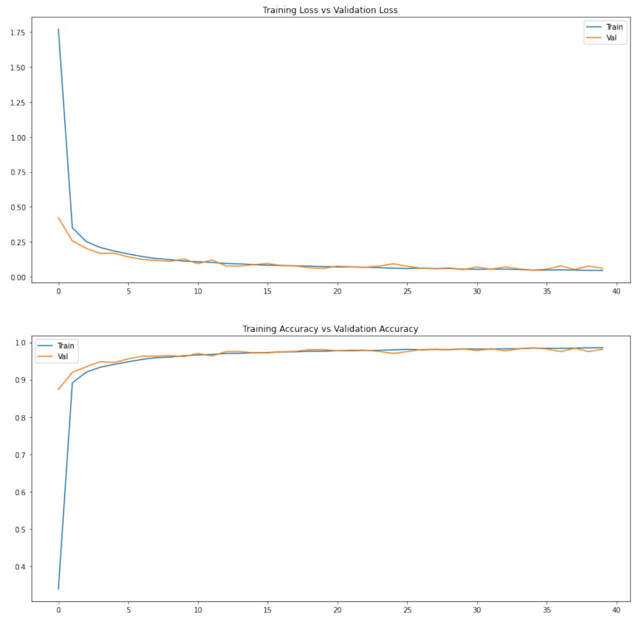

Figure 20: CNN LeNet-5 Training/Testing

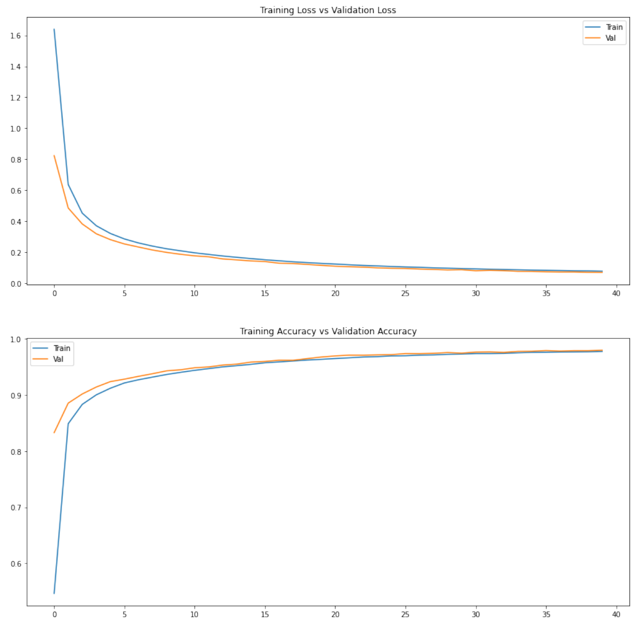

Figure 21: CNN LeNet-1 Training/Testing

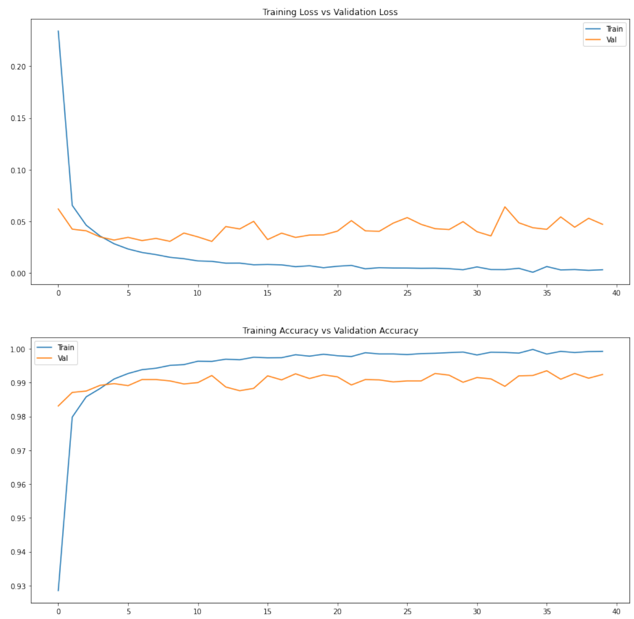

Figure 22: Changing Kernel Size from 5X5 to stacked 3X3

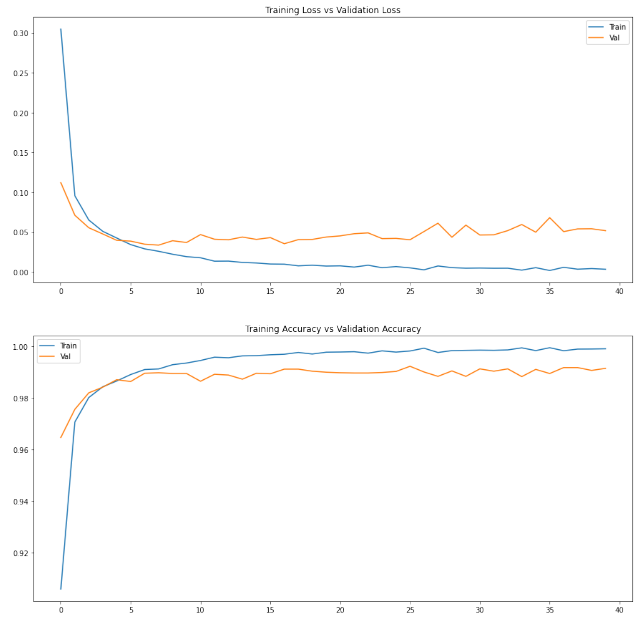

Figure 23: Changing Activation Functions to ReLU

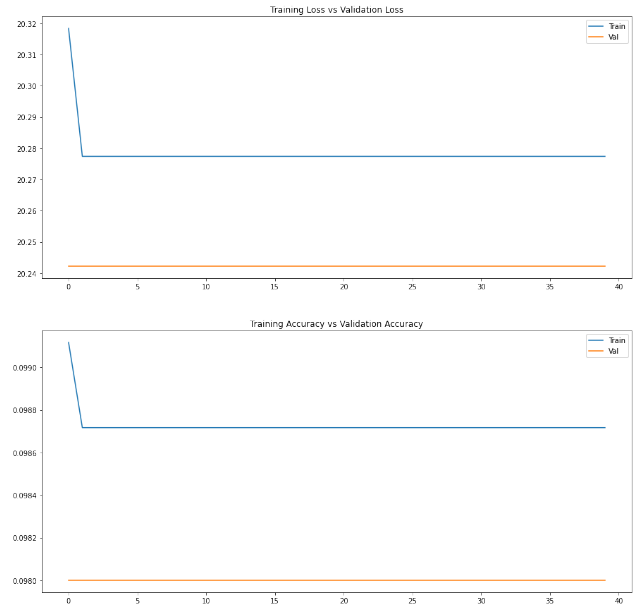

Figure 24: Investigating MSE Loss

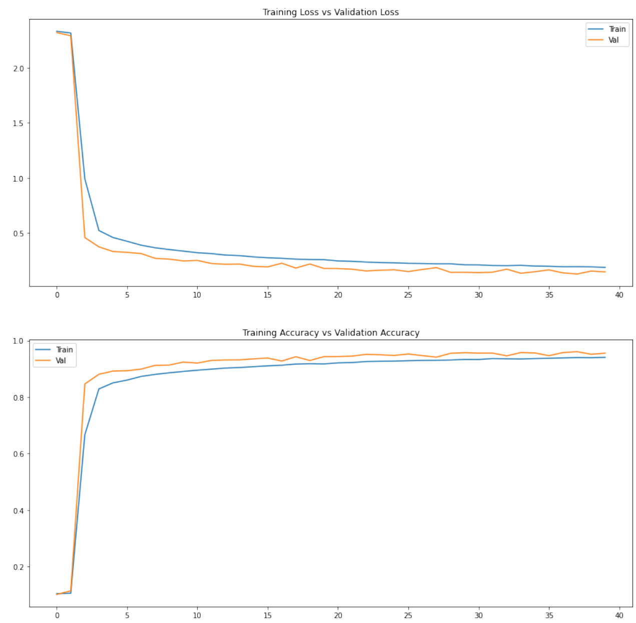

Figure 25: Checking Impact of Dropout layer

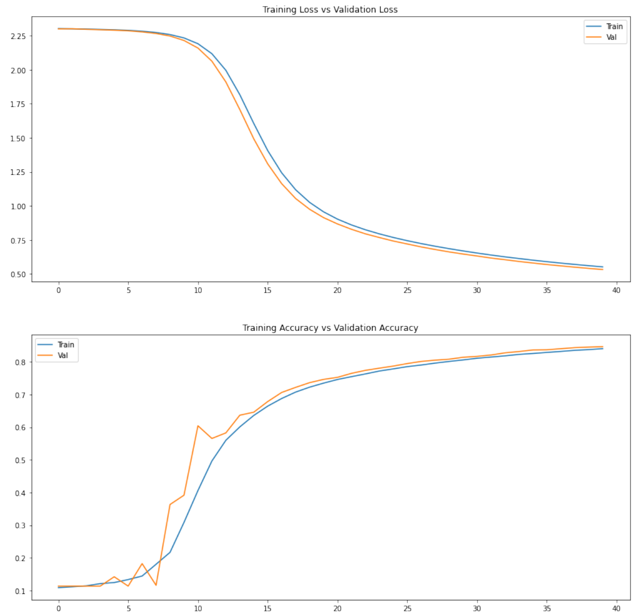

Figure 26: Changing Optimiser to ADAGrad

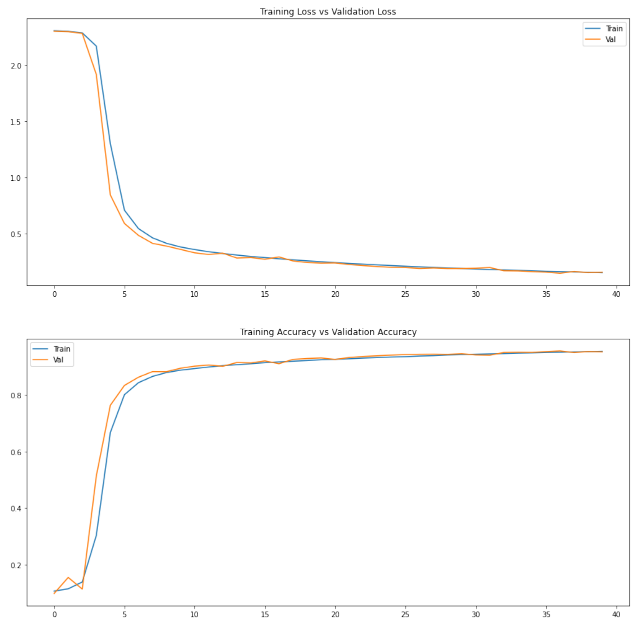

Figure 27: Changing Optimiser to SGD

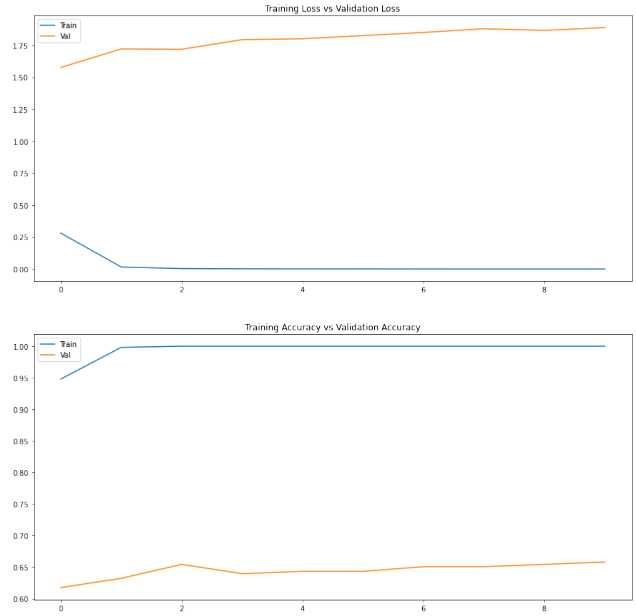

Figure 28: Benchmarking LeNet on Dataset for Transfer Learning

```
Epoch 1/10
35/35 [==============================] - 44s 1s/step - loss: 3.1104 - accuracy: 0.3807 - val_loss: 0.7436 - val_accuracy: 0.7537
Epoch 2/10
35/35 [==============================] - 41s 1s/step - loss: 1.2112 - accuracy: 0.6867 - val_loss: 0.3833 - val_accuracy: 0.8860
Epoch 3/10
35/35 [==============================] - 38s 1s/step - loss: 0.7819 - accuracy: 0.7832 - val_loss: 0.3287 - val_accuracy: 0.9191
Epoch 4/10
35/35 [==============================] - 38s 1s/step - loss: 0.7022 - accuracy: 0.8060 - val_loss: 0.3519 - val_accuracy: 0.8824
Epoch 5/10
35/35 [==============================] - 38s 1s/step - loss: 0.6537 - accuracy: 0.8297 - val_loss: 0.2883 - val_accuracy: 0.9044
Epoch 6/10
35/35 [==============================] - 43s 1s/step - loss: 0.5694 - accuracy: 0.8525 - val_loss: 0.3226 - val_accuracy: 0.9191
Epoch 7/10
35/35 [==============================] - 39s 1s/step - loss: 0.5267 - accuracy: 0.8561 - val_loss: 0.2844 - val_accuracy: 0.9338
Epoch 8/10
35/35 [==============================] - 38s 1s/step - loss: 0.5502 - accuracy: 0.8552 - val_loss: 0.2580 - val_accuracy: 0.9265
Epoch 9/10
35/35 [==============================] - 38s 1s/step - loss: 0.6459 - accuracy: 0.8488 - val_loss: 0.2823 - val_accuracy: 0.9265
Epoch 10/10
35/35 [==============================] - 38s 1s/step - loss: 0.4230 - accuracy: 0.8716 - val_loss: 0.2459 - val_accuracy: 0.9191
```

Figure 29: Transfer Learning using Inception v3

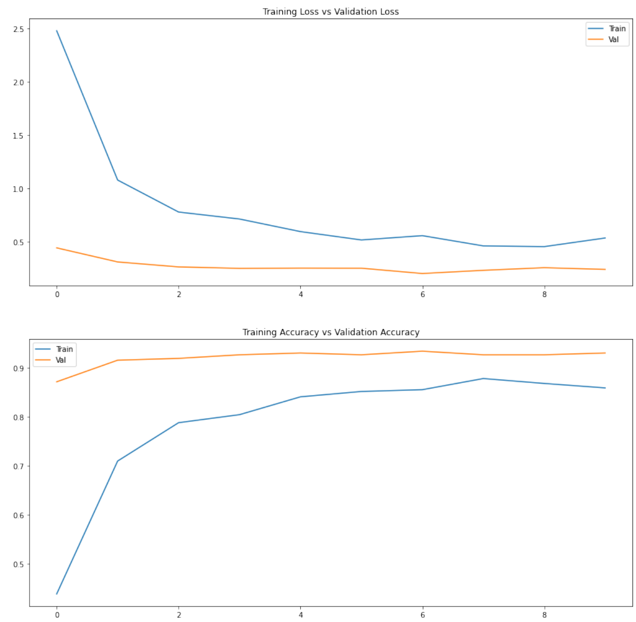

Figure 30: Transfer Learning using Inception v3

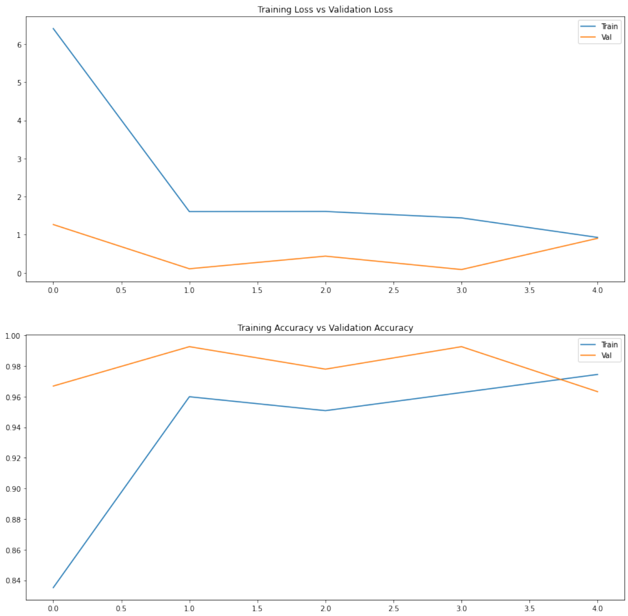

Figure 31: Transfer Learning using Xception

```
Epoch 1/5
35/35 [==============================] - 144s 4s/step - loss: 6.4080 - accuracy: 0.8352 - val_loss: 1.2663 - val_accuracy: 0.9669
Epoch 2/5
35/35 [==============================] - 137s 4s/step - loss: 1.6049 - accuracy: 0.9599 - val_loss: 0.1056 - val_accuracy: 0.9926
Epoch 3/5
35/35 [==============================] - 141s 4s/step - loss: 1.6092 - accuracy: 0.9508 - val_loss: 0.4372 - val_accuracy: 0.9779
Epoch 4/5
35/35 [==============================] - 139s 4s/step - loss: 1.4401 - accuracy: 0.9627 - val_loss: 0.0853 - val_accuracy: 0.9926
Epoch 5/5
35/35 [==============================] - 135s 4s/step - loss: 0.9281 - accuracy: 0.9745 - val_loss: 0.9048 - val_accuracy: 0.9632
```

Figure 32: Transfer Learning using Xception