# ICP9

## Answer 1)

**Program Description**:
A program for plotting the Loss and accuracy for both training data and validation data, by using the history object in the source code:
**Original Source Code:** https://umkc.box.com/s/10nrlk6216fncengv7qxbbw5o9vgc3hs

**Code Explanation:**
Start with setting up environment and importing necessary libraries.

**Keras**: Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow. In this Sequential model is a linear stack of layers.
**MNIST Dataset:** The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning.

```
import warnings
warnings.filterwarnings('ignore')
# Imported the necessary libraries and created our environment
# Keras is a high-level library.
# Sequential model is a linear stack of layers.
# for mnist digit classification dataset imported mnist
from keras import Sequential
from keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
from keras.layers import Dense
from keras.utils import to_categorical
```

**Step 1:**
**Data Loading and Plotting the Digit:**
Loading mnist data set:
The MNIST database contains 60,000 training images and 10,000 testing images. Half of the training set and half of the test set were taken from NIST's training dataset, while the other half of the training set and the other half of the test set were taken from NIST's testing dataset.
Total 60,000 grayscale images (28x28 pixels) of the (0-9) digits, along with a test set of 10,000 images.
Size of mnist_train_images is 60000 and mnist_test_images is 10000

```
# Loading mnist data set
# This is a dataset of 60,000 28x28 grayscale images of the (0-9) digits, along with a test set of 10,000 images.
# Size of mnist_train_images is 60000 and mnist_test_images is 10000
(train_images,train_labels),(test_images, test_labels) = mnist.load_data()
# printing the images shape.
print('Pixel Values: ', train_images.shape[1:])
# printing the number of train_images
print('Number of train_images and pixel values: ',train_images.shape)
# printing the number of test_images
print('Number of test_images and pixel values: ',test_images.shape)
```

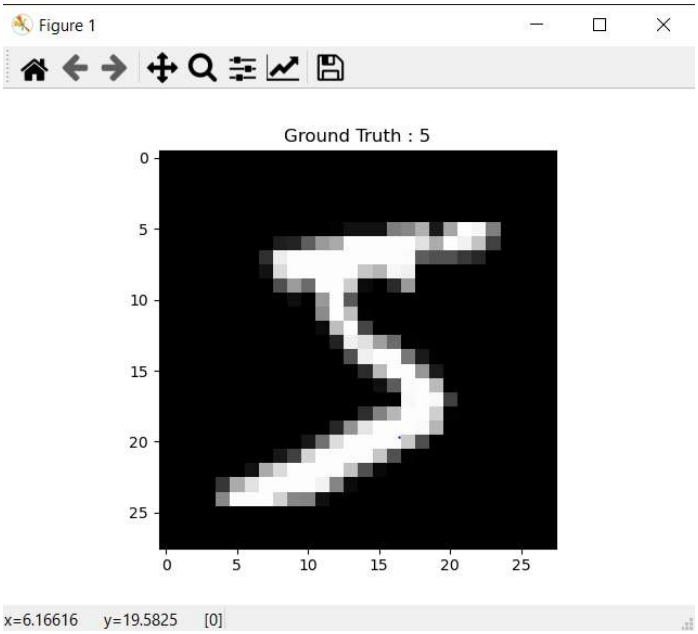
**OUTPUT:**

```
Pixel Values:  (28, 28)
Number of train_images and pixel values:  (60000, 28, 28)
Number of test_images and pixel values:  (10000, 28, 28)
```

Now visualizing an image from train_images, by using **imshow** to display the image.
**cmap:** This parameter is a colormap instance or registered colormap name used to map scalar data to colors.

```
# Visualizing a image from train_images.
# imshow to display the image
# cmap : This parameter is a colormap instance or registered colormap name used to map scalar data to colors. .
plt.imshow(train_images[0,:,:],cmap='gray')
plt.title('Ground Truth : {}'.format(train_labels[0]))
plt.show()
```

**OUTPUT:**



Above output shows the greyscale visualization of an image from train_images.

**Step 2:**
**Processing the Data:**
The number of input neuron (dimension)  here is 28*28 = 784
np.prod: return the product of array elements over a given array.

We need to convert each image of shape 28*28 to 784 dimensional which will be fed to the network as a single feature.

```
# Get the number of input neuron(dimension) here 28*28 = 784
dimData = np.prod(train_images.shape[1:])
# Printing dimdata
print('dimdata: ', dimData)
# Convert each image of shape 28*28 to 784 dimensional which will be fed to the network as a single feature.
train_data = train_images.reshape(train_images.shape[0],dimData)
test_data = test_images.reshape(test_images.shape[0],dimData)
```

**OUTPUT:**

```
dimdata:  784
```

This shows the dimdata as 784. (28*28).

Next we are making sure that the values are float that we can get decimal points after division.
Converting the data to float and scaling values between 0 and 1

```
# Making sure that the values are float so that we can get decimal points after division
#convert data to float and scale values between 0 and 1
train_data = train_data.astype('float')
test_data = test_data.astype('float')
```

**Scaling the data:**
The pixel values in images must be scaled prior to providing the images as input to a deep learning neural network model during the training or evaluation of the model.
For most image data, the pixel values are integers with values between 0 and 255.
Neural networks process inputs using small weight values, and inputs with large integer values can disrupt or slow down the learning process. As such it is good practice to normalize the pixel values so that each pixel value has a value between 0 and 1.

```
# Scale data
# Original data is uint8 (0-255). Scale it to range [0,1].
# Normalizing the RGB codes by dividing it to the max RGB value.
train_data /=255.0
test_data /=255.0
```

**Changing the labels from integer to one- hot encoding:**
**one- hot encoding:** A one hot encoding is a representation of categorical variables as binary vectors.
First thing it requires is that the categorical values be mapped to integer values.
Then, each integer value is represented as a binary vector that is all zero values except the index of the integer, which is marked with a 1.

```
#change the labels from integer to one-hot encoding
# one hot encoding is a representation of categorical variables as binary vectors.
train_labels_one_hot = to_categorical(train_labels)
test_labels_one_hot = to_categorical(test_labels)
```

**Step3:**
**Creating a Model**
We are creating sequential Neural Network with input and output layers. We are using Dense Layer which is one of the type of layer and all of its nodes/neurons are fully connected.
For defining first i.e. input layer we are mentioning 512 units are used by model. Input dimension is 784 units and activation = relu
There is another hidden layer with 512 units and again activation function = relu.

Output layer has 10 output units and activation = softmax

```
# Creating network
# Creating model....
# Sequential is a linear stack of layers.
model = Sequential()
# Adding dense layer of 512 units and activation = relu
# And input dimension is 784 units.
model.add(Dense(512, activation='relu', input_shape=(dimData,)))
# Adding dense layer of 512 unit and activation = relu
model.add(Dense(512, activation='relu'))
# output layer
# 10 output units and activation = softmax
model.add(Dense(10, activation='softmax'))
```

Let's compile our model. Here we are using **rmsprop** optimizer and **binary_crossentropy** loss function. To monitor the accuracy we are passing ['accuracy'] to metrics argument.

```
# Compile the model
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
```

Let's fit the model with train data. We are training model over 20 epochs (iteration) over all the samples of train data and one-hot labels with 256 batches of sample train data.

```
# fit the model
history = model.fit(train_data, train_labels_one_hot, batch_size=256, epochs=20, verbose=1,
                    validation_data=(test_data, test_labels_one_hot))
```

Next evaluating the result on test data and get the loss and accuracy values.

```
# Evaluating the result on test data and get the loss and accuracy values
[test_loss, test_acc] = model.evaluate(test_data, test_labels_one_hot)
print("Evaluation result on Test Data : Loss = {}, accuracy = {}".format(test_loss, test_acc))
```

**Step 4:**
**Graphical Comparison**
Lets plot  accuracy and loss values graphs to draw comparisons between Train and validation data values.

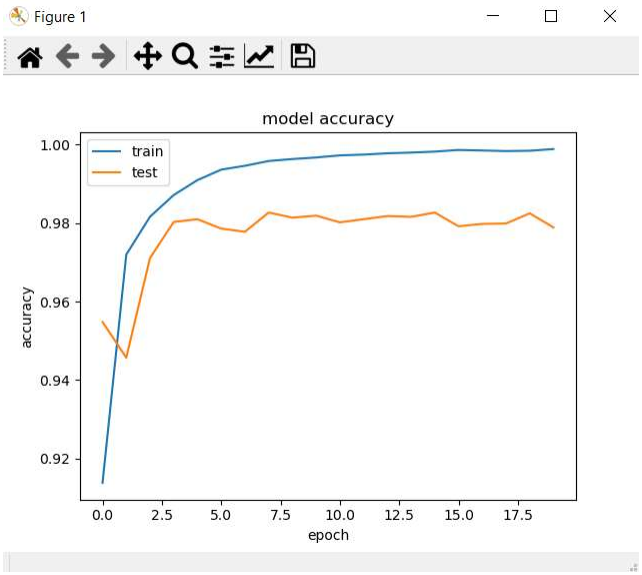
First plot History for accuracy

```
# Plotting history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
# Printing Title
plt.title('model accuracy')
# y label as accuracy
plt.ylabel('accuracy')
# x label as epoch
plt.xlabel('epoch')
# Placing a legend on 'train' and 'test'
plt.legend(['train', 'test'], loc='upper left')
# To show the graph
plt.show()
```

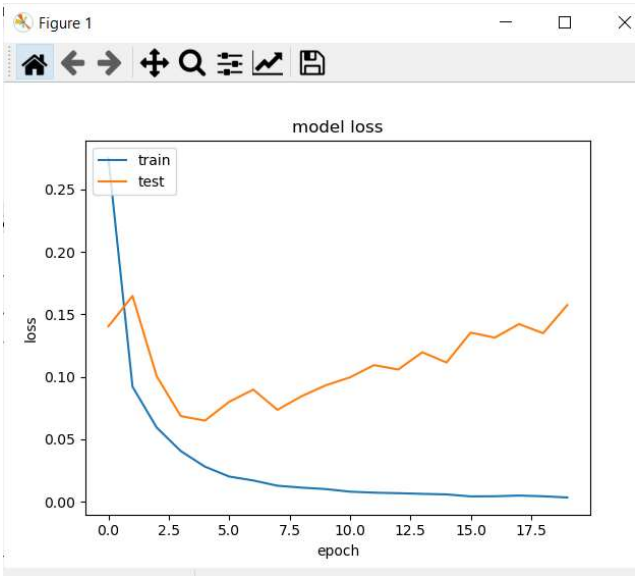
Next plot history for loss

```
# Plotting History for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
# Printing Title
plt.title('model loss')
# y label as loss
plt.ylabel('loss')
# x label as epoch
plt.xlabel('epoch')
# Placing a legend on 'train' and 'test'
plt.legend(['train', 'test'], loc='upper left')
# To show the graph
plt.show()
```

**Accuracy graph:**
Overall accuracy for Validation data is less than that for training data. After certain epochs validation accuracy seizes to increase and stay almost constant similar to what happens for training data with increase in epochs.




**Loss graph:**
Overall Loss for Validation data is more than that for training data. After certain epochs, validation Loss increases slowly and steadily, over increase in epoch. For Training data, loss stays almost constant after certain epochs.




**OUTPUT:**

```
59392/60000 [============================>.] - ETA: 0s - loss: 0.0033 - accuracy: 0.9989
60000/60000 [==============================] - 4s 66us/step - loss: 0.0034 - accuracy: 0.9989 - val_loss: 0.1574 - val_accuracy: 0.9789

   32/10000 [..............................] - ETA: 0s
  960/10000 [=>............................] - ETA: 0s
 2016/10000 [=====>........................] - ETA: 0s
 3072/10000 [========>.....................] - ETA: 0s
 4128/10000 [==========>...................] - ETA: 0s
 5088/10000 [==============>...............] - ETA: 0s
 5984/10000 [================>.............] - ETA: 0s
 7008/10000 [====================>.........] - ETA: 0s
 8064/10000 [========================>.....] - ETA: 0s
 9120/10000 [==========================>...] - ETA: 0s
10000/10000 [==============================] - 1s 51us/step
Evaluation result on Test Data : Loss = 0.15742653221499905, accuracy = 0.9789000153541565
```

Loss: 0.15742...
Accuracy: 0.978900...

## ANSWER2)

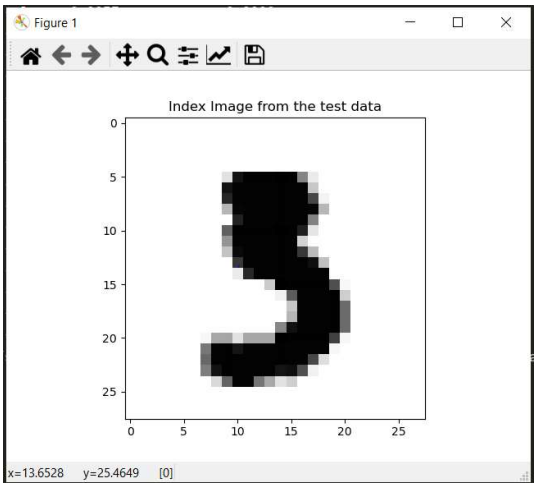
**Program Description:**
A program to plot one of the images from the test data, then doing inferencing to check the prediction of the model on that single image in the test data.

**Code Explanation:**
All code will be the same as above, apart from plotting history of accuracy and loss for training data and validation data.

Plotting one of the images from the test data.
We can select anything up to 60,000. Here picked a random image 450 from test data and then printing the Index Image from the test data.

```
# Plotting one of the images in the test data
# we can select anything up to 60,000, picked a random image 450 from test data
image_index = 450
plt.imshow(test_data[image_index].reshape(28,28),cmap='Greys')
# Printing title as Index Image from the test data.
plt.title('Index Image from the test data')
# To show the graph
plt.show()
```

**OUTPUT:**




Above output shows the 450 index_image from test data which is 3.

Next step is the model prediction on the single image (450) from the test data.
We created a flatten copy that is reshaped to (1,784). We will feed this copy into our model and then obtain the predicted data and print it out.
We are using argmax operation which finds the argument that gives the maximum value from a target function.

```
# Model prediction on the single image (450) in the test data.
# We create a flatten copy that is reshaped to (1,784).
pred = model.predict(test_data[image_index].reshape(1,784))
# We feed this copy into our model, next we obtain the predicted data and print it out.
# We are using argmax operation which that finds the argument that gives the maximum value from a target function.
print('Pridicted Data: ',pred.argmax())
# Printing the history keys.
print('history keys: ', history.history.keys())
```

**OUTPUT:**

```
 3328/10000 [========>.....................] - ETA: 0s
 4448/10000 [============>.................] - ETA: 0s
 5536/10000 [===============>..............] - ETA: 0s
 6496/10000 [==================>...........] - ETA: 0s
 7488/10000 [====================>.........] - ETA: 0s
 8576/10000 [========================>.....] - ETA: 0s
 9696/10000 [============================>.] - ETA: 0s
10000/10000 [==============================] - 0s 48us/step
Evaluation result on Test Data : Loss = 0.14922311627002377, accuracy = 0.9814000129699707
Pridicted Data:  3
history keys:  dict_keys(['val_loss', 'val_accuracy', 'loss', 'accuracy'])
```

The above output shows the **predicted image value as 3.**
**Loss = 0.14922**
**Accuracy = 0.981400**

## ANSWER 3)

**Program Description**:
A program, to check the difference after changing the: Number of hidden layer and the activation to tanh or sigmoid.

**Code Explanation:**
Source code will be almost similar as in first question except in this code we will be adding more dense layers.

**Adding two more hidden layer :**

Input dimension is 784 units.
Adding dense layer of 122 units and activation= tanh
Adding dense layer of 142 units and activation= sigmoid

Output layer:
10 output units and activation = softmax

```
#creating network
# creating model....
# Adding two more hidden layer with unit 122 and 142 and activation as tanh and sigmoid.
# Sequential is a linear stack of layers.
model = Sequential()
# Adding dense layer of 512 units and activation = relu
# And input dimension is 784 units.
model.add(Dense(512, activation='relu', input_shape=(dimData,)))
# Adding dense layer of 512 unit and activation = relu
model.add(Dense(512, activation='relu'))
# Adding dense layer 122 and activation= tanh
model.add(Dense(122, activation='tanh'))
# Adding dense layer 142 and activation= sigmoid
model.add(Dense(142, activation='sigmoid'))
# output layer
# 10 output units and activation = softmax
model.add(Dense(10, activation='softmax'))
```

**Graphical Comparison**

Let's plot  accuracy and loss values graphs to draw comparisons between Train and validation data values.

Plotting history for accuracy:

```
# Plotting history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
# Printing Title
plt.title('model accuracy')
# y label as accuracy
plt.ylabel('accuracy')
# x label as epoch
plt.xlabel('epoch')
# Placing a legend on 'train' and 'test'
plt.legend(['train', 'test'], loc='upper left')
# To show the graph
plt.show()
```

Plotting History for loss:

```
# Plotting History for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
# Printing Title
plt.title('model loss')
# y label as loss
plt.ylabel('loss')
# x label as epoch
plt.xlabel('epoch')
# Placing a legend on 'train' and 'test'
plt.legend(['train', 'test'], loc='upper left')
# To show the graph
plt.show()
```

**OUTPUT:**

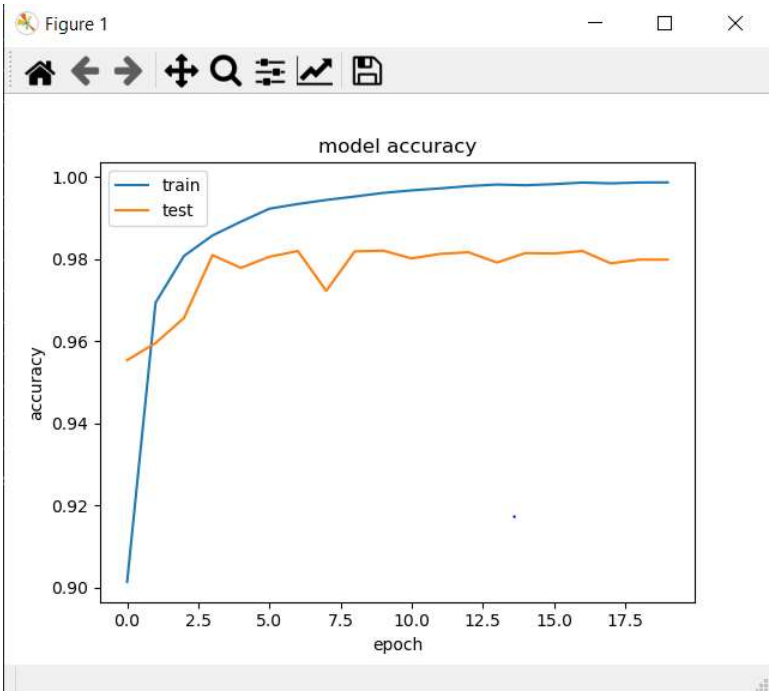
```
 4992/10000 [=============>................] - ETA: 0s
 5888/10000 [==============>...............] - ETA: 0s
 6880/10000 [===================>..........] - ETA: 0s
 7936/10000 [======================>.......] - ETA: 0s
 8992/10000 [=========================>....] - ETA: 0s
 9984/10000 [============================>.] - ETA: 0s
10000/10000 [==============================] - 1s 52us/step
Evaluation result on Test Data with Scaling : Loss = 0.10779970787606726, accuracy = 0.9812999963760376
```

After adding the 2 dense layers :
**Accuracy : 0.981299**
**Loss: 0.107799**

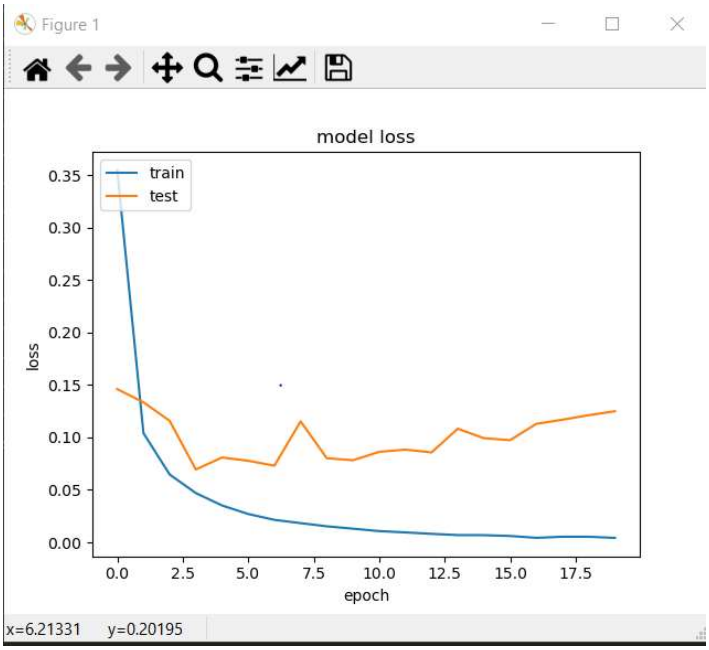
**Comparison before and after adding hidden layers in the model.**
Overall the accuracy got increased.
Before adding the new hidden layers accuracy : 0.9789000
After adding the new hidden layers accuracy : 0.981299

Overall the loss got reduced.
Before adding the new hidden layers loss : 0.1574265
After adding the new hidden layers loss: 0.107799

**Accuracy Graph:**




**Loss Graph:**

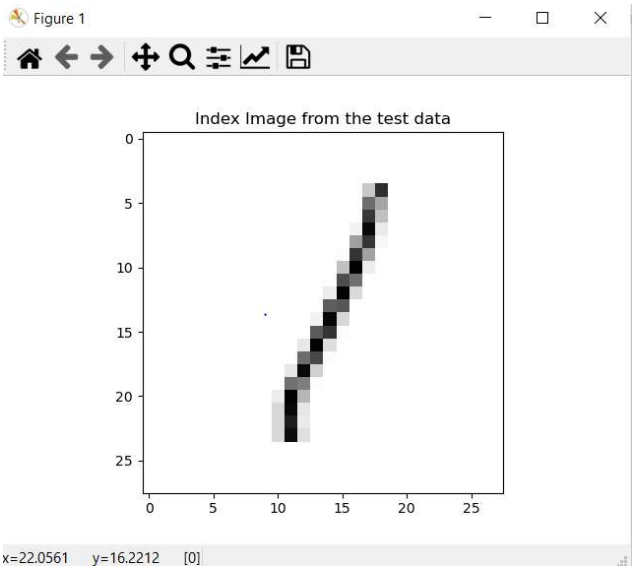


Let's run same program for different image index.
Plotting one of the image in the test data.

```
# Plotting one of the images in the test data
# we can select anything up to 60,000, picked a random image 224 from test data
image_index = 224
plt.imshow(test_data[image_index].reshape(28,28),cmap='Greys')
# Printing title as Index Image from the test data.
plt.title('Index Image from the test data')
# To show the graph
plt.show()

# Model prediction on the single image (224) in the test data.
# We create a flatten copy that is reshaped to (1,784).
pred = model.predict(test_data[image_index].reshape(1,784))
# We feed this copy into our model, next we obtain the predicted data and print it out.
# We are using argmax operation which that finds the argument that gives the maximum value from a target function.
print('Pridicted Data: ',pred.argmax())
# Printing the history keys.
print('history keys: ', history.history.keys())
```

**OUTPUT:**




Above output shows the 224 index_image from test data.

**OUTPUT:**
The below output shows the predicted image value as 1.

```
 8992/10000 [=========================>....] - ETA: 0s
 9984/10000 [==========================>.] - ETA: 0s
10000/10000 [==============================] - 1s 52us/step
Evaluation result on Test Data with Scaling : Loss = 0.10779970787606726, accuracy = 0.9812999963760376
Pridicted Data:  1
history keys:  dict_keys(['val_loss', 'val_accuracy', 'loss', 'accuracy'])
```

## ANSWER 4)

**Program Description:** A program, to check the accuracy and loss changes without scaling the images.

**Code Explanation:** Source code will be almost similar as in third question except in this code we checking the accuracy change without scaling the data.

Commented the part which was scaling the data.

```
# Commenting the lines which used to scale the data.
# Scale data
# Scale it to range [0,1].
# Normalizing the RGB codes by dividing it to the max RGB value.
#train_data /=255.0
#test_data /=255.0
```

**Graphical Comparison**

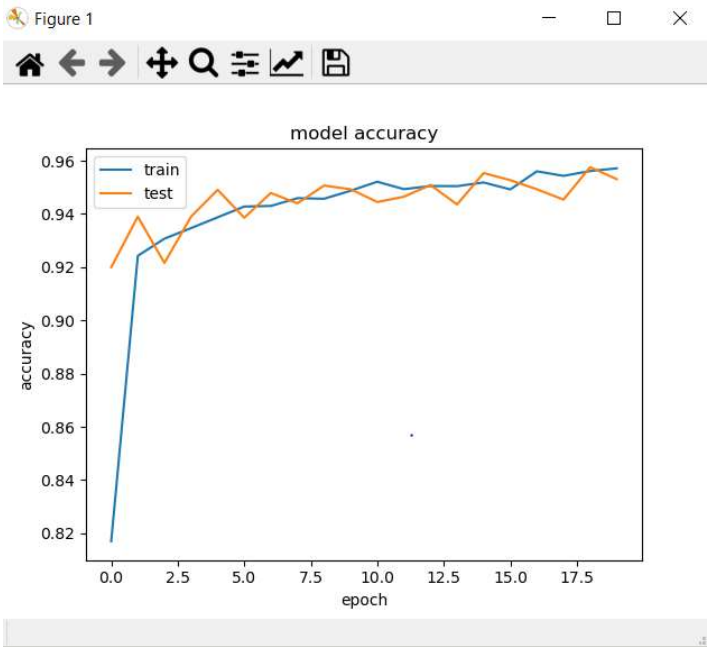
Let's plot accuracy and loss values graphs to draw comparisons between Train and validation data values.

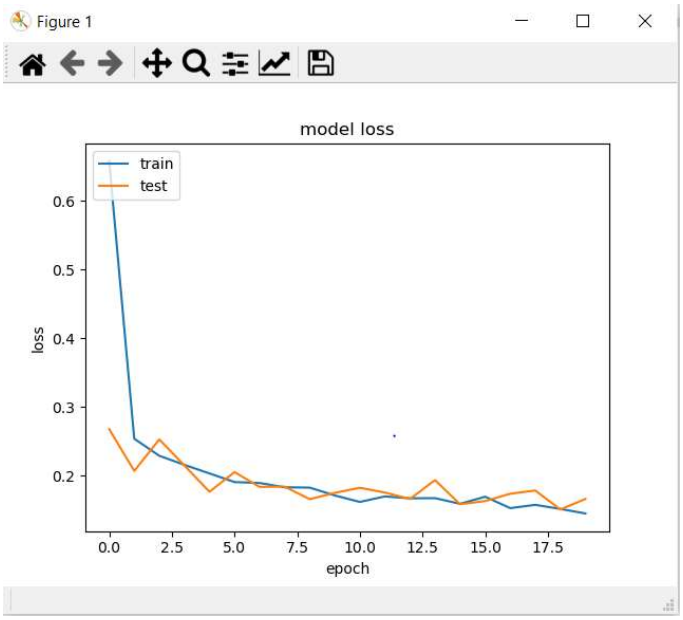
Plotting History for Accuracy comparison between Training and Validation data.

```
# Plotting history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
# Printing Title
plt.title('model accuracy')
# y label as accuracy
plt.ylabel('accuracy')
# x label as epoch
plt.xlabel('epoch')
# Placing a legend on 'train' and 'test'
plt.legend(['train', 'test'], loc='upper left')
# To show the graph
plt.show()
```

Plotting History for Loss comparison between Training and Validation data.

```
# Plotting History for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
# Printing Title
plt.title('model loss')
# y label as loss
plt.ylabel('loss')
# x label as epoch
plt.xlabel('epoch')
# Placing a legend on 'train' and 'test'
plt.legend(['train', 'test'], loc='upper left')
# To show the graph
plt.show()
```



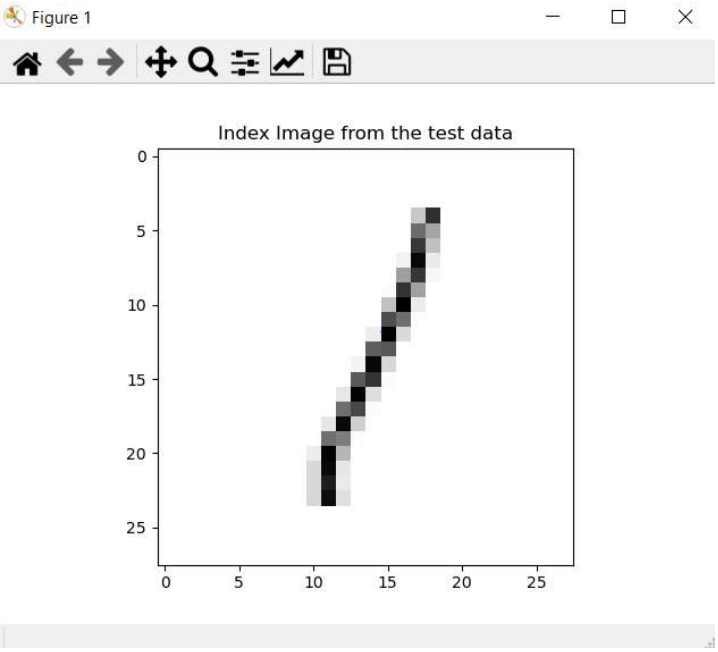



Plotting one of the image from the test data.

```
# Plotting one of the images in the test data
# we can select anything up to 60,000, picked a random image 224 from test data
image_index = 224
plt.imshow(test_data[image_index].reshape(28,28),cmap='Greys')
# Printing title as Index Image from the test data.
plt.title('Index Image from the test data')
# To show the graph
plt.show()

# Model prediction on the single image (224) in the test data.
# We create a flatten copy that is reshaped to (1,784).
pred = model.predict(test_data[image_index].reshape(1,784))
# We feed this copy into our model, next we obtain the predicted data and print it out.
# We are using argmax operation which that finds the argument that gives the maximum value from a target function.
print('Pridicted Data: ',pred.argmax())
# Printing the history keys.
print('history keys: ', history.history.keys())
```

**OUTPUT:**




Above output shows the 224 index_image from test data.

The below output shows the predicted image value as 1.

```
 5824/10000 [================>.............] - ETA: 0s
 6656/10000 [=================>............] - ETA: 0s
 7680/10000 [=======================>......] - ETA: 0s
 8704/10000 [==========================>....] - ETA: 0s
 9664/10000 [============================>..] - ETA: 0s
10000/10000 [==============================] - 1s 53us/step
Evaluation result on Test Data : Loss = 0.16613920537233354, accuracy = 0.9531000256538391
Pridicted Data:  1
history keys:  dict_keys(['val_loss', 'val_accuracy', 'loss', 'accuracy'])
```

**Comparison Conclusion:**

Overall the accuracy got increased with Scaled data .
Scaled data Accuracy : 0.98129999
Accuracy without Scaled data : 0.9531000..

Overall the loss got reduced with Scaled data.
Scaled data Loss : 0.10779970
Loss without Scaled data: 0.166139..