```
from keras.datasets import mnist
from keras import models
from keras import layers
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
```

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

```
print(train_images.shape)
test_images.shape
```

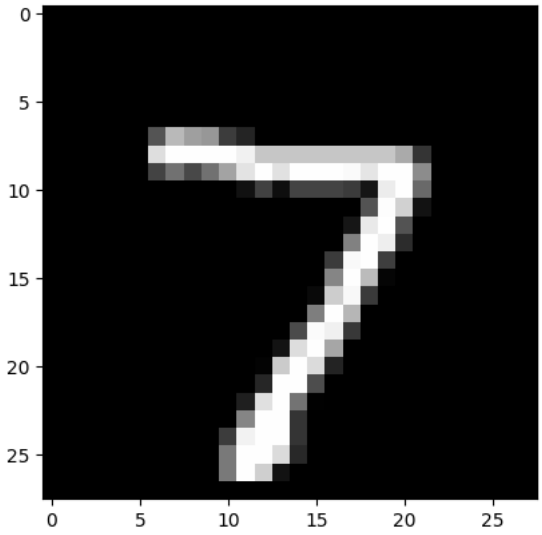
```
(60000, 28, 28)
(10000, 28, 28)
```

```
print("train_images_dim=",train_images.ndim)
print('train_labels_dim=',train_labels.ndim)
```

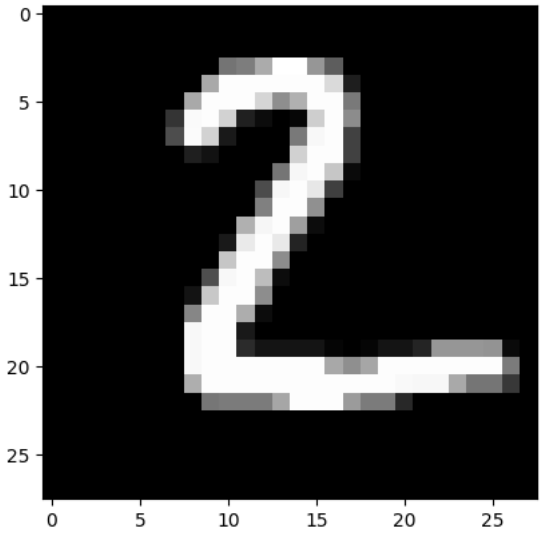
```
train_images_dim= 3
train_labels_dim= 1
```

```
for i in range(5):
    print("location of image=",i)   #if you want to see your image in image form
    img = test_images[i].reshape((28,28))
    plt.imshow(img, cmap="gray") # use jet in place of gray
    plt.show()
```

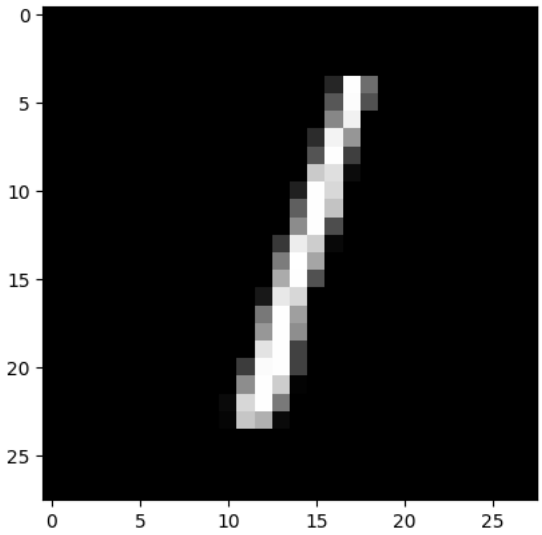
location of image= 0



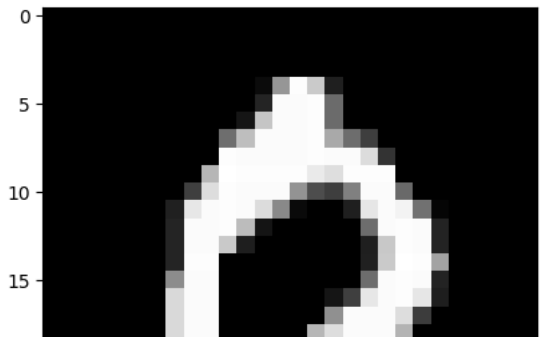
location of image= 1

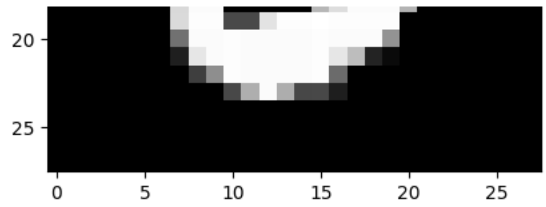


location of image= 2

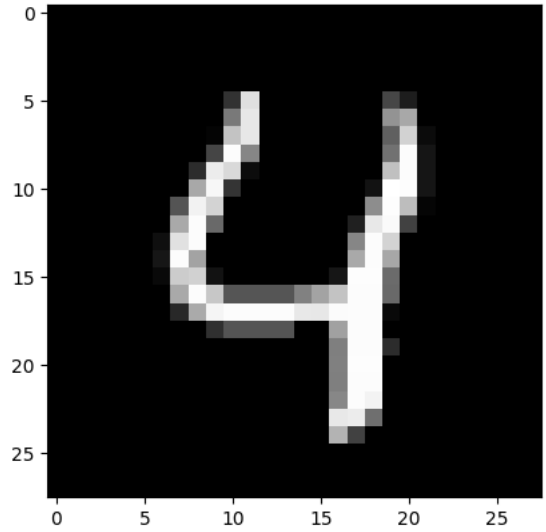


location of image= 3






```
location of image= 4
```



Before training, we'll preprocess the data by reshaping it into the shape the network expects and scaling it so that all values are in the [0, 1] interval. Previously, our training images, for instance, were stored in an array of shape (60000, 28, 28) of type uint8 with values in the [0, 255] interval. We transform it into a float32 array of shape (60000, 28 * 28) with values between 0 and 1.

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images / 255.0  #re-scale the image data to values between (0.0,1.0]

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images / 255.0
```

```
train_images[1] #now you can check the image in location 1
```

```
array([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.        , 0.2       , 0.62352941, 0.99215686,
       0.62352941, 0.19607843, 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.18823529,
       0.93333333, 0.98823529, 0.98823529, 0.98823529, 0.92941176,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.21176471, 0.89019608, 0.99215686, 0.98823529,
       0.9372549 , 0.91372549, 0.98823529, 0.22352941, 0.02352941,
       0.        , 0.        , 0.        , 0.        , 0.        ,
```

```
0.        , 0.        , 0.        , 0.        , 0.        ,
0.        , 0.        , 0.        , 0.        , 0.        ,
0.        , 0.        , 0.03921569, 0.23529412, 0.87843137,
0.98823529, 0.99215686, 0.98823529, 0.79215686, 0.32941176,
0.98823529, 0.99215686, 0.47843137, 0.        , 0.        ,
0.        , 0.        , 0.        , 0.        , 0.        ,
0.        , 0.        , 0.        , 0.        , 0.        ,
0.        , 0.        , 0.        , 0.        , 0.        ,
0.63921569, 0.98823529, 0.98823529, 0.98823529, 0.99215686,
0.98823529, 0.98823529, 0.37647059, 0.74117647, 0.99215686,
0.65490196, 0.        , 0.        , 0.        , 0.        ,
0.        , 0.        , 0.        , 0.        , 0.        ,
0.        , 0.        , 0.        , 0.        , 0.        ,
0.        , 0.        , 0.2       , 0.93333333, 0.99215686,
0.99215686, 0.74509804, 0.44705882, 0.99215686, 0.89411765,
0.18431373, 0.30980392, 1.        , 0.65882353, 0.        ,
0.        , 0.        , 0.        , 0.        , 0.        ,
0.        , 0.        , 0.        , 0.        , 0.        ,
0.        , 0.        , 0.        , 0.        , 0.18823529,
```

In addition to reshaping our data, we will also need to encode it. For this , we will use categorical encoding, which in essence turns a number of features in numerical representations.

```
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

With our training and test data set-up, we are now ready to build our model. here We initialize a sequential model called network.

And we add our NN layers. For this example, we will be using dense layers. A dense layer simply means that each neuron receives input from all the neurons in the previous layer. [784] and [10] refer to the dimensionality of the output space, we can think of this as the number of inputs for the subsequent layers, and since we are trying to solve a classification problem with 10 possible categories (numbers 0 to 9) the final layer has a potential output of 10 units. The activation parameter refers to the activation function we want to use, in essence, an activation function calculates an output based on a given input. And finally, the input shape of [28 * 28] refers to the image's pixel width and height.

```
network = models.Sequential()

network.add(layers.Dense(784, activation='relu', input_shape=(28 * 28,)))

network.add(layers.Dense(784, activation='relu', input_shape=(28 * 28,)))

network.add(layers.Dense(10, activation='softmax'))
```

```
#network.pop() # FOR removing the layer if you executed many times
print(len(network.layers))
#network.weights
#network.summary()
```

```
3
```

Once our model is defined then we simply compile the model with our optimizer of choice, our loss function of choice, and the metrics we want to use to judge our model's performance.

To make the network ready for training, we need to pick three things, as part of the compilation step:

A loss function—How the network will be able to measure its performance on the training data, and thus how it will be able to steer itself in the right direction. **An optimizer**—The mechanism through which the network will update itself based on the data it sees and its loss function. **Metrics** to monitor during training and testing—Here, we'll only care about accuracy (the fraction of the images that were correctly classified).

```
network.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

we are now ready to train our NN! To do this we will call the fit function and pass in our required parameters.

```
network.fit(train_images, train_labels, epochs=10, batch_size=128)
```

```
Epoch 1/10
469/469 [==============================] - 12s 24ms/step - loss: 0.0448 - accuracy: 0.9325
Epoch 2/10
469/469 [==============================] - 11s 24ms/step - loss: 0.0148 - accuracy: 0.9777
Epoch 3/10
469/469 [==============================] - 11s 23ms/step - loss: 0.0089 - accuracy: 0.9864
Epoch 4/10
469/469 [==============================] - 11s 24ms/step - loss: 0.0065 - accuracy: 0.9902
Epoch 5/10
469/469 [==============================] - 11s 24ms/step - loss: 0.0042 - accuracy: 0.9942
Epoch 6/10
```

```
469/469 [==============================] - 11s 24ms/step - loss: 0.0032 - accuracy: 0.9958
Epoch 7/10
469/469 [==============================] - 11s 24ms/step - loss: 0.0027 - accuracy: 0.9963
Epoch 8/10
469/469 [==============================] - 11s 24ms/step - loss: 0.0026 - accuracy: 0.9962
Epoch 9/10
469/469 [==============================] - 11s 24ms/step - loss: 0.0022 - accuracy: 0.9969
Epoch 10/10
469/469 [==============================] - 11s 24ms/step - loss: 0.0020 - accuracy: 0.9970
<keras.callbacks.History at 0x7f19b03dab10>
```

```
test_loss, test_acc = network.evaluate(test_images, test_labels, verbose=1)
print('\nTest accuracy:', test_acc)
print("\nTest loss:", test_loss)
```

```
313/313 [==============================] - 2s 5ms/step - loss: 0.0138 - accuracy: 0.9828

Test accuracy: 0.9828000068664551

Test loss: 0.013772514648735523
```

for fitting both dataset (training and testing)

```
# network.fit(train_images, train_labels,
#           batch_size=128,
#           epochs=5,
#           verbose=1,
#           validation_data=(test_images, test_labels))
```

```
print('congratulations ! your MNIST neural network is ready now.')
```

```
congratulations ! your MNIST neural network is ready now.
```