Assignment No. 3

Problem Statement:

Build the Image classification model by dividing the model into following 4 stages:

1. Loading and preprocessing the image data
2. Defining the model's architecture
3. Training the model
4. Estimating the model's performance

Objective:

1. To be able to apply deep learning algorithms to solve problems of moderate complexity
2. Understand how a model is trained and evaluated.
3. Classifying images from the image dataset.
4. Our main goal is to train a neural network (using Keras) to obtain > 90% accuracy on image dataset..
5. To apply the algorithms to a real-world problem, optimize the models learned and report on the expected accuracy that can be achieved by applying the models

Outcomes:

At the end of the assignment the students should able-

1. Learn and Use various Deep Learning tools and packages.
2. Build and train a deep Neural Network models for use in various applications.
3. Apply Deep Learning techniques like CNN, RNN Auto encoders to solve real word Problems.
4. Evaluate the performance of the model build using Deep Learning.

Solution Expected

Implement and train a Convulutional neural network (CNN) on an hand-written digits image dataset called MNIST and improve model generalisation by achieving increased accuracy and decreased loss

where model gains good confidence with the prediction.

Methodology to be used

- Deep Learning
- Convulutional Neural Network

Infrastructure: Desktop/ laptop system with Linux /Ubuntu 16.04 or higher (64-bit)/ Windows OS/Mac OS

Software used: LINUX/ Windows OS/ Virtual Machine/ IOS, Anaconda distribution, Jupyter notebook, python 3.9.12

Theory:

Deep Learning has been proved that its a very powerful tool due to its ability to handle huge amounts of data. The use of hidden layers exceeds traditional techniques, especially for pattern recognition. One of the most popular Deep Neural Networks is Convolutional Neural Networks (CNN).

Convolutional Neural Networks (CNNs)

A convolutional neural network (CNN) is a type of Artificial Neural Network (ANN) used in image recognition and processing which is specially designed for processing data (pixels). The goal of a CNN is to learn higher-order features in the data via convolutions. They are well suited to object recognition with images and consistently top image classification competitions. They can identify faces, individuals, street signs, platypuses, and many other aspects of visual data. CNNs overlap with text analysis via optical character recognition, but they are also useful when analyzing words6 as discrete textual units. They're also good at analyzing sound. The efficacy of CNNs in image recognition is one of the main reasons why the world recognizes the power of deep learning. CNNs are good at building position and (somewhat) rotation invariant features from raw image data. CNNs are powering major advances in machine vision, which has obvious applications for self-driving cars, robotics, drones, and treatments for the visually impaired. The structure of image data allows us to change the architecture of a neural network in a way that we can take advantage of this structure. With CNNs, we can arrange the neurons in a three-dimensional structure for which we have the following:

1. Width
2. Height
3. Depth

These attributes of the input match up to an image structure for which we have:

1. Image width in pixels
2. Image height in pixels
3. RGB channels as the depth

We can consider this structure to be a three-dimensional volume of neurons. A significant aspect to how CNNs evolved from previous feed-forward variants is how they achieved computational efficiency with new layer types.

CNN Architecture Overview

CNNs transform the input data from the input layer through all connected layers into a set of class scores given by the output layer. There are many variations of the CNN architecture, but they are based on the pattern of layers, as demonstrated in Figure 1 (below).

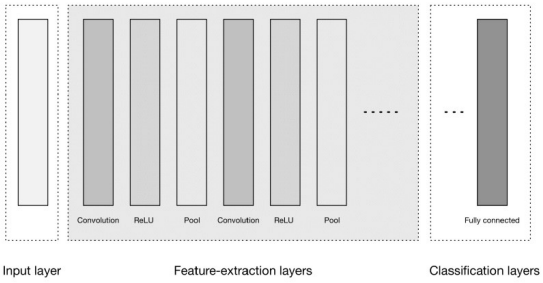


Figure 1. High-level general CNN architecture

Figure 1 depicts three major groups:

1. Input layer

- An image matrix (volume) of dimension **(h x w x d)**
- A filter **(f_h x f_w x d)**
- Outputs a volume dimension **(h - f_h + 1) x (w - f_w + 1) x 1**

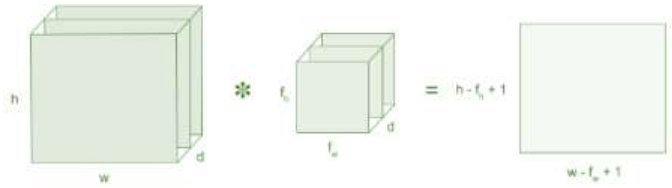


Figure 2: Image matrix multiplies kernel or filter matrix

The input layer accepts three-dimensional input generally in the form spatially of the size (width × height) of the image and has a depth representing the color channels (generally three for RGB color channels) as shown in fig 2 above.

2. Feature-extraction (learning) layers

The feature-extraction layers have a general repeating pattern of the sequence as shown in fig 1 above of -:

A. Convolution layer

We express the Rectified Linear Unit (ReLU) activation function as a layer in the diagram in fig 1. convolutional layers transform the input data by using a patch of locally connecting neurons from the previous layer

B. Pooling layer

These layers find a number of features in the images and progressively construct higher-order features. This corresponds directly to the ongoing theme in deep learning by which features are automatically learned as opposed to traditionally hand engineered.

3. Classification layers

Finally we have the classification layers in which we have one or more fully connected layers to take the higher-order features and produce class probabilities or scores. These layers are fully connected to all of the neurons in the previous layer, as their name implies. The output of these layers produces typically a two dimensional output of the dimensions [b × N], where b is the number of examples in the mini-batch and N is the number of classes we're interested in scoring.

Multilayer neural networks vs CNN

In traditional multilayer neural networks, the layers are fully connected and every neuron in a layer is connected to every neuron in the next layer whereas The neurons in the layers of a CNN are arranged in three dimensions to match the input volumes. Here, depth means the third dimension of the activation volume, not the number of layers, as in a multilayer neural network.

Evolution of the connections between layers

Another change is how we connect layers in a convolutional architecture. Neurons in a layer are connected to only a small region of neurons in the layer

before it. CNNs retain a layer-oriented architecture, as in traditional multilayer networks, but have different types of layers. Each layer transforms the 3D input volume from the previous layer into a 3D output volume of neuron activations with some differentiable function that might or might not have parameters, as demonstrated in Figure 1.

Input Layers

Input layers are where we load and store the raw input data of the image for processing in the network. This input data specifies the width, height, and number of channels. Typically, the number of channels is three, for the RGB values for each pixel.

Convolutional Layers

Convolutional layers are considered the core building blocks of CNN architectures. As Figure 2 illustrates, convolutional layers transform the input data by using a patch of locally connecting neurons from the previous layer. The layer will compute a dot product between the region of the neurons in the input layer and the weights to which they are locally connected in the output layer.

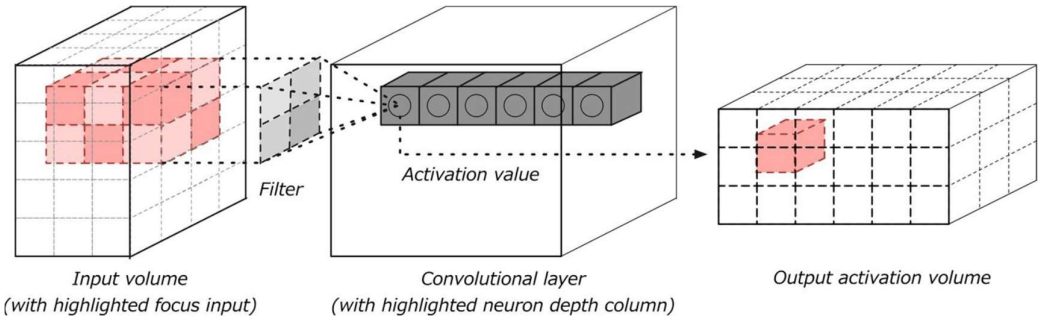


Figure 2. Convolution layer with input and output volumes

The resulting output generally has the same spatial dimensions (or smaller spatial dimensions) but sometimes increases the number of elements in the third dimension of the output (depth dimension).

TensorFlow is an open-source platform for machine learning. Keras is the high-level application programming interface (API) of TensorFlow. Using Keras, we can rapidly develop a prototype system and test it out. This is the first in a three-part series on using TensorFlow for supervised classification tasks.

STEPS: To implement Convolutional Neural Network for image classification

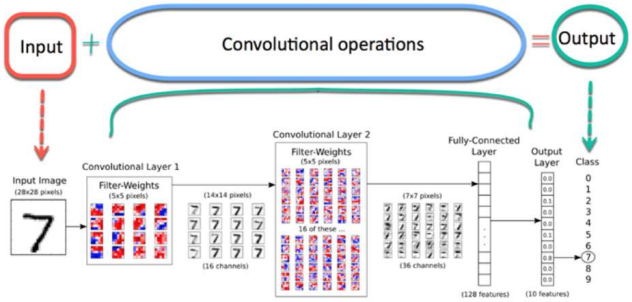


Figure 2. Image Classification model using CNN with python

CNN:

Now imagine there is an image of a bird, and you want to identify it whether it is really a bird or something other. The first thing you should do is feed the pixels of the image in the form of arrays to the input layer of the neural network (MLP networks used to classify such things). The hidden layers carry Feature Extraction by performing various calculations and operations. There are multiple hidden layers like the convolution, the ReLU, and the pooling layer that performs feature extraction from your image. So finally, there is a fully connected layer that you can see which identifies the exact object in the image. You can understand very easily from the following figure:

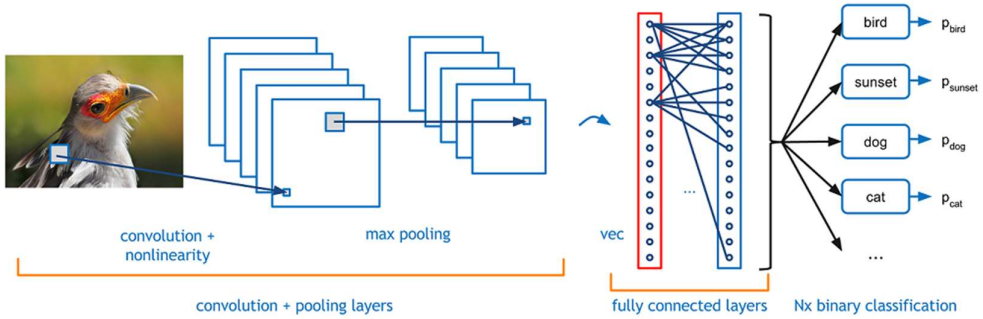


Figure 2. Convolution and pooling layers in CNN

Convolution:-

Convolution Operation involves matrix arithmetic operations and every image is represented in the form of an array of values(pixels).

Let us understand example:

a = [2,5,8,4,7,9]

b = [1,2,3]

In Convolution Operation, the arrays are multiplied one by one element-wise, and the product is grouped or summed to create a new array that represents **a*b**.

The first three elements of matrix **a** are now multiplied by the elements of matrix **b**. The product is summed to get the result and stored in a new array of **a*b.**

This process remains continuous until the operation gets completed.

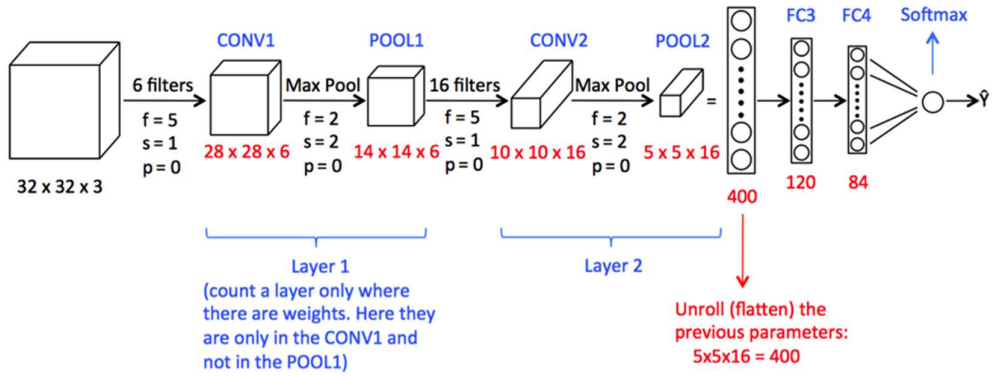


Figure 2. Sequence of Convolution and pooling layers in CNN

Pooling:

After the convolution, there is another operation called pooling. So, in the chain, convolution and pooling are applied sequentially on the data in the interest of extracting some features from the data. After the sequential convolutional and pooling layers, the data is flattened into a feed-forward neural network which is also called a Multi-Layer Perceptron.

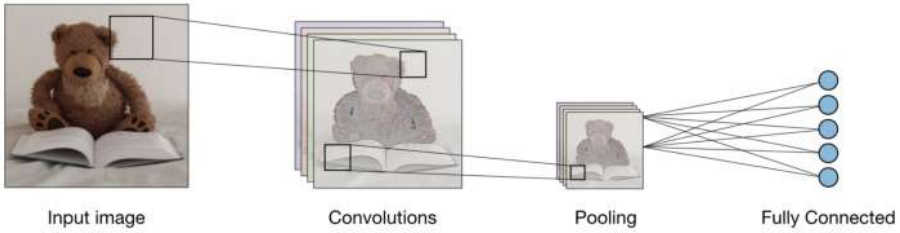


Fig: Data flattening into a feed-forward neural network

Thus, we have seen steps that are important for building CNN model.

Code Snippets

#Importing Libraries

```
import numpy as np
import pandas as pd
import random
import tensorflow as tf
import matplotlib.pyplot as plt

from sklearn.metrics import accuracy_score

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Conv2D, Dense,
MaxPooling2D
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import mnist

# Loading and preprocessing the image data
```

Loading the MNIST dataset is very simple using Keras. The dataset is so popular, that you can use access it through datasets.mnist and use the load_data() function to get a train and a test set.

The dataset contains 28x28 images showing handwritten digits.

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

If you print the shape the any of the sets you will see more information about the samples.

For example, printing the shape of the train set will get you (60000, 28, 28):

60000: This is the number of samples in the set.
28: This is the height of each image.
28: This is the width of each image.

So we have 60,000 28x28 images.

print(X_train.shape)
(60000, 28, 28)
Anytime you are using a neural network, you should pay special attention to the range of the input values you will be feeding it. In our case, each image is a matrix of 28x28 pixels.

Let's print the range of these values to understand what's the scale we are working with:

X_train[0].min(), X_train[0].max()
(0, 255)

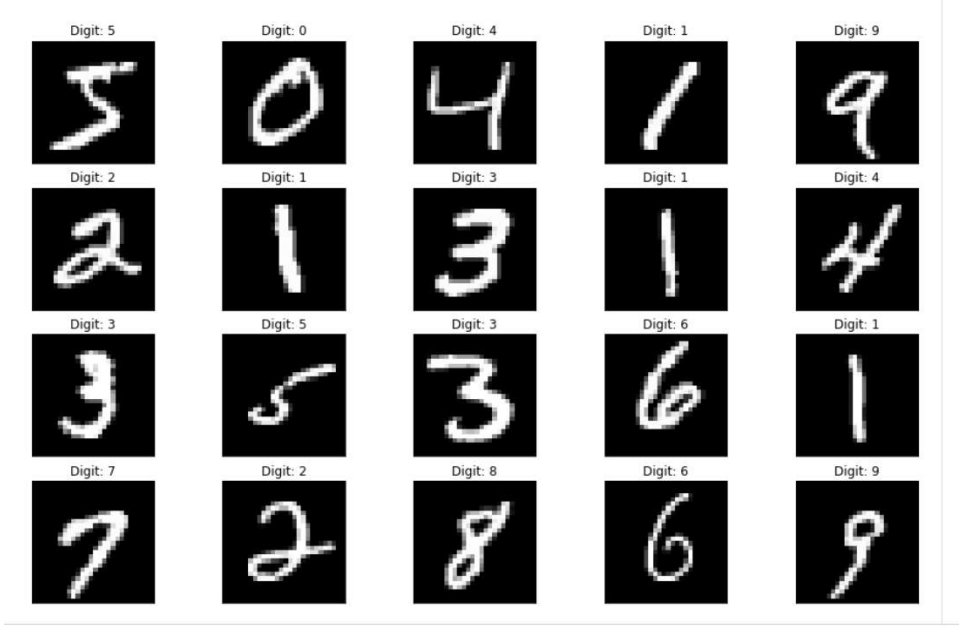
The pixel values in our images are between 0 (black) to 255 (white).

Neural networks have a much easier time converging when they work with values that don't vary a lot in scale. It's a common practice to scale every input to fit a small range like 0 to 1 or -1 to 1.
We should do that here, and scale our pixels to a range that goes from 0 (black) to 1 (white).
Here is the formula to scale a value: scaled_value = (original_value – min) / (max – min). In our case, the minimum value is 0 and the maximum value is 255.

Let's use this to scale our train and test sets (notice that you can get rid of the 0.0 in the formula below but I'm leaving them there for clarity purposes):
X_train = (X_train - 0.0) / (255.0 - 0.0)
X_test = (X_test - 0.0) / (255.0 - 0.0)
X_train[0].min(), X_train[0].max()
(0.0, 1.0)

We can now plot the first 20 images on the train set:

```
def plot_digit(image, digit, plt, i):
    plt.subplot(4, 5, i + 1)
    plt.imshow(image, cmap=plt.get_cmap('gray'))
    plt.title(f"Digit: {digit}")
    plt.xticks([])
    plt.yticks([])
plt.figure(figsize=(16, 10))
for i in range(20):
    plot_digit(X_train[i], y_train[i], plt, i)
plt.show()
```



In Computer Vision, we usually use 4 dimensions to represent a set of images:

The total number of images (we call this "batch size")

The width of each image

The height of each image

The number of channels of each image

As you saw before, our train set has 3 dimensions only; we are missing the number of channels. We need to transform our data by adding that fourth dimension. Since these images are grayscale, that fourth dimension will be 1.

We can use numpy's reshape() function to reshape all of the data by adding

that extra dimension.

X_train = X_train.reshape((X_train.shape + (1,)))

X_test = X_test.reshape((X_test.shape + (1,)))

Finally, let's take a look at the format of our target values (y_train). Let's print the first 20 samples in our train set:

y_train[0:20]

array([5, 0, 4, 1, 9, 2, 1, 3, 1, 4, 3, 5, 3, 6, 1, 7, 2, 8, 6, 9],

dtype=uint8)

We are representing the target digits using integers (the digit 5 is represented with value 5, the digit 0 with value 0, etc.) This is important because it determines which loss function we should use to optimize our neural network.

We have two options:

Use integers for our target values (as they are now), and make sure we use the "Sparse Categorical Cross-Entropy" loss function.

One-hot encode the targets and use the "Categorical Cross-Entropy" loss function.

The easiest solution is to leave the targets as they are, so let's move on to creating the model.

Defining the model's architecture

There are several ways to create a model in Keras. In this example, we are going to use Kera's Sequential API because it's very simple to use.

Let's break down the definition of model below step by step:

First, we are going to define the first hidden layer of our network: A convolutional layer with 32 filters and a 3x3 kernel. This layer will use a ReLU activation function. The goal of this layer is to generate 32 different representations of an image, each one of 26x26. The 3x3 kernel will discard a pixel on each side of the original image and that's way we get 26x26 squares instead of 28x28.

Notice how we also need to define the input shape of the network as part of that first layer. Remember that our images are 28x28 with a single color channel, so that leads to the (28, 28, 1) shape.

Right after that first layer, we are going to do a 2x2 max pooling to downsample the amount of information generated by the convolutional layer. This operation will half the size of the filters. Remember we start with 32 filters of 26x26, so after this operation will have 32 filters of 13x13.
We then take the (13, 13, 32) vector and flatten it to a (5408,) vector. Notice that 13 x 13 x 32 = 5408.
Finally, we add a couple more fully-connected layers (also called Dense layers.) Notice how the output layer has size 10 (one for each of our possible digit values) and a softmax activation. Softmax ensures we get a probability distribution indicating the most likely digit in the image.

```
model = Sequential([
   Conv2D(32, (3, 3), activation="relu", input_shape=(28, 28, 1)),
   MaxPooling2D((2, 2)),
   Flatten(),
   Dense(100, activation="relu"),
   Dense(10, activation="softmax")
])
```

We now have our model. The next step is to define how we want to train it: Let's use an SGD optimizer (Stochastic Gradient Descent) with 0.01 as the learning rate.
As we discussed before, we need to use the sparse_categorical_crossentropy loss because our target values are represented as integers.
And we are going to compute accuracy of our model as we train it.
Notice in the summary of the model the shape of the vectors as they move through the layers we defined. They should look familiar after reading the explanation of our model above.

```
optimizer = SGD(learning rate=0.01, momentum=0.9)
model. Compile(
   optimizer=optimizer,
   loss="sparse_categorical_crossentropy",
   metrics=["accuracy"]
)
```

model.summary()

Model: "sequential_1"

_____
Layer (type) Output Shape Param #
===
conv2d_1 (Conv2D) (None, 26, 26, 32) 320
_____
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 32) 0
_____
flatten_1 (Flatten) (None, 5408) 0
_____
dense_2 (Dense) (None, 100) 540900
_____
dense_3 (Dense) (None, 10) 1010
===
Total params: 542,230
Trainable params: 542,230
Non-trainable params: 0
_____

#Training and testing the model

At this point we are ready to fit our model on the train set.

For this example, we are going to run batches of 32 samples through our model for 10 iterations (epochs.) This should be enough to get a model with good predictive capabilities.

Note: This network is fairly shallow so it shouldn't take a long time to train it on a CPU. If you have access to a GPU it should be much faster.

model.fit(X_train, y_train, epochs=10, batch_size=32)
Epoch 1/10
1875/1875 [==============================] - 7s 3ms/step - loss: 0.2412 - accuracy: 0.9252
Epoch 2/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0847 - accuracy: 0.9748
Epoch 3/10

1875/1875 [==========] - 6s 3ms/step - loss: 0.0535 - accuracy: 0.9831

Epoch 4/10

1875/1875 [==========] - 6s 3ms/step - loss: 0.0383 - accuracy: 0.9886

Epoch 5/10

1875/1875 [==========] - 6s 3ms/step - loss: 0.0288 - accuracy: 0.9909

Epoch 6/10

1875/1875 [==========] - 6s 3ms/step - loss: 0.0212 - accuracy: 0.9936

Epoch 7/10

1875/1875 [==========] - 6s 3ms/step - loss: 0.0163 - accuracy: 0.9951

Epoch 8/10

1875/1875 [==========] - 6s 3ms/step - loss: 0.0129 - accuracy: 0.9961

Epoch 9/10

1875/1875 [==========] - 6s 3ms/step - loss: 0.0092 - accuracy: 0.9973

Epoch 10/10

1875/1875 [==========] - 6s 3ms/step - loss: 0.0072 - accuracy: 0.9983

<keras.callbacks.History at 0x7ff9e24c1a20>

At this point you should have a model that scored above 99% accuracy on the train set.

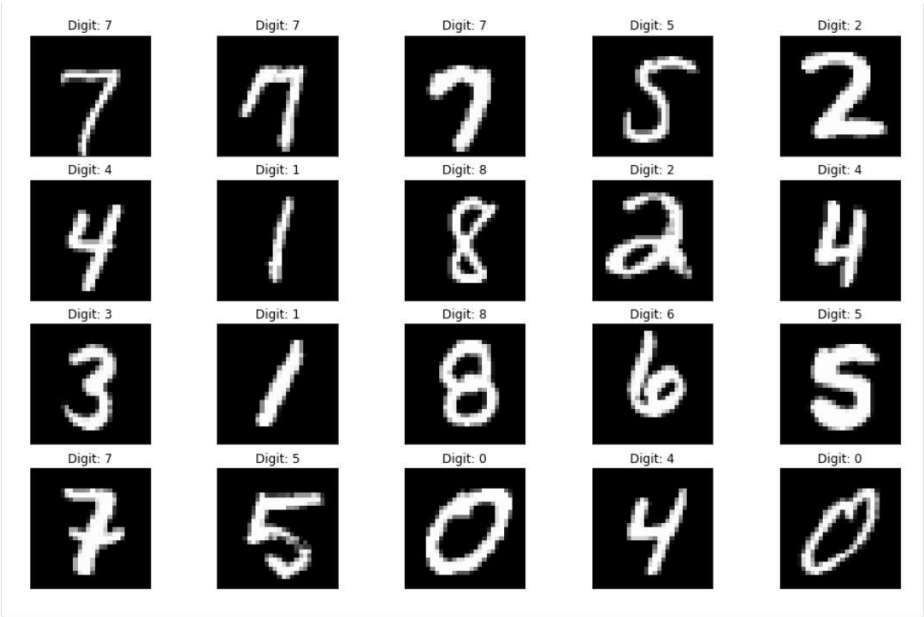
Now it's time to test it with a few of the images that we set aside on our test set. Let's run 20 random images through the model and display them together with the predicted digit:

```
plt.figure(figsize=(16, 10))
for i in range(20):
    image = random.choice(X_test).squeeze()
    digit = np.argmax(model.predict(image.reshape((1, 28, 28, 1)))[0],
```

axis=-1)

plot_digit(image, digit, plt, i)

plt.show()



The results look pretty good!

To get a much better idea about the quality of the predictions, we can run the entire test set (10,000 images) through the model and compute the final accuracy. To do this we can use the accuracy_score() function from SciKit-Learn passing a couple of arguments:

True values: The correct digit expected for each image. These are the values we have stored in y_test.

Predicted values: The predictions that our model made. These are the results of our model.

The final accuracy will be the value printed after running the cell.

predictions = np.argmax(model.predict(X_test), axis=-1)

accuracy_score(y_test, predictions)

0.9855

 # Estimating the model's performance

Now the trained model needs to be evaluated in terms of performance.

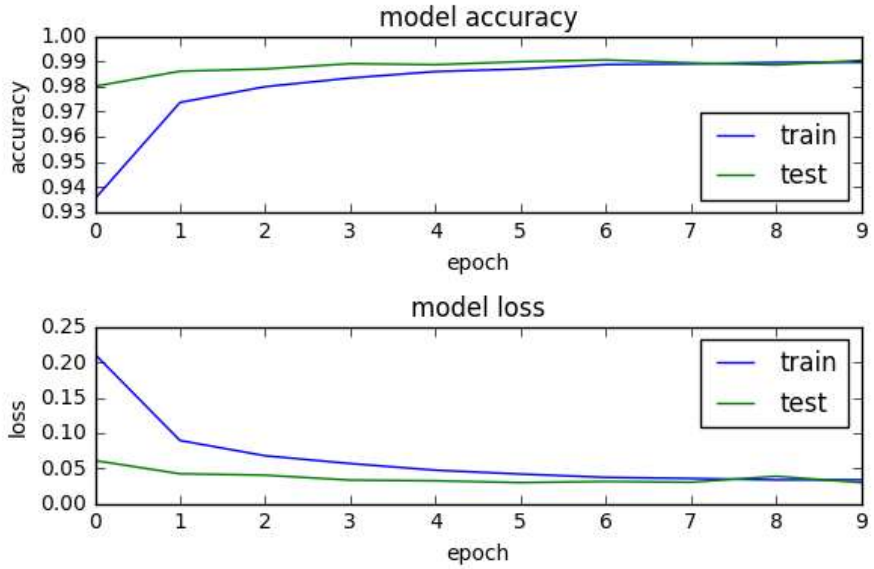
score = model.evaluate(X_test, y_test, verbose=0)

```
print('Test loss:', score[0]) #Test loss: 0.0296396646054
print('Test accuracy:', score[1]) #Test accuracy: 0.9904
```

Test accuracy 99%+ implies the model is trained well for prediction. If we visualize the whole training log, then with more number of epochs the loss and accuracy of the model on training and testing data converged thus making the model a stable one.

```
import os
# plotting the metrics
fig = plt.figure()
plt.subplot(2,1,1)
plt.plot(model_log.history['acc'])
plt.plot(model_log.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='lower right')plt.subplot(2,1,2)
plt.plot(model_log.history['loss'])
plt.plot(model_log.history['val_loss'])
```

```
.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')plt.tight_layout()fig
```



Saving the model to disk for reuse

Now, the trained model needs to be serialized. The architecture or structure of the model will be stored in a json file and the weights will be stored in hdf5 file format.

```
#Save the model
# serialize model to JSON
model_digit_json = model.to_json()
with open("model_digit.json", "w") as json_file:
    json_file.write(model_digit_json)
# serialize weights to HDF5
model.save_weights("model_digit.h5")
print("Saved model to disk")
```

.

Hence the saved model can be reused later or easily ported to other environments too

Conclusion:

Thus, we have implemented the Image classification model using CNN. With above code we can see that sufficient accuracy has been met. Throughout the epochs, our model accuracy increases and loss

decreases that is good since our model gains confidence with our prediction

This indicates the model is trained in a good way

1. The loss is decreasing and the accuracy is increasing with every epoch.
2. The test accuracy is the measure of how good the model is predicting so, it is observed that the model is well trained after 10 epochs

References:

1. https://www.analyticsvidhya.com/blog/2021/06/image-classification-using-convolutional-neural-network-with-python/
2. Josh Patterson, Adam Gibson "Deep Learning: A Practitioner's Approach", O'ReiIIy Media, 2017
3. https://deepnote.com/@svpino/MNIST-with-Convolutional-Neural-Networks-a4b3c412-b802-4185-9806-02640fbba02e
4. https://towardsdatascience.com/a-simple-2d-cnn-for-mnist-digit-recognition-a998dbc1e79a