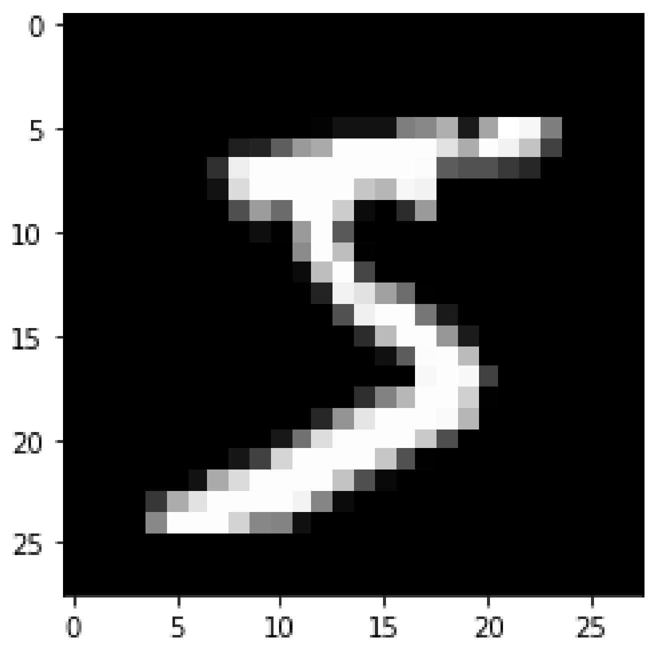
```
import cv2
import numpy as np
from keras.datasets import mnist
from keras.layers import Dense, Flatten, MaxPooling2D, Dropout
from keras.layers.convolutional import Conv2D
from keras.models import Sequential
from keras.utils import to_categorical
import matplotlib.pyplot as plt
```

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

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

```
## Looking at a sample
plt.imshow(X_train[0], cmap="gray")
plt.show()
print (y_train[0])
```



Data Preprocessing

Reshaping Stuff We need to reshape our dataset inputs (X_train and X_test) to the shape that our model expects when we train the model. The first number is the number of images (X_train -> 60000, X_test -> 10000). Then comes the shape of each image i.e. (28, 28). The last number 1 signifies that the image is greyscale

```
## Checking out the shapes involved in dataset
print ("Shape of X_train: {}".format(X_train.shape))
print ("Shape of y_train: {}".format(y_train.shape))
print ("Shape of X_test: {}".format(X_test.shape))
print ("Shape of y_test: {}".format(y_test.shape))
```

```
Shape of X_train: (60000, 28, 28)
Shape of y_train: (60000,)
Shape of X_test: (10000, 28, 28)
Shape of y_test: (10000,)
```

```
# Reshaping so as to convert images for our model
X_train = X_train.reshape(60000, 28, 28, 1)
X_test = X_test.reshape(10000, 28, 28, 1)


print ("Shape of X_train: {}".format(X_train.shape))
print ("Shape of y_train: {}".format(y_train.shape))
print ("Shape of X_test: {}".format(X_test.shape))
print ("Shape of y_test: {}".format(y_test.shape))
```

```
Shape of X_train: (60000, 28, 28, 1)
Shape of y_train: (60000,)
Shape of X_test: (10000, 28, 28, 1)
Shape of y_test: (10000,)
```

One-Hot Encoding

We need to hot encode our target variables. Basically, a column will be created for each kind of output and a binary variable is inputted for each kind. For example, if the image is of the number 6, then the label instead of being = 6, it will have a value 1 in column 7 and 0 in rest of the columns, like [0,0,0,0,0,0,1,0,0]

```
### Lets one hot encode labels
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

Building the model

Let's build the model

```
## Declare the model
model = Sequential()
```

```python
## Declare the layers
layer_1 = Conv2D(64, kernel_size=3, activation='relu', input_shape=(28, 28, 1))
layer_2 = MaxPooling2D(pool_size=2)
layer_3 = Conv2D(32, kernel_size=3, activation='relu')
layer_4 = MaxPooling2D(pool_size=2)
layer_5 = Dropout(0.5)
layer_6 = Flatten()
layer_7 = Dense(128, activation="relu")
layer_8 = Dropout(0.5)
layer_9 = Dense(10, activation='softmax')

## Add the layers to the model
model.add(layer_1)
model.add(layer_2)
model.add(layer_3)
model.add(layer_4)
model.add(layer_5)
model.add(layer_6)
model.add(layer_7)
model.add(layer_8)
model.add(layer_9)
```

The model type that we will be using is Sequential.

Sequential is the easiest way to build a model in Keras. It allows to build the model layer by layer. add() function is used for adding successive layers.

Kernel Size is the size of the filter matrix for our convolution. So, kernel size 3 means that a 3x3 filter matrix is going to be used.

Pool Size is the size of the filter window which will be used by MaxPooling Layers for the max pooling operation. So, pool size 2 means that a 2x2 window will be used for performing each iteration of max pooling operation.

Activation is the activation function for the layer. The activation function here being used for the first 2 layers is the ReLU, or Rectified Linear Activation. This activation function is known for performing well in terms of speed and output in the neural nets.

Flow of the model

The first layer takes in an input shape, here, being 28, 28, 1 where 1 signifies greyscale. Then, comes the Max Pooling layer 1 which simplifies the previous layer by taking the maximum number out of each filter window of the size 2x2 and create a matrix out of those max. numbers. Then, this max pooling layer 1 will be fed to the next Convolutional layer 2 of 32 nodes which will perform convolution operation on it using a window of size 3x3 Then, comes the max pooling layer 2 which will do the same max pooling operation as max pooling layer 1 but for the convolutional layer 2. This is for more simplification. Then, comes the dropout layer which simplify the network further by performing dropout regularisation. It basically, drops out random nodes from the network. The number of the nodes to be dropped depends on the comparison between each node achieves and probability we give. Basically, for node to drop out, node's prob. > drop out probability. So, when drop out prob. = 1 => None of the nodes will be dropped; 0 => all the nodes will be dropped. Here, we're giving that probability to be 0.5. So, almost half of the nodes will be dropped. Now, there is a "Flatten" layer. Flatten serves as a connection between convolutional and dense layers. Now, we are going to use "Dense" layer with "relu" activation. Again, we will use another dropout layer with the same drop out probability = 0.5 Now, comes the final dense layer of the activation "softmax". This will act as the output layer for our network. We will have 10 nodes in our output layer, one for each possible outcome (0-9) The activation function is 'softmax'. Softmax makes the output sum up to 1, so that the output contains a series of probabilities. The model will predict the one with the highest probability. Compiling the model Compiling the model takes three parameters

Optimizer - It controls the learning rate. We will be using 'adam' optimizer. It is a very good optimizer as it utilises the perks of both Stochastic gradient and RMSprop optimizers.

Loss function - We will be using 'categorical_crossentropy' loss function. It is the most common choice for classification. A lower score corresponds to better performance.

Metrics - To make things easier to interpret, we will be using 'accuracy' metrix to see the accuracy score on the validation set while training the model.

```
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
```

Training the model

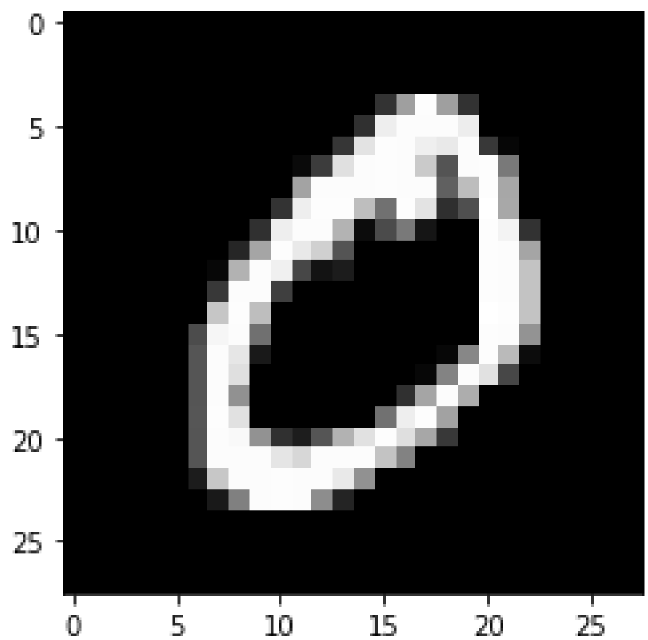
```
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=3)
```

```
Epoch 1/3
1875/1875 [==============================] - 85s 45ms/step - loss: 0.9582 -
accuracy: 0.7720 - val_loss: 0.1465 - val_accuracy: 0.9575
Epoch 2/3
1875/1875 [==============================] - 88s 47ms/step - loss: 0.2840 -
accuracy: 0.9151 - val_loss: 0.0862 - val_accuracy: 0.9742
Epoch 3/3
1875/1875 [==============================] - 84s 45ms/step - loss: 0.2130 -
accuracy: 0.9368 - val_loss: 0.0653 - val_accuracy: 0.9802
```

Predicting and Testing

```
example = X_train[1]
prediction = model.predict(example.reshape(1, 28, 28, 1))
print ("Prediction (Softmax) from the neural network:\n\n
{}".format(prediction))
hard_maxed_prediction = np.zeros(prediction.shape)
hard_maxed_prediction[0][np.argmax(prediction)] = 1
print ("\n\nHard-maxed form of the prediction: \n\n
{}".format(hard_maxed_prediction))

print ("\n\n--------- Prediction --------- \n\n")
plt.imshow(example.reshape(28, 28), cmap="gray")
plt.show()
print("\n\nFinal Output: {}".format(np.argmax(prediction)))
```

```
1/1 [==============================] - 0s 103ms/step
Prediction (Softmax) from the neural network:
[[9.9999738e-01 9.5464214e-10 2.2212889e-07 6.9435202e-10 6.2112551e-09
  5.3277843e-10 8.0331188e-07 2.5352099e-07 1.1882761e-06 8.3343465e-08]]


Hard-maxed form of the prediction:

 [[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]


--------- Prediction ---------
```



Final Output: 0

Let's test our model on a real image
For that first of all, we will preprocess the image
These are the steps **for** preprocessing the image:

Convert that image to greyscale
Binarize(threshold) the greyscaled image **in** such a way that only the digits **in** the image are white **and** rest **is** black
Using the binarized image, find contours **in** the image. Here, contours will provide us the individual digits **in** the image
Now, we have the digits. But we have to modify it further **in** such a way that it becomes a lot more similar to the images present **in** the training dataset.
Now, looking at an image **in** dataset. We can infer that the image has to be of shape (28, 28), it should contain the digit white colored **and** background black colored, **and** the digit **in** the image **is** **not** stretched to the boundaries, instead, around the digit, **in** each of the four sides, there **is** a 5 pixel region (padding) of black color. (You''ll understand this fully **if** you check out any of the image **from** the dataset).
So, now **for** modifying our image, we'll resize it to (18,18)
Then, we will add a padding of zeros (black color) of 5 pixels **in** each direction (top, bottom, left, right).

```
So, the final padded image will be of the size (5+18+5, 5+18+5) =
(28, 28), which is what we wanted.
```

Let's test our model on a real image

For that first of all, we will preprocess the image These are the steps for preprocessing the image:

Convert that image to greyscale

Binarize(threshold) the greyscaled image in such a way that only the digits in the image are white and rest is black

Using the binarized image, find contours in the image. Here, contours will provide us the individual digits in the image

Now, we have the digits. But we have to modify it further in such a way that it becomes a lot more similar to the images present in the training dataset.

Now, looking at an image in dataset. We can infer that the image has to be of shape (28, 28), it should contain the digit white colored and background black colored, and the digit in the image is not stretched to the boundaries, instead, around the digit, in each of the four sides, there is a 5 pixel region (padding) of black color. (You"ll understand this fully if you check out any of the image from the dataset).

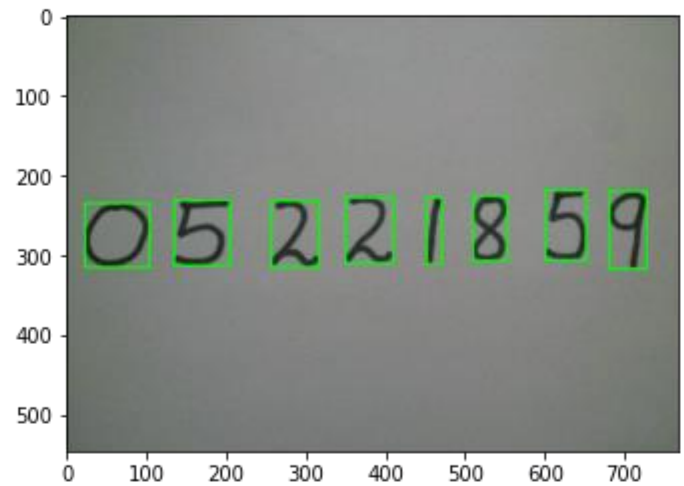
So, now for modifying our image, we'll resize it to (18,18) Then, we will add a padding of zeros (black color) of 5 pixels in each direction (top, bottom, left, right).

So, the final padded image will be of the size (5+18+5, 5+18+5) = (28, 28), which is what we wanted.

```
image = cv2.imread('/content/test_image.jpg')
grey = cv2.cvtColor(image.copy(), cv2.COLOR_BGR2GRAY)

ret, thresh = cv2.threshold(grey.copy(), 75, 255, cv2.THRESH_BINARY_INV)
contours,hierarchy= cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

preprocessed_digits = []

for c in contours:
    x,y,w,h = cv2.boundingRect(c)
# Creating a rectangle around the digit in the original image (for
displaying the digits fetched via contours)
    cv2.rectangle(image, (x,y), (x+w, y+h), color=(0, 255, 0),
thickness=2)
```

```
    # Cropping out the digit from the image corresponding to the
current contours in the for loop
    digit = thresh[y:y+h, x:x+w]

    # Resizing that digit to (18, 18)
    resized_digit = cv2.resize(digit, (18,18))

    # Padding the digit with 5 pixels of black color (zeros) in
each side to finally produce the image of (28, 28)
    padded_digit = np.pad(resized_digit, ((5,5),(5,5)), "constant",
constant_values=0)

    # Adding the preprocessed digit to the list of preprocessed
digits
    preprocessed_digits.append(padded_digit)

print("\n\n\n----------------Contoured Image--------------------")
plt.imshow(image, cmap="gray")
plt.show()

inp = np.array(preprocessed_digits)
```

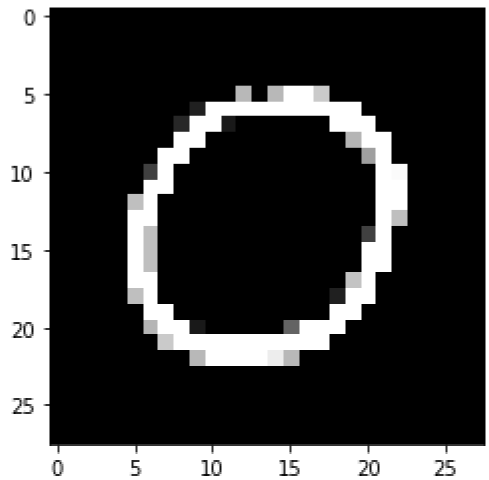
```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
 in
      1 image = cv2.imread('/content/test_image.jpg')
----> 2 grey = cv2.cvtColor(image.copy(), cv2.COLOR_BGR2GRAY)
      3
      4 ret, thresh = cv2.threshold(grey.copy(), 75, 255,
cv2.THRESH_BINARY_INV)
      5 contours,hierarchy= cv2.findContours(thresh.copy(),
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

AttributeError: 'NoneType' object has no attribute 'copy'
```




```
for digit in preprocessed_digits:
    prediction = model.predict(digit.reshape(1, 28, 28, 1))
    print ("\n\n---------------------------------------\n\n")
    print ("=========PREDICTION============ \n\n")
    plt.imshow(digit.reshape(28, 28), cmap="gray")
    plt.show()
    print("\n\nFinal Output: {}".format(np.argmax(prediction)))
    print ("\nPrediction (Softmax) from the neural network:\n\n
{}".format(prediction))
    hard_maxed_prediction = np.zeros(prediction.shape)
    hard_maxed_prediction[0][np.argmax(prediction)] = 1
    print ("\n\nHard-maxed form of the prediction: \n\n
{}".format(hard_maxed_prediction))
    print ("\n\n---------------------------------------\n\n")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
 in
----> 1 for digit in preprocessed_digits:
      2     prediction = model.predict(digit.reshape(1, 28, 28, 1))
      3     print ("\n\n---------------------------------------\n\n")
      4     print ("=========PREDICTION============ \n\n")
      5     plt.imshow(digit.reshape(28, 28), cmap="gray")

NameError: name 'preprocessed_digits' is not defined
```

=========PREDICTION============



Final Output: 0

Prediction (Softmax) from the neural network:

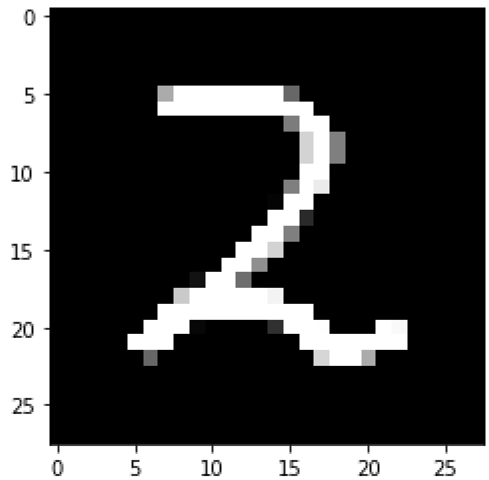
[[9.9999917e-01 3.5340183e-14 2.2112522e-08 2.8153077e-11 1.4685572e-11 4.8874613e-11 5.5869581e-08 1.5508652e-12 6.5314953e-07 1.1440835e-07]]

Hard-maxed form of the prediction:

[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]

1/1 [==============================] - 0s 19ms/step

=========PREDICTION============



Final Output: 2

Prediction (Softmax) from the neural network:

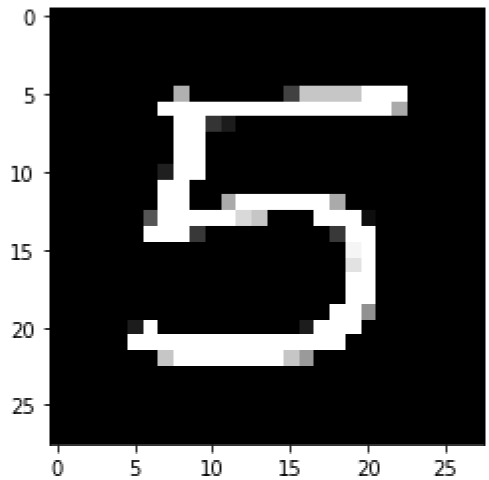
[[3.0629899e-13 6.8132201e-12 1.0000000e+00 3.6194525e-09 8.8187543e-21
8.2901249e-18 2.4562090e-18 2.6661509e-09 3.6261760e-09 5.7934094e-16]]

Hard-maxed form of the prediction:

[[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]]

1/1 [==============================] - 0s 15ms/step

=========PREDICTION============



Final Output: 5

Prediction (Softmax) from the neural network:

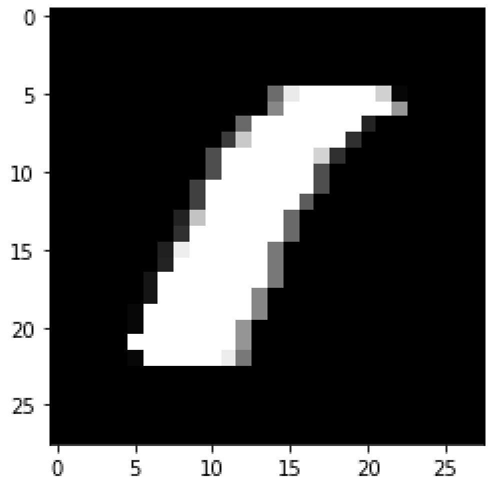
[[8.9878327e-15 5.1411508e-17 2.4887940e-16 1.5858398e-06 7.6144713e-15
9.9999845e-01 6.0097496e-11 4.5582556e-14 3.0063843e-09 8.6659159e-11]]

Hard-maxed form of the prediction:

[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]]

1/1 [==============================] - 0s 16ms/step

=========PREDICTION============



Final Output: 8

Prediction (Softmax) from the neural network:

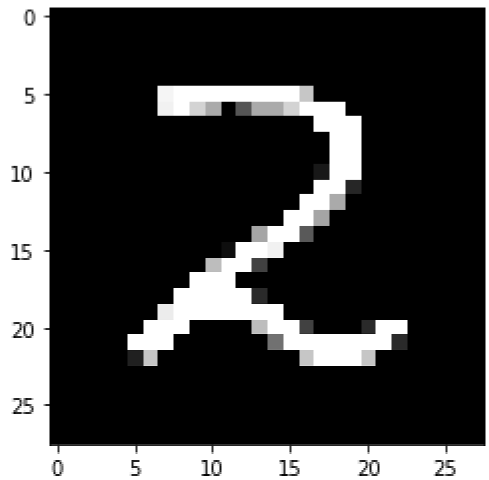
[[3.2720964e-02 1.9043766e-02 4.3137711e-03 2.1025266e-04 5.3455061e-03 3.0700490e-02 1.5814583e-01 6.6169095e-04 7.4381024e-01 5.0475220e-03]]

Hard-maxed form of the prediction:

[[0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]

1/1 [==============================] - 0s 16ms/step

=========PREDICTION============



Final Output: 2

Prediction (Softmax) from the neural network:

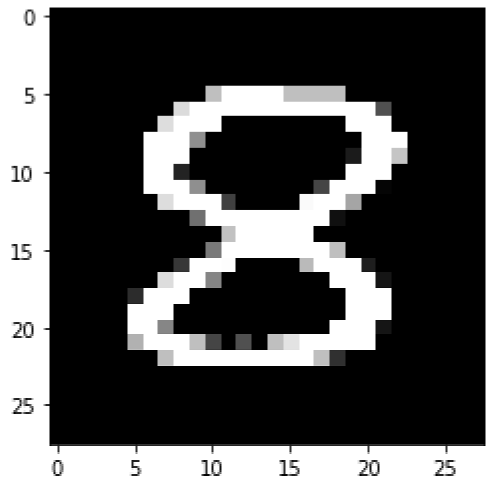
[[5.2333778e-13 3.4270063e-12 9.9999988e-01 6.2104180e-09 3.2207634e-19 3.7055385e-17 1.8095418e-17 2.5537053e-08 1.5839066e-07 5.3073573e-15]]

Hard-maxed form of the prediction:

[[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]]

1/1 [==============================] - 0s 20ms/step

=========PREDICTION============



Final Output: 8

Prediction (Softmax) from the neural network:

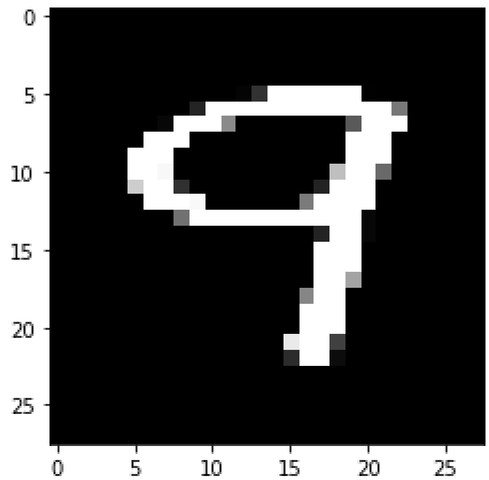
[[8.9141777e-10 1.1559232e-10 2.4292819e-07 2.3460484e-03 4.2623162e-12 7.3766969e-05 5.9465695e-09 5.9620919e-10 9.9757993e-01 2.0580067e-08]]

Hard-maxed form of the prediction:

[[0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]

1/1 [==============================] - 0s 16ms/step

=========PREDICTION============



Final Output: 9

Prediction (Softmax) from the neural network:

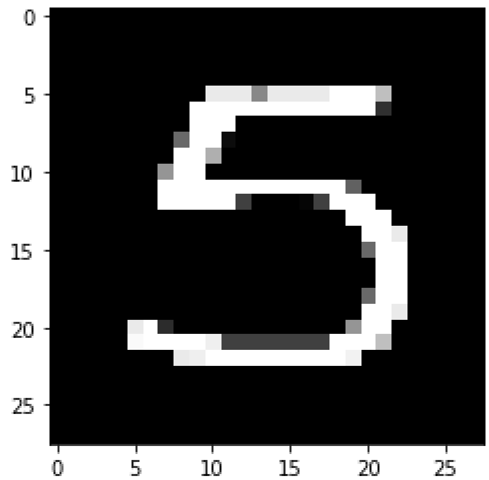
[[7.4751202e-05 3.1806519e-03 2.0419380e-03 4.8388895e-02 2.5094576e-02 3.4303032e-03 1.5223634e-05 4.2743430e-01 6.4502405e-03 4.8388919e-01]]

Hard-maxed form of the prediction:

[[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]

1/1 [==============================] - 0s 16ms/step

=========PREDICTION============



Final Output: 5

Prediction (Softmax) from the neural network:

[[1.23008131e-11 7.92557480e-14 4.88042386e-13 7.41563213e-07 1.02033054e-13
9.99999285e-01 7.48859974e-10 1.09584522e-10 1.14892789e-08 5.65052805e-09]]

Hard-maxed form of the prediction:

[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]]