CSC 3304 MACHINE LEARNING

Semester 1 2019/2020

Section 1

# Digit Recognition Using CONVOLUTIONAL NEURAL NETWORK

Lecturer:

ASSOC. PROF. DR. AMELIA RITAHANI BINTI ISMAIL

**Group Members:**

| | |
|---|---|
| Zian MD Afique Amin | - 1631005 |
| Khan Nasik Sami | - 1638153 |
| Md Sariful Islam | - 1626667 |
| Abderrahmane El Massaoudi | -1919859 |
| Suhib Ahmad Abdulla | -1423183 |

## TABLE OF CONTENTS

## 1. Introduction

Digit recognition system is the working of a machine to train itself or recognizing the digits from different sources like emails, bank cheque, papers, images, etc.

### 1.1 **Problem Background**

The handwritten digits are not always of the same size, width, orientation and justified to margins as they differ from writing of person to person, so the general problem would be while classifying the digits due to the similarity between digits such as 1 and 7, 5 and 6, 3 and 8, 2 and 5, 2 and 7, etc. This problem is faced more when many people write a single digit with a variety of different handwritings. Lastly, the uniqueness and variety in the handwriting of different individuals also influence the formation and appearance of the digits. With the use of deep learning and machine learning, human effort can be reduced in recognizing, learning, predictions and many more areas. We are going to present recognizing the handwritten digits (0 to 9) from the famous MNIST dataset, comparing classifiers like KNN, PSVM, NN and Convolution Neural Network (CNN) on the basis of performance, accuracy, time, with using different parameters with the classifiers.

### 1.2 Objective

The main objective is to recognize a digit properly. We are going to improve the readability and the automatic recognition of handwritten document images, preprocessing steps are imperative. We will train the machine with big data so that it can predict our expectations

1. To understand the datasets properly and analysis the data
2. Select the suitable algorithm (CNN) for the dataset and the problem.
3. To develop an algorithm utilizing Convolutional Neural Network (CNN) to extract significant features from the dataset.
4. To analyze the results obtained by the algorithm based on the developed model

### 1.3 Expected Outcome:

From the CNN processes , we will come out with a flattened matrix that will be used as an input and we will train those inputs to recognize the expected digit. So, after the project Digit recognition system, it will act as a machine to train itself or recognizing the digits from different sources like emails, bank cheque, papers, images, etc. and in different real-world scenarios for online handwriting recognition on computer tablets or system, recognize number plates of vehicles, processing bank cheque amounts, numeric entries in forms filled up by hand. It will automatically detect which digit it is. We expect accuracy will rise by 90%.

# 2 Literature review

## 2.1 The drastic growth of data

Technology has come a long way since the beginning of the digital era. The fact that we now have the ability to create new virtual world using virtual reality and are able to work more productively and efficiently is proof of the technical advancements. However, with the increasing usage of technology, there is also an exponential increase in the size of data created by technological devices, which is used to provide services to users such as social media and online entertainment. As long as the information is in a digital format, it is possible to manipulate it and store it in a very efficient way. The problem actually arises when the data is not in the digital format, rather, it is available in the physical world. For the longest time, it has been difficult to keep track of all the information and would take a lot of hours and people to convert all the information into digital data in order to provide better services. It is also becoming more and more difficult since the growth of data is not slowing down at all. Before long, it will impossible to deal with unless a solution is found.

## 2.2 Introduction to hand-written digit recognition

With the help of technology, the manipulation of data and its storage is far more efficient than it used to be, making it currently the best and most efficient way to deal with data. The introduction of Artificial Intelligence and eventually, Machine learning, has led to the development of what is known as hand-written digit recognition. The recognition of digits is also a well-researched area in Machine Learning and is considered by some as the introduction to Machine Learning algorithms. Using Machine Learning, it is possible to design algorithms that will be utilized to interpret the digits in images. This involves computer vision, which is used for object recognition.

## 2.3 Why Convolutional Neural Network

There are several algorithms that people used for image classification before CNN became popular. People used to create features from images and then feed those features into some classification algorithm like SVM. Some algorithm also used the pixel level values of images as a feature vector too.

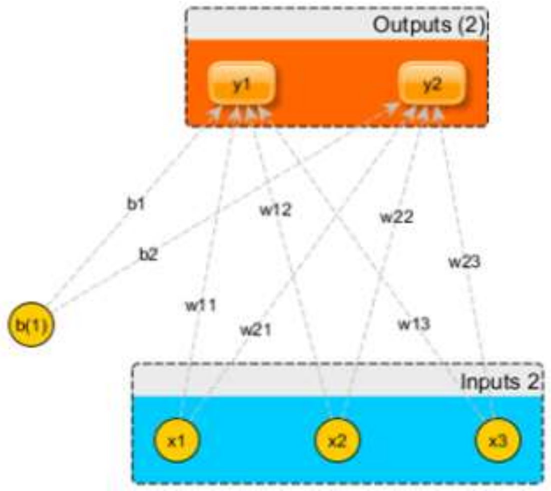
The main reason behind CNN is feature engineering is not required. Before CNN, we need to spend so much time on feature selection (algorithm for features extraction). When we compare handcrafted features with CNN, CNN performance well and it gives better accuracy. It is covering local and global features. It also learns different features from images.

In algorithm based image classification, we need to select the features (local, global) and classifiers. In some cases, global features worked well and in some cases the local features worked well.

Other learning algorithms or models can also be used for image classification. However CNN has emerged as the model of choice for multiple reasons. These include the multiple uses of the convolution operator in image processing, The CNN architecture implicitly combines the benefits obtained by a standard neural network training with the convolution operation to efficiently classify images. Further, being a neural network, CNN (and its variants) are also scalable for large datasets, which is often the case when images are to be classified.

## 2.4 Neural Network:

A Neural Network is put together by hooking together many of our simple "neurons," so that the output of a neuron can be the input of another. It consists of an input layer, multiple hidden layers, and an output layer. Every node in one layer is connected to every other node in the right next layer.



$$y_1 = [(w_{11}.x_1) + (w_{12}.x_2) + (w_{13}.x_3)] + b1$$
$$y_2 = [(w_{21}.x_1) + (w_{22}.x_2) + (w_{23}.x_3)] + b2$$

$$\underbrace{\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}}_{\text{One collumn per x dimension}} . \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \therefore$$

$$H(X) = (W.x) + b^T$$
$$H(X) = (W^T.x) + b$$

$$(\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} . \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix}) + \begin{bmatrix} b_1 & b_2 \end{bmatrix} = \begin{bmatrix} y_1 & y_2 \end{bmatrix} \therefore$$
$$H(X) = (x.W^t) + b$$

Figure 1: Neural Network

The important thing about neurons is that they can learn. Basically, a neural network is a system of interconnected artificial neurons that exchange the messages between them. The connections consist of numeric weights that are tuned during the training process, so the proper trained network could respond correctly when presented with an image to recognize or detect.

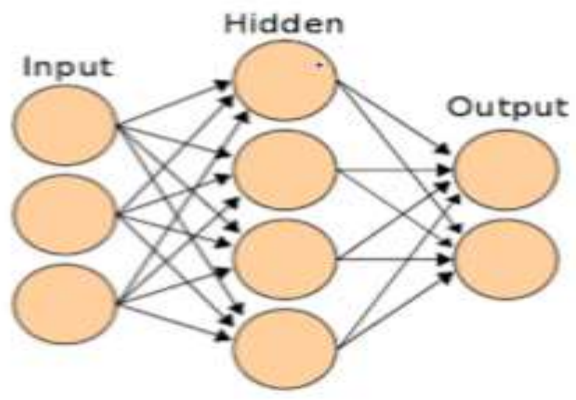


Figure 2: ANN, a single hidden layer

Figure shows the neural network that is built on a single hidden layer. First layer detects the pattern of the input, the second layer detects the patterns of those patterns and third layer detects the patterns of those patterns.

### 2.4.1 Training A Neural network:

Targets: Determine the weights for the network.

Method: Using gradient descent to minimize the error function.

Deep Learning: Applying multiple hidden layers of neural network.

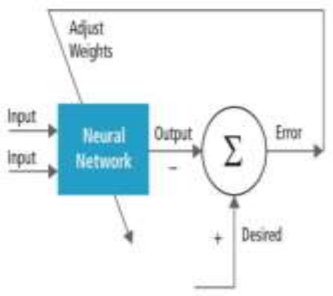


Figure 2: Training of neural network [1]

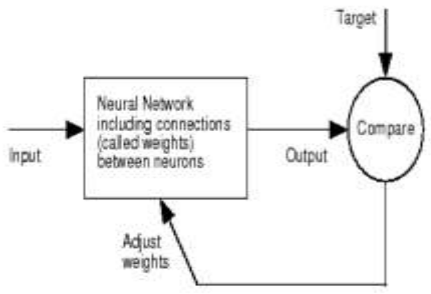

Target
Input
Neural Network including connections (called weights) between neurons
Output
Compare
Adjust weights

Figure 3: Training of neural network [2]

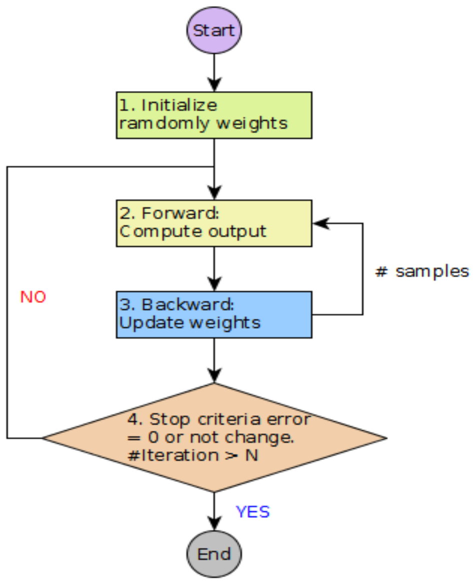

Start
1. Initialize ramdomly weights
2. Forward: Compute output
3. Backward: Update weights
# samples
NO
4. Stop criteria error = 0 or not change. #Iteration > N
YES
End

Figure 4: Flow Chart

### 2.5 Convolutional Neural Network:

Convolutional neural networks were inspired by biological processes in which the connectivity pattern between neurons is inspired by the organization of the animal visual corte

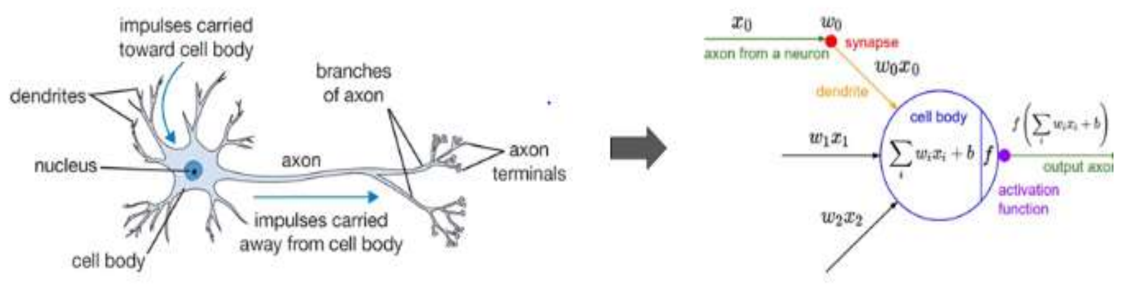


Figure 5: The biological process

Convolutional neural networks (CNNs) have a remote origin. They developed while multi-layer perceptrons were perfected, and the first concrete example is the neocognitron.

The neocognitron is a hierarchical, multilayered Artificial Neural Network (ANN), and was introduced in a 1980 paper by Prof. Fukushima and has the following principal features:

It is the year 1994, and this is one of the very first convolutional neural networks, and what propelled the field of Deep Learning. This pioneering work by Yann LeCun was named LeNet5 after many previous successful iterations.

The main component of a CNN is a convolutional layer. Its job is to detect important features in the image pixels. Layers that are deeper (closer to the input) will learn to detect simple features such as edges and color gradients, whereas higher layers will combine simple features into more complex features. Finally, dense layers at the top of the network will combine very high level features and produce classification predictions.

CNNs are usually applied to image data. Every image is a matrix of pixel values. With colored images, particularly RGB (Red, Green, Blue)-based images, the presence of separate color channels (3 in the case of RGB images) introduces an additional 'depth' field to the data, making the input 3-dimensional. Hence, for a given RGB image of size, say 255×255 (Width x Height) pixels, we'll have 3 matrices associated with each image, one for each of the color channels. Thus, the image in it's entirety, constitutes a 3-dimensional structure called the Input Volume (255x255x3).

### 2.5.1 Features:

A feature is a distinct and useful observation or pattern obtained from the input data that aids in performing the desired image analysis. The CNN learns the features from the input images. Typically, they emerge repeatedly from the data to gain prominence.

### 2.5.2 Receptive field:

It is impractical to connect all neurons with all possible regions of the input volume. It would lead to too many weights to train, and produce too high a computational complexity.

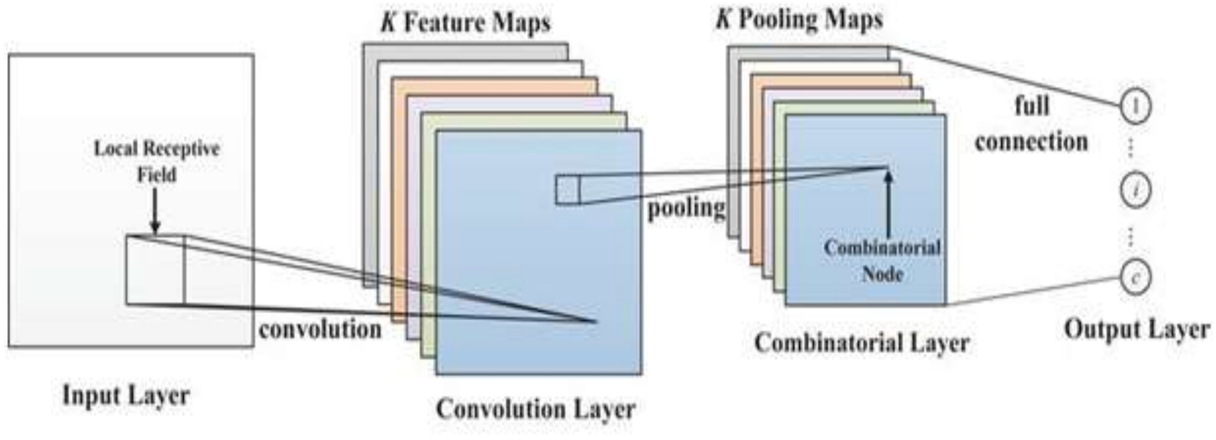


Figure 6: Different layers

Thus, instead of connecting each neuron to all possible pixels, we specify a 2 dimensional region called the 'receptive field[14]' (say of size 5×5 units) extending to the entire depth of the input (5x5x3 for a 3 colour channel input), within which the encompassed pixels are fully connected to the neural network's input layer.

**2.5. 3 Activation Layer:**

Activation Layer is an activation function that decides the final value of a neuron. Suppose a cell value should be 1 ideally, however it has a value of 0.85, since you can never achieve a probability of 1 in CNN thus we apply an activation function. E.g. if cell value is greater than 0.7 make 1 else make it 0. In this way one can easily achieve an image with sharp features.

**2.5.4 Pooling Layer:**

The pooling layer is usually placed after the Convolutional layer. Its primary utility lies in reducing the spatial dimensions (Width x Height) of the Input Volume for the next Convolutional Layer. It does not affect the depth dimension of the Volume.

Back propagation is the process in which we try to bring the error down. By error, I mean the difference in y and y'. This will help w, to fit the data set that we gave to the network. We perform Back propagation using Gradient descent process. This process tries to bring the error value close to zero.

**2.5.5 Fully Connected Layer:**

At the end of convolution and pooling layers, networks generally use fully-connected layers in which each pixel is considered as a separate neuron just like a regular neural network. The last fully-connected layer will contain as many neurons as the number of classes to be predicted. For instance, in CIFAR-10 case, the last fully-connected layer will have 10 neurons.

There are three hyper parameters control the size of the output volume: the depth, stride and zero-padding.

- Depth (D) of the output volume is a hyper parameter: it corresponds to the number of filters we would like to use, each learning to look for something different in the input.

- Stride (S) with which we slide the filter. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially.

- Zero-Padding refers to the process of symmetrically adding zeroes to the input matrix. It’s a commonly used modification that allows the size of the input to be adjusted to our requirement. It is mostly used in designing the CNN layers when the dimensions of the input volume need to be preserved in the output volume.

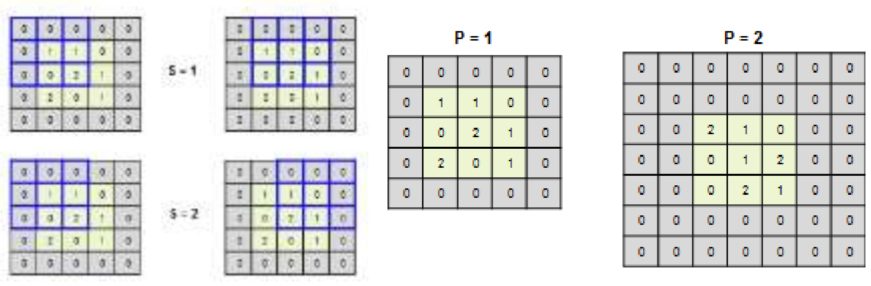


Figure7: Spatial Arrangement

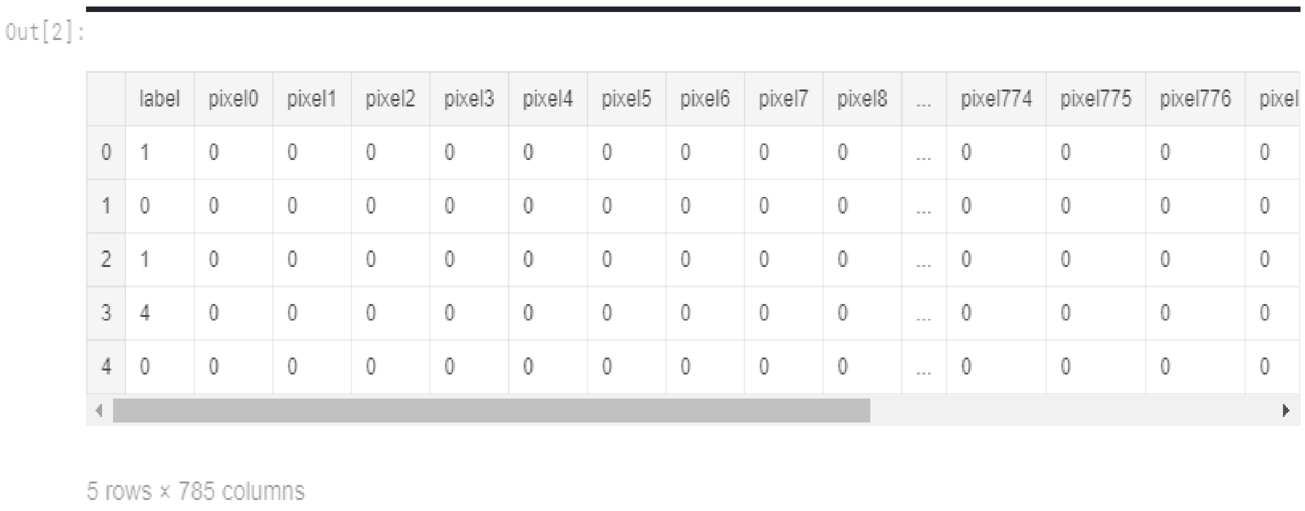
By using this algorithm, we can recognize any digit (0-9)

## 3 Experimental Design

### 3.1 Data description:

This dataset is a digit recognition dataset originating from the digit recognizer .The data files train.csv and test.csv contain gray-scale images of hand-drawn digits, from zero through nine. Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255, inclusive. The training data set, (train.csv), has 785 columns.

The test data set, (test.csv), is the same as the training set, except that it does not contain the "label" column. The first column, called "label", is the digit that was drawn by the user. The rest of the columns contain the pixel-values of the associated image.



Out[2]:

| | label | pixel0 | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 | pixel7 | pixel8 | ... | pixel774 | pixel775 | pixel776 | pixel |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 |
| 3 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 |

5 rows × 785 columns

Figure 8: Data Set

## 3.2 Data Preprocessing:

The dataset is a large dataset and it reasonable to assume that it needed to be preprocessed before being used. Preprocessing is usually done to handle missing values or incomplete data. Our data is originally from the kaggle, but it has already been processed by the author, Poonam Ligade at Kaggle before uploading. Thus, the training data set, (train.csv), has 785 columns.

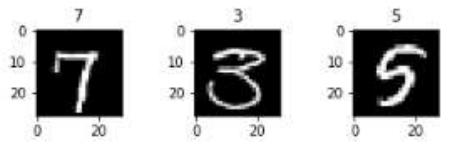
## 3.3 Data Visualization:

Data visualization is the practice of presenting data using pictorial or graphical formats. This helps decision makers to understand difficult concepts or identify new patterns, as the data is presented visually. For our project we segmented some of our data and for visualization we use Matplotlib, which is a 2-D plotting library that helps in visualizing figure.

%matplotlib inline sets the backend of matplotlib to the 'inline' backend: With this backend, the output of plotting commands is displayed inline within frontends like the Jupyter notebook, directly below the code cell that produced it. The resulting plots will then also be stored in the notebook document.

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

import matplotlib.pyplot as plt
%matplotlib inline
```

```
#Convert train datset to (num_images, img_rows, img_cols) format
X_train = X_train.reshape(X_train.shape[0], 28, 28)

for i in range(6, 9):
    plt.subplot(330 + (i+1))
    plt.imshow(X_train[i], cmap=plt.get_cmap('gray'))
    plt.title(y_train[i]);
```



```
#expand 1 more dimention as 1 for colour channel gray
X_train = X_train.reshape(X_train.shape[0], 28, 28,1)
X_train.shape
```

```
(42000, 28, 28, 1)
```

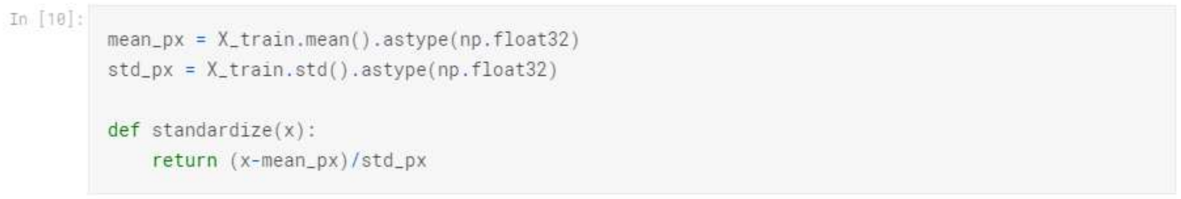
```
X_test = X_test.reshape(X_test.shape[0], 28, 28,1)
X_test.shape
```

```
(28000, 28, 28, 1)
```

Figure 9: Data Visualization

## 3.4 Feature Standardization

It is important preprocessing step. It is used to centre the data around zero mean and unit variance. It is a step of Data Pre Processing which is applied to independent variables or features of data. It basically helps to normalise the data within a particular range. Sometimes, it also helps in speeding up the calculations in an algorithm.



```
In [10]:
mean_px = X_train.mean().astype(np.float32)
std_px = X_train.std().astype(np.float32)

def standardize(x):
    return (x-mean_px)/std_px
```

Figure 10: Feature Standardization

## 3.5 One Hot encoding of labels.

A one-hot vector is a vector which is 0 in most dimensions, and 1 in a single dimension. In this case, the nth digit will be represented as a vector which is 1 in the nth dimension.

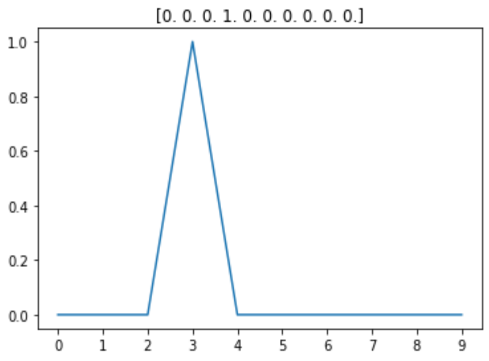
For example, 3 would be [0,0,0,1,0,0,0,0,0,0].

```
In [11]:  from keras.utils.np_utils import to_categorical
          y_train= to_categorical(y_train)
          num_classes = y_train.shape[1]
          num_classes
```

```
Out[11]:  10
```

Lets plot 10th label.

```
In [12]:  plt.title(y_train[9])
          plt.plot(y_train[9])
          plt.xticks(range(10));
```



Oh its 3 !

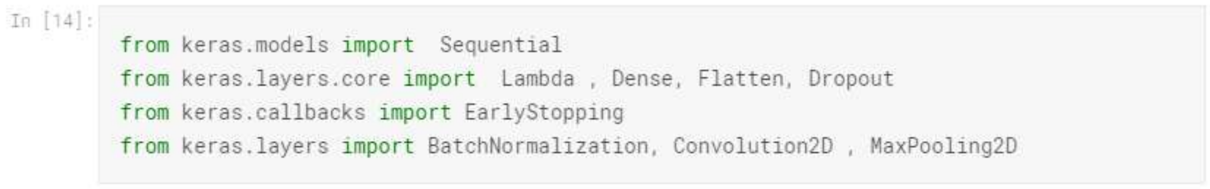
Figure 11: One Hot encoding of labels.

### 3.6 Designing Neural Network Architecture

```
In [13]:  # fix random seed for reproducibility
          seed = 43
          np.random.seed(seed)
```

Figure 12: Fixing random seed

## 3.7 Linear Model:



```
In [14]:
from keras.models import  Sequential
from keras.layers.core import  Lambda , Dense, Flatten, Dropout
from keras.callbacks import EarlyStopping
from keras.layers import BatchNormalization, Convolution2D , MaxPooling2D
```

Figure 13: Model

Lets create a simple model from Keras Sequential layer.

1. Lambda layer performs simple arithmetic operations like sum, average, exponentiation etc.

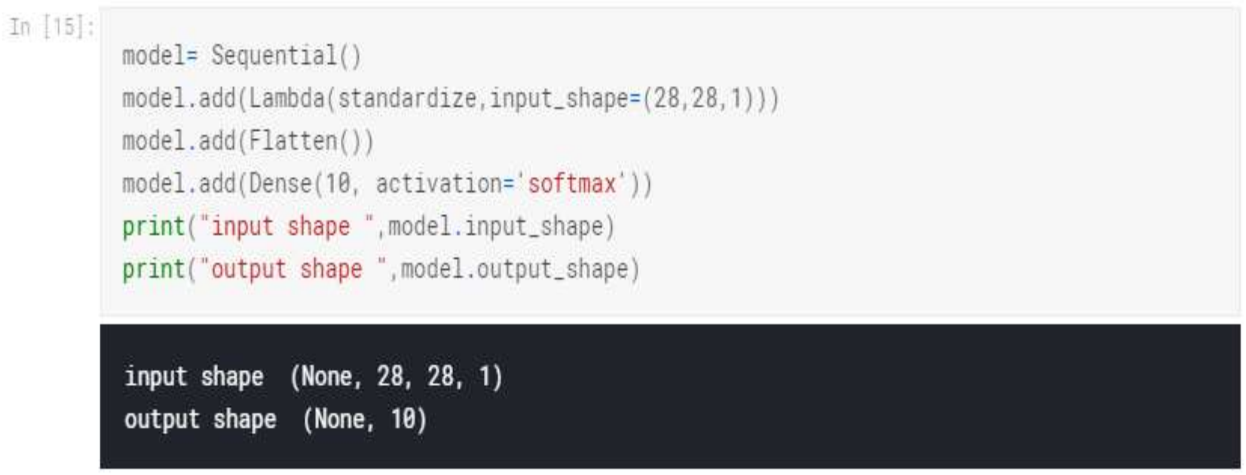
   In 1st layer of the model we have to define input dimensions of our data in

   (rows, columns, color channel) format.

2. Flatten will transform input into 1D array.
3. Dense is fully connected layer that means all neurons in previous layers will be connected to all neurons in fully connected layer. In the last layer we have to specify output dimensions/classes of the model. Here it's 10, since we have to output 10 different digit labels.

Code:

```
from keras.models import  Sequential
from keras.layers.core import  Lambda , Dense, Flatten, Dropout
from keras.callbacks import EarlyStopping
from keras.layers import BatchNormalization, Convolution2D , MaxPooling2D

model= Sequential()
model.add(Lambda(standardize,input_shape=(28,28,1)))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))
print("input shape ",model.input_shape)
print("output shape ",model.output_shape)
```



```
In [15]:
model= Sequential()
model.add(Lambda(standardize,input_shape=(28,28,1)))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))
print("input shape ",model.input_shape)
print("output shape ",model.output_shape)
```

```
input shape  (None, 28, 28, 1)
output shape  (None, 10)
```

Figure 14: Linear Model

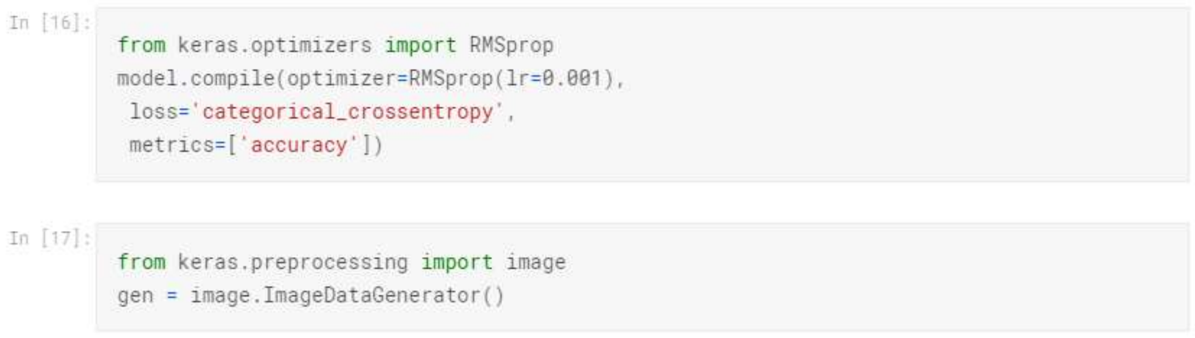
### 3.8 Compile network:

Before making network ready for training we have to make sure to add below things:

1. A loss function: to measure how good the network is
2. An optimizer: to update network as it sees more data and reduce loss value
3. Metrics: to monitor performance of network

Code:

```
from keras.optimizers import RMSprop
model.compile(optimizer=RMSprop(lr=0.001),
 loss='categorical_crossentropy',
 metrics=['accuracy'])
from keras.preprocessing import image
gen = image.ImageDataGenerator()
```



```
In [16]:
from keras.optimizers import RMSprop
model.compile(optimizer=RMSprop(lr=0.001),
 loss='categorical_crossentropy',
 metrics=['accuracy'])
```

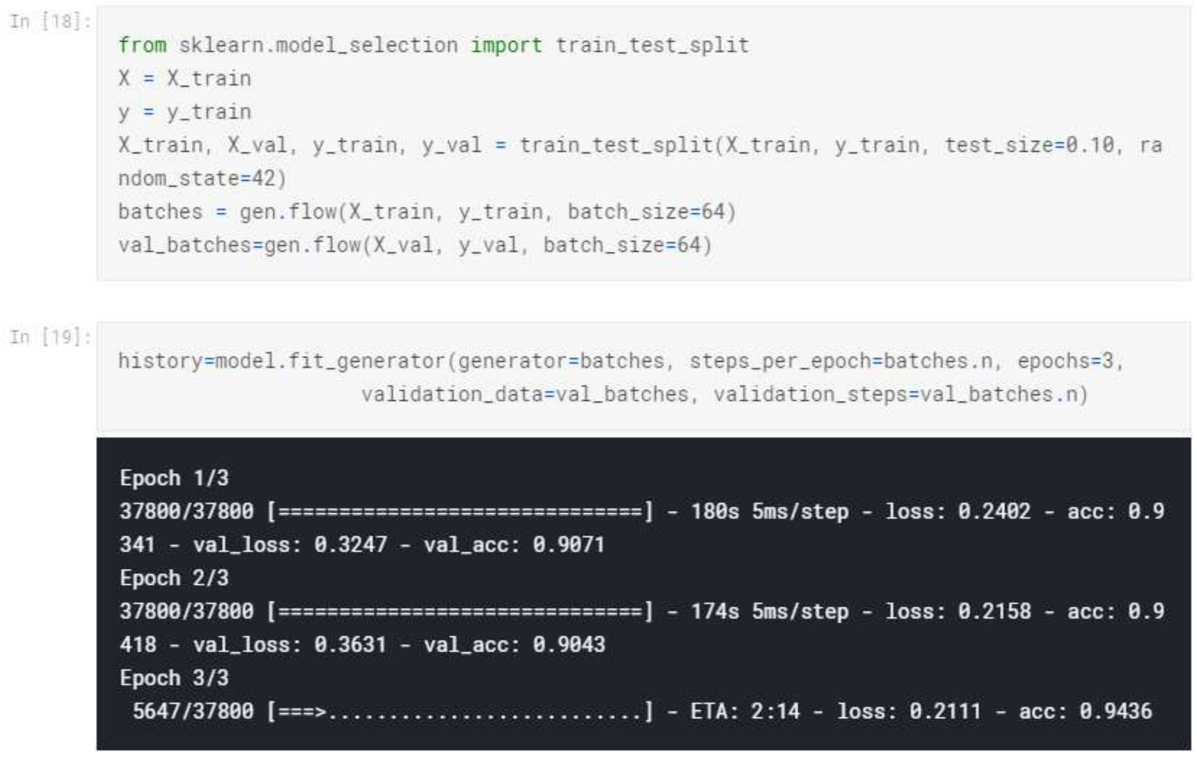
```
In [17]:
from keras.preprocessing import image
gen = image.ImageDataGenerator()
```

Figure 15: Compile Network

## 3.9 Cross Validation:

**Code:**

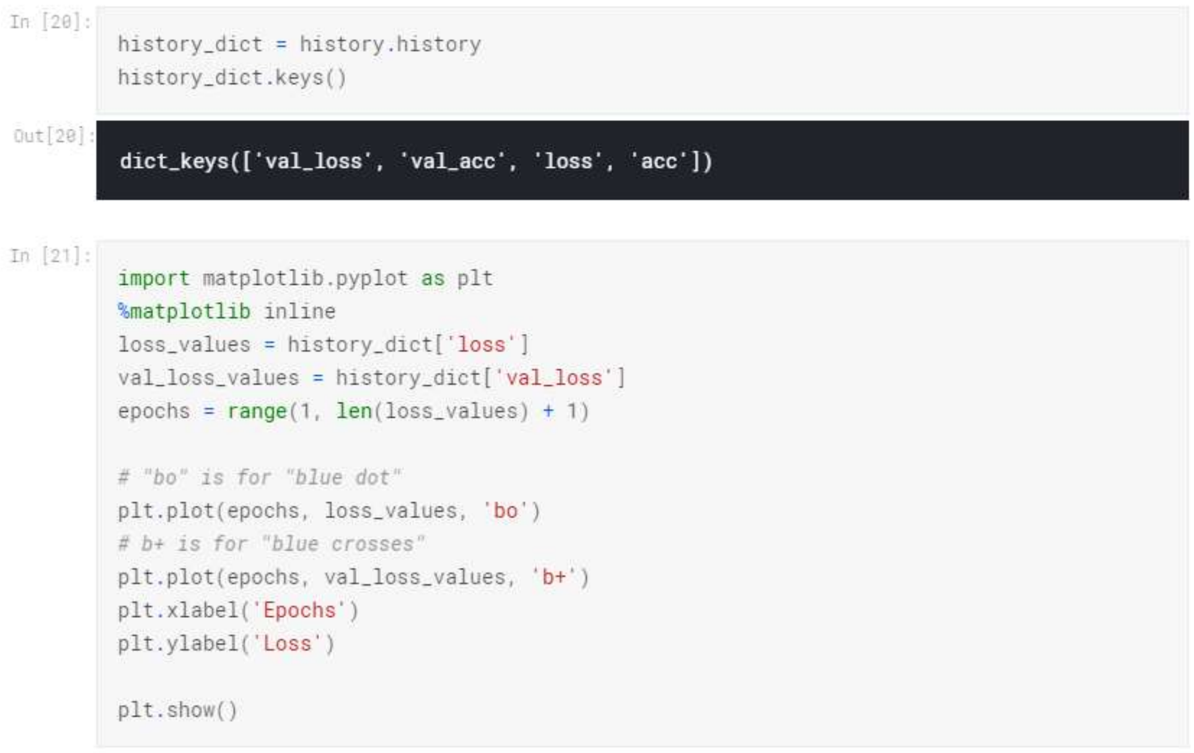
```
from sklearn.model_selection import train_test_split
X = X_train
y = y_train
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.10, r
andom_state=42)
batches = gen.flow(X_train, y_train, batch_size=64)
val_batches=gen.flow(X_val, y_val, batch_size=64)


history=model.fit_generator(generator=batches, steps_per_epoch=batches.n, epochs=3,
                    validation_data=val_batches, validation_steps=val_batches.n)


history_dict = history.history
history_dict.keys()
```



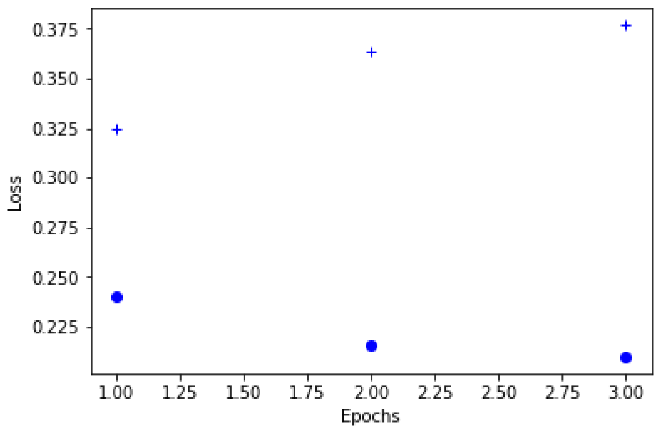
```
In [18]:
from sklearn.model_selection import train_test_split
X = X_train
y = y_train
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.10, ra
ndom_state=42)
batches = gen.flow(X_train, y_train, batch_size=64)
val_batches=gen.flow(X_val, y_val, batch_size=64)

In [19]:
history=model.fit_generator(generator=batches, steps_per_epoch=batches.n, epochs=3,
                    validation_data=val_batches, validation_steps=val_batches.n)

Epoch 1/3
37800/37800 [==============================] - 180s 5ms/step - loss: 0.2402 - acc: 0.9
341 - val_loss: 0.3247 - val_acc: 0.9071
Epoch 2/3
37800/37800 [==============================] - 174s 5ms/step - loss: 0.2158 - acc: 0.9
418 - val_loss: 0.3631 - val_acc: 0.9043
Epoch 3/3
 5647/37800 [===>..........................] - ETA: 2:14 - loss: 0.2111 - acc: 0.9436
```

**Code: Fo**

```
import matplotlib.pyplot as plt
%matplotlib inline
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']
epochs = range(1, len(loss_values) + 1)

# "bo" is for "blue dot"
plt.plot(epochs, loss_values, 'bo')
# b+ is for "blue crosses"
plt.plot(epochs, val_loss_values, 'b+')
plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.show()
```



```
In [20]:
history_dict = history.history
history_dict.keys()

Out[20]:
dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])

In [21]:
import matplotlib.pyplot as plt
%matplotlib inline
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']
epochs = range(1, len(loss_values) + 1)

# "bo" is for "blue dot"
plt.plot(epochs, loss_values, 'bo')
# b+ is for "blue crosses"
plt.plot(epochs, val_loss_values, 'b+')
plt.xlabel('Epochs')
plt.ylabel('Loss')

plt.show()
```



```
In [22]:
plt.clf()   # clear figure
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']

plt.plot(epochs, acc_values, 'bo')
plt.plot(epochs, val_acc_values, 'b+')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

plt.show()
```

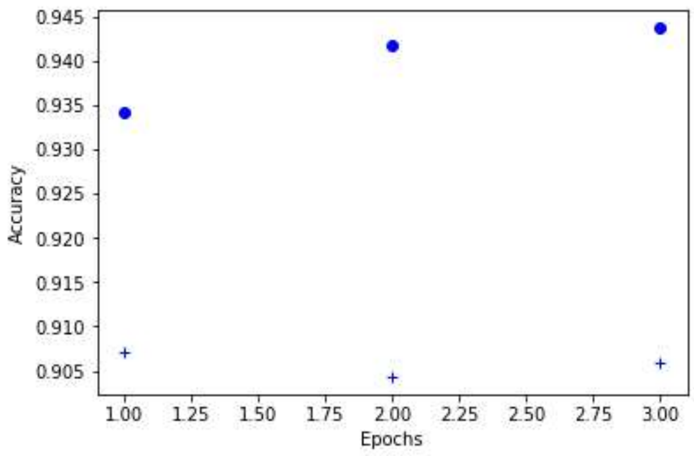


**Figure 16: Cross Validation**

## 3.10 Fully Connected Model:

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Adding another Dense Layer to model.

Code:

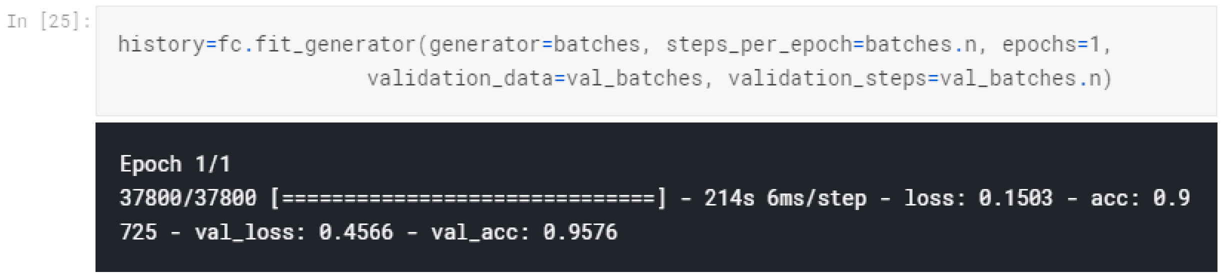
```
def get_fc_model():
    model = Sequential([
        Lambda(standardize, input_shape=(28,28,1)),
        Flatten(),
        Dense(512, activation='relu'),
        Dense(10, activation='softmax')
        ])
    model.compile(optimizer='Adam', loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

```
In [24]:
fc = get_fc_model()
fc.optimizer.lr=0.01
```

```
In [25]:
history=fc.fit_generator(generator=batches, steps_per_epoch=batches.n, epochs=1,
                    validation_data=val_batches, validation_steps=val_batches.n)
```

```
In [23]: def get_fc_model():
             model = Sequential([
                 Lambda(standardize, input_shape=(28,28,1)),
                 Flatten(),
                 Dense(512, activation='relu'),
                 Dense(10, activation='softmax')
                 ])
             model.compile(optimizer='Adam', loss='categorical_crossentropy',
                           metrics=['accuracy'])
             return model
```

```
In [24]: fc = get_fc_model()
         fc.optimizer.lr=0.01
```



```
In [25]: history=fc.fit_generator(generator=batches, steps_per_epoch=batches.n, epochs=1,
                              validation_data=val_batches, validation_steps=val_batches.n)
```

```
Epoch 1/1
37800/37800 [==============================] - 214s 6ms/step - loss: 0.1503 - acc: 0.9
725 - val_loss: 0.4566 - val_acc: 0.9576
```

**Figure: Connected Model**

## 3.11

## Convolutional Neural Network:

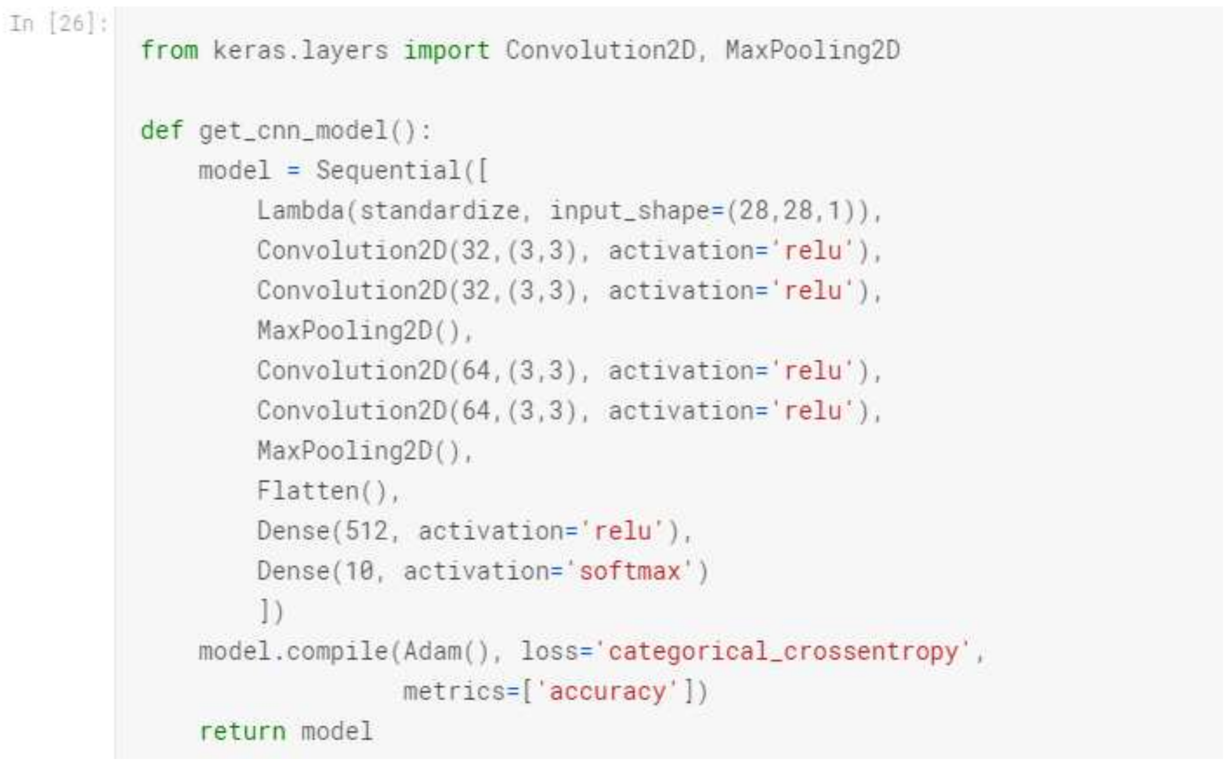
CNNs are extremely efficient for images.

Code:

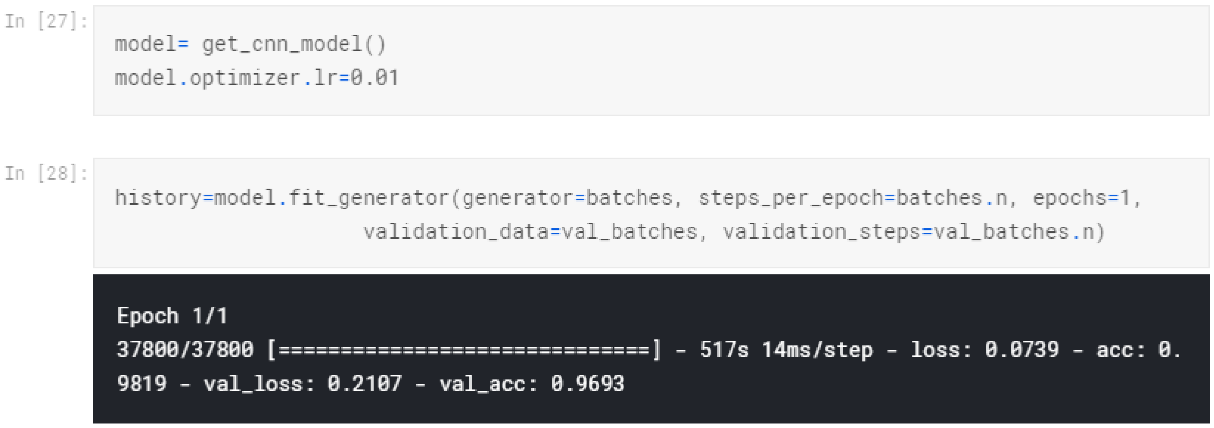
```
from keras.layers import Convolution2D, MaxPooling2D

def get_cnn_model():
    model = Sequential([
        Lambda(standardize, input_shape=(28,28,1)),
        Convolution2D(32,(3,3), activation='relu'),
        Convolution2D(32,(3,3), activation='relu'),
        MaxPooling2D(),
        Convolution2D(64,(3,3), activation='relu'),
        Convolution2D(64,(3,3), activation='relu'),
        MaxPooling2D(),
        Flatten(),
        Dense(512, activation='relu'),
        Dense(10, activation='softmax')
        ])
    model.compile(Adam(), loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

```
model= get_cnn_model()
model.optimizer.lr=0.01
```

```
history=model.fit_generator(generator=batches, steps_per_epoch=batches.n, epochs=1,
                   validation_data=val_batches, validation_steps=val_batches.n)
```



```
In [26]:
from keras.layers import Convolution2D, MaxPooling2D

def get_cnn_model():
    model = Sequential([
        Lambda(standardize, input_shape=(28,28,1)),
        Convolution2D(32,(3,3), activation='relu'),
        Convolution2D(32,(3,3), activation='relu'),
        MaxPooling2D(),
        Convolution2D(64,(3,3), activation='relu'),
        Convolution2D(64,(3,3), activation='relu'),
        MaxPooling2D(),
        Flatten(),
        Dense(512, activation='relu'),
        Dense(10, activation='softmax')
        ])
    model.compile(Adam(), loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```



```
In [27]:
model= get_cnn_model()
model.optimizer.lr=0.01
```

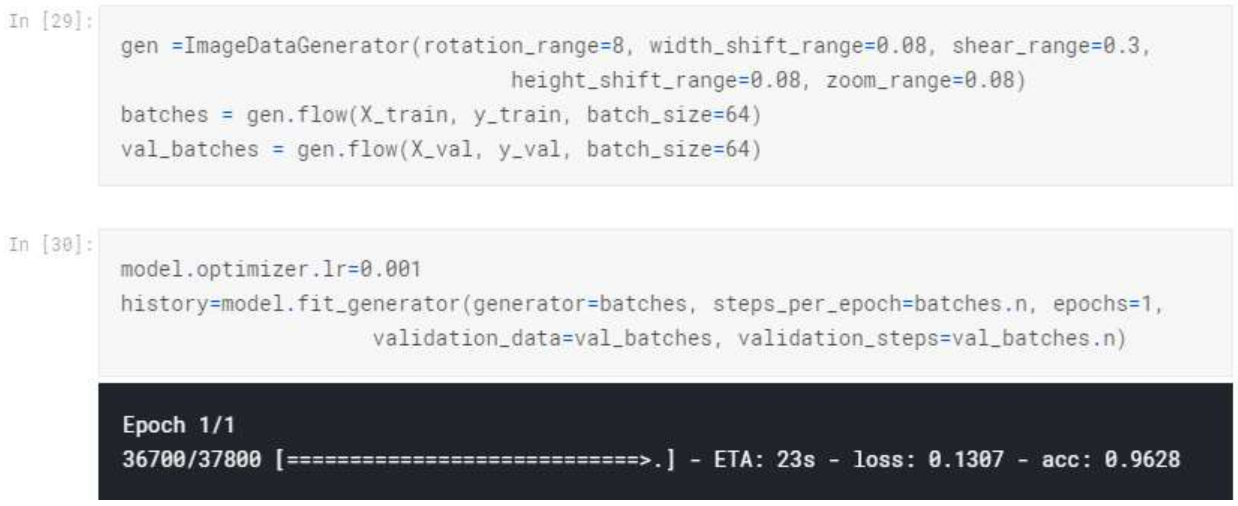
```
In [28]:
history=model.fit_generator(generator=batches, steps_per_epoch=batches.n, epochs=1,
                    validation_data=val_batches, validation_steps=val_batches.n)
```

```
Epoch 1/1
37800/37800 [==============================] - 517s 14ms/step - loss: 0.0739 - acc: 0.
9819 - val_loss: 0.2107 - val_acc: 0.9693
```

Figure 17: CNN

### 3.12 Data Augmentation:

It is technique of showing slightly different or new images to neural network to avoid overfitting. And to achieve better generalization. In case you have very small dataset, you can use different kinds of data augmentation techniques to increase your data size. Neural networks perform better if you provide them more data.



```
In [29]:
gen =ImageDataGenerator(rotation_range=8, width_shift_range=0.08, shear_range=0.3,
                               height_shift_range=0.08, zoom_range=0.08)
batches = gen.flow(X_train, y_train, batch_size=64)
val_batches = gen.flow(X_val, y_val, batch_size=64)
```

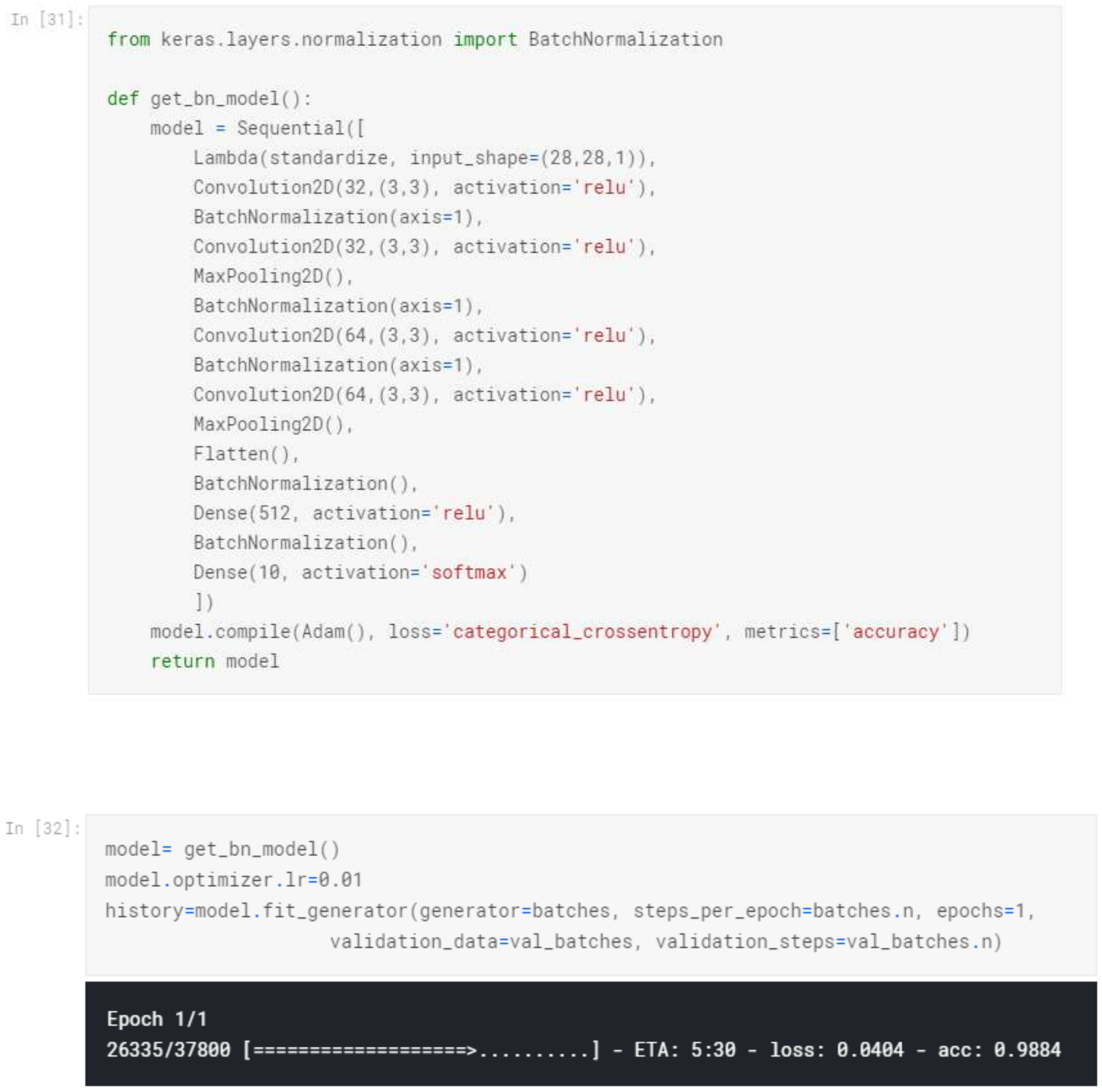
```
In [30]:
model.optimizer.lr=0.001
history=model.fit_generator(generator=batches, steps_per_epoch=batches.n, epochs=1,
                    validation_data=val_batches, validation_steps=val_batches.n)
```

```
Epoch 1/1
36700/37800 [============================>.] - ETA: 23s - loss: 0.1307 - acc: 0.9628
```

Figure 18: Data Augmentation

### 3.13 Adding Batch Normalization:

BN helps to fine tune hyperparameters more better and train really deep neural networks.



```
In [31]:
from keras.layers.normalization import BatchNormalization

def get_bn_model():
    model = Sequential([
        Lambda(standardize, input_shape=(28,28,1)),
        Convolution2D(32,(3,3), activation='relu'),
        BatchNormalization(axis=1),
        Convolution2D(32,(3,3), activation='relu'),
        MaxPooling2D(),
        BatchNormalization(axis=1),
        Convolution2D(64,(3,3), activation='relu'),
        BatchNormalization(axis=1),
        Convolution2D(64,(3,3), activation='relu'),
        MaxPooling2D(),
        Flatten(),
        BatchNormalization(),
        Dense(512, activation='relu'),
        BatchNormalization(),
        Dense(10, activation='softmax')
        ])
    model.compile(Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

```
In [32]:
model= get_bn_model()
model.optimizer.lr=0.01
history=model.fit_generator(generator=batches, steps_per_epoch=batches.n, epochs=1,
                    validation_data=val_batches, validation_steps=val_batches.n)
```

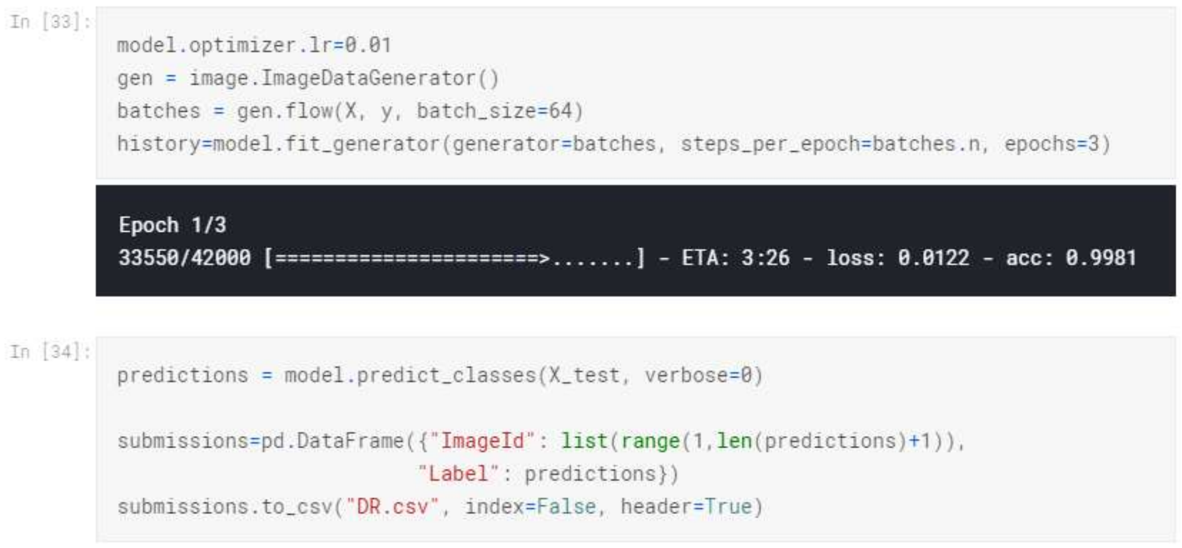
```
Epoch 1/1
26335/37800 [==================>.........] - ETA: 5:30 - loss: 0.0404 - acc: 0.9884
```

**Figure 19: Adding Batch Normalization**

### 3.13 Submitting Prediction

**Code:**

```
model.optimizer.lr=0.01
gen = image.ImageDataGenerator()
batches = gen.flow(X, y, batch_size=64)
history=model.fit_generator(generator=batches, steps_per_epoch=batches.n, epochs=
3)
```

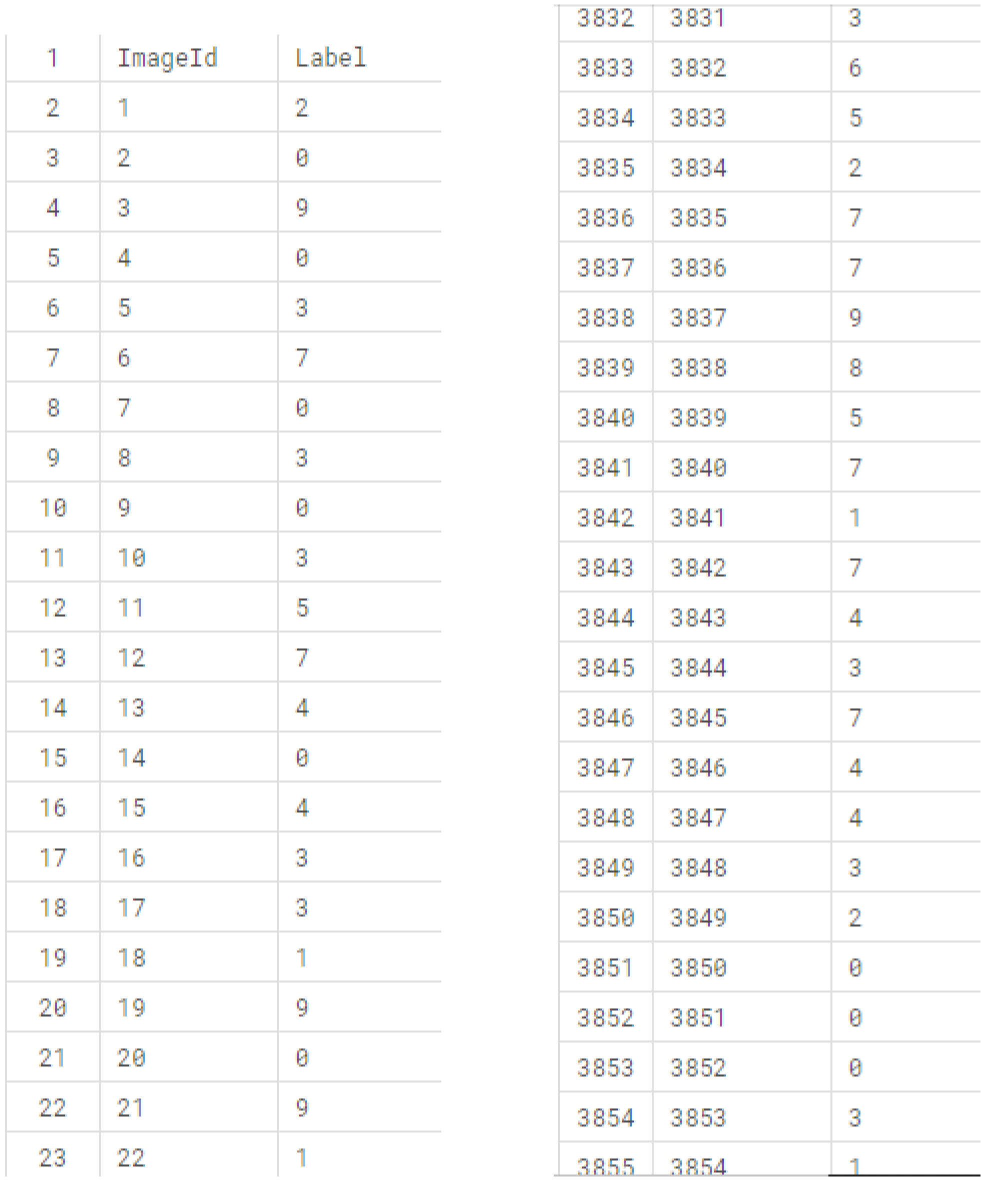


```
In [33]:
model.optimizer.lr=0.01
gen = image.ImageDataGenerator()
batches = gen.flow(X, y, batch_size=64)
history=model.fit_generator(generator=batches, steps_per_epoch=batches.n, epochs=3)
```

```
Epoch 1/3
33550/42000 [======================>.......] - ETA: 3:26 - loss: 0.0122 - acc: 0.9981
```

```
In [34]:
predictions = model.predict_classes(X_test, verbose=0)

submissions=pd.DataFrame({"ImageId": list(range(1,len(predictions)+1)),
                         "Label": predictions})
submissions.to_csv("DR.csv", index=False, header=True)
```

**Figure 20: prediction and output**



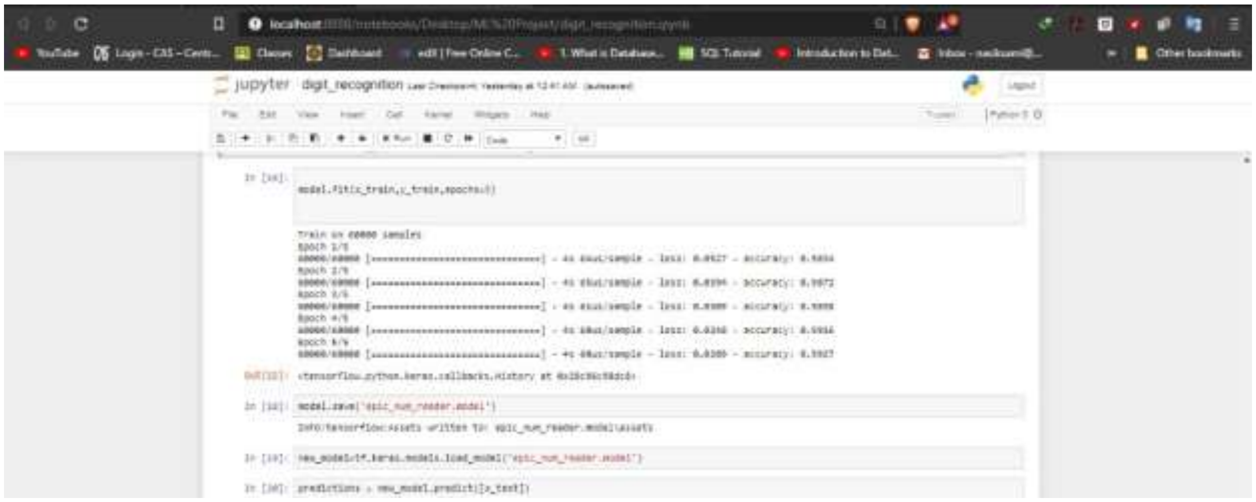
| 1 | ImageId | Label |
|---|---|---|
| 2 | 1 | 2 |
| 3 | 2 | 0 |
| 4 | 3 | 9 |
| 5 | 4 | 0 |
| 6 | 5 | 3 |
| 7 | 6 | 7 |
| 8 | 7 | 0 |
| 9 | 8 | 3 |
| 10 | 9 | 0 |
| 11 | 10 | 3 |
| 12 | 11 | 5 |
| 13 | 12 | 7 |
| 14 | 13 | 4 |
| 15 | 14 | 0 |
| 16 | 15 | 4 |
| 17 | 16 | 3 |
| 18 | 17 | 3 |
| 19 | 18 | 1 |
| 20 | 19 | 9 |
| 21 | 20 | 0 |
| 22 | 21 | 9 |
| 23 | 22 | 1 |

| | | |
|---|---|---|
| 3832 | 3831 | 3 |
| 3833 | 3832 | 6 |
| 3834 | 3833 | 5 |
| 3835 | 3834 | 2 |
| 3836 | 3835 | 7 |
| 3837 | 3836 | 7 |
| 3838 | 3837 | 9 |
| 3839 | 3838 | 8 |
| 3840 | 3839 | 5 |
| 3841 | 3840 | 7 |
| 3842 | 3841 | 1 |
| 3843 | 3842 | 7 |
| 3844 | 3843 | 4 |
| 3845 | 3844 | 3 |
| 3846 | 3845 | 7 |
| 3847 | 3846 | 4 |
| 3848 | 3847 | 4 |
| 3849 | 3848 | 3 |
| 3850 | 3849 | 2 |
| 3851 | 3850 | 0 |
| 3852 | 3851 | 0 |
| 3853 | 3852 | 0 |
| 3854 | 3853 | 3 |
| 3855 | 3854 | 1 |

Figure 21: Output

### 3.14 Other Algorithm and Accuracy:

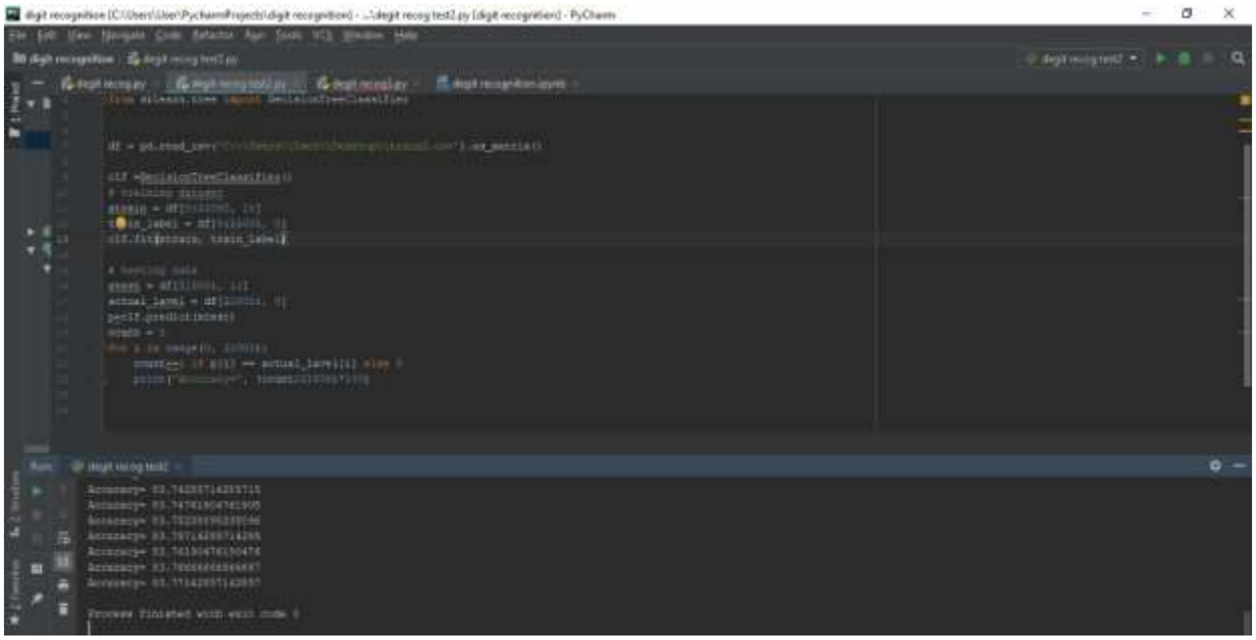
**Neural Network:**



```
In [18]: model.fit(x_train,y_train,epochs=5)

Train on 60000 samples
Epoch 1/5
60000/60000 [==============================] - 4s [illegible]us/sample - loss: 0.0527 - accuracy: 0.9834
Epoch 2/5
60000/60000 [==============================] - 4s [illegible]us/sample - loss: 0.0396 - accuracy: 0.9872
Epoch 3/5
60000/60000 [==============================] - 4s [illegible]us/sample - loss: [illegible] - accuracy: [illegible]
Epoch 4/5
60000/60000 [==============================] - 4s [illegible]us/sample - loss: 0.0268 - accuracy: 0.9916
Epoch 5/5
60000/60000 [==============================] - 4s [illegible]us/sample - loss: 0.0209 - accuracy: 0.9927
Out[18]: <tensorflow.python.keras.callbacks.History at [illegible]>

In [19]: model.save('epic_num_reader.model')
INFO:tensorflow:Assets written to: epic_num_reader.model\assets

In [19]: new_model=tf.keras.models.load_model('epic_num_reader.model')

In [20]: predictions = new_model.predict([x_test])
```

Figure 22: Accuracy By Neural Network

**Decision Tree:**



```
from sklearn.tree import DecisionTreeClassifier


df = pd.read_csv("[illegible]").as_matrix()

clf =DecisionTreeClassifier()
# training dataset
xtrain = df[0:21000, 1:]
train_label = df[0:21000, 0]
clf.fit(xtrain, train_label)

# testing data
xtest = df[21000:, 1:]
actual_label = df[21000:, 0]
p=clf.predict(xtest)
count = 0
for i in range(0, 21000):
    count+=1 if p[i] == actual_label[i] else 0
    print("Accuracy=", (count/21000)*100)
```

```
Accuracy= 83.74285714285715
Accuracy= 83.74761904761905
Accuracy= 83.75238095238096
Accuracy= 83.75714285714285
Accuracy= 83.76190476190476
Accuracy= 83.76666666666667
Accuracy= 83.77142857142857

Process finished with exit code 0
```

Figure 23: Decision tree

**Difference:**

| **Algorithm** | **Accuracy** | **Accuracy** | **Accuracy** |
|---|---|---|---|
| **CNN** | **0.9628** | **0.9693** | **0.9884** |
| Decision tree | 0.8373 | 0.8375 | 0.8377 |
| Neural network | 0.9298 | 0.97 | 0.98 |

## 4 References:

1. https://ieeexplore.ieee.org/document/8286426
2. https://machinelearningmastery.com/handwritten-digit-recognition-using-convolutional-neural-networks-python-keras/
3. https://www.opentechguides.com/how-to/article/dataanalytics/179/jupyter-notebook-pandas.html
4. https://en.wikipedia.org/wiki/Matplotlib
5. https://towardsdatascience.com/building-a-convolutional-neural-network-cnn-in-keras-329fbbadc5f5
6. https://machinelearningmastery.com/display-deep-learning-model-training-history-in-keras/
7. https://towardsdatascience.com/a-simple-2d-cnn-for-mnist-digit-recognition-a998dbc1e79a
8. https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/
9. https://towardsdatascience.com/image-classification-in-10-minutes-with-mnist-dataset-54c35b77a38d