1. Introduction

In this project, deep neural networks (NN) are utilized to recognize handwritten digits using the MNIST dataset. Three type of deep NN are used; Multi layer perceptron, Deep NN and Convolutional Neural Networks. These NNs are implemented by using high-level neural networks API, Keras.

The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. Each image is a 28 by 28 pixel square (784 pixels total). Images of digits were taken from a variety of scanned documents. The digits have been size-normalized and centered in a fixed-size image.

It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK,or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.

TensorFlow and Theano backends are deployed by Keras in the project.

2. Requirements

We have to install Anaconda 3.x as python 3.x environment, Keras, Tensorflow, Theano. Then we need Blas and Graphviz for Keras. We have to install them as follows:

```
conda install -c conda-forge blas
```

```
conda install -c conda-forge python-graphviz
```

If you do not set your path in your OS, following path definition should be used in program for graphviz, please set your path in python code.

```
import os
```

```
os.environ["PATH"] += os.pathsep +"C:\\Users\\Enespc\\Anaconda3\\Library\\bin\\graphviz"
```

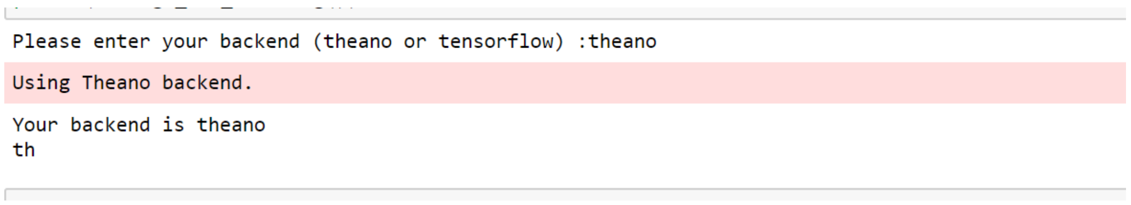
I prefered Jupyter notebook as ide for the project.

3. Description of the Python program

Our program contains main three parts. First part consist of MNIST Handwritten Digit Recognition in Keras by using Multilayer Perceptron.

PART I :

Firstly, backend of Keras have to be selected, theano or tensorflow.



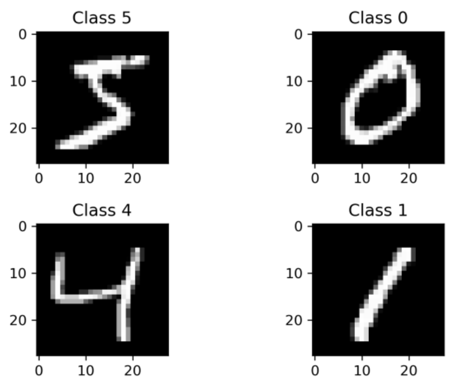
```
Please enter your backend (theano or tensorflow) :theano
Using Theano backend.
Your backend is theano
th
```

MNIST dataset is loaded using Keras function;

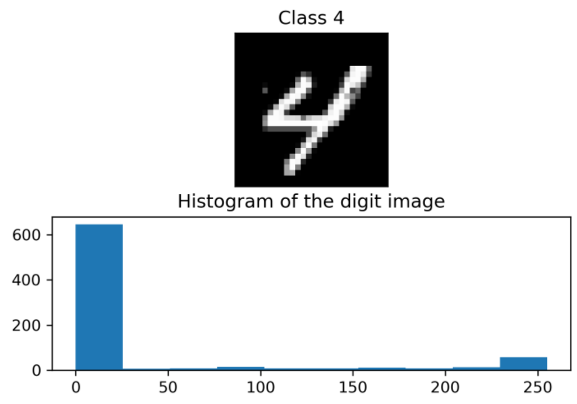
(X_train, y_train), (X_test, y_test) = mnist.load_data()

The function splits the MNIST data into train and test sets.

We can now start to explore the dataset. In order to validate downloaded MNIST dataset, some digits are plotted.



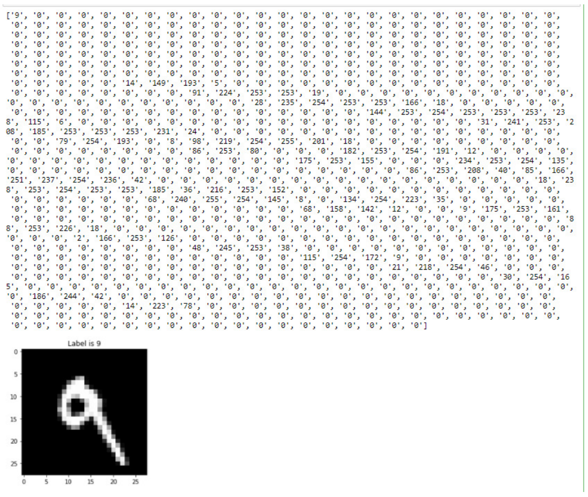
One digit is ploted and a histogram of the digits is calculated. We become familiar to MNIST datasets. The pixel values range from 0 to 255. The background majority close to 0, and others close to 255 representing the digit.



In this stage, MNIST datasets are stored into two CVS files. The format of files is: label, pix-11, pix-12, pix-13, ... where pix-ij is the pixel in the ith row and jth column. Every column has one digit.

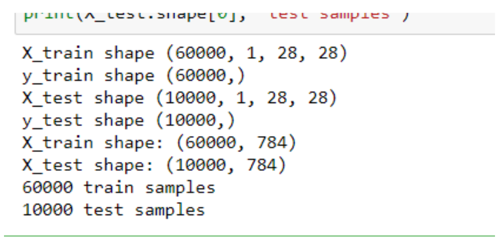
The Program creates two files; mnist_train.csv and mnist_test.csv on the working directory. Please set your working directory from your IDE environment.

On next step, CVS file is opened and checked. Everything is very well or not? The program finds first digit 9 from the CVS file 'mnist_test.csv' and plots it.



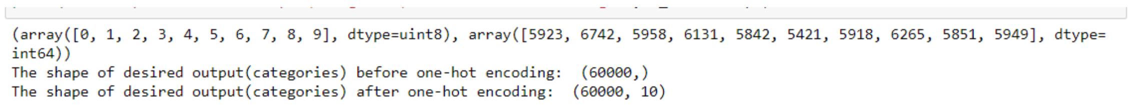
```
['9', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '14', '149', '193', '5', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '91', '224', '253', '253', '19', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
'0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '28', '235', '254', '253', '253', '166', '18', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '144', '253', '254', '253', '253', '253', '23
8', '115', '6', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '31', '241', '253', '2
08', '185', '253', '253', '253', '231', '24', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '79', '254', '193', '0', '8', '98', '219', '254', '255', '201', '18', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '86', '253', '80', '0', '0', '0', '182', '253', '254', '191', '12', '0', '0', '0', '0',
'0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '175', '253', '155', '0', '0', '0', '234', '253', '254', '135',
'0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '86', '253', '208', '40', '85', '166',
'251', '237', '254', '236', '42', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '18', '23
8', '253', '254', '253', '253', '185', '36', '216', '253', '152', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '68', '240', '255', '254', '145', '8', '0', '134', '254', '223', '35', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '68', '158', '142', '12', '0', '0', '9', '175', '253', '161',
 '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '8
8', '253', '226', '18', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
'0', '0', '0', '2', '166', '253', '126', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '48', '245', '253', '38', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '115', '254', '172', '9', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '21', '218', '254', '46', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '30', '254', '16
5', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
'0', '186', '244', '42', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '14', '223', '78', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
 '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0']
```

The train and test data are reshaped according to ordering of Tensorflow or Theano. Theano ordering (channels, rows, cols) and TensorFlow ordering (rows, cols, channels). Then, the input data is normalized.



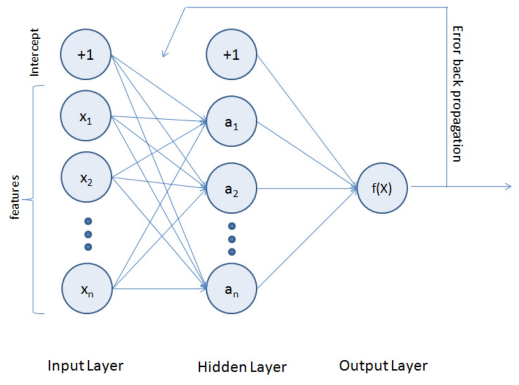
```
print(X_test.shape[0], 'test samples')

X_train shape (60000, 1, 28, 28)
y_train shape (60000,)
X_test shape (10000, 1, 28, 28)
y_test shape (10000,)
X_train shape: (60000, 784)
X_test shape: (10000, 784)
60000 train samples
10000 test samples
```

Let's check desired output y, our categories. It still holds integer values from 0 to 9. Our categories are reshaped - digits from 0 to 9 by using one-hot encoding. The result is a vector with a length equal to the number of categories. The vector is all zeroes except in the position for the respective category. Thus a '4' will be represented by [0,0,0,1,0,0,0,0,0].

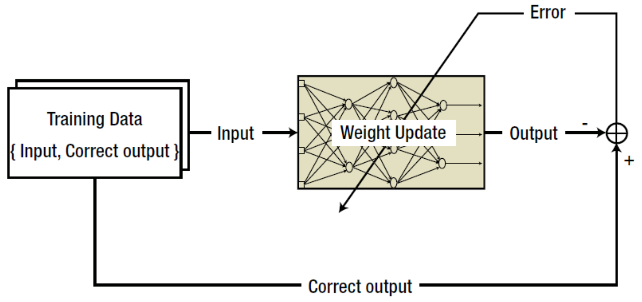


```
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8), array([5923, 6742, 5958, 6131, 5842, 5421, 5918, 6265, 5851, 5949], dtype=
int64))
The shape of desired output(categories) before one-hot encoding:  (60000,)
The shape of desired output(categories) after one-hot encoding:  (60000, 10)
```

Typical MLP architecture:

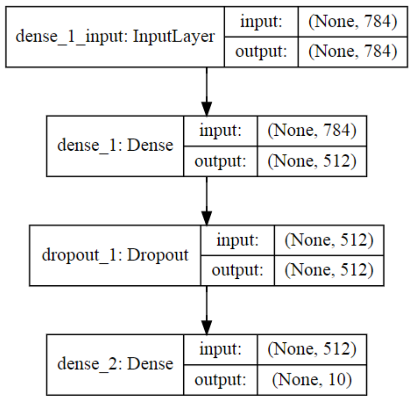


Typical training of MLP scheme:

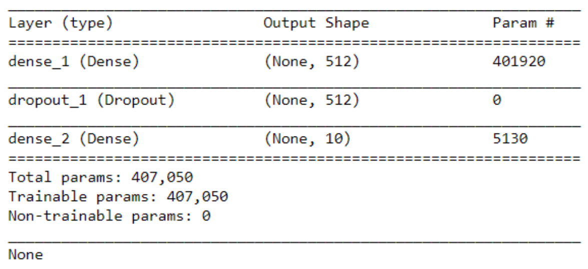


We can start to create the Neural Network, MLP by building a linear stack of layers with the sequential model of Keras. We can stack layers using the .add() method. When adding the first layer in the Sequential Model we need to specify the input shape so Keras can create the appropriate matrices. For all remaining layers the shape is inferred automatically.
In order to introduce nonlinearities into the network and elevate it beyond the capabilities of a simple perceptron we also add activation functions to the *hidden layers.* The differentiation for the training via backpropagation is happening behind the scenes without having to implement the details.
We also add *dropout* as a way to prevent overfitting. Here we randomly keep some network weights fixed when we would normally update them so that the network doesn't rely too much on very few nodes.

The last layer consists of connections for our 10 classes and the softmax activation which is standard for multi-class targets. Then the model is compiled and visualized by SVG.

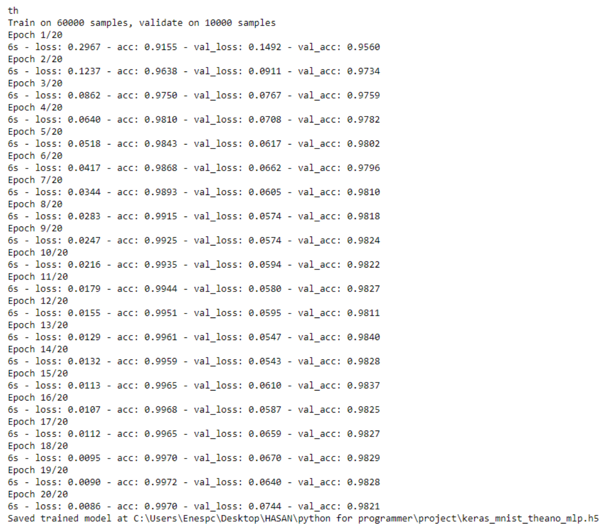


MLP has 784 (number of pixels in digit) neurons in input layer, 512 in hidden layer and 10 neurons (number of category, 0-9) in output layer. Total parameters 407,050 is trained. Summary of the model as follows:



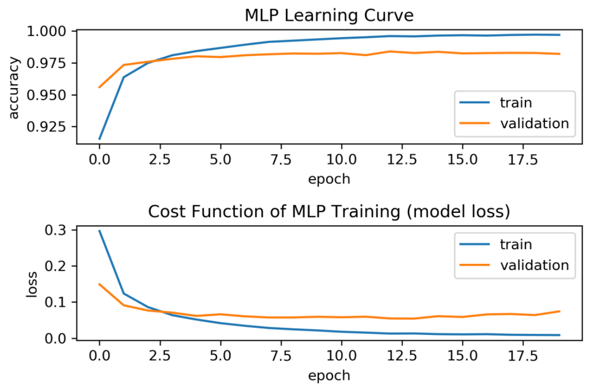
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
None
```

We train the Model and save the model and training history. In order to work with the trained model and evaluate its performance the model is saved in the following directory; save_dir = "C:\\Users\\Enespc\\Desktop\\HASAN\\python for programmer\\project". The training history is as follows

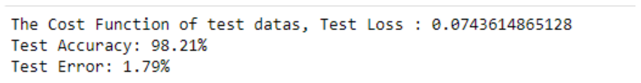


```
th
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
6s - loss: 0.2967 - acc: 0.9155 - val_loss: 0.1492 - val_acc: 0.9560
Epoch 2/20
6s - loss: 0.1237 - acc: 0.9638 - val_loss: 0.0911 - val_acc: 0.9734
Epoch 3/20
6s - loss: 0.0862 - acc: 0.9750 - val_loss: 0.0767 - val_acc: 0.9759
Epoch 4/20
6s - loss: 0.0640 - acc: 0.9810 - val_loss: 0.0708 - val_acc: 0.9782
Epoch 5/20
6s - loss: 0.0518 - acc: 0.9843 - val_loss: 0.0617 - val_acc: 0.9802
Epoch 6/20
6s - loss: 0.0417 - acc: 0.9868 - val_loss: 0.0662 - val_acc: 0.9796
Epoch 7/20
6s - loss: 0.0344 - acc: 0.9893 - val_loss: 0.0605 - val_acc: 0.9810
Epoch 8/20
6s - loss: 0.0283 - acc: 0.9915 - val_loss: 0.0574 - val_acc: 0.9818
Epoch 9/20
6s - loss: 0.0247 - acc: 0.9925 - val_loss: 0.0574 - val_acc: 0.9824
Epoch 10/20
6s - loss: 0.0216 - acc: 0.9935 - val_loss: 0.0594 - val_acc: 0.9822
Epoch 11/20
6s - loss: 0.0179 - acc: 0.9944 - val_loss: 0.0580 - val_acc: 0.9827
Epoch 12/20
6s - loss: 0.0155 - acc: 0.9951 - val_loss: 0.0595 - val_acc: 0.9811
Epoch 13/20
6s - loss: 0.0129 - acc: 0.9961 - val_loss: 0.0547 - val_acc: 0.9840
Epoch 14/20
6s - loss: 0.0132 - acc: 0.9959 - val_loss: 0.0543 - val_acc: 0.9828
Epoch 15/20
6s - loss: 0.0113 - acc: 0.9965 - val_loss: 0.0610 - val_acc: 0.9837
Epoch 16/20
6s - loss: 0.0107 - acc: 0.9968 - val_loss: 0.0587 - val_acc: 0.9825
Epoch 17/20
6s - loss: 0.0112 - acc: 0.9965 - val_loss: 0.0659 - val_acc: 0.9827
Epoch 18/20
6s - loss: 0.0095 - acc: 0.9970 - val_loss: 0.0670 - val_acc: 0.9829
Epoch 19/20
6s - loss: 0.0090 - acc: 0.9972 - val_loss: 0.0640 - val_acc: 0.9828
Epoch 20/20
6s - loss: 0.0086 - acc: 0.9970 - val_loss: 0.0744 - val_acc: 0.9821
Saved trained model at C:\Users\Enespc\Desktop\HASAN\python for programmer\project\keras_mnist_theano_mlp.h5
```

The program plots the learning curve of MLP (NN). Training has two step train and validation. 20% of train data is used for validation (validation_split=0.2 parameter manage the ratio).

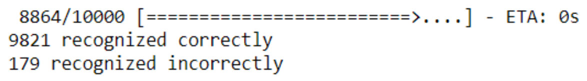


It is the time to evaluate Performance of the Model (MLP). we load the model and obtain results as floows;

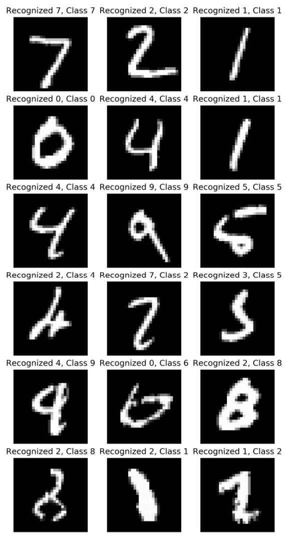


```
The Cost Function of test datas, Test Loss : 0.0743614865128
Test Accuracy: 98.21%
Test Error: 1.79%
```

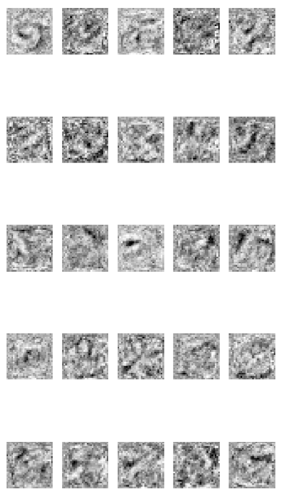
Visualization of recognized correctly and incorrectly sample of digits are done. Firstly, the model is loaded and create recognitions on the test set. Over all recognition performance of MLP is;



```
 8864/10000 [========================>....] - ETA: 0s
9821 recognized correctly
179 recognized incorrectly
```



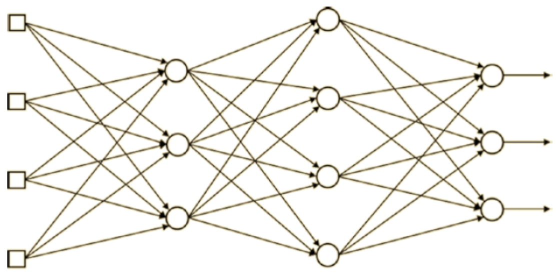
Then, we plot that first layer of MLP neurons what learned during training. Let's visualize them.

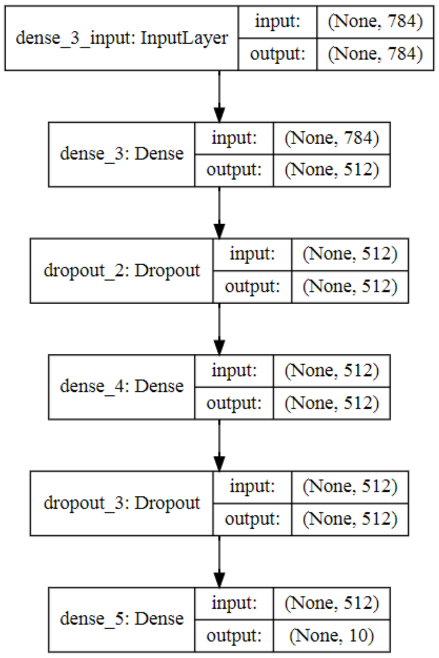


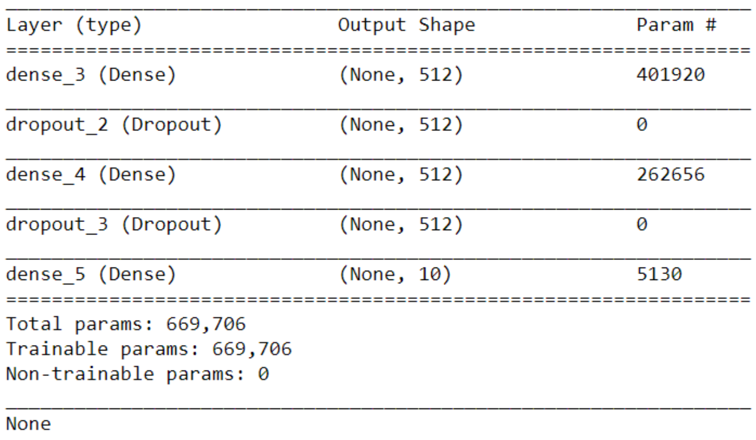
PARTII :

We implement Deep Neural Network for MNIST Handwritten Digit Recognition in part II.

Typical Deep NN architecture:

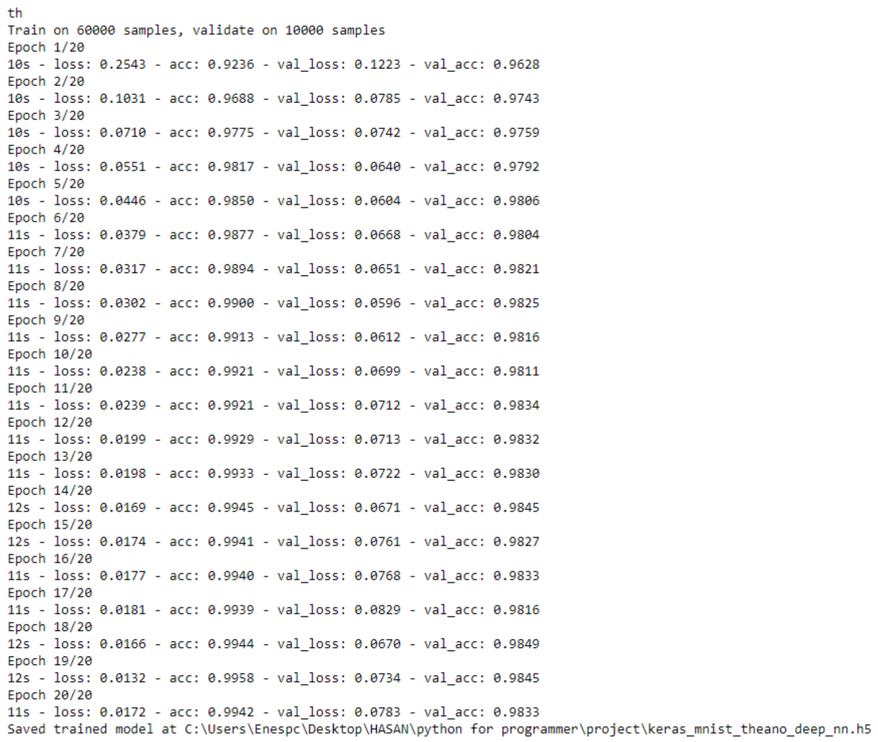






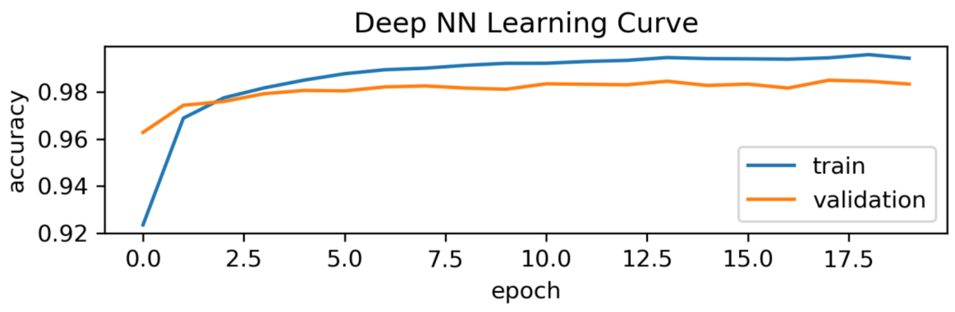
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_3 (Dense)              (None, 512)               401920    
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_4 (Dense)              (None, 512)               262656    
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_5 (Dense)              (None, 10)                5130      
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
_________________________________________________________________
None
```

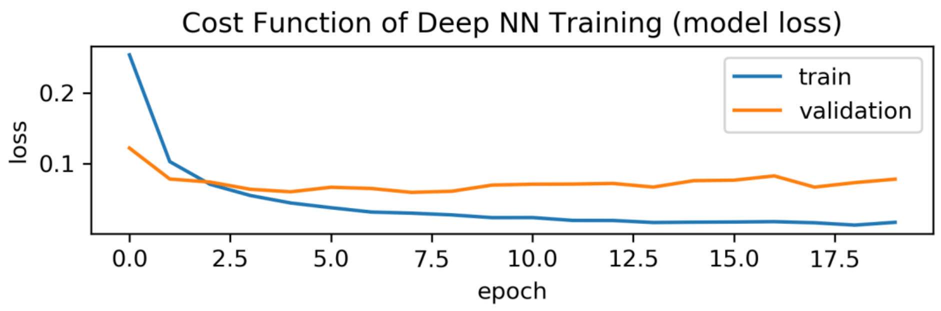
Deep NN has 4 layers, input, hidden1, hidden2 and output layer. Input layer has 784 neurons, hidden1 and hidden2 have 512 neurons, output layer has 10 neurons. Total parameter of deep NN is 669,706. Training history of deep NN is as follows;



```
th
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
10s - loss: 0.2543 - acc: 0.9236 - val_loss: 0.1223 - val_acc: 0.9628
Epoch 2/20
10s - loss: 0.1031 - acc: 0.9688 - val_loss: 0.0785 - val_acc: 0.9743
Epoch 3/20
10s - loss: 0.0710 - acc: 0.9775 - val_loss: 0.0742 - val_acc: 0.9759
Epoch 4/20
10s - loss: 0.0551 - acc: 0.9817 - val_loss: 0.0640 - val_acc: 0.9792
Epoch 5/20
10s - loss: 0.0446 - acc: 0.9850 - val_loss: 0.0604 - val_acc: 0.9806
Epoch 6/20
11s - loss: 0.0379 - acc: 0.9877 - val_loss: 0.0668 - val_acc: 0.9804
Epoch 7/20
11s - loss: 0.0317 - acc: 0.9894 - val_loss: 0.0651 - val_acc: 0.9821
Epoch 8/20
11s - loss: 0.0302 - acc: 0.9900 - val_loss: 0.0596 - val_acc: 0.9825
Epoch 9/20
11s - loss: 0.0277 - acc: 0.9913 - val_loss: 0.0612 - val_acc: 0.9816
Epoch 10/20
11s - loss: 0.0238 - acc: 0.9921 - val_loss: 0.0699 - val_acc: 0.9811
Epoch 11/20
11s - loss: 0.0239 - acc: 0.9921 - val_loss: 0.0712 - val_acc: 0.9834
Epoch 12/20
11s - loss: 0.0199 - acc: 0.9929 - val_loss: 0.0713 - val_acc: 0.9832
Epoch 13/20
11s - loss: 0.0198 - acc: 0.9933 - val_loss: 0.0722 - val_acc: 0.9830
Epoch 14/20
12s - loss: 0.0169 - acc: 0.9945 - val_loss: 0.0671 - val_acc: 0.9845
Epoch 15/20
12s - loss: 0.0174 - acc: 0.9941 - val_loss: 0.0761 - val_acc: 0.9827
Epoch 16/20
11s - loss: 0.0177 - acc: 0.9940 - val_loss: 0.0768 - val_acc: 0.9833
Epoch 17/20
11s - loss: 0.0181 - acc: 0.9939 - val_loss: 0.0829 - val_acc: 0.9816
Epoch 18/20
12s - loss: 0.0166 - acc: 0.9944 - val_loss: 0.0670 - val_acc: 0.9849
Epoch 19/20
12s - loss: 0.0132 - acc: 0.9958 - val_loss: 0.0734 - val_acc: 0.9845
Epoch 20/20
11s - loss: 0.0172 - acc: 0.9942 - val_loss: 0.0783 - val_acc: 0.9833
Saved trained model at C:\Users\Enespc\Desktop\HASAN\python for programmer\project\keras_mnist_theano_deep_nn.h5
```

The learning curve of deep NN;





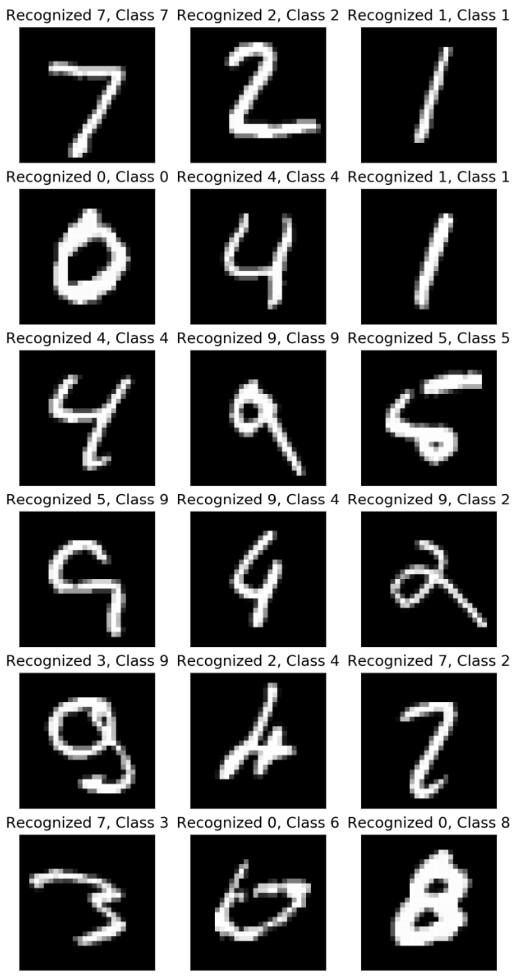
The performance of deep NN;

```
The Cost Function of test datas, Test Loss : 0.0782854418568
Test Accuracy: 98.33%
Test Error: 1.67%
```

Recognition performance of deep NN;

```
 9920/10000 [============================>.] - ETA: 0s
9833 recognized correctly
167 recognized incorrectly
```

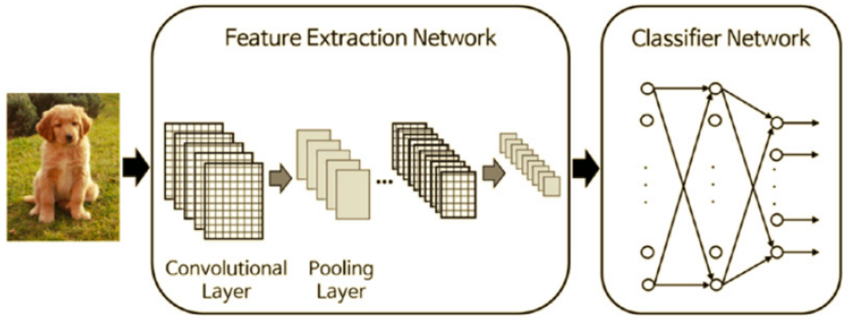
Sample of recognized and unrecognized digits of deep NN;

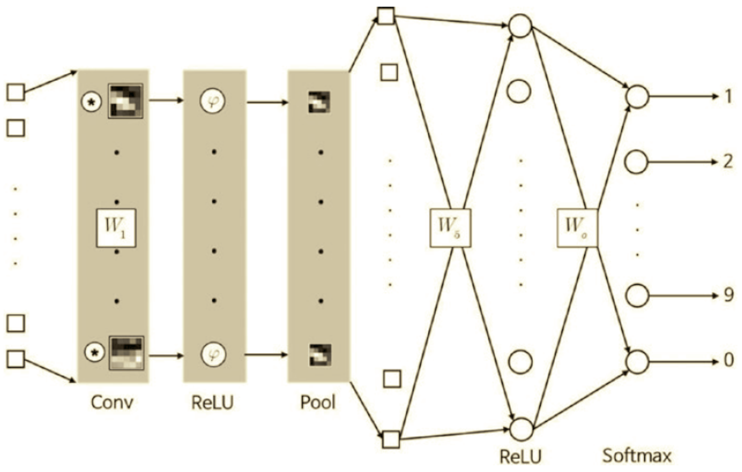


PART III :

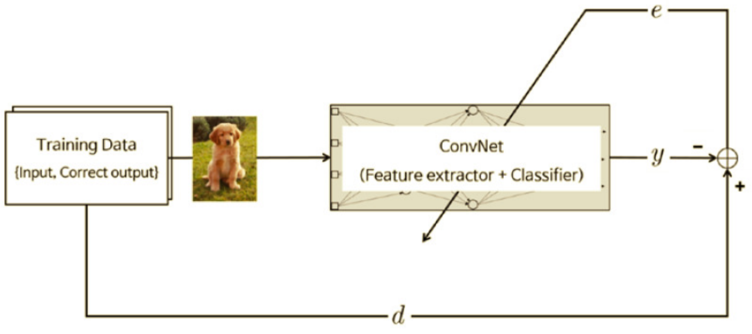
We implement Deep Convolutional Neural Networks (CNN) for MNIST Handwritten Digit Recognition in part III.

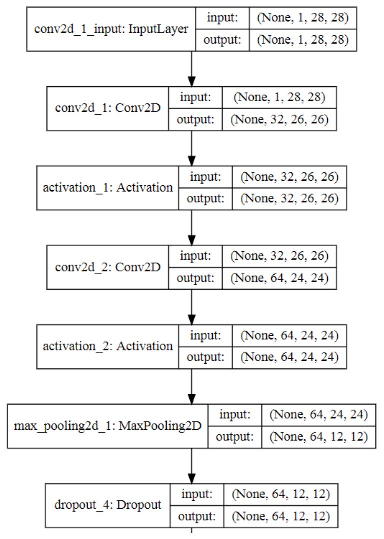
Typical architecture of CNN:

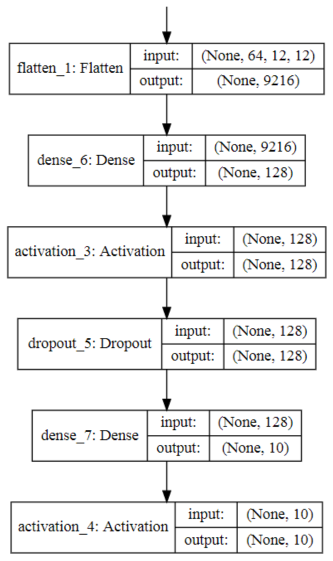




Training scheme of CNN:




conv2d_1_input: InputLayer
input: (None, 1, 28, 28)
output: (None, 1, 28, 28)
conv2d_1: Conv2D
input: (None, 1, 28, 28)
output: (None, 32, 26, 26)
activation_1: Activation
input: (None, 32, 26, 26)
output: (None, 32, 26, 26)
conv2d_2: Conv2D
input: (None, 32, 26, 26)
output: (None, 64, 24, 24)
activation_2: Activation
input: (None, 64, 24, 24)
output: (None, 64, 24, 24)
max_pooling2d_1: MaxPooling2D
input: (None, 64, 24, 24)
output: (None, 64, 12, 12)
dropout_4: Dropout
input: (None, 64, 12, 12)
output: (None, 64, 12, 12)



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 26, 26)        320
_________________________________________________________________
activation_1 (Activation)    (None, 32, 26, 26)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 64, 24, 24)        18496
_________________________________________________________________
activation_2 (Activation)    (None, 64, 24, 24)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 64, 12, 12)        0
_________________________________________________________________
dropout_4 (Dropout)          (None, 64, 12, 12)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 9216)              0
_________________________________________________________________
dense_6 (Dense)              (None, 128)               1179776
_________________________________________________________________
activation_3 (Activation)    (None, 128)               0
_________________________________________________________________
dropout_5 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_7 (Dense)              (None, 10)                1290
_________________________________________________________________
activation_4 (Activation)    (None, 10)                0
=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
_________________________________________________________________
None
```

CNN has two networks, first one is convolutional and second one is classifier, classical deep NN. CNN has two 2D filter bank layers and pooling layer. First 2D layers have 32 filters. Second 2D layers have 64 filters. Pooling layer has 64 filters.

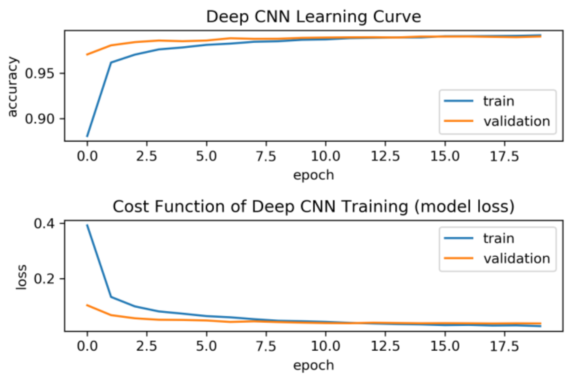
Classifier, classical deep NN has 128 neurons in hidden and 10 neurons in output layer. Number of total parameters of Deep CNN is 1,199,882.

Training sequences are:

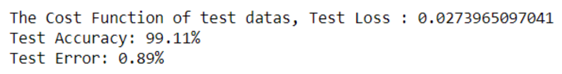
```
Train on 48000 samples, validate on 12000 samples
Epoch 1/20
171s - loss: 0.3923 - acc: 0.8810 - val_loss: 0.1033 - val_acc: 0.9703
Epoch 2/20
171s - loss: 0.1334 - acc: 0.9615 - val_loss: 0.0676 - val_acc: 0.9803
Epoch 3/20
172s - loss: 0.0992 - acc: 0.9701 - val_loss: 0.0561 - val_acc: 0.9838
Epoch 4/20
168s - loss: 0.0811 - acc: 0.9759 - val_loss: 0.0509 - val_acc: 0.9856
Epoch 5/20
169s - loss: 0.0729 - acc: 0.9780 - val_loss: 0.0502 - val_acc: 0.9848
Epoch 6/20
173s - loss: 0.0641 - acc: 0.9809 - val_loss: 0.0482 - val_acc: 0.9856
Epoch 7/20
171s - loss: 0.0597 - acc: 0.9822 - val_loss: 0.0430 - val_acc: 0.9881
Epoch 8/20
173s - loss: 0.0530 - acc: 0.9842 - val_loss: 0.0453 - val_acc: 0.9874
Epoch 9/20
167s - loss: 0.0472 - acc: 0.9848 - val_loss: 0.0425 - val_acc: 0.9875
Epoch 10/20
166s - loss: 0.0456 - acc: 0.9864 - val_loss: 0.0404 - val_acc: 0.9885
Epoch 11/20
167s - loss: 0.0432 - acc: 0.9868 - val_loss: 0.0388 - val_acc: 0.9888
Epoch 12/20
166s - loss: 0.0396 - acc: 0.9882 - val_loss: 0.0382 - val_acc: 0.9891
Epoch 13/20
167s - loss: 0.0373 - acc: 0.9886 - val_loss: 0.0402 - val_acc: 0.9892
Epoch 14/20
174s - loss: 0.0354 - acc: 0.9889 - val_loss: 0.0391 - val_acc: 0.9889
Epoch 15/20
162s - loss: 0.0341 - acc: 0.9890 - val_loss: 0.0381 - val_acc: 0.9899
Epoch 16/20
168s - loss: 0.0315 - acc: 0.9903 - val_loss: 0.0385 - val_acc: 0.9899
Epoch 17/20
172s - loss: 0.0324 - acc: 0.9904 - val_loss: 0.0381 - val_acc: 0.9900
Epoch 18/20
170s - loss: 0.0299 - acc: 0.9905 - val_loss: 0.0374 - val_acc: 0.9896
Epoch 19/20
168s - loss: 0.0306 - acc: 0.9907 - val_loss: 0.0380 - val_acc: 0.9892
Epoch 20/20
166s - loss: 0.0278 - acc: 0.9913 - val_loss: 0.0375 - val_acc: 0.9900
```

```
Saved trained model at C:\Users\Enespc\Desktop\HASAN\python for programmer\project\keras_mnist_theano_cnn.h5
```

Learning Curve of Deep CNN

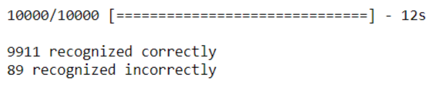


Performance of Deep CNN :



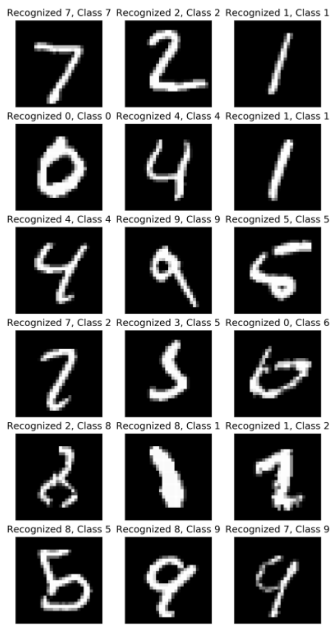
```
The Cost Function of test datas, Test Loss : 0.0273965097041
Test Accuracy: 99.11%
Test Error: 0.89%
```

Recognition Performance of Deep CNN

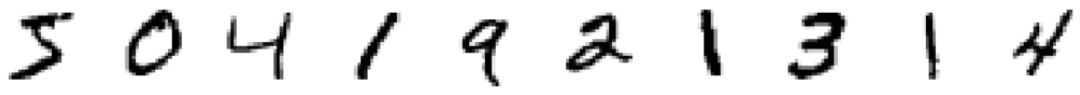


```
10000/10000 [==============================] - 12s

9911 recognized correctly
89 recognized incorrectly
```

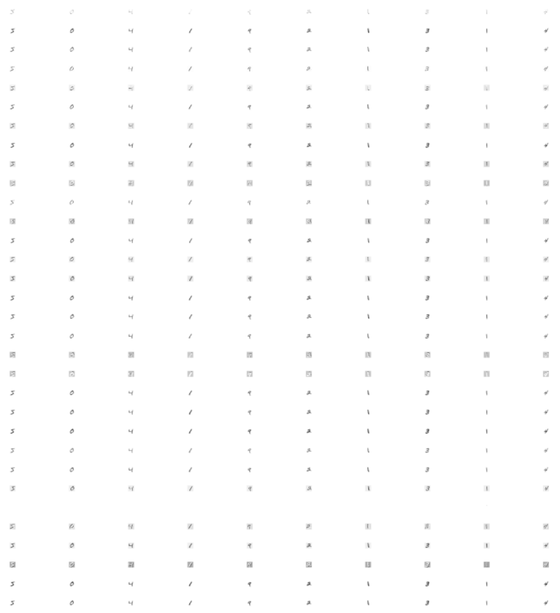
Sample of recognized and unrecognized digits of deep CNN;



visualization of Input Layer of Deep CNN, Original Image

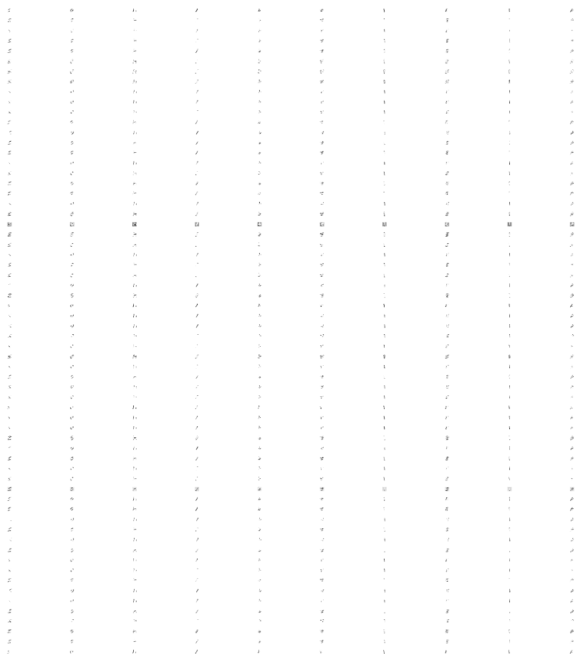


visualization of First Layer with 32 filters



visualization of second Layer with 64 filters

visualization of third layer with 64 filters



4. Screenshots of the program output

All screenshots are given in above section.

5. Conclusion

In this project, I discovered the MNIST handwritten digit recognition problem and deep learning models. They are developed in Python using the Keras library that are capable of achieving excellent results.

6. Python program

Python program "project_mnist_keras_hasan_palaz_01.ipynb" is archived in zip folder.

The folder contains two files;

project_mnist_keras_hasan_palaz_01.ipynb

project_report_hasan_palaz_01.pdf

and trained_folders. The folder consist of trained deep NN models;

keras_mnist_tensorflow_deep_nn.h5

keras_mnist_tensorflow_mlp.h5

keras_mnist_theano_cnn.h5

keras_mnist_theano_deep_nn.h5

keras_mnist_theano_mlp.h5