Megha Tatti
CS 512 – Computer Vision
CWID: A20427027

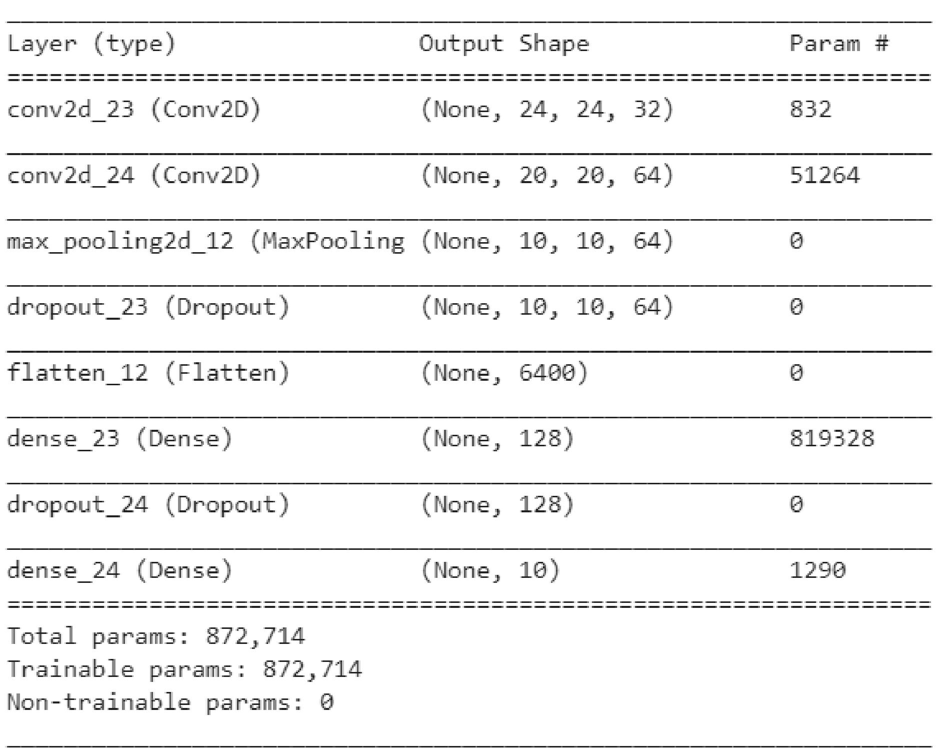
Assignment 4 – CNN

This project is built using the Keras implementation with tensorflow backend.

Deliverable 1: Custom CNN

Model Details:

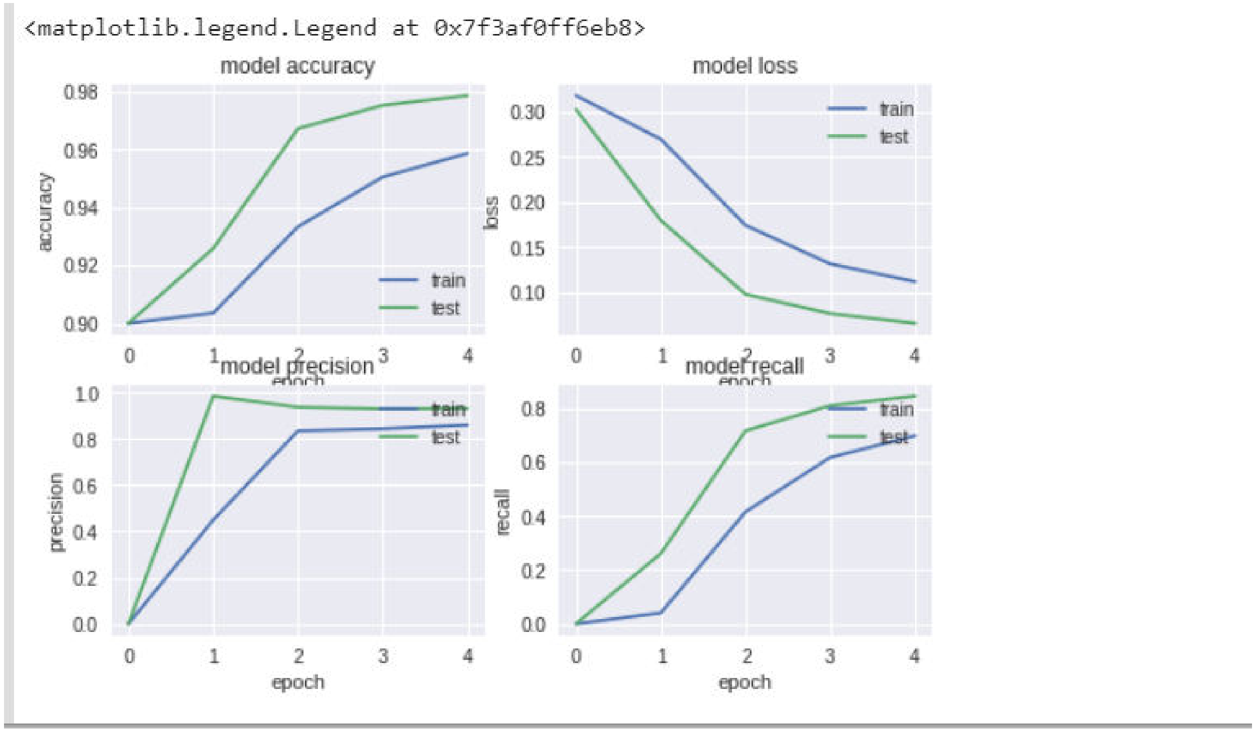
- Here we have used Sequential Convolution 2D model.
- There are 2 layers with 32 and 64 filters of kernel size 5x5.
- The datasets are MNIST dataset from "keras".
- 55000 samples are trained, and 10000 samples are tested for 5 epochs.
- Pooling in both layers should downsample by factor 2 of 2.
- Dropout rate is taken as 40%**.**
- The model Summary is as follows:



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_23 (Conv2D)           (None, 24, 24, 32)        832
_________________________________________________________________
conv2d_24 (Conv2D)           (None, 20, 20, 64)        51264
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 10, 10, 64)        0
_________________________________________________________________
dropout_23 (Dropout)         (None, 10, 10, 64)        0
_________________________________________________________________
flatten_12 (Flatten)         (None, 6400)              0
_________________________________________________________________
dense_23 (Dense)             (None, 128)               819328
_________________________________________________________________
dropout_24 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_24 (Dense)             (None, 10)                1290
=================================================================
Total params: 872,714
Trainable params: 872,714
Non-trainable params: 0
_________________________________________________________________
```

Results:

- The graph plots of the loss, accuracy, precision and recall for training and validation are as follows:



```
<matplotlib.legend.Legend at 0x7f3af0ff6eb8>
```

- The results for every iteration are as follows:

1st Iteration

```
Epoch 1/5
    55000/55000 [==============================] - 8s 150us/step - loss:
    0.3186 - acc: 0.9000 - precision: 0.0000e+00 - recall: 0.0000e+00 -
    val_loss: 0.3031 - val_acc: 0.9000 - val_precision: 0.0000e+00 -
    val_recall: 0.0000e+00
```

2nd Iteration

```
Epoch 2/5
    55000/55000 [==============================] - 7s 128us/step - loss:
    0.2700 - acc: 0.9035 - precision: 0.4494 - recall: 0.0403 - val_loss:
    0.1796 - val_acc: 0.9258 - val_precision: 0.9857 - val_recall: 0.2614
```

3rd Iteration

```
Epoch 3/5
    55000/55000 [==============================] - 7s 129us/step - loss:
    0.1742 - acc: 0.9333 - precision: 0.8352 - recall: 0.4165 - val_loss:
    0.0974 - val_acc: 0.9672 - val_precision: 0.9377 - val_recall: 0.7179
```

4th iteration

```
Epoch 4/5
```

```
55000/55000 [==============================] - 7s 129us/step - loss:
0.1312 - acc: 0.9505 - precision: 0.8443 - recall: 0.6188 - val_loss:
0.0759 - val_acc: 0.9752 - val_precision: 0.9304 - val_recall: 0.8116
```

Final iteration

```
Epoch 5/5
    55000/55000 [==============================] - 7s 129us/step - loss:
    0.1115 - acc: 0.9586 - precision: 0.8604 - recall: 0.6997 - val_loss:
    0.0652 - val_acc: 0.9786 - val_precision: 0.9319 - val_recall: 0.8470
```

- The values of the final step:

```
Test loss: 0.06517968086898326
Test accuracy: 0.9786200025558471
Test precision: 0.9316988312721253
Test recall: 0.847
Train loss: 0.1115
Train accuracy: 0.9586
Train precision: 0.8604
Train recall: 0.8470
```

This model is saved in the .h5 format to use in Deliverable 3.

Discussion:

- In the deliverable, we use the “binary_crossentropy” to calculate loss.
- The optimizer used is keras.optimizers.SGD(lr=0.001) with learning rate =0.01.
- The batch_size = 128, num_classes=10.

By using the above-mentioned parameters, we can see that:

- Loss is around 0.065
- Accuracy is around 0.98
- Precision is around 0.93
- Recall is around 0.84

Deliverable 2: Parameter tuning

Model Details:

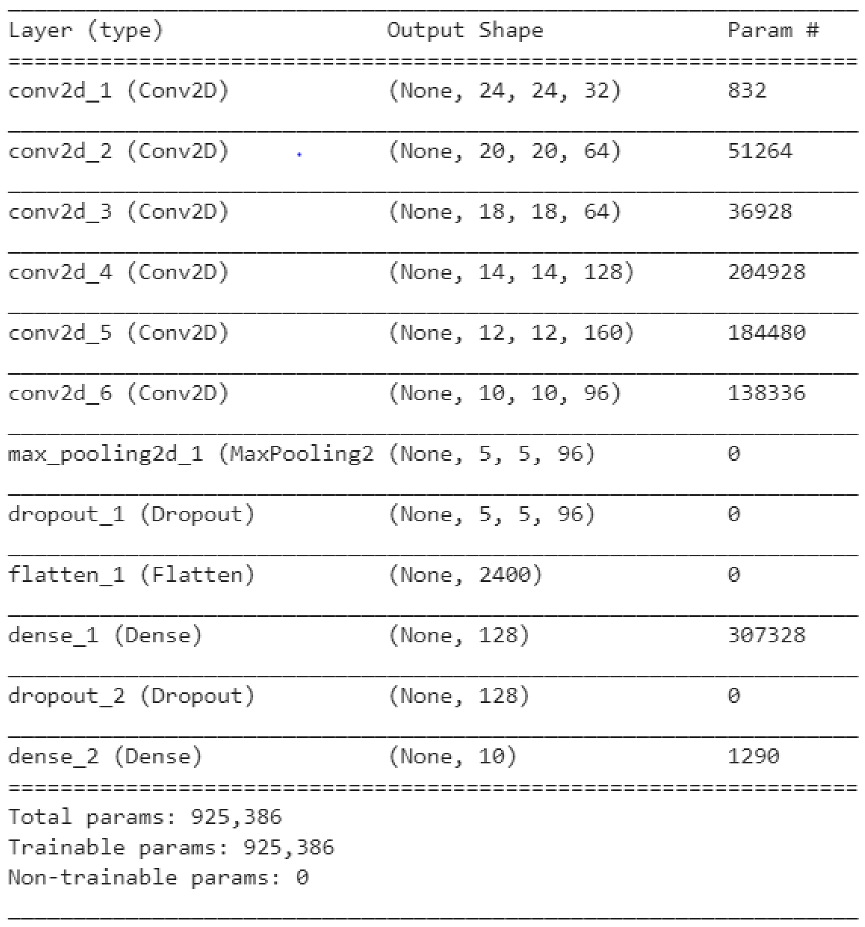
1. **Changing the network architecture**

To the deliverable 1, the layers are added, and the organization has been altered.

The following code has been added to add more layers:

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(128, (5, 5), activation='relu'))
model.add(Conv2D(160, (3, 3), activation='relu'))
model.add(Conv2D(96, (3, 3), activation='relu'))

The model summary after adding this is:



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 24, 24, 32)        832
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 20, 20, 64)        51264
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 18, 18, 64)        36928
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 14, 14, 128)       204928
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 12, 12, 160)       184480
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 10, 10, 96)        138336
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 96)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 5, 5, 96)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2400)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               307328
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 925,386
Trainable params: 925,386
Non-trainable params: 0
_________________________________________________________________
```

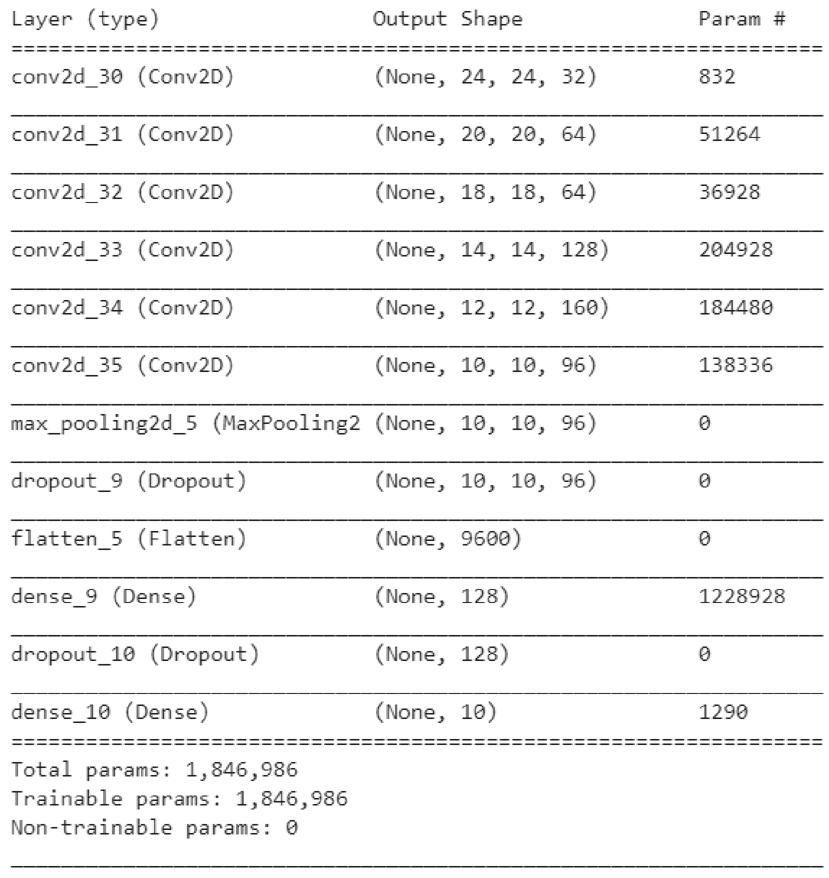
2. Changing the receptive field and stride parameters

After the 1st step in Deliverable 2, to the same code, we alter the following to change the receptive field and stride parameters.

The pooling is downsampled to factor 2 of 2.

model.add(MaxPooling2D(pool_size=(1, 1)))

The stride parameter is added.
model.add(Conv2D(32, kernel_size=(5, 5), activation='relu', input_shape=input_shape, strides=(1,1)))



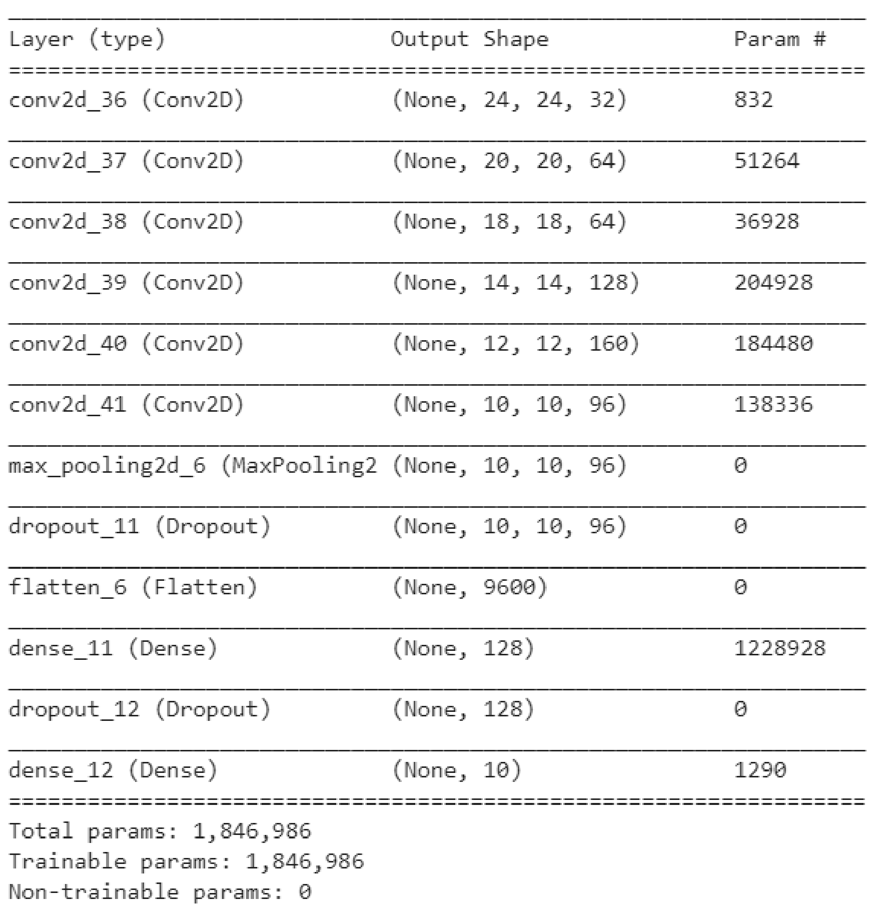
```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_30 (Conv2D)           (None, 24, 24, 32)        832
_________________________________________________________________
conv2d_31 (Conv2D)           (None, 20, 20, 64)        51264
_________________________________________________________________
conv2d_32 (Conv2D)           (None, 18, 18, 64)        36928
_________________________________________________________________
conv2d_33 (Conv2D)           (None, 14, 14, 128)       204928
_________________________________________________________________
conv2d_34 (Conv2D)           (None, 12, 12, 160)       184480
_________________________________________________________________
conv2d_35 (Conv2D)           (None, 10, 10, 96)        138336
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 10, 10, 96)        0
_________________________________________________________________
dropout_9 (Dropout)          (None, 10, 10, 96)        0
_________________________________________________________________
flatten_5 (Flatten)          (None, 9600)              0
_________________________________________________________________
dense_9 (Dense)              (None, 128)               1228928
_________________________________________________________________
dropout_10 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_10 (Dense)             (None, 10)                1290
=================================================================
Total params: 1,846,986
Trainable params: 1,846,986
Non-trainable params: 0
_________________________________________________________________
```

3. Changing optimizer and loss function

We change the optimizer to from "SGD "(keras.optimizers.SGD) to "Adam" (keras.optimizers.Adam).

We change the loss function from "binary_crossentropy" to "mean_squared_error".

model.compile(loss="mean_squared_error", optimizer=keras.optimizers.Adam(lr=0.01), metrics=['accuracy',precision,recall])



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_36 (Conv2D)           (None, 24, 24, 32)        832       
_________________________________________________________________
conv2d_37 (Conv2D)           (None, 20, 20, 64)        51264     
_________________________________________________________________
conv2d_38 (Conv2D)           (None, 18, 18, 64)        36928     
_________________________________________________________________
conv2d_39 (Conv2D)           (None, 14, 14, 128)       204928    
_________________________________________________________________
conv2d_40 (Conv2D)           (None, 12, 12, 160)       184480    
_________________________________________________________________
conv2d_41 (Conv2D)           (None, 10, 10, 96)        138336    
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 10, 10, 96)        0         
_________________________________________________________________
dropout_11 (Dropout)         (None, 10, 10, 96)        0         
_________________________________________________________________
flatten_6 (Flatten)          (None, 9600)              0         
_________________________________________________________________
dense_11 (Dense)             (None, 128)               1228928   
_________________________________________________________________
dropout_12 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_12 (Dense)             (None, 10)                1290      
=================================================================
Total params: 1,846,986
Trainable params: 1,846,986
Non-trainable params: 0
```

4. Varying various parameters

We alter the values of dropout, learning rate, number of filters, number of epochs

model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))

model.add(Conv2D(160, (3, 3), activation='relu'))
model.add(Conv2D(96, (3, 3), activation='relu'))
epochs = 6

model.compile(loss="binary_crossentropy", optimizer=keras.optimizers.Adam(lr=0.02), metrics=['accuracy',precision,recall])

5. Adding Batch and layer Normalization

We change the batch_size from 128 to 150

batch_size=150

Using BatchNormalization function as follows:

keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, center=True, scale=True, beta_initializer='zeros', gamma_initializer='ones', moving_mean_initializer='zeros', moving_variance_initializer='ones', beta_regularizer=None, gamma_regularizer=None, beta_constraint=None, gamma_constraint=None)

6. Using different weight initializers

Using the Xavier normal weight initializer. It draws samples from a truncated normal distribution centered on 0 with `stddev = sqrt(2 / (fan_in + fan_out))` where `fan_in` is the number of input units in the weight tensor and `fan_out` is the number of output units in the weight tensor.

The following code is used.

keras.initializers.glorot_normal(seed=None)

7. Using features from pretrained model

Using VGG16 model, with weights pre-trained on ImageNet. The following code is used.

keras.applications.vgg16.VGG16(include_top=True, weights='imagenet', input_tensor=None, input_shape=None, pooling=None, classes=1000)

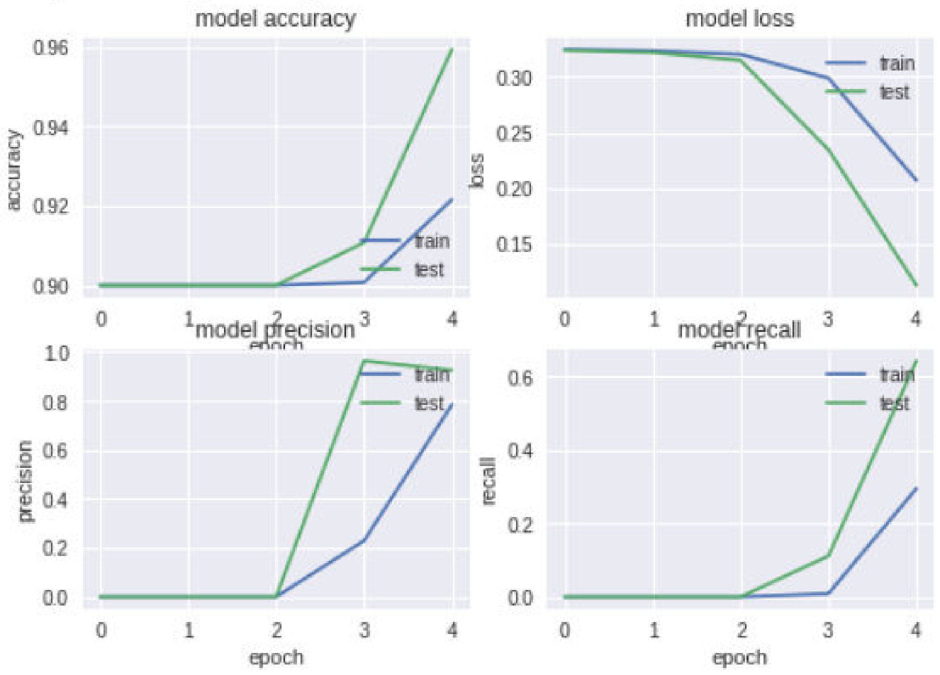
This model can be built both with `'channels_first'` data format (channels, height, width) or `'channels_last'`data format (height, width, channels).

The default input size for this model is 224x224.

Results:

1. The graph plots of the loss, accuracy, precision and recall for training and validation are as follows:

```
<matplotlib.legend.Legend at 0x7ff5c22c9c50>
```



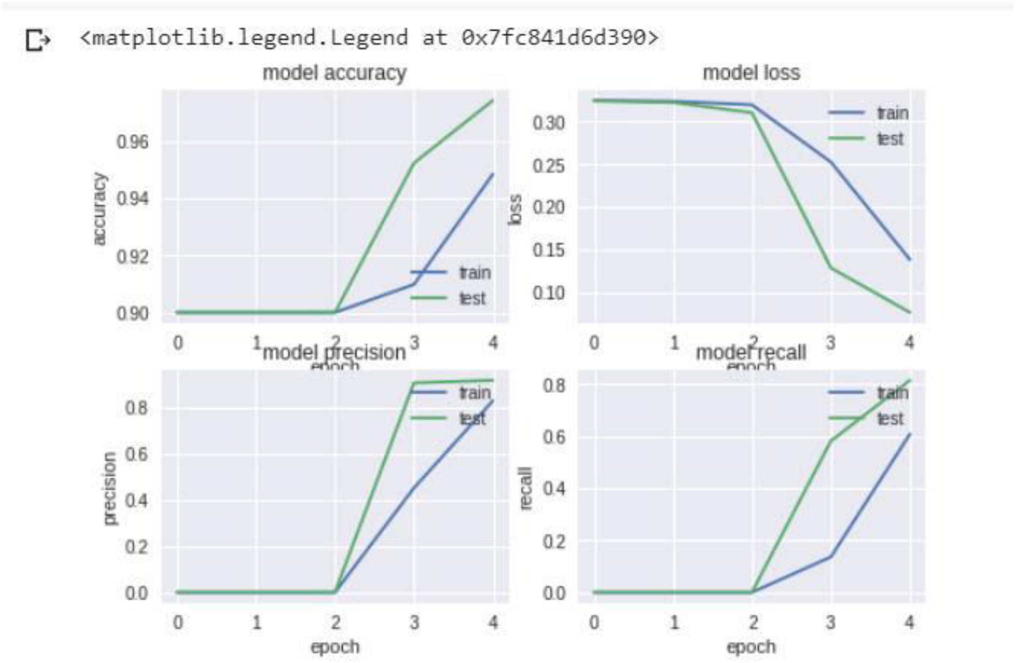
The final iteration is:

```
Epoch 5/5
   55000/55000 [==============================] - 21s 390us/step - loss:
   0.2073 - acc: 0.9215 - precision: 0.7848 - recall: 0.2947 - val_loss:
   0.1130 - val_acc: 0.9593 - val_precision: 0.9260 - val_recall: 0.6423
```

- The values of the final step:

```
Test loss: 0.1129913817167282
Test accuracy: 0.959289994430542
Test precision: 0.92493919506073
Test recall: 0.6423
Train loss: 0.2073
Train accuracy: 0.9215
Train precision: 0.9260
Train recall: 0.6423
```

2. The graph plots of the loss, accuracy, precision and recall for training and validation are as follows:



```
<matplotlib.legend.Legend at 0x7fc841d6d390>
```

The final iteration is:

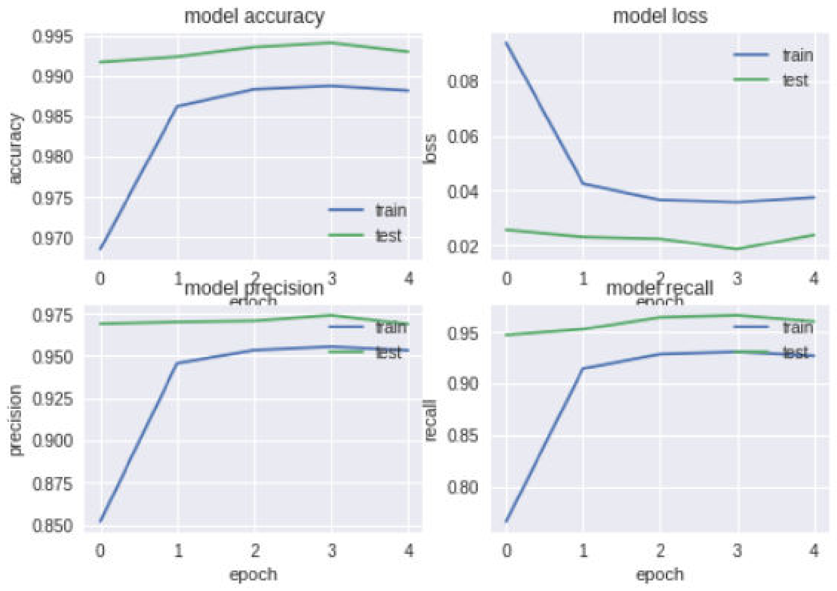
```
Epoch 5/5
   55000/55000 [==============================] - 22s 409us/step - loss:
   0.1385 - acc: 0.9484 - precision: 0.8296 - recall: 0.6074 - val_loss:
   0.0766 - val_acc: 0.9743 - val_precision: 0.9181 - val_recall: 0.8138
```

- The values of the final step:

```
Test loss: 0.07663719896376132
Test accuracy: 0.9742700033187867
Test precision: 0.9180297658920288
Test recall: 0.8138
Train loss: 0.1385
Train accuracy: 0.9484
Train precision: 0.8296
Train recall: 0.6074
```

3. The graph plots of the loss, accuracy, precision and recall for training and validation are as follows:

```
<matplotlib.legend.Legend at 0x7fc82f4f4358>
```



The final iteration is:

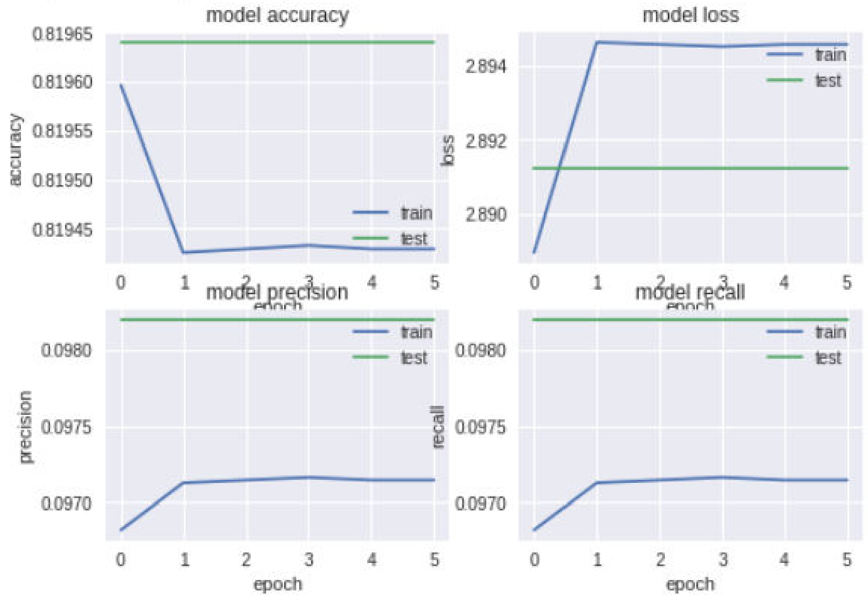
```
Epoch 5/5
55000/55000 [==============================] - 22s 405us/step - loss:
0.0375 - acc: 0.9882 - precision: 0.9533 - recall: 0.9272 - val_loss:
0.0238 - val_acc: 0.9930 - val_precision: 0.9690 - val_recall: 0.9607
```

- The values of the final step:

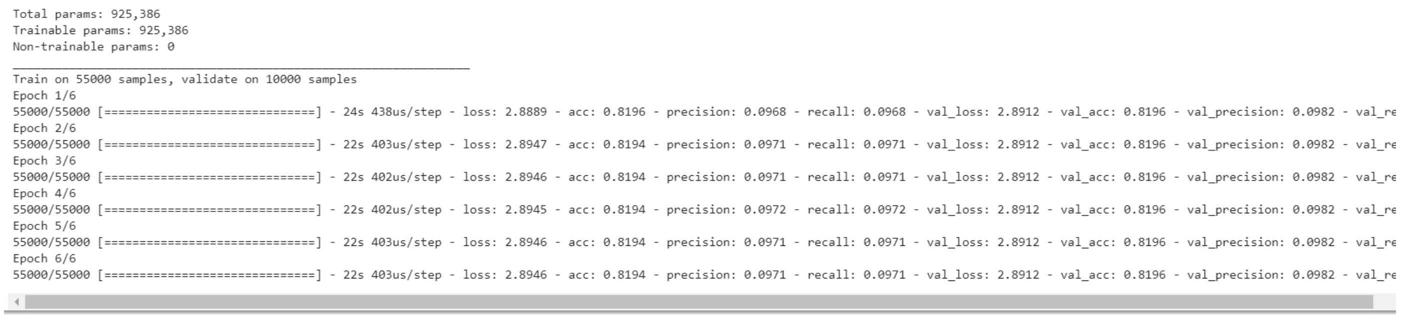
```
Test loss: 0.023781839976796983
Test accuracy: 0.9929999998092651
Test precision: 0.968920813369751
Test recall: 0.9607
Train loss: 0.0375
Train accuracy: 0.9930
Train precision: 0.9690
Train recall: 0.9607
```

4. The graph plots of the loss, accuracy, precision and recall for training and validation are as follows:

```
<matplotlib.legend.Legend at 0x7fc82be95f28>
```



Training:



```
Total params: 925,386
Trainable params: 925,386
Non-trainable params: 0
_________________________________________________________________
Train on 55000 samples, validate on 10000 samples
Epoch 1/6
55000/55000 [==============================] - 24s 438us/step - loss: 2.8889 - acc: 0.8196 - precision: 0.0968 - recall: 0.0968 - val_loss: 2.8912 - val_acc: 0.8196 - val_precision: 0.0982 - val_re
Epoch 2/6
55000/55000 [==============================] - 22s 403us/step - loss: 2.8947 - acc: 0.8194 - precision: 0.0971 - recall: 0.0971 - val_loss: 2.8912 - val_acc: 0.8196 - val_precision: 0.0982 - val_re
Epoch 3/6
55000/55000 [==============================] - 22s 402us/step - loss: 2.8946 - acc: 0.8194 - precision: 0.0971 - recall: 0.0971 - val_loss: 2.8912 - val_acc: 0.8196 - val_precision: 0.0982 - val_re
Epoch 4/6
55000/55000 [==============================] - 22s 402us/step - loss: 2.8945 - acc: 0.8194 - precision: 0.0972 - recall: 0.0972 - val_loss: 2.8912 - val_acc: 0.8196 - val_precision: 0.0982 - val_re
Epoch 5/6
55000/55000 [==============================] - 22s 403us/step - loss: 2.8946 - acc: 0.8194 - precision: 0.0971 - recall: 0.0971 - val_loss: 2.8912 - val_acc: 0.8196 - val_precision: 0.0982 - val_re
Epoch 6/6
55000/55000 [==============================] - 22s 403us/step - loss: 2.8946 - acc: 0.8194 - precision: 0.0971 - recall: 0.0971 - val_loss: 2.8912 - val_acc: 0.8196 - val_precision: 0.0982 - val_re
```

The final iteration is:

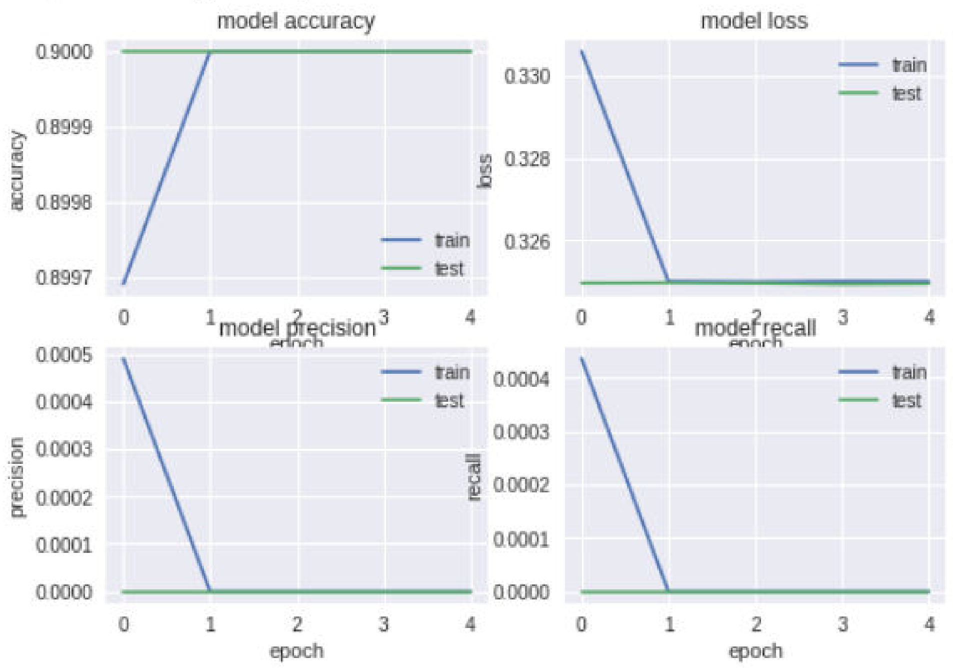
```
Epoch 6/6
55000/55000 [==============================] - 22s 403us/step - loss:
2.8946 - acc: 0.8194 - precision: 0.0971 - recall: 0.0971 - val_loss:
2.8912 - val_acc: 0.8196 - val_precision: 0.0982 - val_recall: 0.0982
```

- The values of the final step:

```
Test loss: 2.8912137603759764
Test accuracy: 0.8196400192260742
Test precision: 0.0982
Test recall: 0.0982
Train loss: 2.8946
Train accuracy: 0.8194
Train precision: 0.0971
Train recall: 0.0971
```

5. The graph plots of the loss, accuracy, precision and recall for training and validation are as follows:

```
<matplotlib.legend.Legend at 0x7fc82cd14c18>
```



The final iteration is:

```
Epoch 5/5
```

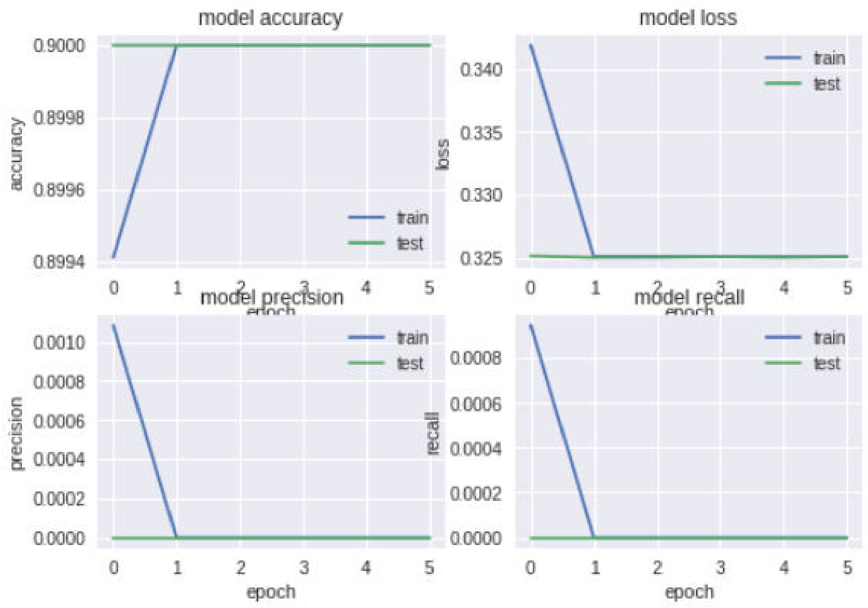
```
55000/55000 [==============================] - 22s 402us/step - loss:
0.3250 - acc: 0.9000 - precision: 0.0000e+00 - recall: 0.0000e+00 -
val_loss: 0.3249 - val_acc: 0.9000 - val_precision: 0.0000e+00 -
val_recall: 0.0000e+00
```

- The values of the final step:

```
Test loss: 0.32494929628372193
Test accuracy: 0.9000099761581421
Test precision: 0.0
Test recall: 0.0
Train loss: 0.3250
Train accuracy: 0.9000
Train precision: 0.0000
Train recall: 0.0000
```

6. The graph plots of the loss, accuracy, precision and recall for training and validation are as follows:

```
<matplotlib.legend.Legend at 0x7fc82b4c20b8>
```



The final iteration:

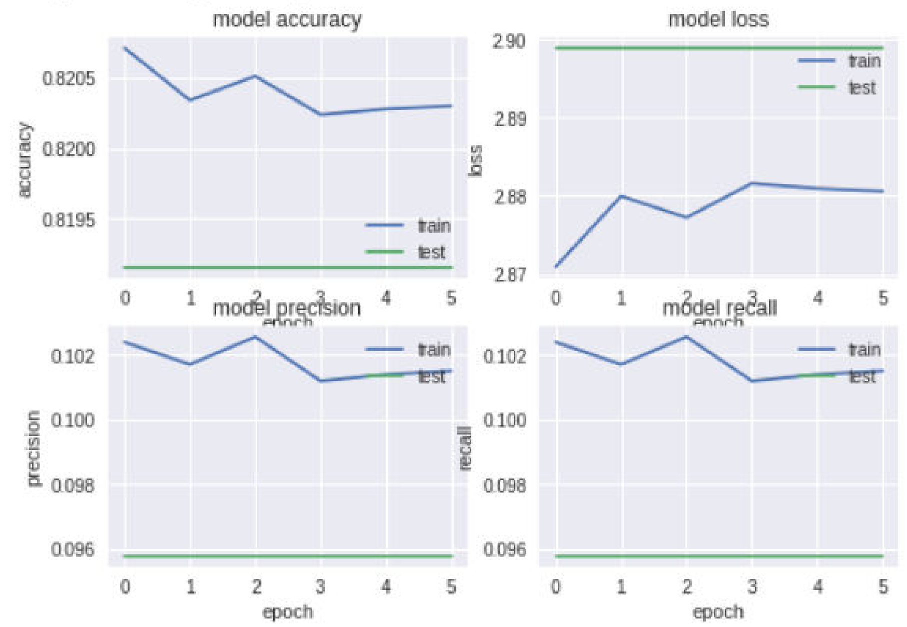
```
Epoch 6/6
55000/55000 [==============================] - 22s 401us/step -
loss: 0.3251 - acc: 0.9000 - precision: 0.0000e+00 - recall:
0.0000e+00 - val_loss: 0.3251 - val_acc: 0.9000 - val_precision:
0.0000e+00 - val_recall: 0.0000e+00
```

- The values of the final step:

```
Test loss: 0.3250725396156311
Test accuracy: 0.8999999761581421
Test precision: 0.0
Test recall: 0.0
Train loss: 0.3251
Train accuracy: 0.9000
Train precision:0.0
Train recall:0.0
```

7. The graph plots of the loss, accuracy, precision and recall for training and validation are as follows:

```
<matplotlib.legend.Legend at 0x7fc82ce674e0>
```



The final iteration:

```
Epoch 6/6
55000/55000 [==============================] - 22s 408us/step - loss:
2.8806 - acc: 0.8203 - precision: 0.1015 - recall: 0.1015 - val_loss:
2.8989 - val_acc: 0.8192 - val_precision: 0.0958 - val_recall: 0.0958
```

- The values of the final step:

```
Test loss: 2.8989082710266114
Test accuracy: 0.819160019493103
Test precision: 0.0958
Test recall: 0.0958
Train loss: 2.8806
Train precision: 0.1015
Total recall: 0.1015
```

Deliverable 3: Applications

Implementation Details:

First step is to connect to google colab using the following command and give authorization.

1. Accept an image as an input using the following command
   image=cv2.imread(path,1)

2. Using OpenCV, we do the following:
   We resize the image to fit the model's image size requirements.
   pix_resol = cv2.resize(image, (28,28))
   We then convert the image to grayscale and then form a binary image. We use **GuassianBlur() and binary threshold**
   gray_scale = cv2.cvtColor(pix_resol,cv2.COLOR_BGR2GRAY)
   blurr = cv2.GaussianBlur(gray_scale,(3,3),0)
   retval, threshold = cv2.threshold(blurr, 70, 255, cv2.THRESH_BINARY)

   Then the original and the Binary images are shown in 2 separate windows
   plt.subplot(121),plt.imshow(image),plt.title('Original')
   plt.xticks([]), plt.yticks([])
   plt.subplot(122),plt.imshow(threshold),plt.title('Preprocessed Image')
   plt.xticks([]), plt.yticks([])
   plt.show()

3. Classifying the Binary image using CNN
   Here, we first load our **Deliverable 1** model saved as "**cnn.h5**". We use the following code:
   from keras.models import load_model
   model = load_model('/content/drive/My Drive/apps/cnn.h5')
   We use the following code to classify the binary image
   threshold=np.reshape(threshold,(-1,28,28,1))
   predictions=model.predict_classes(threshold)

4. Predicting whether the image is odd/even and printing it.
   threshold=np.reshape(threshold,(-1,28,28,1))
   predictions=model.predict_classes(threshold)
   #print(predictions[0])
   label = ''
   label = "even" if predictions[0] %2 == 0 else "odd"
   print("The label is: ", label)

5. Program terminates on pressing 'q' or 'esc' button
   while True:
   path = input("\nPlease enter file path or press esc or q for exit: ")
   #print(keyboard.is_pressed('q'))

```
        if path == 'q':
            print("\n\nProgram Terminated")
            break
        else:
            test(path)
```

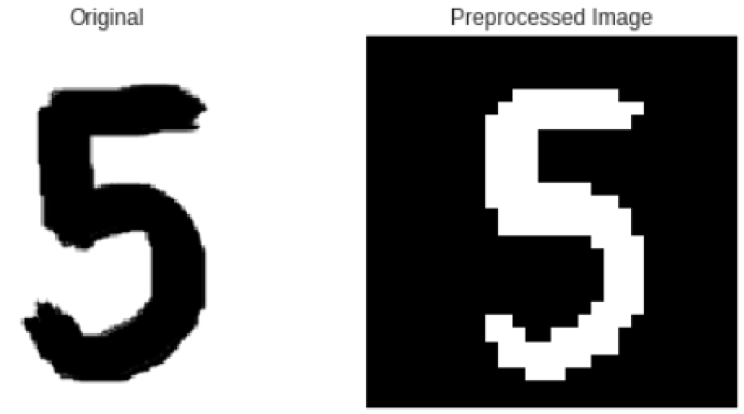
Results:

The image path is taken as input:

```
Please enter file path or press q for exit: |
```

Testing the prediction of odd numbers:

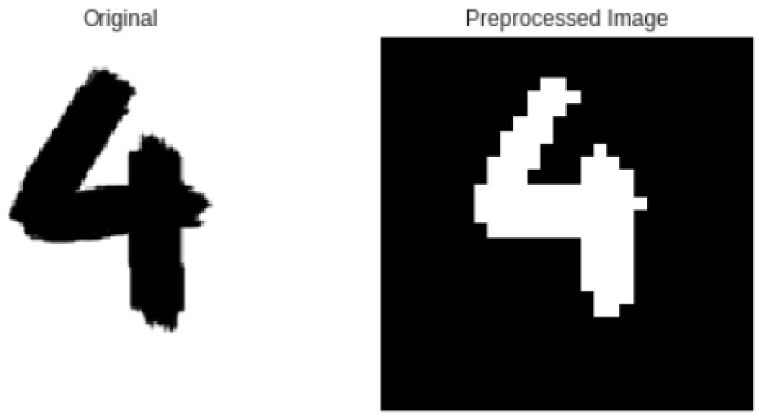
```
Please enter file path or press q for exit: /content/drive/My Drive/apps/input/five.png
```



```
The number is:  odd
```

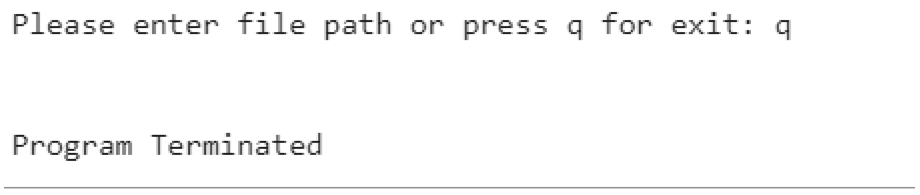
Testing the prediction of even numbers

```
Please enter file path or press q for exit: /content/drive/My Drive/apps/input/four.png
```



```
The number is:  even
```

To terminate.



```
Please enter file path or press q for exit: q


Program Terminated
```

Discussions:

We first loaded our model “cnn.h5”
The prediction was done based on this model. It first asks for the image path and then preprocesses it to fit the model’s image requirements and then predicts if the handwritten number in the image is odd or even. On Pressing the ‘q’ key on the keyboard, the program terminates.