UNIVERSITY OF

18 56

MARYLAND

PROJECT 2

CMSC828C/ENEE633 Statistical Pattern Recognition

December 13, 2021

Instructors:
Behtash Babadi

Student:
Siddharth Telang

Course:
Statistical Pattern Recognition, Fall 2021

Course code:
CMSC828C/ENEE633

Contents

1 Introduction

- In this project we implement the below classifiers to recognize Hand-written digits
  - Logistic Regression
  - Kernel SVM
  - Convolution Neural Network(CNN)
  - Transfer learning using CNN

  As the image data is of high dimension, we use the below dimensionality reduction methods for Logistic regression and SVM
  - PCA
  - MDA

2 Data sets

We are provided with three data sets

- MNIST Hand written data set having 60,000 training images and 10,000 testing images
- Kaggle Slothkong 10 species Monkey data set, having 1098 training images and 272 testing images

3 Logistic Regression

Logistic regression takes into account the posterior probability for the classification. The class which has the maximum posterior probability is assigned the respective label. This is achieved by using the sigmoid function to calculate the conditional probability, which is $\sigma(t) = \frac{1}{1+e^{\theta^T X}}$. For a multiclass problem, we use the softmax function, so as to ensure that each row sums up to 1. The softmax is given by

$$P(w_m|x) = \frac{e^{\theta_m^T X}}{\sum_{j=1}^{M} e^{\theta_j^T X}} \tag{1}$$

Procedure followed -

- Perform dimentionality reduction using PCA and MDA
- Initialize a weight vector θ, and multiply our data matrix with weight vector
- Calculate softmax for each element in the multiplied matrix
- Compute gradient including the regularization term
- Perform gradient descent (steepest descent) to update the weight vector
- Continue till the defined number of iterations

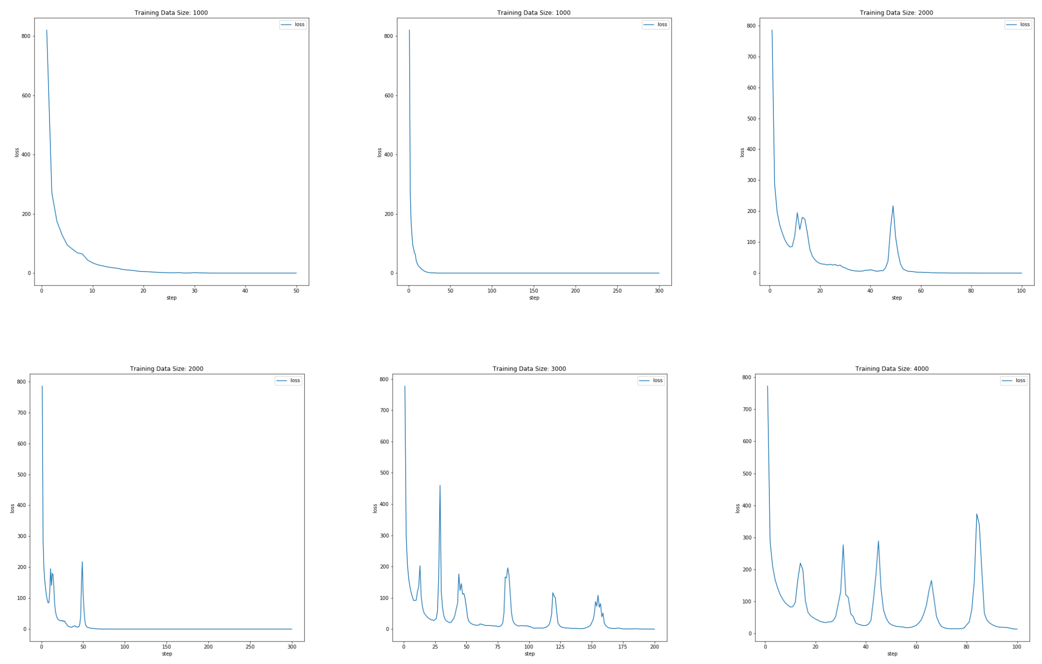


Figure 1: Training loss - Training size and number of Iterations step size 0.1 (PCA)

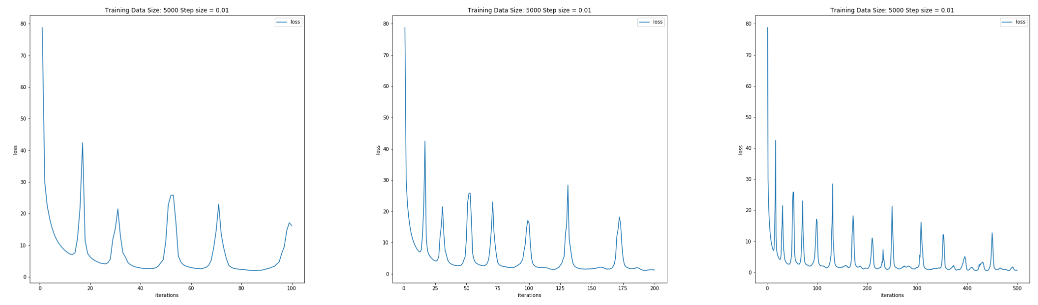


Figure 2: Training loss - Training size and number of Iterations step size 0.01 (PCA)

3.1 Training

Figure 1 shows the Training loss on various training data sizes, with a step size of 0.1, with respect to the number of iterations. We see that the large spikes in the loss when we increase the size of the training data and the loss curve is not smooth. We further check the effect of step size on the loss and set the step size to 0.01 from 0.1

Figure 2 shows the training loss on data size of 5000 but with modified step size. We observe that the spikes are still there but there is some shrinking.
Next, we check the same but with MDA.

Figure 3 shows the testing loss when we apply MDA to reduce the dimensions with two step sizes. Clearly, we see a significant improvement in the performance, as there are no spikes, and the loss curve is smooth. Even if we increase the data size, the loss curve exponentially decreases. When we decrease the step size, we see that the loss decreases quickly.

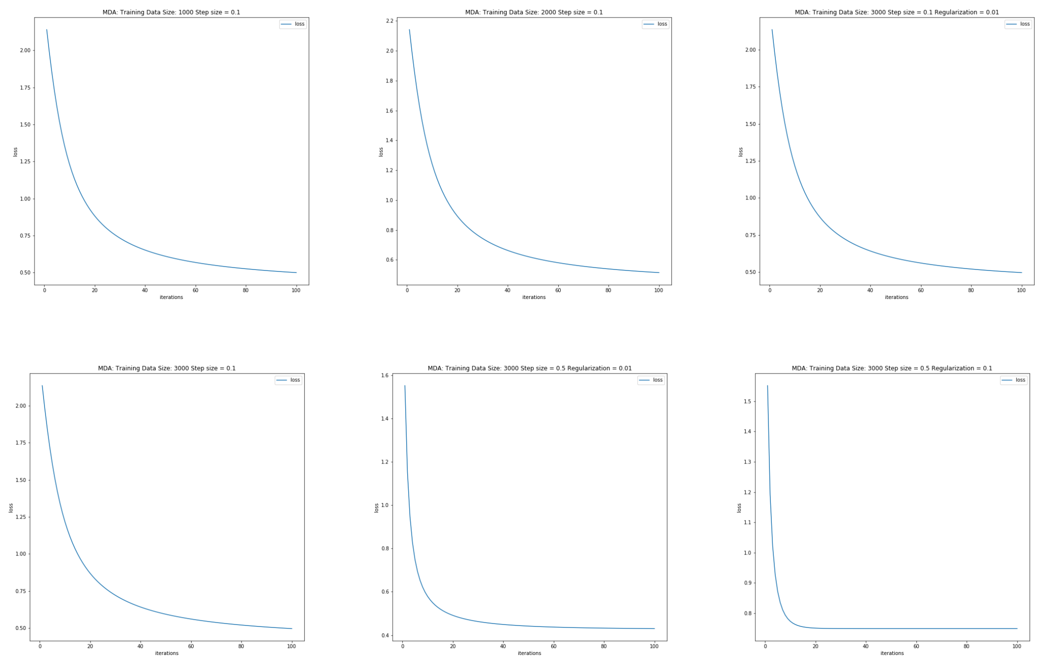


Figure 3: Training loss - Training size and number of Iterations (MDA)

3.2 Validation/Testing

We test our trained weights on test sizes ranging from 200 to 1000. The training data size is of 4000 and 7500.

Figure 4 shows the how the accuracy improves when we increase our training data size from 4000 to 7500. The colored lines show variations in the testing data for a fixed number of training data sizes. We see that as we increase the number of training samples, the accuracy improves.

We follow the same, but now with MDA. Figure 5 shows various graphs of accuracy vs training data size for varying testing data sizes(colored lines). Here, we include more samples in training as well as in testing, and found similar results.

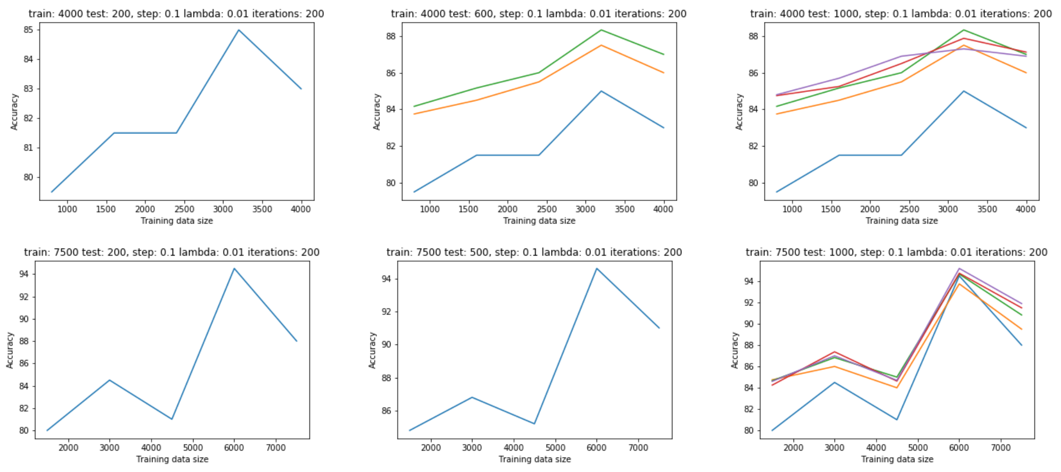


Figure 4: Accuracy vs the Training data size (PCA)

Table 1: Accuracy with PCA

Training Data size	Testing Data size	Accuracy(PCA)
4500	200	85%
4500	600	87%
4500	1000	88%
7500	200	94%
7500	1000	95%

Table 2: Accuracy with MDA

Training Data size	Testing Data size	Accuracy(MDA)
5000	500	89%
5000	1500	90%
5000	2500	91%
10000	2000	91%
10000	2500	92%

Table 3: Linear and Sigmoid kernel accuracies with PCA and MDA

Kernel	Margin	Accuracy(MDA)	Accuracy(PCA)
Sigmoid	0.5	83.32%	16.4%
Sigmoid	0.6	82.96%	16.0%
Sigmoid	0.8	82.16%	16.5%
Sigmoid	1.0	81.48%	16.0%
Linear	0.5	85.58%	11.6%
Linear	0.5	85.58%	11.6%
Linear	0.6	85.5%	11.6%
Linear	0.8	85.58%	11.6%
Linear	1	85.52%	11.6%

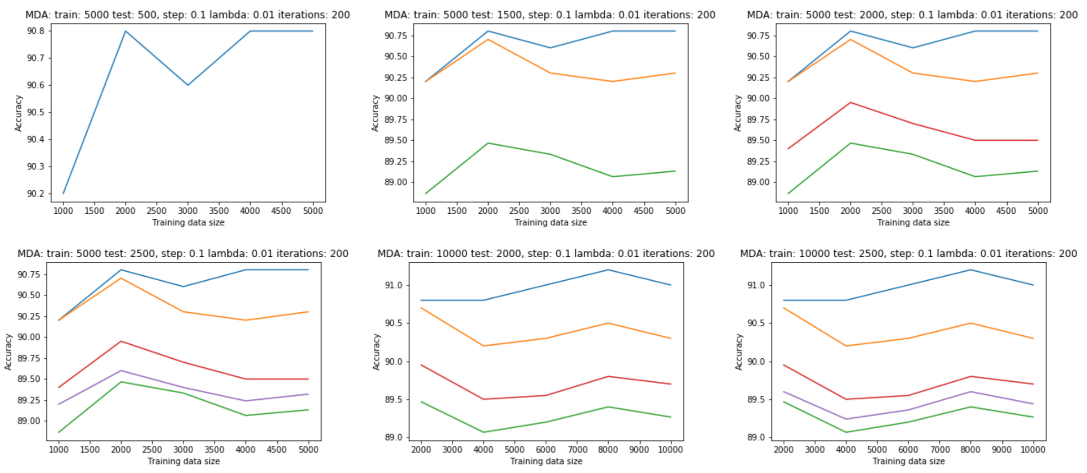


Figure 5: Accuracy vs the Training data size Logistic Regression (MDA)

We observe that our accuracy was capped at 94%. Had we increased the number of training data, the accuracy might have got improved. However, due to computational restrictions, we did not perform that.

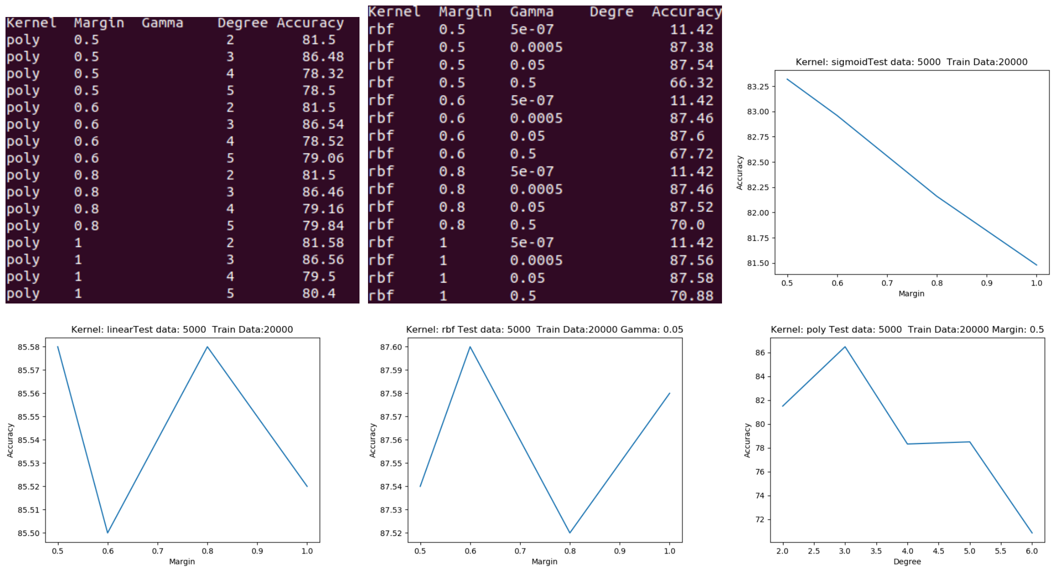
4 SVM Classifier

We try the following kernels for SVM Classifiers - Linear, Polynomial, RBF, and Sigmoid.

Table 3 show the accuracy with Linear and Sigmoid kernels when MDA and PCA are used for dimension reduction.
Clearly, we see that we get worst accuracy with PCA. Similar observation is seen on other kernels as well. Even if we increase the training data size, the results remain almost the same.
Figure 6 shows the below

- RBF Kernel - Accuracy with various margin and Gamma parameter (terminal screenshot)
- Polynomial Kernel - Accuracy with various margins and Degrees

- Accuracy of RBF and Poly kernel on 20000 training data and 5000 testing data : Maximum accuracy is achieved with polynomial degree 3, 0.05 gamma value, and a soft margin of 0.5
- Accuracy of Sigmoid and Linear kernels: Linear(maximum achieved at margin of 0.5 and 0.8), while for sigmoid it decreased as we increase the margin from 0.5



```
Kernel  Margin  Gamma       Degree Accuracy
poly    0.5                  2        81.5
poly    0.5                  3        86.48
poly    0.5                  4        78.32
poly    0.5                  5        78.5
poly    0.6                  2        81.5
poly    0.6                  3        86.54
poly    0.6                  4        78.52
poly    0.6                  5        79.06
poly    0.8                  2        81.5
poly    0.8                  3        86.46
poly    0.8                  4        79.16
poly    0.8                  5        79.84
poly    1                    2        81.58
poly    1                    3        86.56
poly    1                    4        79.5
poly    1                    5        80.4
```

```
Kernel  Margin  Gamma     Degre  Accuracy
rbf     0.5     5e-07             11.42
rbf     0.5     0.0005            87.38
rbf     0.5     0.05              87.54
rbf     0.5     0.5               66.32
rbf     0.6     5e-07             11.42
rbf     0.6     0.0005            87.46
rbf     0.6     0.05              87.6
rbf     0.6     0.5               67.72
rbf     0.8     5e-07             11.42
rbf     0.8     0.0005            87.46
rbf     0.8     0.05              87.52
rbf     0.8     0.5               70.0
rbf     1       5e-07             11.42
rbf     1       0.0005            87.56
rbf     1       0.05              87.58
rbf     1       0.5               70.88
```

Kernel: sigmoidTest data: 5000 Train Data:20000
Accuracy
Margin
Kernel: linearTest data: 5000 Train Data:20000
Accuracy
Margin
Kernel: rbf Test data: 5000 Train Data:20000 Gamma: 0.05
Accuracy
Margin
Kernel: poly Test data: 5000 Train Data:20000 Margin: 0.5
Accuracy
Degree

Figure 6: Accuracy of Polynomial, RBF Kernel with various parameters, graphs

5 Convolution Neural Network (CNN)

5.1 Number of convolution layers

We observe that as we increase the number of convolution layers, the accuracy improves. For instance, the first CNN has only 2 convolution layers, each with 4 and 12 filters respectively. It is combined with max pooling and a dense layer at the end to get the results shown in figure 7

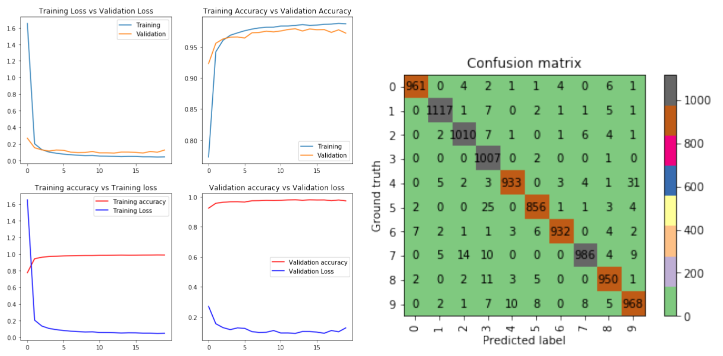


Figure 7: Basic CNN with 2 convolution layers - Accuracy, Loss Graphs and Confusion matrix

We observe that the training, validation and testing accuracy were 98.67%, 97.17%, and 97.2%. The confusion matrix enlists the number of correct and incorrect classifications. We can see that

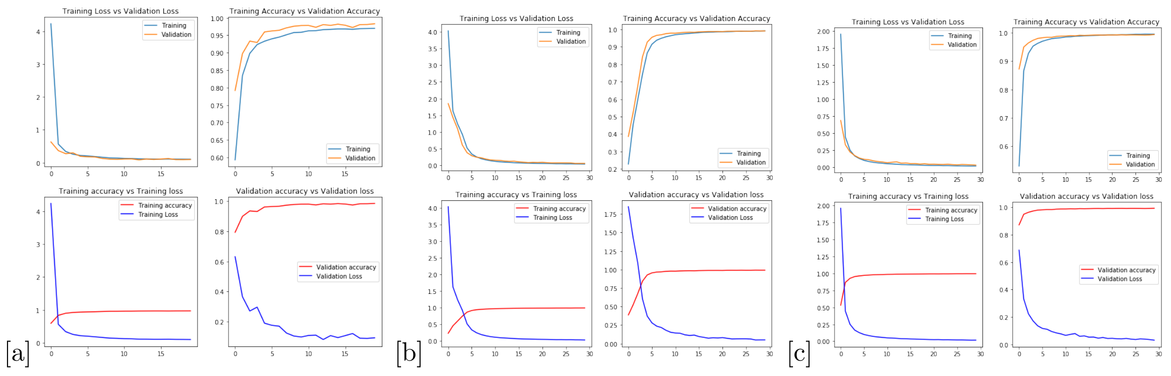


Figure 8: Accuracy and loss plots as we increase the number of convolution layers

Table 4: Accuracy with increasing number of convolution layers

Sub-figure	Training accuracy	Validation accuracy	Testing accuracy
(a)	96.98%	98.30%	98.38%
(b)	99.09%	99.04%	99.11%
(c)	99.43%	99.28%	99.33%

there are numerous mistakes that our classifier has made(out of 10,000 samples 280 were incorrectly classified).
Next, we increase the number of convolution layers and observe the results in figure 8 and table 4

- (a) has 2 convolution layers - 12 and 24 filters, each of size 5
- (b) has 3 convolution layers - 24, 48, and 48 filters, each of size 3
- (c) has 4 convolution layers - 24, 24, 48, and 48, each of size 3

We observe that with the increase in the number of convolution layers, our accuracy gets improved.

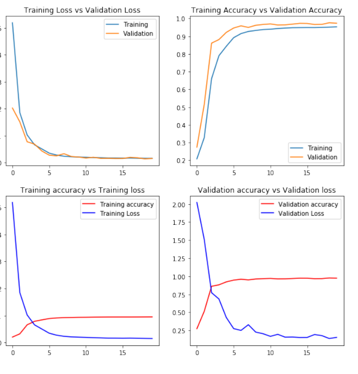
5.2 Dropout

In the CNN with only two layers with the accuracy shown in figure 7, we add dropout layers after each convolution and see an increase in the testing accuracy from 97.2% to 97.45%. Hence, adding a dropout layer further increases the accuracy and makes sure that the model does not over fit. The graph is shown in figure 9(a)

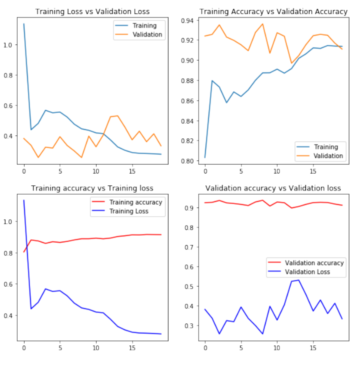
5.3 Learning Rate

By default, the Adam optimizer uses a learning rate of 0.001. We update this to 0.01 and 0.0001 and observe the graphs in figure 9
Table 5 points out the difference. We see that as soon as we update the learning rate to 0.01, the accuracy falls to 91%. The learning curves are also pretty bad, as in figure 9(b), we see many spikes, which is not the case when lr=0.001 and 0.0001.

[a]



[b]



[c]

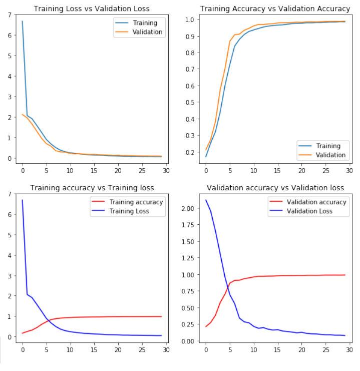


Figure 9: Learning curves with rate (a) 0.001 (b)0.01 (c) 0.0001

Table 5: Accuracy with different learning rates

Learning Rate	Training accuracy	Validation accuracy	Testing accuracy
0.001	99.3%	99.28%	98.6%
0.01	91.35%	91.09%	91.51%
0.0001	99.4%	99.3%	98.97%

5.4 Batch Normalization

With the addition of batch normalization, there was no such significant change in the accuracy and learning in our case

5.5 Batch size

Table 6 enlists the experiment with various batch size. We see that the best testing accuracy was with a smaller batch of 16 and the lowest with size of 64.

5.6 Training data Size

From the table 7 we clearly observe that neural network is data hungry, the more we provide the data, the more better it gets(provided we do not over-fit it).

5.7 Number of epochs

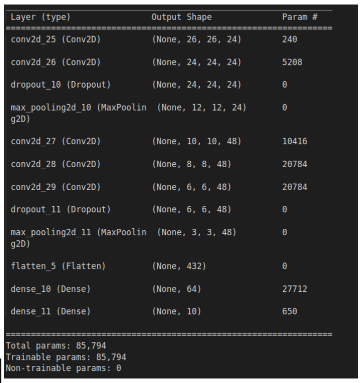
Observation was made that as we increase the number of epochs, our training got better and the testing accuracy increased.

Table 6: Accuracy with batch size

Batch size	Training accuracy	Validation accuracy	Testing accuracy
16	99.68%	99.19%	99.35%
32	99.18%	99.61%	99.30%
64	99.62%	99.16%	99.26%

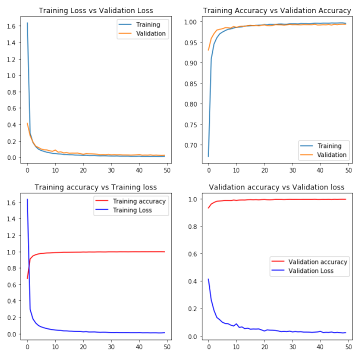
Table 7: Accuracy with training data size (30 epochs)

Training size	Training accuracy	Validation accuracy	Testing accuracy
1000	86%	86.67%	88.19%
10000	97.67%	97.53%	97.87%
30000	99.29%	98.84%	98.98%
60000	99.48%	99.21%	99.35%

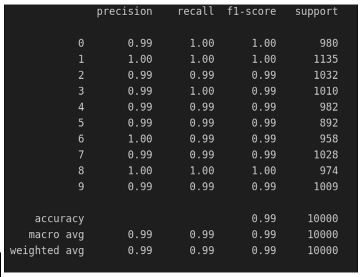


[a]

```
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_25 (Conv2D)          (None, 26, 26, 24)        240

 conv2d_26 (Conv2D)          (None, 24, 24, 24)        5208

 dropout_10 (Dropout)        (None, 24, 24, 24)        0

 max_pooling2d_10 (MaxPoolin  (None, 12, 12, 24)       0
 g2D)

 conv2d_27 (Conv2D)          (None, 10, 10, 48)        10416

 conv2d_28 (Conv2D)          (None, 8, 8, 48)          20784

 conv2d_29 (Conv2D)          (None, 6, 6, 48)          20784

 dropout_11 (Dropout)        (None, 6, 6, 48)          0

 max_pooling2d_11 (MaxPoolin  (None, 3, 3, 48)         0
 g2D)

 flatten_5 (Flatten)         (None, 432)               0

 dense_10 (Dense)            (None, 64)                27712

 dense_11 (Dense)            (None, 10)                650

=================================================================
Total params: 85,794
Trainable params: 85,794
Non-trainable params: 0
```

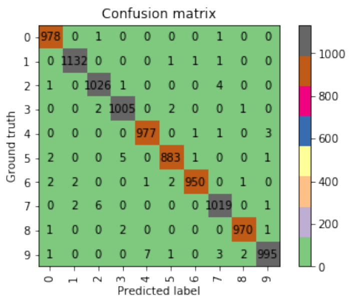


[b]



[c]

```
              precision    recall  f1-score   support

           0       0.99      1.00      1.00       980
           1       1.00      1.00      1.00      1135
           2       0.99      0.99      0.99      1032
           3       0.99      1.00      0.99      1010
           4       0.99      0.99      0.99       982
           5       0.99      0.99      0.99       892
           6       1.00      0.99      0.99       958
           7       0.99      0.99      0.99      1028
           8       1.00      1.00      1.00       974
           9       0.99      0.99      0.99      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```



[d]

Figure 10: Final Network trained (a) Architecture (b) Learning curves (c) Precision Recall scores (d) Confusion Matrix

5.8 Early stopping

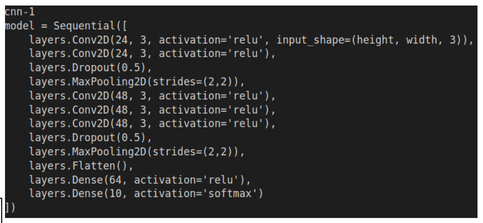
Sometimes the Neural Network can not do better after reaching a stage, and hence it makes no sense to continue the training on the specified number of epochs. Hence, we stop the training at this stage. The metric used here is accuracy. We can also use validation accuracy or loss for early stopping. If we set the number of epochs to 100, we observe that the framework was trained at 47th epoch and no training was further required.

5.9 Dense layers

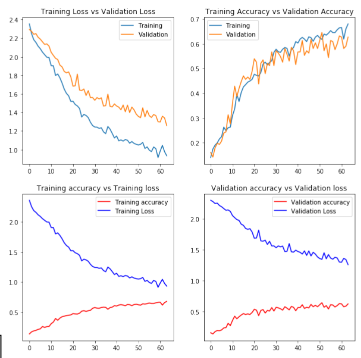
This is one of the critical layers that plays a vital role in the training. As soon as we add more dense layer to our neural network, the testing accuracy reached 99.44%, which is only 56 mistakes! The figure 10 shows the final network used, the training curves, confusion matrix, and final classification report.

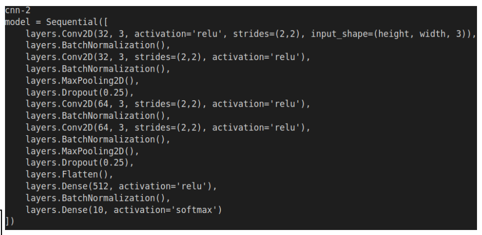
Table 8: Accuracy with CNN-1 and CNN-2 in 11

CNN	Training accuracy	Validation accuracy	Testing accuracy
1	67%	62.67%	63.97%
2	85.93%	65.90%	74.26%

[a]


```
cnn-1
model = Sequential([
    layers.Conv2D(24, 3, activation='relu', input_shape=(height, width, 3)),
    layers.Conv2D(24, 3, activation='relu'),
    layers.Dropout(0.5),
    layers.MaxPooling2D(strides=(2,2)),
    layers.Conv2D(48, 3, activation='relu'),
    layers.Conv2D(48, 3, activation='relu'),
    layers.Conv2D(48, 3, activation='relu'),
    layers.Dropout(0.5),
    layers.MaxPooling2D(strides=(2,2)),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

[b]


[c]


```
cnn-2
model = Sequential([
    layers.Conv2D(32, 3, activation='relu', strides=(2,2), input_shape=(height, width, 3)),
    layers.BatchNormalization(),
    layers.Conv2D(32, 3, strides=(2,2), activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(),
    layers.Dropout(0.25),
    layers.Conv2D(64, 3, strides=(2,2), activation='relu'),
    layers.BatchNormalization(),
    layers.Conv2D(64, 3, strides=(2,2), activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(),
    layers.Dropout(0.25),
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.BatchNormalization(),
    layers.Dense(10, activation='softmax')
])
```

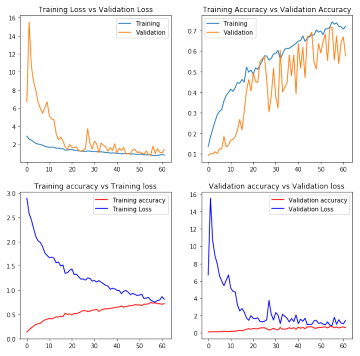
[d]


Figure 11: CNN used (a) 1st Architecture (b) Learning curves (c) 2nd Architecture (d) Learning curves

6 Transfer Learning

6.1 Without pre-trained model

We use two different convolution networks as shown in figure 11 to train our model on the monkey data set. The observations are summarized in table 8. We observe a testing accuracy of 64% with CNN-1 and 74% with CNN-2.

6.2 With Trained Model

6.2.1 Without Fine-tuning

We import the pre-trained VGG-16 model from keras, remove it's last dense layers, and freeze the model so that it is not trained. We then add our dense network and the prediction layer to train the model on the monkey data set. Initially, we observe that even after one dense layer, no significant improvement was seen over the CNN-1 and CNN-2 in previous section 11 and table 8. This is because, the CNN-2 already has a lot of convolution layers. So, we add more dense layers to our new model and observe the results shown in table 10 (excludes the output 10 layers). We see that the testing

Table 9: Testing Accuracy as we add dense layers to our model with VGG-16 as pre-model

Dense layers	Testing accuracy
128	73.16%
256	74.26%
512, 256, 64	76%

Table 10: Fine tuning our model with VGG-16

Dense layers	Training accuracy	Validation accuracy	Testing accuracy
128	98.18%	85.25%	88.24%
256	99.21%	86.64%	90.81%
512, 256, 64	99.43%	85.29%	87.5%

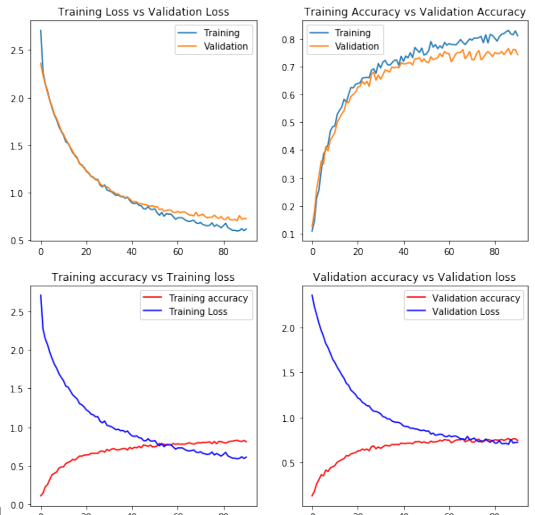
accuracy was improved with respect to CNN-1, but still was capped at 76%. To further increase the performance, we fine tune the network by un-freezing the last 3 layers of our pre-trained model.

6.2.2 Fine Tuning

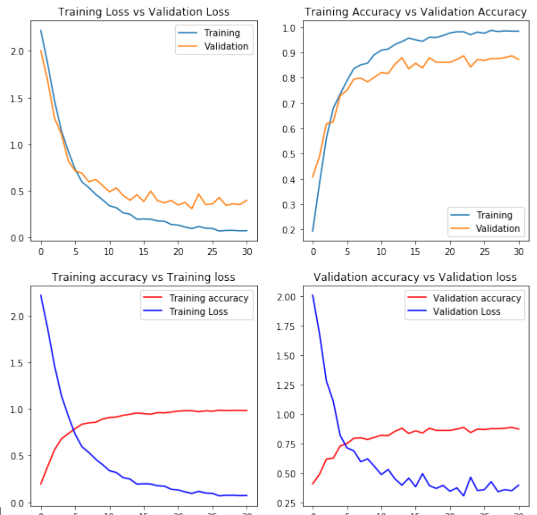
We now un-freeze the last three layers of VGG-16 model to better fine tune it with our monkey data set, and train it over 30 epochs. Following are the results with various dense layers added on top of VGG-16, summarized in table 10
We observe that with fine tuning we have a significant improvement over our previous models. The training accuracy reached 99.43% and test accuracy was 90.81%, which is pretty good given the small data set. The learning curves and the architecture is shown in figure 12

7 Dataset and Code

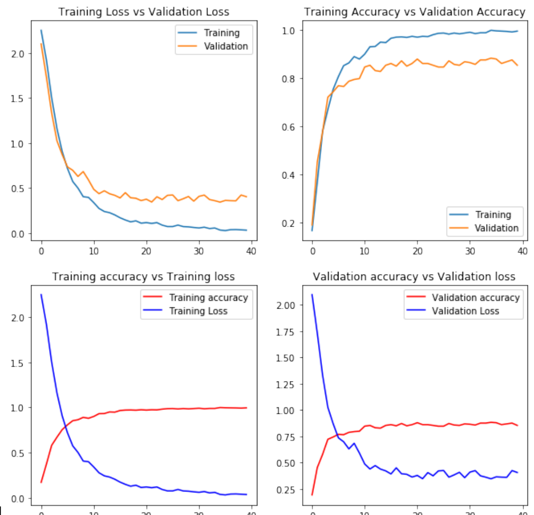
Open source code: `https://github.com/siddharthtelang/Hand-Written-Digits-Recognition`
Data set: `https://drive.google.com/file/d/1Dx19LPN9bCo9OBpq9wDku3oMzrkFG8mP/view?usp=sharing`



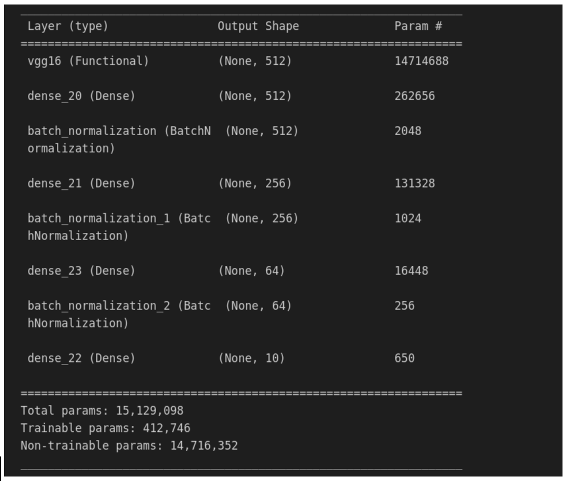
[a]



[b]



[c]



```
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 512)               14714688

 dense_20 (Dense)            (None, 512)               262656

 batch_normalization (BatchN  (None, 512)              2048
 ormalization)

 dense_21 (Dense)            (None, 256)               131328

 batch_normalization_1 (Batc  (None, 256)              1024
 hNormalization)

 dense_23 (Dense)            (None, 64)                16448

 batch_normalization_2 (Batc  (None, 64)               256
 hNormalization)

 dense_22 (Dense)            (None, 10)                650

=================================================================
Total params: 15,129,098
Trainable params: 412,746
Non-trainable params: 14,716,352
```

[d]

Figure 12: Dense layers in fine tuning (a) 128 (b) 256 (c) 512, 256, 64 (d) Architecture