Arabic Handwritten Character Recognition

DOCUMENTATION PROJECT FOR:

Selected -2

FACULTY NAME:

Computers and artificial intelligence Helwan

TEAMWORK

	name	ID
1	Mohamed Ashraf Khalifa	**202000729**
2	Mohamed Zakaria Kamel	202000761
3	Mohamed Emadeline Abdelhamid	202000806
4	Ahmed Ashraf Taha	202000013
5	Mostafa Kamal Fahmy	202000907
6	Mostafa AlaaElden Mostafa	202000906

Our paper

Name:

Arabic handwriting recognition system using convolutional neural network

Publisher:

[Najwa Altwaijry (0000-0002-7386-1886) (orcid.org)]

[Isra Al-Turaiki (0000-0003-0550-5115) (orcid.org)]

Link:
[Arabic handwriting recognition system using convolutional neural network | SpringerLink]

Received: 11 October 2019 / Accepted: 3 June 2020 / Published online: 28 June 2020 © The Author(s) 2020

CONTENTS

Definition

Overview

The automatic recognition of text on scanned images has enabled many applications such as searching for words in large volumes of documents, automatic sorting of postal mail, and convenient editing of previously printed documents. The domain of handwriting in the Arabic script presents unique technical challenges and has been addressed more recently than other domains. Many different methods have been proposed and applied to various types of images.
Here we will focus on the recognition part of handwritten Arabic letters recognition that face several challenges, including the unlimited variation in human handwriting and the large public databases.
In this project we will use the unique datasets **Arabic Letters** for arabic digits respectively which are available at kaggle kernels.
There are many related papers that introduced some solutions for that problem such as :

- **HMM Based Approach for Handwritten Arabic Word Recognition**
- **An Arabic handwriting synthesis system**
- **Normalization-Cooperated Gradient Feature Extraction for Handwritten**

Character Recognition

Problem Statement

In this project we want to build a model which can classify a new image to an arabic letter.
We can use machine learning classification algorithms or deep learning algorithms like CNN to build a strong model able to recognize the images

which high accuracy.
My solution will use a CNN which performs better with images classification problems as they can successfully boil down a given image into a highly abstracted representation through the layer filters which make the prediction process more accurate.
There are some related kernels which discuss this problem on kaggle ,
To Conclude our input of the model will be an image and the output will be one of the arabic letters or digits which represents the image.

Data Exploration

I. Datasets and Inputs

Arabic Letters Dataset is composed of 16,800 characters written by 60 participants, the age range is between 19 to 40 years, and 90% of participants are right-hand. Each participant wrote each character (from ’alef’ to ’yeh’) ten times.
The images were scanned at the resolution of 300 dpi. Each block is segmented automatically using Matlab 2016a to determining the coordinates for each block. The database is partitioned into two sets: a training set (13,440 characters to 480 images per class)
and a test set (3,360 characters to 120 images per class). Writers of training set and test set are exclusive. Ordering of including writers to test set are randomized to make sure that writers of test set are not from a single institution to ensure variability of the test set.

II. Loading Arabic Letters Dataset

There are 13440 training arabic images of 32x32 pixels and 3360 testing

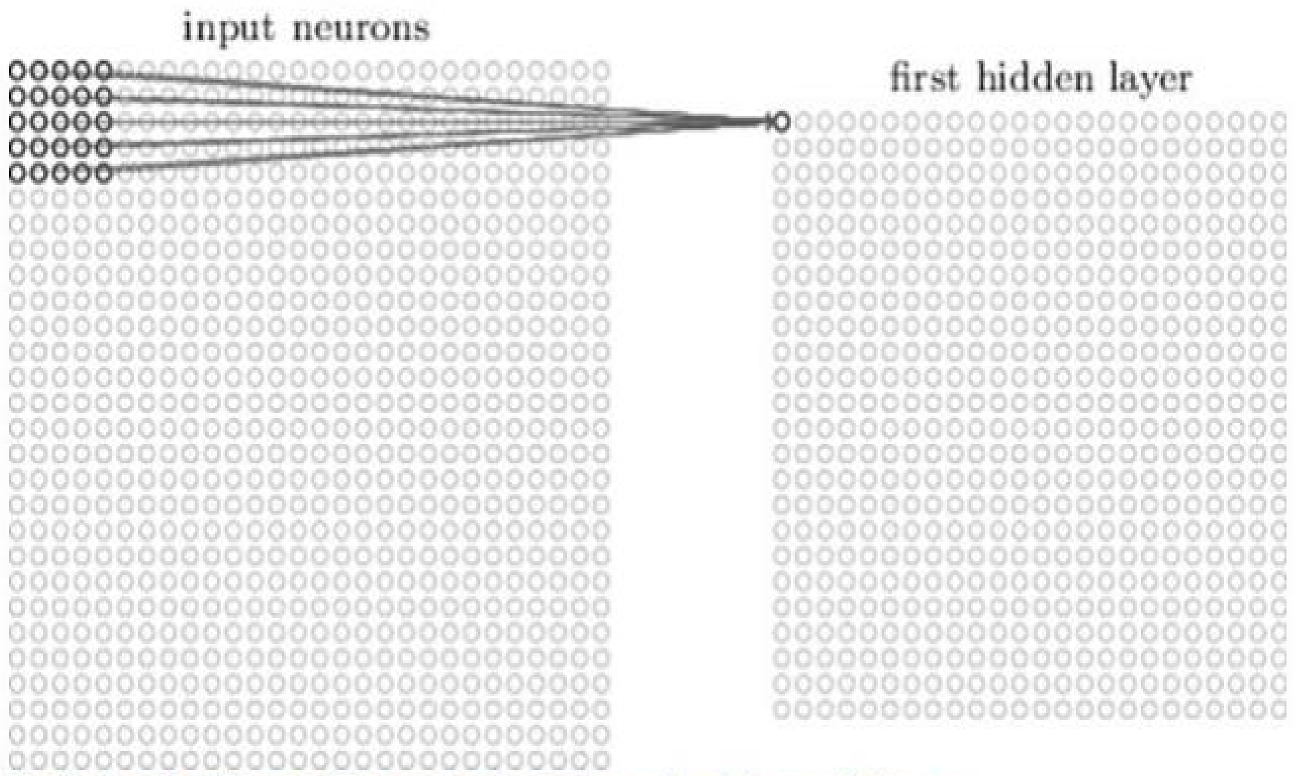
Algorithms and Techniques

We can use machine learning classification algorithms such as (SVM, Decision Tree, .. etc). But here we will use deep learning neural networks which often provide better results than ML algorithms especially when dealing with complex patterns like the images. So we will build a CNN model to able to recognize the images which high performance. We will discuss the model details and architecture in the following sections but let’s discuss some concepts like convolution, activation layers, dropout and optimizers. A more detailed overview of what CNNs do would be that you take the image, pass it through a series of convolutional, nonlinear, pooling (downsampling), and fully connected layers, and get an output. As we said earlier, the output can be a single class or a probability of classes that best describes the image. Now, the hard part is understanding what each of these layers do. So let’s get into the most important one.

What is Convolutional layer ?

The first layer in a CNN is a Convolutional Layer. Now, the best way to explain a conv layer is to imagine a flashlight that is shining over the top left of the image.
Let’s say that the light this flashlight shines covers a 5 x 5 area. And now, let’s imagine this flashlight sliding across all the areas of the input image. In machine learning terms, this flashlight is called a filter (or sometimes referred to as a neuron or a kernel) and the region that it is shining over is called the receptive field. Now this filter is also an array of numbers (the numbers are called weights or parameters). A very important note is that the depth of this filter has to be the same as the depth of the input (this makes sure that the math works out), so the dimensions of this filter is 5 x 5 x 3. Now, let’s take the first position the filter is in for example. It would be the top left corner. As the filter is sliding, or convolving, around the input image, it is multiplying the values in the filter with the original pixel values of the image (aka computing element wise multiplications). These multiplications are all summed up (mathematically speaking, this would be 75 multiplications in total). So now you have a single number. Remember, this number is just representative of when the filter is at the top left of the image. Now, we repeat this process for every location on the input volume. (Next step would be moving the filter to the right by 1 unit, then right again by 1, and so on). Every unique location on the input volume produces a number.

After sliding the filter over all the locations, you will find out that what you're left with is a 28 x 28 x 1 array of numbers, which we call an activation map or feature map. The reason you get a 28 x 28 array is that there are 784 different locations that a 5 x 5 filter can fit on a 32 x 32 input image. These 784 numbers are mapped to a 28 x 28 array.



Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

What is Activation layers ?

It’s just a node that you add to the output end of any neural network. It is also known as Transfer Function. It can also be attached in between two Neural Networks. It is used to determine the output of neural network like yes or no. It maps the resulting values in between 0 to 1 or -1 to 1 etc. (depending upon the function). There are many activation functions used such as relu, linear, sigmoid, softmax, etc

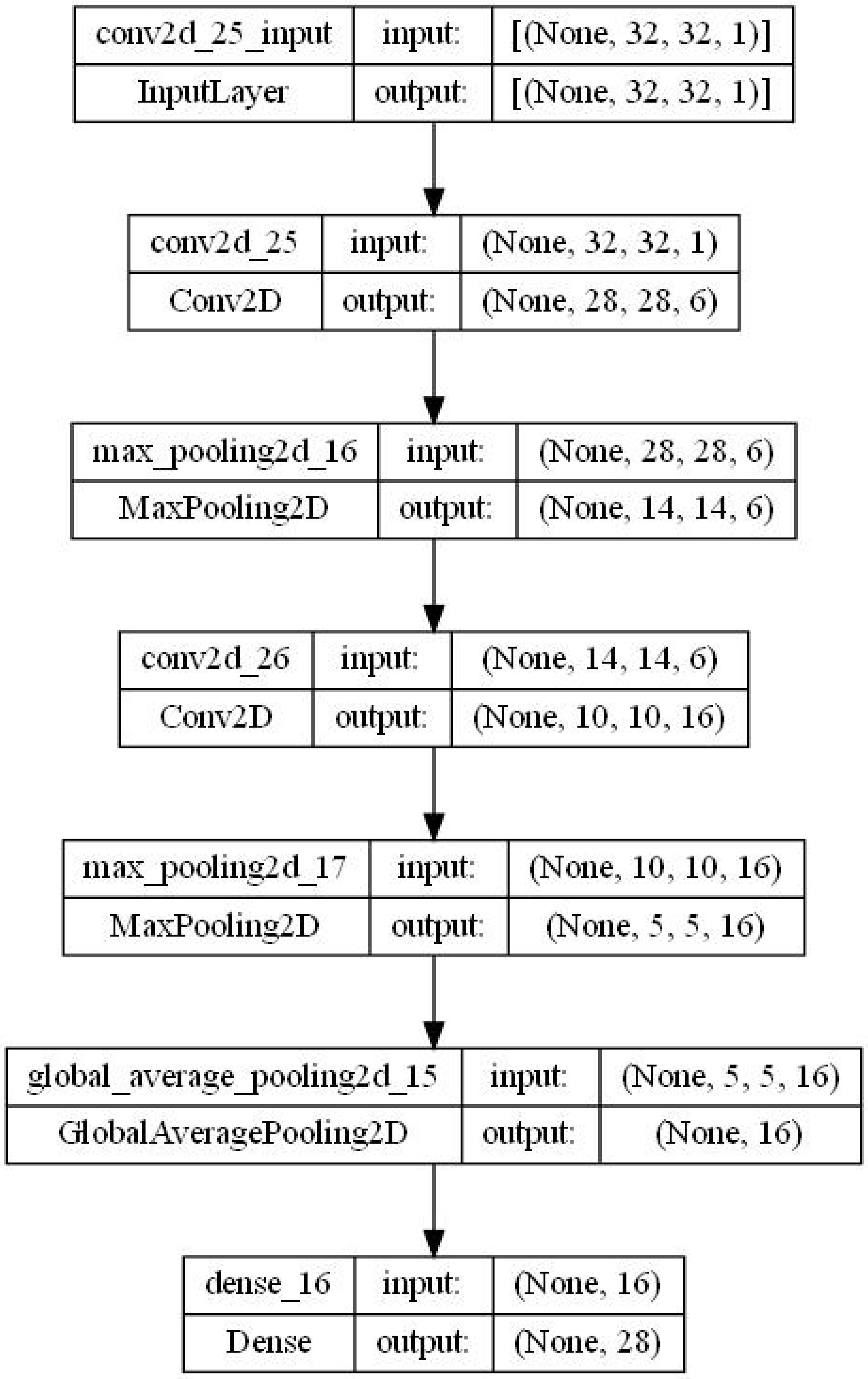
What is Dropout ?

Dropout is a regularization technique for reducing overfitting in neural networks by preventing complex co-adaptations on training data. The term "dropout" refers to dropping out units (both hidden and visible) in a neural network.

LetNET-5

We will use a very simple (vanilla) CNN model as benchmark and Train/test it using
the same data that you have used for our model solution.
Out Vanilla CNN will consist of:
-
Single Convolutional layer of 6 filters and window size of 5 and to capture the
basic patterns like edges from the input images.
And hidden layer of 16 filters and windoe size of 5

MaxPooling layer,pool size=2.
an activation function which is relu.
-
Single Pooling Layer to down-sample the input to enable the model to make
assumptions about the features so as to reduce overfitting. It also reduces the
number of parameters to learn, reducing the training time.
-
The last layer is the output layer with 28 neurons (number of output
classes) and it uses softmax activation function as we have multi-classes.
each neuron will give the probability of that class.
We will train our model using Adam Optimizer, cross entropy (Log loss) as loss
function
, batch size of 2 (to reduce training time and overfitting) and finally
using 10 epochs as we want a simple model just to capture the basic patterns


conv2d_25_input | InputLayer | input: [(None, 32, 32, 1)] | output: [(None, 32, 32, 1)]
conv2d_25 | Conv2D | input: (None, 32, 32, 1) | output: (None, 28, 28, 6)
max_pooling2d_16 | MaxPooling2D | input: (None, 28, 28, 6) | output: (None, 14, 14, 6)
conv2d_26 | Conv2D | input: (None, 14, 14, 6) | output: (None, 10, 10, 16)
max_pooling2d_17 | MaxPooling2D | input: (None, 10, 10, 16) | output: (None, 5, 5, 16)
global_average_pooling2d_15 | GlobalAveragePooling2D | input: (None, 5, 5, 16) | output: (None, 16)
dense_16 | Dense | input: (None, 16) | output: (None, 28)

Name of the dataset

Arabic Handwritten Character Recognition

Number of classes:

28

Total Number of Samples :

Tot:	16800 sample	
65%	Train:	10702 sample
16%	Cross Validation:	2688 sample
19%	Test:	3460 sample

Images Size :

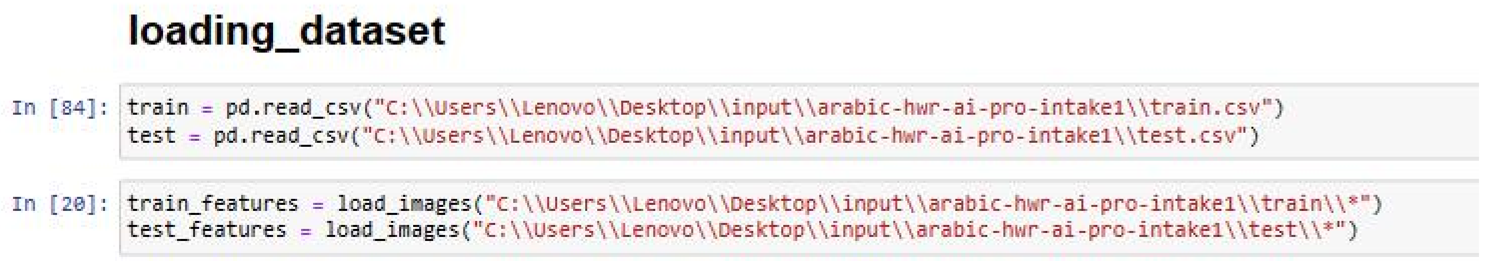
32*32*3

Dataset link :

[Arabic Handwritten Characters Recognition | Kaggle]

The First Step Preparing the data :

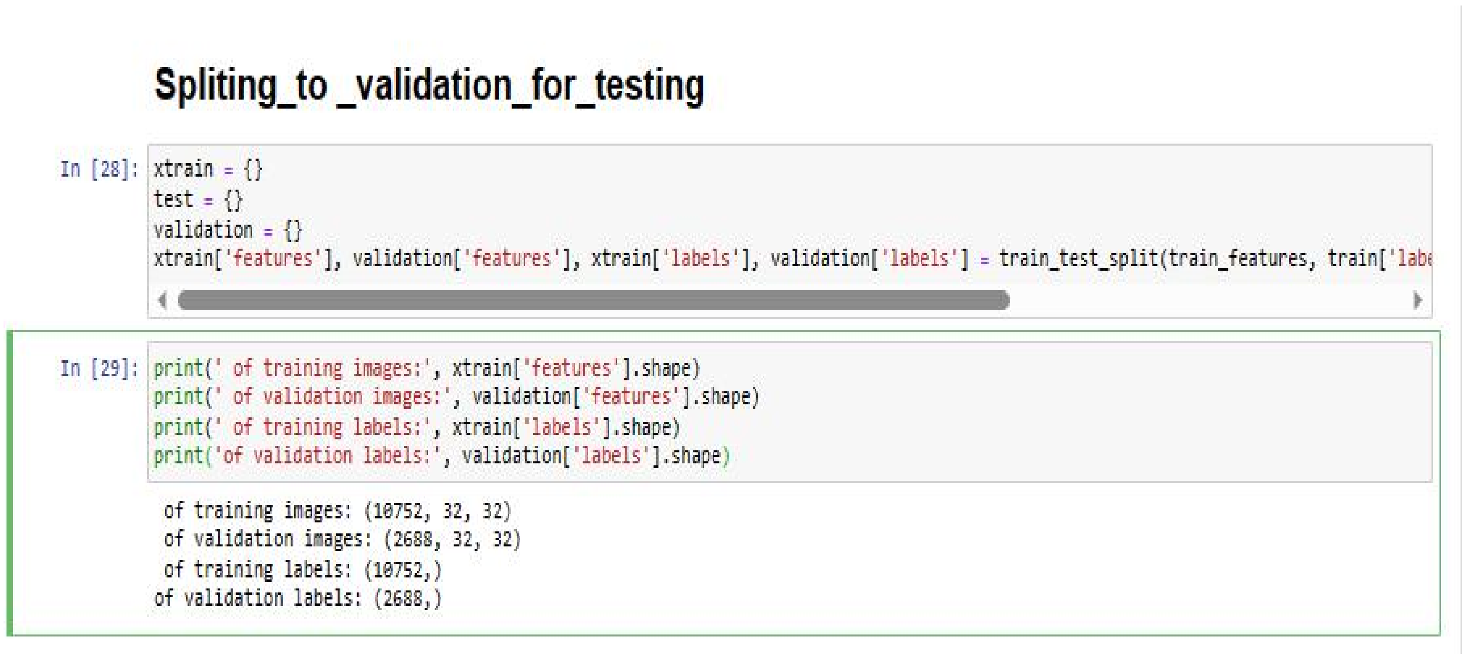
We define A variable train and validation to load the dataset then we split the data into features and target using .Iloc so we can put the features in X and the target in y.



loading_dataset

```
In [84]: train = pd.read_csv("C:\\Users\\Lenovo\\Desktop\\input\\arabic-hwr-ai-pro-intake1\\train.csv")
         test = pd.read_csv("C:\\Users\\Lenovo\\Desktop\\input\\arabic-hwr-ai-pro-intake1\\test.csv")
```

```
In [20]: train_features = load_images("C:\\Users\\Lenovo\\Desktop\\input\\arabic-hwr-ai-pro-intake1\\train\\*")
         test_features = load_images("C:\\Users\\Lenovo\\Desktop\\input\\arabic-hwr-ai-pro-intake1\\test\\*")
```

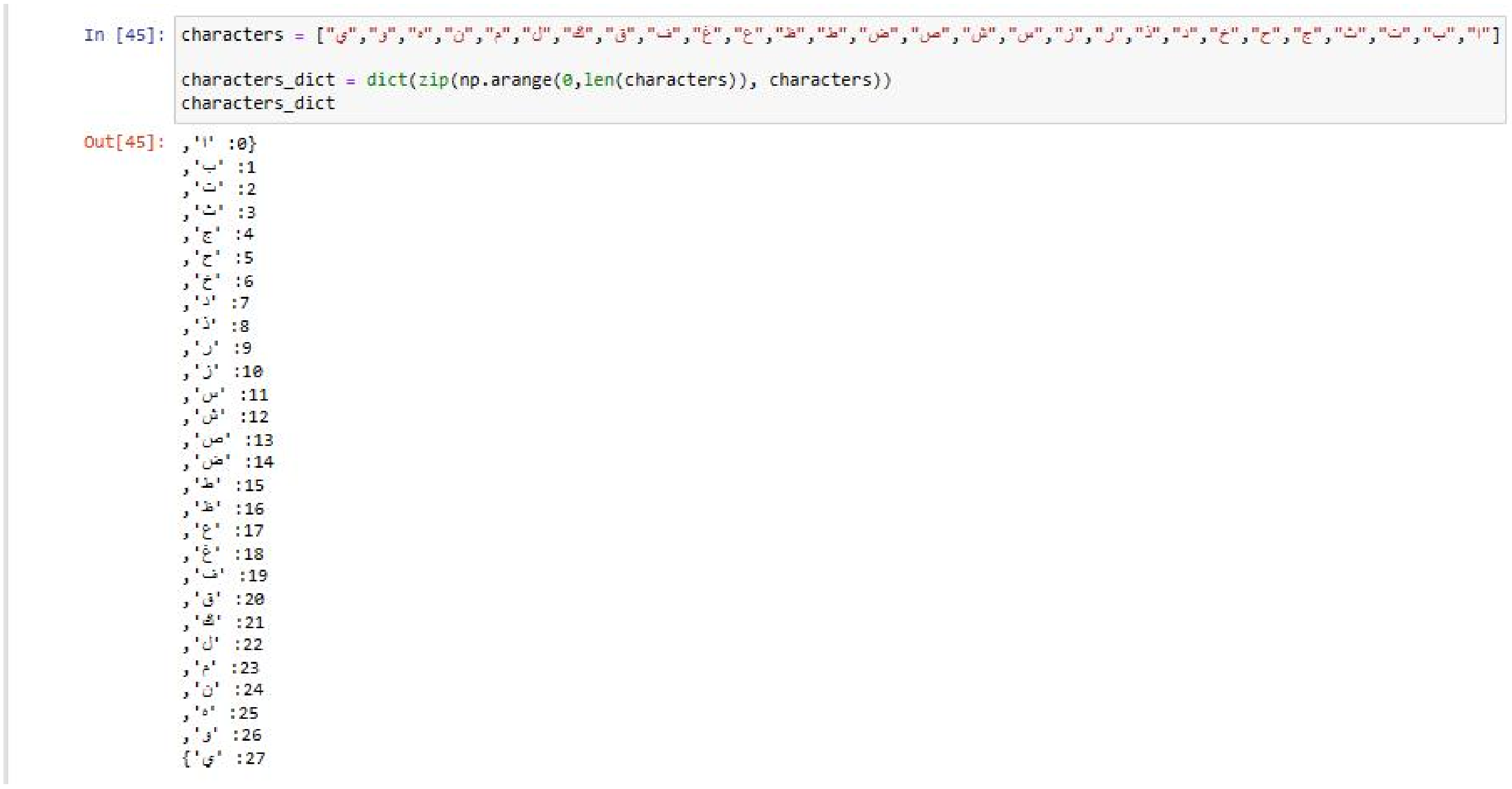


Spliting_to _validation_for_testing

```
In [28]: xtrain = {}
         test = {}
         validation = {}
         xtrain['features'], validation['features'], xtrain['labels'], validation['labels'] = train_test_split(train_features, train['labe
```

```
In [29]: print(' of training images:', xtrain['features'].shape)
         print(' of validation images:', validation['features'].shape)
         print(' of training labels:', xtrain['labels'].shape)
         print('of validation labels:', validation['labels'].shape)

          of training images: (10752, 32, 32)
          of validation images: (2688, 32, 32)
          of training labels: (10752,)
         of validation labels: (2688,)
```

Converting: we converted the output which was numbers representing the letters into actual letters so we can



```
In [45]: characters = ["ا","ب","ت","ث","ج","ح","خ","د","ذ","ر","ز","س","ش","ص","ض","ط","ظ","ع","غ","ف","ق","ك","ل","م","ن","ه","و","ي"]

         characters_dict = dict(zip(np.arange(0,len(characters)), characters))
         characters_dict
```

```
Out[45]: {0: 'ا',
          1: 'ب',
          2: 'ت',
          3: 'ث',
          4: 'ج',
          5: 'ح',
          6: 'خ',
          7: 'د',
          8: 'ذ',
          9: 'ر',
          10: 'ز',
          11: 'س',
          12: 'ش',
          13: 'ص',
          14: 'ض',
          15: 'ط',
          16: 'ظ',
          17: 'ع',
          18: 'غ',
          19: 'ف',
          20: 'ق',
          21: 'ك',
          22: 'ل',
          23: 'م',
          24: 'ن',
          25: 'ه',
          26: 'و',
          27: 'ي'}
```

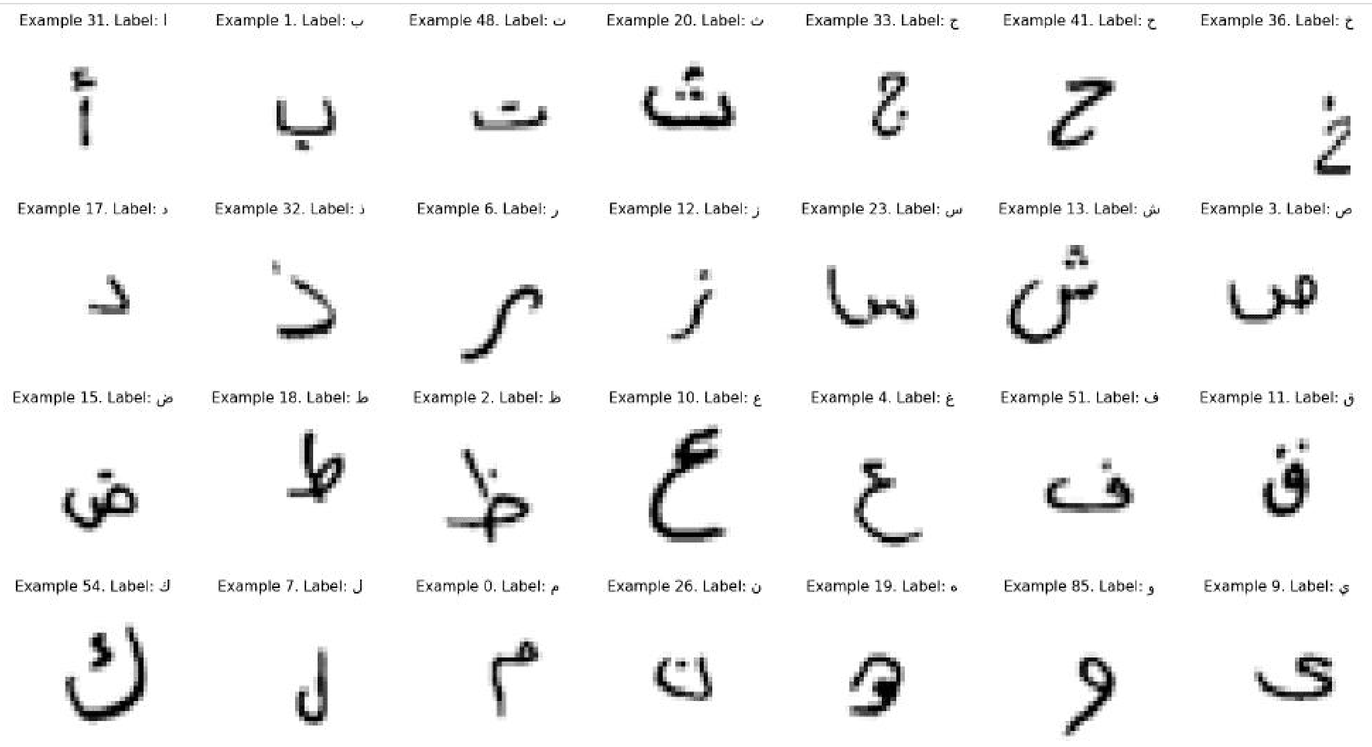
I.We visualise the classes samples

II.Then we divided the X_train and X_test by 255 to normalize them

by this the computation becomes easier and faster since all the numbers became in range of 0 and 1

III.Then we Reshaping Input Images

When using TensorFlow as backend, Keras CNNs require a 4D array (which we'll also refer to as a 4D tensor) as input, with shape
(nb_samples,rows,columns,channels)
where nb_samples corresponds to the total number of images (or samples), and rows, columns, and channels correspond to the number of rows, columns, and channels for each image, respectively.
So we will reshape the input images to a 4D tensor with following shape
(nb_samples, 32, 32 ,1) as we use grayscale images of 32x32 pixels.



Normalize_and_reshaping

```
In [30]: images_train = xtrain['features'].reshape((-1, 32, 32, 1))
         print("images shape: {}".format(images_train.shape))
         images_train = images_train/255

         images shape: (10752, 32, 32, 1)
```

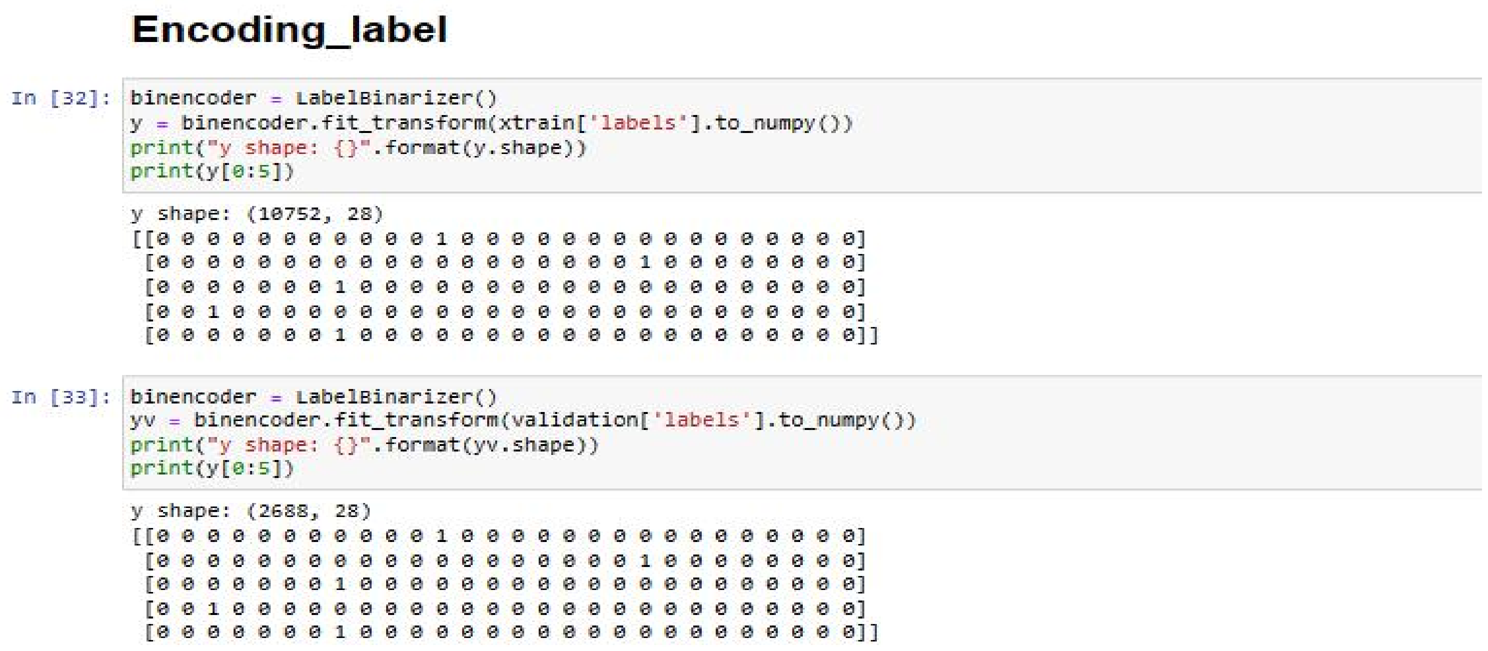
```
In [31]: image_val=validation['features'].reshape((-1, 32, 32, 1))
         print("images shape: {}".format(image_val.shape))
         images_val = image_val/255

         images shape: (2688, 32, 32, 1)
```

From the labels csv files we can see that labels are categorical values and it is a multi-class classification problem.

Our outputs are in the form of:

● Letters from ’alef’ to ’yeh’ have categories numbers from 0 to 27

Here we will encode these categories values using One Hot Encoding with keras. One-hot encoding transforms integer to a binary matrix where the array contains only one ‘1’ and the rest elements are ‘0’.

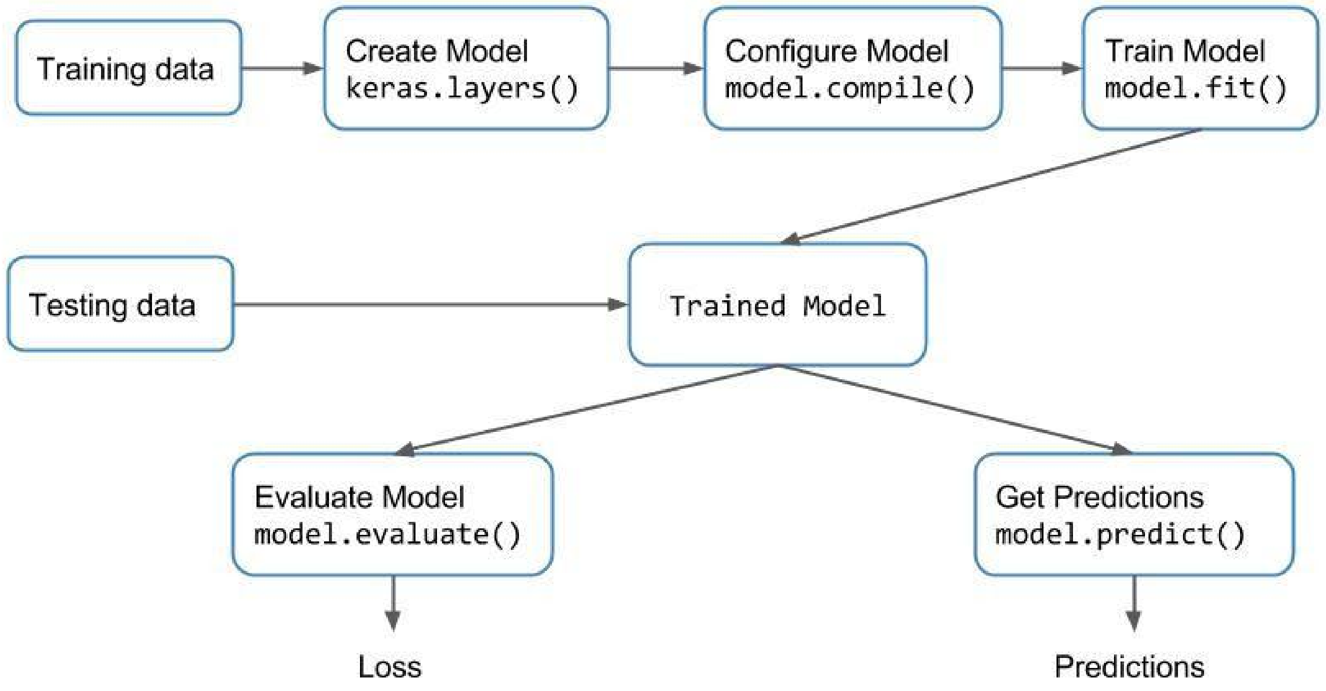


Encoding_label

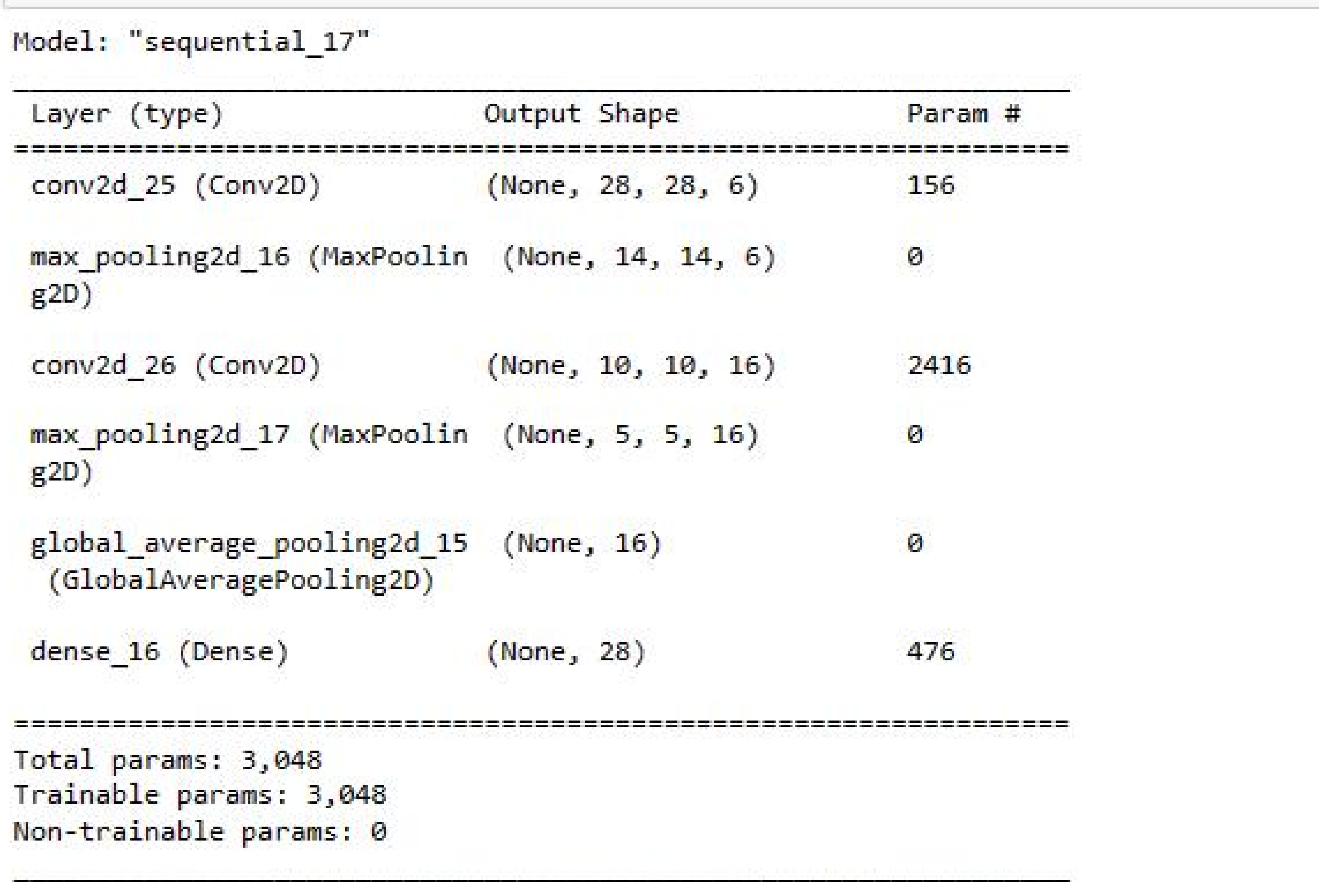
```
In [32]: binencoder = LabelBinarizer()
         y = binencoder.fit_transform(xtrain['labels'].to_numpy())
         print("y shape: {}".format(y.shape))
         print(y[0:5])

         y shape: (10752, 28)
         [[0 0 0 0 0 0 0 0 0 0 0 1 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 1 0 0 0 0 0 0 0 0]
          [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
          [0 0 1 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 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]

In [33]: binencoder = LabelBinarizer()
         yv = binencoder.fit_transform(validation['labels'].to_numpy())
         print("y shape: {}".format(yv.shape))
         print(y[0:5])

         y shape: (2688, 28)
         [[0 0 0 0 0 0 0 0 0 0 0 1 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 1 0 0 0 0 0 0 0 0]
          [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
          [0 0 1 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 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

Implementation

We will follow the following workflow to build our model and be ready to use it



We want to apply one of the primitive algorithms in CNN:



```
Model: "sequential_17"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_25 (Conv2D)          (None, 28, 28, 6)         156

 max_pooling2d_16 (MaxPoolin  (None, 14, 14, 6)        0
 g2D)

 conv2d_26 (Conv2D)          (None, 10, 10, 16)        2416

 max_pooling2d_17 (MaxPoolin  (None, 5, 5, 16)         0
 g2D)

 global_average_pooling2d_15  (None, 16)               0
  (GlobalAveragePooling2D)

 dense_16 (Dense)            (None, 28)                476

=================================================================
Total params: 3,048
Trainable params: 3,048
Non-trainable params: 0
_________________________________________________________________
```

LetNET-5

```
Epoch 1/10
6048/6048 [==============================] - 20s 3ms/step - loss: 2.9881 - accuracy: 0.1371 - val_loss: 2.7011 - val_accuracy:
0.2016
Epoch 2/10
6048/6048 [==============================] - 20s 3ms/step - loss: 2.4087 - accuracy: 0.2994 - val_loss: 2.2344 - val_accuracy:
0.3564
Epoch 3/10
6048/6048 [==============================] - 17s 3ms/step - loss: 2.0616 - accuracy: 0.4011 - val_loss: 1.9187 - val_accuracy:
0.4345
Epoch 4/10
6048/6048 [==============================] - 18s 3ms/step - loss: 1.8347 - accuracy: 0.4653 - val_loss: 1.7216 - val_accuracy:
0.5074
Epoch 5/10
6048/6048 [==============================] - 21s 3ms/step - loss: 1.6703 - accuracy: 0.5097 - val_loss: 1.5862 - val_accuracy:
0.5290
Epoch 6/10
6048/6048 [==============================] - 21s 3ms/step - loss: 1.5404 - accuracy: 0.5431 - val_loss: 1.5417 - val_accuracy:
0.5312
Epoch 7/10
6048/6048 [==============================] - 21s 3ms/step - loss: 1.4348 - accuracy: 0.5751 - val_loss: 1.3989 - val_accuracy:
0.5751
Epoch 8/10
6048/6048 [==============================] - 19s 3ms/step - loss: 1.3469 - accuracy: 0.5969 - val_loss: 1.3462 - val_accuracy:
0.5863
Epoch 9/10
6048/6048 [==============================] - 17s 3ms/step - loss: 1.2732 - accuracy: 0.6202 - val_loss: 1.2908 - val_accuracy:
0.5975
Epoch 10/10
6048/6048 [==============================] - 17s 3ms/step - loss: 1.2116 - accuracy: 0.6343 - val_loss: 1.2250 - val_accuracy:
0.6243
Baseline Model Test Accuracy: 0.651289701461792
Baseline Model Test Loss: 1.1665055751800537
```

Second Step Building the model:

1. Design Model Architecture

In this section we will discuss our CNN Model.

● The first hidden layer is a convolutional layer.

The layer has 32 feature maps, which with the size of 5×5 and an activation function which is relu. This is the input layer, expecting images with the structure outlined above.

● The second layer is Batch Normalization which solves having distributions of the features vary across the training and test data,

which breaks the IID assumption. We use it to help in two ways faster learning and higher overall accuracy.

● The third layer is the MaxPooling layer. MaxPooling layer is used to down-sample the input to enable the model to make assumptions about the features so as to reduce overfitting. It also reduces the number of parameters to learn, reducing the training time.

● The next layer is a Regularization layer using dropout. It is configured to randomly exclude 10% of neurons in the layer in order to reduce overfitting. 12

● Another hidden layer with 64 feature maps with the size of 3×3 and a relu activation function to capture more features from the image. ● Other hidden layers with 256 feature maps with the size of 3×3 and a relu activation function to capture complex patterns from the image which will describe the digits and letters later.

● More MaxPooling, Batch Normalization, Regularization and GlobalAveragePooling2D layers

.● The last layer is the output layer with 28 neurons (number of output classes) and it uses softmax activation function as we have multi-classes. Each neuron will give the probability of that class.

We used categorical_crossentropy as a loss function because its a multi-class classification problem. I also used accuracy as metrics to improve the performance of our neural network.

Notice the hyperparameters used for the model:

- Dropout rate: 10% of the layer nodes.

- Epochs: 10 then we will fit the model incrementally on 100 more epochs.

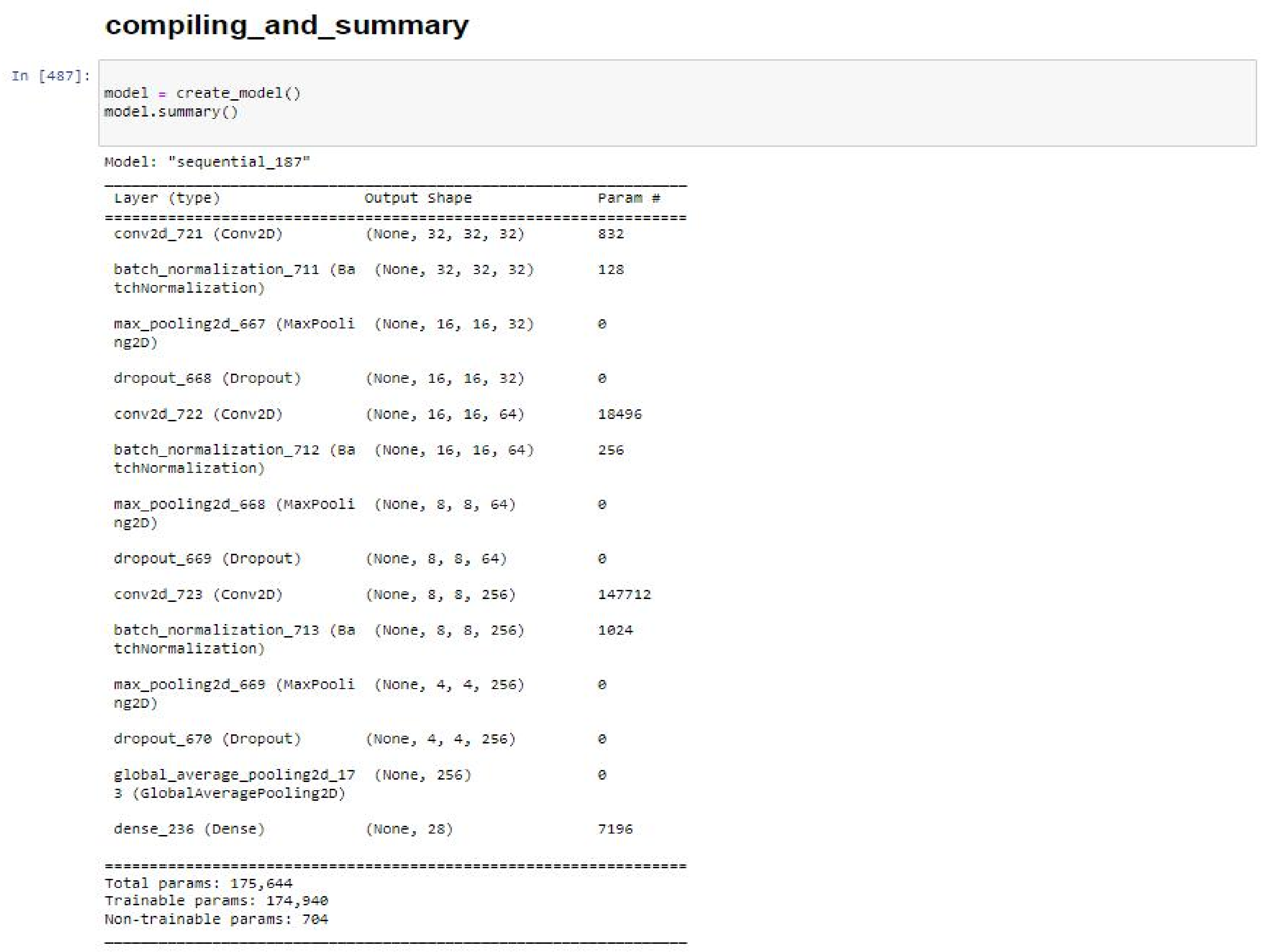
- Batch size: 20 as it is enough amount and divisible by the size of total training dataset and also the size of total testing dataset.

- Optimizer: After the refinement section we will see the best optimizer we will use is RMSprop.

- Activation Layer: After the refinement section we will see the best activation we will use is relu.

- Kernel initializer: After the refinement section we will see the best kernel initializer we will use is normal.

2. Model Summary

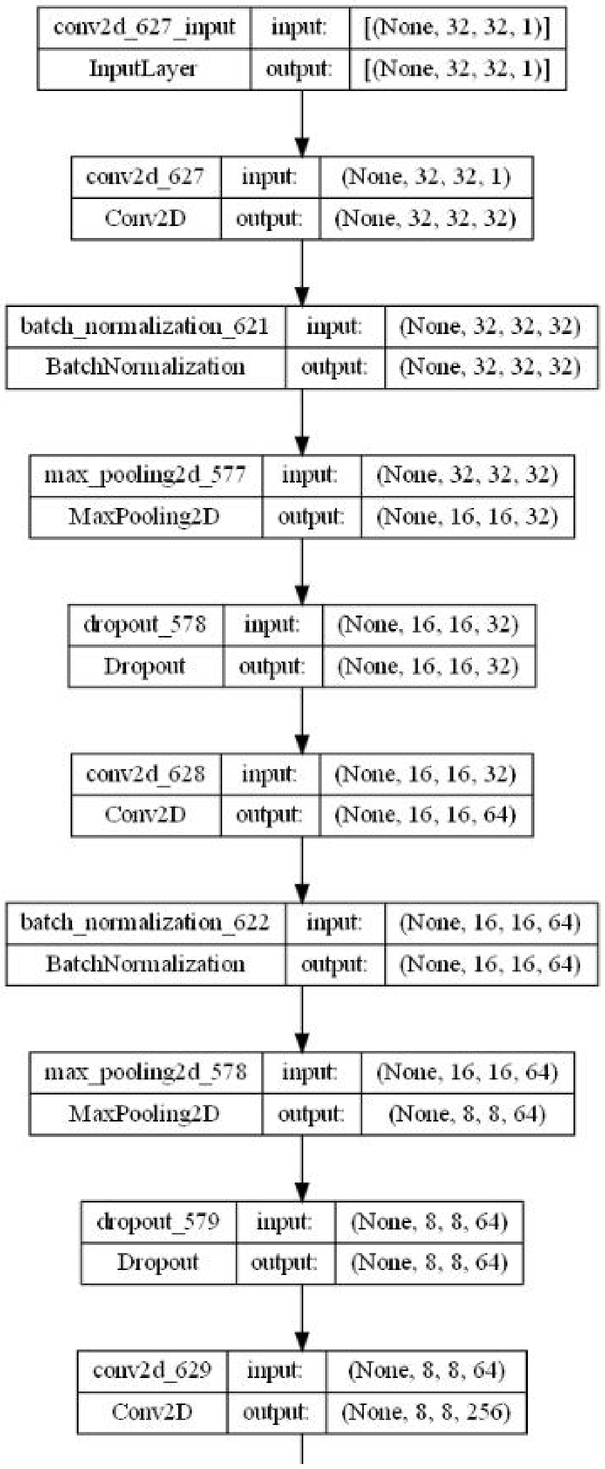


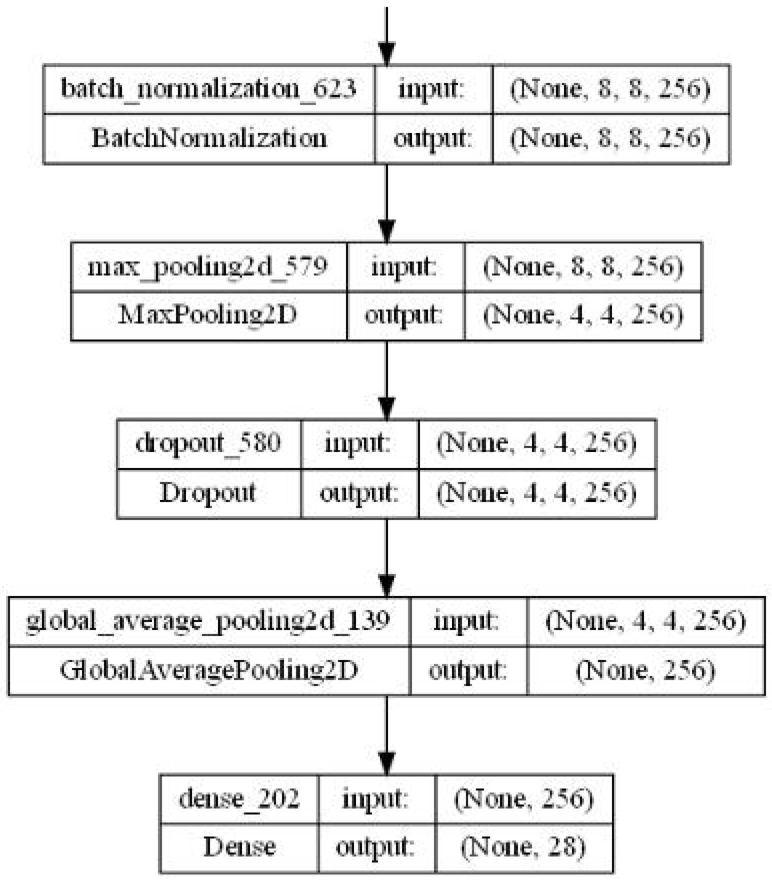
compiling_and_summary

In [487]:

```
model = create_model()
model.summary()
```

```
Model: "sequential_187"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_721 (Conv2D)         (None, 32, 32, 32)        832

 batch_normalization_711 (Ba  (None, 32, 32, 32)       128
 tchNormalization)

 max_pooling2d_667 (MaxPooli  (None, 16, 16, 32)       0
 ng2D)

 dropout_668 (Dropout)       (None, 16, 16, 32)        0

 conv2d_722 (Conv2D)         (None, 16, 16, 64)        18496

 batch_normalization_712 (Ba  (None, 16, 16, 64)       256
 tchNormalization)

 max_pooling2d_668 (MaxPooli  (None, 8, 8, 64)         0
 ng2D)

 dropout_669 (Dropout)       (None, 8, 8, 64)          0

 conv2d_723 (Conv2D)         (None, 8, 8, 256)         147712

 batch_normalization_713 (Ba  (None, 8, 8, 256)        1024
 tchNormalization)

 max_pooling2d_669 (MaxPooli  (None, 4, 4, 256)        0
 ng2D)

 dropout_670 (Dropout)       (None, 4, 4, 256)         0

 global_average_pooling2d_17  (None, 256)              0
 3 (GlobalAveragePooling2D)

 dense_236 (Dense)           (None, 28)                7196

=================================================================
Total params: 175,644
Trainable params: 174,940
Non-trainable params: 704
_________________________________________________________________
```


conv2d_627_input | input: | [(None, 32, 32, 1)]
InputLayer | output: | [(None, 32, 32, 1)]
conv2d_627 | input: | (None, 32, 32, 1)
Conv2D | output: | (None, 32, 32, 32)
batch_normalization_621 | input: | (None, 32, 32, 32)
BatchNormalization | output: | (None, 32, 32, 32)
max_pooling2d_577 | input: | (None, 32, 32, 32)
MaxPooling2D | output: | (None, 16, 16, 32)
dropout_578 | input: | (None, 16, 16, 32)
Dropout | output: | (None, 16, 16, 32)
conv2d_628 | input: | (None, 16, 16, 32)
Conv2D | output: | (None, 16, 16, 64)
batch_normalization_622 | input: | (None, 16, 16, 64)
BatchNormalization | output: | (None, 16, 16, 64)
max_pooling2d_578 | input: | (None, 16, 16, 64)
MaxPooling2D | output: | (None, 8, 8, 64)
dropout_579 | input: | (None, 8, 8, 64)
Dropout | output: | (None, 8, 8, 64)
conv2d_629 | input: | (None, 8, 8, 64)
Conv2D | output: | (None, 8, 8, 256)


batch_normalization_623
BatchNormalization
input:
output:
(None, 8, 8, 256)
(None, 8, 8, 256)
max_pooling2d_579
MaxPooling2D
input:
output:
(None, 8, 8, 256)
(None, 4, 4, 256)
dropout_580
Dropout
input:
output:
(None, 4, 4, 256)
(None, 4, 4, 256)
global_average_pooling2d_139
GlobalAveragePooling2D
input:
output:
(None, 4, 4, 256)
(None, 256)
dense_202
Dense
input:
output:
(None, 256)
(None, 28)

Refinement We will tune the parameters optimizer, kernel_initializer and activation function to see the effect of changing them on our model architecture.

We will run GridSearch on these parameters with the following possible values:

optimizer : ['RMSprop', 'Adam', 'Adagrad', 'Nadam']

kernel_initializer : ['normal', 'uniform']

activation : ['relu', 'linear', 'tanh']

Here is more details about the results of some parameters:

```
In [444]:
epochs = 5
batch_size = 20
for a,b,c in [(x,y,z) for x in optimizer for z in activation for y in kernel_initializer]:
    params = {'optimizer' : a , 'kernel_initializer' : b , 'activation' : c}
    print(params)
    curr_model = create_model(a, b, c)
    curr_model.fit(images_train, y,
                   validation_data=(images_val, yv),
                   epochs=epochs, batch_size=batch_size, verbose=1)
    print("==============================================================================")
```

```
{'optimizer': 'RMSprop', 'kernel_initializer': 'normal', 'activation': 'relu'}
Epoch 1/5
605/605 [==============================] - 14s 22ms/step - loss: 1.0745 - accuracy: 0.6905 - val_loss: 3.0755 - val_accuracy:
0.3817
Epoch 2/5
605/605 [==============================] - 13s 22ms/step - loss: 0.2999 - accuracy: 0.9110 - val_loss: 1.3495 - val_accuracy:
0.6280
Epoch 3/5
605/605 [==============================] - 13s 22ms/step - loss: 0.1951 - accuracy: 0.9391 - val_loss: 0.6479 - val_accuracy:
0.8051
Epoch 4/5
605/605 [==============================] - 13s 22ms/step - loss: 0.1471 - accuracy: 0.9540 - val_loss: 0.4729 - val_accuracy:
0.8616
Epoch 5/5
605/605 [==============================] - 14s 22ms/step - loss: 0.1234 - accuracy: 0.9611 - val_loss: 0.2464 - val_accuracy:
0.9219
==============================================================================
{'optimizer': 'RMSprop', 'kernel_initializer': 'uniform', 'activation': 'relu'}
Epoch 1/5
605/605 [==============================] - 15s 23ms/step - loss: 0.8978 - accuracy: 0.7418 - val_loss: 7.1076 - val_accuracy:
0.3028
Epoch 2/5
605/605 [==============================] - 14s 23ms/step - loss: 0.2519 - accuracy: 0.9269 - val_loss: 6.5737 - val_accuracy:
0.2961
Epoch 3/5
605/605 [==============================] - 14s 23ms/step - loss: 0.1696 - accuracy: 0.9501 - val_loss: 1.2450 - val_accuracy:
0.6168
Epoch 4/5
605/605 [==============================] - 14s 23ms/step - loss: 0.1312 - accuracy: 0.9597 - val_loss: 1.5237 - val_accuracy:
0.5818
Epoch 5/5
605/605 [==============================] - 14s 23ms/step - loss: 0.1108 - accuracy: 0.9657 - val_loss: 0.6295 - val_accuracy:
0.8192
==============================================================================
{'optimizer': 'RMSprop', 'kernel_initializer': 'normal', 'activation': 'linear'}
Epoch 1/5
605/605 [==============================] - 15s 24ms/step - loss: 1.8037 - accuracy: 0.4647 - val_loss: 3.8106 - val_accuracy:
0.1198
Epoch 2/5
605/605 [==============================] - 14s 24ms/step - loss: 0.9036 - accuracy: 0.7313 - val_loss: 1.2105 - val_accuracy:
0.6190
Epoch 3/5
605/605 [==============================] - 14s 24ms/step - loss: 0.6149 - accuracy: 0.8134 - val_loss: 1.9670 - val_accuracy:
0.5000
Epoch 4/5
605/605 [==============================] - 15s 24ms/step - loss: 0.4888 - accuracy: 0.8504 - val_loss: 1.4926 - val_accuracy:
0.5521
Epoch 5/5
605/605 [==============================] - 15s 25ms/step - loss: 0.3992 - accuracy: 0.8776 - val_loss: 2.2640 - val_accuracy:
0.4159
==============================================================================
```

From the obtained results we can see that best parameters are:

- **Optimizer: RMSprop**
- **Kernel_initializer: uniform**
- **Activation: relu**

Which will produce train accuracy of 0.9983%
and validation accuracy of 0.97545%

Results Model Evaluation and Validation

We will Train the model using batch_size=20 to reduce used memory and make the training more quick.

We will train the model first on 100 epochs to see the accuracy that we will obtain then we will increase number of epochs to be trained on to improve the accuracy.

Here are the results after 100 epochs.

```
0.9754
Epoch 86/100
605/605 [==============================] - ETA: 0s - loss: 0.0068 - accuracy: 0.9981
Epoch 86: val_accuracy did not improve from 0.97545
605/605 [==============================] - 20s 32ms/step - loss: 0.0068 - accuracy: 0.9981 - val_loss: 0.1933 - val_accuracy:
0.9643

Epoch 87/100
605/605 [==============================] - ETA: 0s - loss: 0.0073 - accuracy: 0.9972
Epoch 87: val_accuracy did not improve from 0.97545
605/605 [==============================] - 20s 32ms/step - loss: 0.0073 - accuracy: 0.9972 - val_loss: 0.1715 - val_accuracy:
0.9717
Epoch 88/100
605/605 [==============================] - ETA: 0s - loss: 0.0073 - accuracy: 0.9976
Epoch 88: val_accuracy did not improve from 0.97545
605/605 [==============================] - 19s 32ms/step - loss: 0.0073 - accuracy: 0.9976 - val_loss: 0.2230 - val_accuracy:
0.9665
Epoch 89/100
605/605 [==============================] - ETA: 0s - loss: 0.0067 - accuracy: 0.9973
Epoch 89: val_accuracy did not improve from 0.97545
605/605 [==============================] - 19s 32ms/step - loss: 0.0067 - accuracy: 0.9973 - val_loss: 0.1895 - val_accuracy:
0.9658
Epoch 90/100
605/605 [==============================] - ETA: 0s - loss: 0.0080 - accuracy: 0.9976
Epoch 90: val_accuracy did not improve from 0.97545
605/605 [==============================] - 19s 32ms/step - loss: 0.0080 - accuracy: 0.9976 - val_loss: 0.1509 - val_accuracy:
0.9695
Epoch 91/100
605/605 [==============================] - ETA: 0s - loss: 0.0073 - accuracy: 0.9980
Epoch 91: val_accuracy did not improve from 0.97545
605/605 [==============================] - 19s 32ms/step - loss: 0.0073 - accuracy: 0.9980 - val_loss: 0.2122 - val_accuracy:
0.9680
Epoch 92/100
605/605 [==============================] - ETA: 0s - loss: 0.0084 - accuracy: 0.9974
Epoch 92: val_accuracy did not improve from 0.97545
605/605 [==============================] - 19s 32ms/step - loss: 0.0084 - accuracy: 0.9974 - val_loss: 0.2621 - val_accuracy:
0.9621
Epoch 93/100
605/605 [==============================] - ETA: 0s - loss: 0.0058 - accuracy: 0.9981
Epoch 93: val_accuracy did not improve from 0.97545
605/605 [==============================] - 19s 32ms/step - loss: 0.0058 - accuracy: 0.9981 - val_loss: 1.7248 - val_accuracy:
0.7738
Epoch 94/100
605/605 [==============================] - ETA: 0s - loss: 0.0068 - accuracy: 0.9974
Epoch 94: val_accuracy did not improve from 0.97545
605/605 [==============================] - 19s 32ms/step - loss: 0.0068 - accuracy: 0.9974 - val_loss: 0.1727 - val_accuracy:
0.9732
Epoch 95/100
605/605 [==============================] - ETA: 0s - loss: 0.0055 - accuracy: 0.9981
Epoch 95: val_accuracy did not improve from 0.97545
605/605 [==============================] - 20s 32ms/step - loss: 0.0055 - accuracy: 0.9981 - val_loss: 0.3608 - val_accuracy:
0.9412
Epoch 96/100
605/605 [==============================] - ETA: 0s - loss: 0.0055 - accuracy: 0.9984
Epoch 96: val_accuracy did not improve from 0.97545
605/605 [==============================] - 19s 32ms/step - loss: 0.0055 - accuracy: 0.9984 - val_loss: 0.1860 - val_accuracy:
0.9606
Epoch 97/100
605/605 [==============================] - ETA: 0s - loss: 0.0059 - accuracy: 0.9979
Epoch 97: val_accuracy did not improve from 0.97545
605/605 [==============================] - 19s 32ms/step - loss: 0.0059 - accuracy: 0.9979 - val_loss: 0.1881 - val_accuracy:
0.9725
```

method that is based on adaptive estimation of first-order and second-order moments.

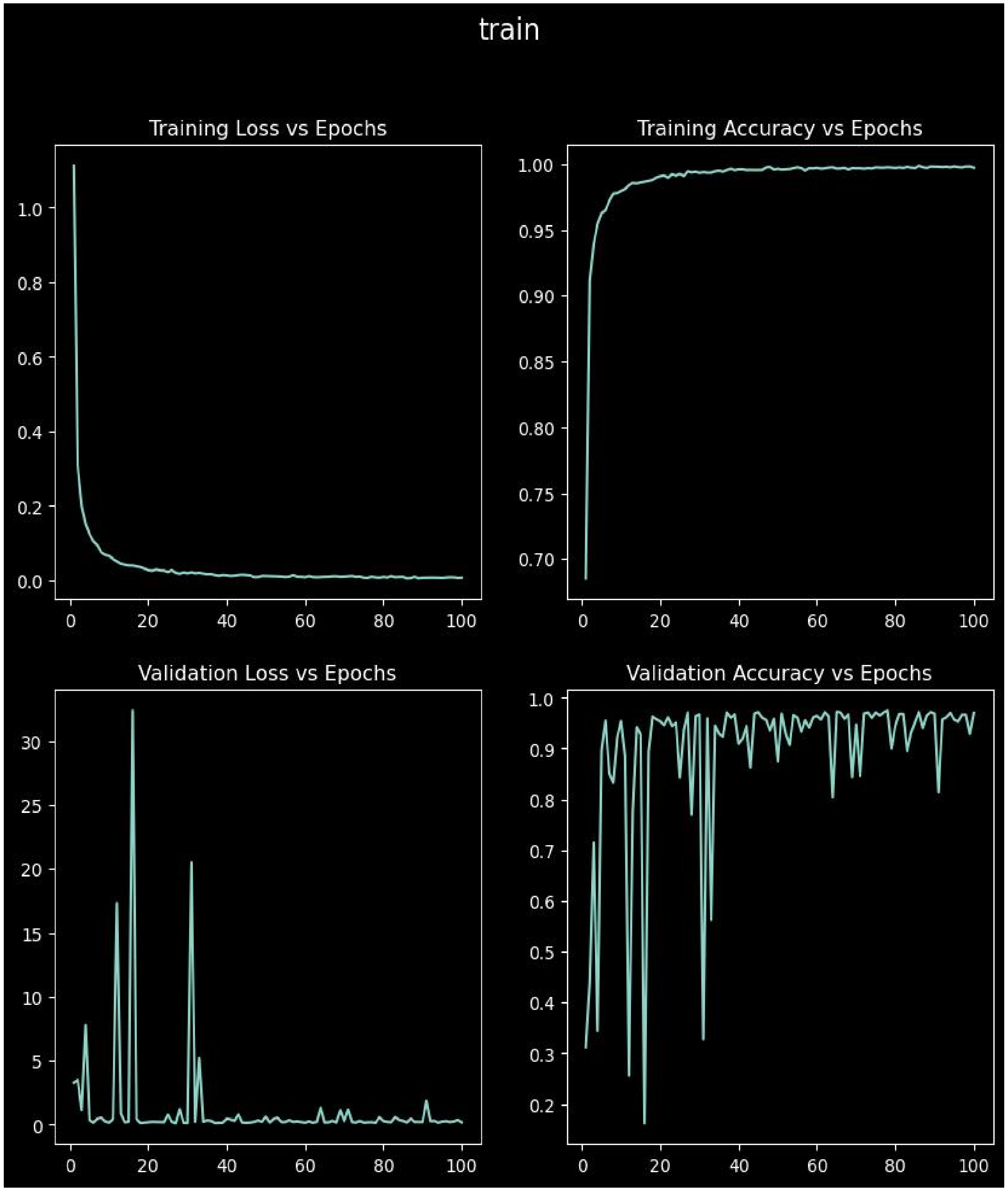
The optimizer generates the hyperparameters and reached the best values for the best

accuracy, save them in weights2.h5 verbose is 1 to see the output progress bar while saving the weights.

Let’s Plot loss and accuracy curves with epochs to see the changes easily

We can see that we may encounter overfitting if we continue to increase the number of epochs.

So we will stop fitting the model at that point as we have already got very satisfied score on the testindataset (we got test accuracy of 97.545%).


train
Training Loss vs Epochs
Training Accuracy vs Epochs
Validation Loss vs Epochs
Validation Accuracy vs Epochs

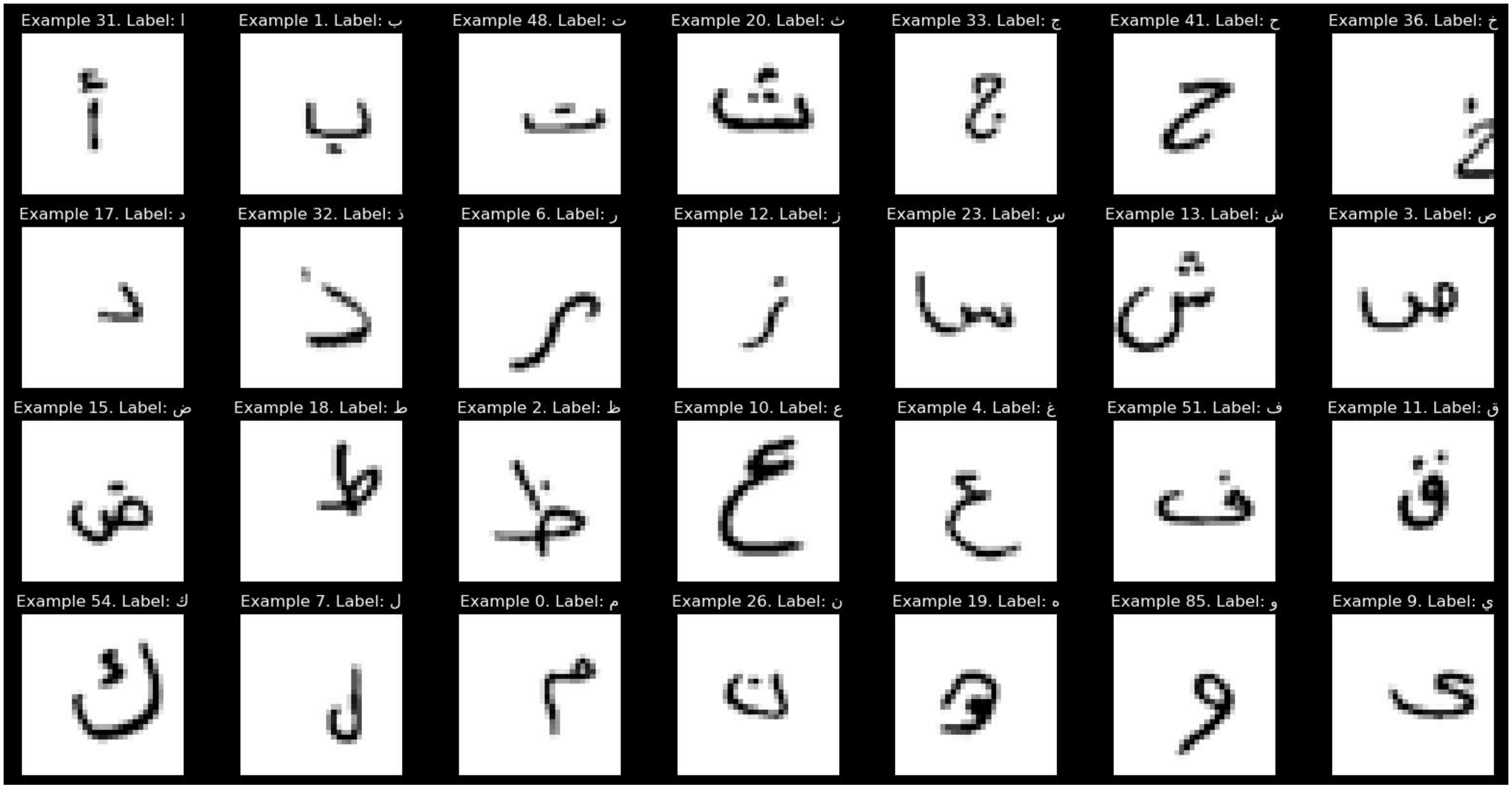
model validation Accuracy

Metric	Final Model	LetNET-5
Accuracy	97.745%	65%
Loss value	0.1881	1.116

The above results represent the average per class.

For more details about each single class, Find below detailed table results.

In test images we want to evaluate this model and Final model


Example 31. Label: ا
Example 1. Label: ب
Example 48. Label: ت
Example 20. Label: ث
Example 33. Label: ج
Example 41. Label: ح
Example 36. Label: خ
Example 17. Label: د
Example 32. Label: ذ
Example 6. Label: ر
Example 12. Label: ز
Example 23. Label: س
Example 13. Label: ش
Example 3. Label: ص
Example 15. Label: ض
Example 18. Label: ط
Example 2. Label: ظ
Example 10. Label: ع
Example 4. Label: غ
Example 51. Label: ف
Example 11. Label: ق
Example 54. Label: ك
Example 7. Label: ل
Example 0. Label: م
Example 26. Label: ن
Example 19. Label: ه
Example 85. Label: و
Example 9. Label: ي

```
In [436]: images_test = test_features.reshape((-1, 32, 32, 1))
          print("images shape: {}".format(images_test.shape))
          images_test = images_test/255

          images shape: (3360, 32, 32, 1)
```

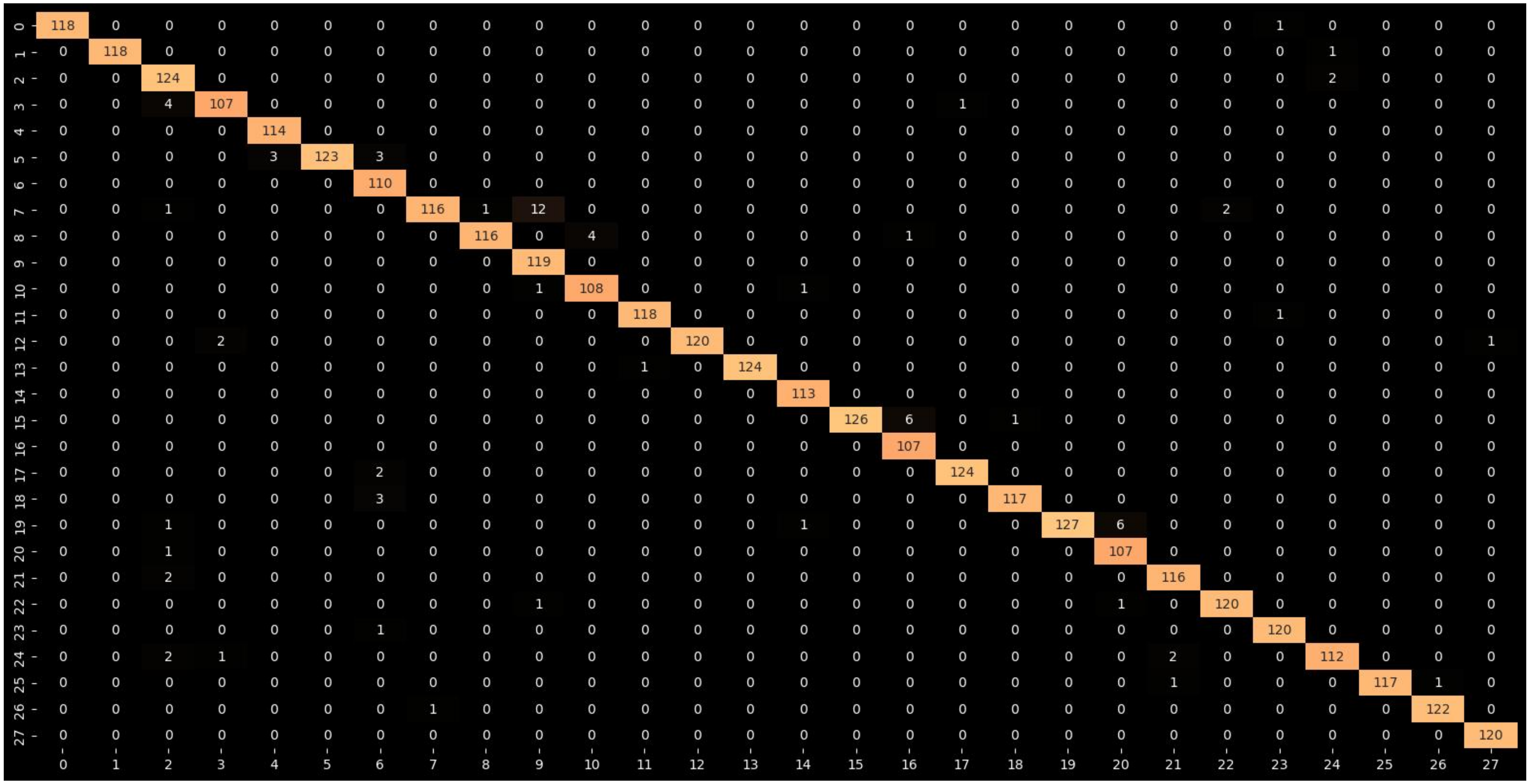
```
In [438]: yt = binencoder.fit_transform(test['label'].to_numpy())
          print("y shape: {}".format(yt.shape))
          print(y[0:5])

          y shape: (3360, 28)
          [[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 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 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
           [0 0 0 0 0 0 0 1 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 1 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 1 0 0 0 0 0 0 0]]
```

ACC_VALIDTION EVALUTE

```
In [525]: model.evaluate(images_test, yt)

          105/105 [==============================] - 1s 8ms/step - loss: 0.1120 - accuracy: 0.9771

Out[525]: [0.11196564137935638, 0.9770833253860474]
```

Each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class, or vice versa – both variants are found in the literature.

The name stems from the fact that it makes it easy to see whether the system is confusing two classes (i.e. commonly mislabeling one as another).



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
0	118	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	0	118	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
2	0	0	124	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0
3	0	0	4	107	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	114	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	3	123	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	1	0	0	0	0	116	1	12	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0
8	0	0	0	0	0	0	0	0	116	0	4	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	119	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	1	108	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	118	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
12	0	0	0	2	0	0	0	0	0	0	0	0	120	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
13	0	0	0	0	0	0	0	0	0	0	0	1	0	124	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	113	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	126	6	0	1	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	107	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	124	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	117	0	0	0	0	0	0	0	0	0
19	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	127	6	0	0	0	0	0	0	0
20	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	107	0	0	0	0	0	0	0
21	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	116	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	120	0	0	0	0	0
23	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	120	0	0	0	0
24	0	0	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	112	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	117	1	0
26	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	122	0
27	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	120

We will use the following metrics (Accuracy, Precision, Recall and F1-score). Log loss might also be a practical metric to be used when we tune/refine our model solution, So we will use Log loss as well. Let's study each metric carefully with our problem.

1. Accuracy: Accuracy is the most intuitive performance measure and it is simply a ratio of correctly predicted observation to the total observations. We can use it here if we don't care about misclassification of letter or digit specially.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

2. Precision - Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. We use it here to evaluate false positive rate As High precision relates to the low false positive rate.

$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

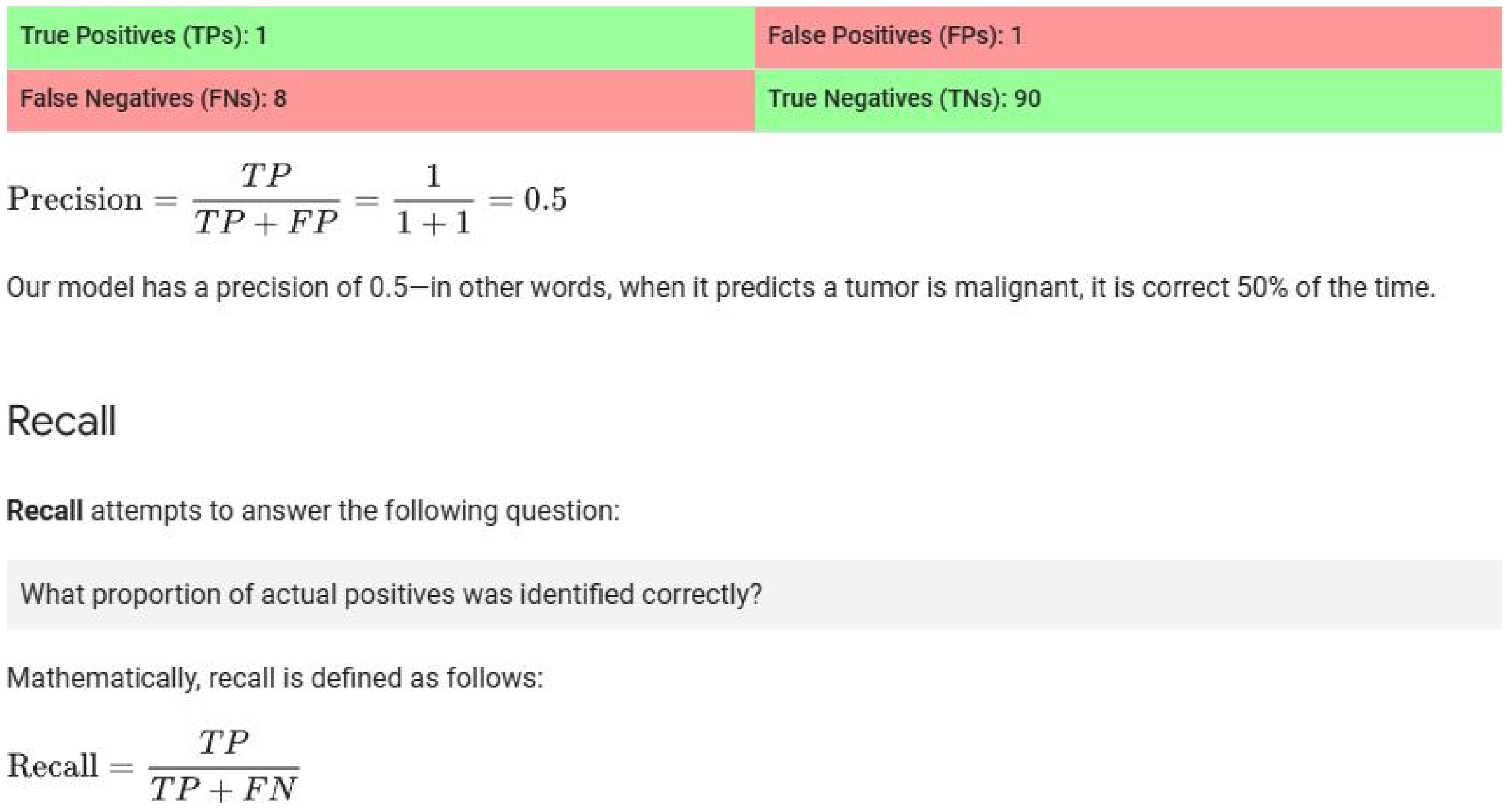
3. Recall (Sensitivity) - Recall is the ratio of correctly predicted positive observations to the all observations in actual class. We use it to select our best model when there is a high cost associated with False Negative/misclassified image.

$$\text{Recall} = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

4. F1 score - F1 Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. F1 is usually more useful than accuracy, especially if you have an uneven class distribution. Accuracy works best if false positives and false negatives have similar cost. If the cost of false

positives and false negatives are very different, it’s better to look at both Precision and Recall which means using F1 score.

$$F1 = 2 \times \frac{Precision * Recall}{Precision + Recall}$$



True Positives (TPs): 1	False Positives (FPs): 1
False Negatives (FNs): 8	True Negatives (TNs): 90

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{1}{1+1} = 0.5$$

Our model has a precision of 0.5—in other words, when it predicts a tumor is malignant, it is correct 50% of the time.

Recall

Recall attempts to answer the following question:

What proportion of actual positives was identified correctly?

Mathematically, recall is defined as follows:

$$\text{Recall} = \frac{TP}{TP + FN}$$

Reporting_Accuracy

In [526]:
```
y_pred = model.predict(images_test)

pred = np.argmax(y_pred, axis=1) + 1
ground = np.argmax(yt, axis=1) + 1

print(classification_report(ground,pred))
```

```
105/105 [==============================] - 1s 8ms/step
              precision    recall  f1-score   support

           1       1.00      0.99      1.00       119
           2       1.00      0.99      1.00       119
           3       0.92      0.98      0.95       126
           4       0.97      0.96      0.96       112
           5       0.97      1.00      0.99       114
           6       1.00      0.95      0.98       129
           7       0.92      1.00      0.96       110
           8       0.99      0.88      0.93       132
           9       0.99      0.96      0.97       121
          10       0.89      1.00      0.94       119
          11       0.96      0.98      0.97       110
          12       0.99      0.99      0.99       119
          13       1.00      0.98      0.99       123
          14       1.00      0.99      1.00       125
          15       0.98      1.00      0.99       113
          16       1.00      0.95      0.97       133
          17       0.94      1.00      0.97       107
          18       0.99      0.98      0.99       126
          19       0.99      0.97      0.98       120
          20       1.00      0.94      0.97       135
          21       0.94      0.99      0.96       108
          22       0.97      0.98      0.98       118
          23       0.98      0.98      0.98       122
          24       0.98      0.99      0.99       121
          25       0.97      0.96      0.97       117
          26       1.00      0.98      0.99       119
          27       0.99      0.99      0.99       123
          28       0.99      1.00      1.00       120

    accuracy                           0.98      3360
   macro avg       0.98      0.98      0.98      3360
weighted avg       0.98      0.98      0.98      3360
```

A ROC curve is constructed by plotting the true positive rate (TPR) against the false positive rate (FPR). The true positive rate is the proportion of observations that were correctly predicted to be positive out of all positive observations (TP/(TP + FN)). Similarly, the false positive rate is the proportion of observations that are incorrectly predicted to be positive out of all negative observations (FP/(TN + FP)). For example, in medical testing, the true positive rate is the rate in which people are correctly identified to test positive for the disease in question.

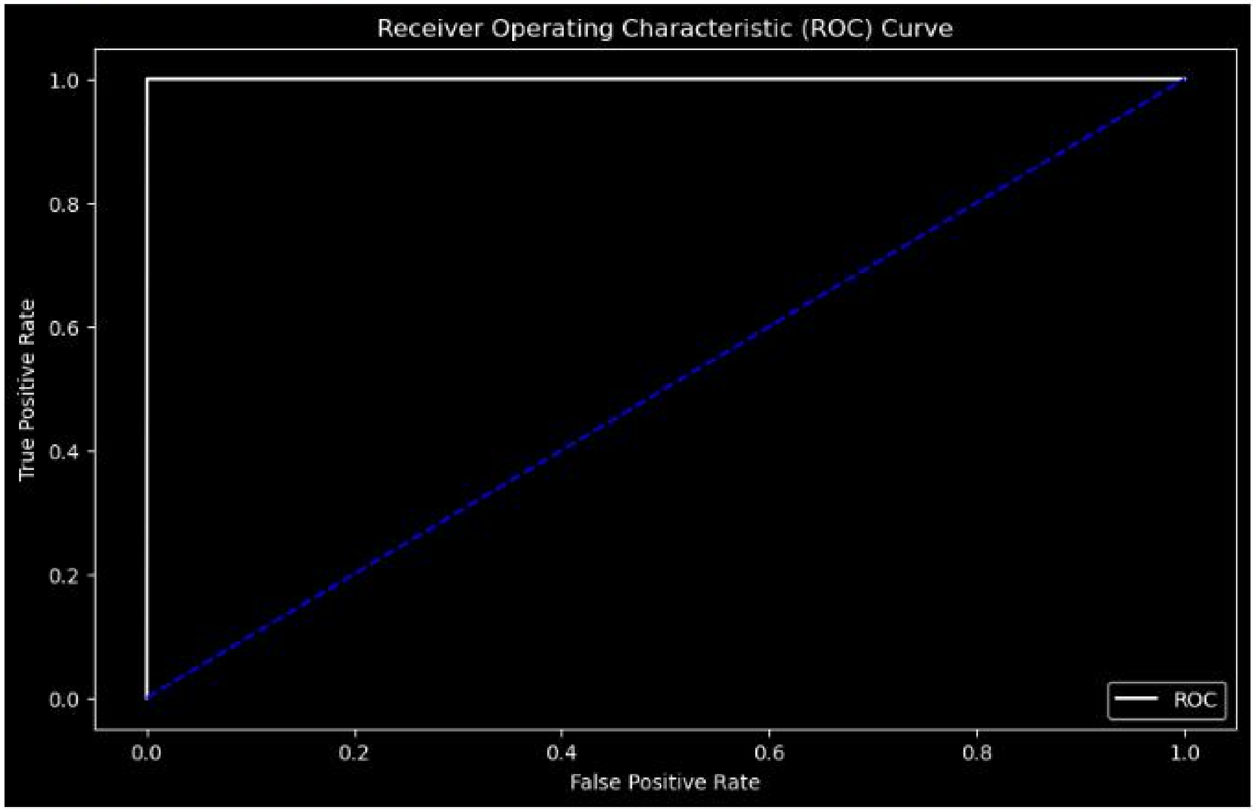
The ROC curve shows the trade-off between sensitivity (or TPR) and specificity (1 – FPR). Classifiers that give curves closer to the top-left corner indicate a better performance. As a baseline, a random classifier is expected to give points lying along the diagonal (FPR = TPR). The closer the curve comes to the 45-degree diagonal of the ROC space, the less accurate the test.

ACC

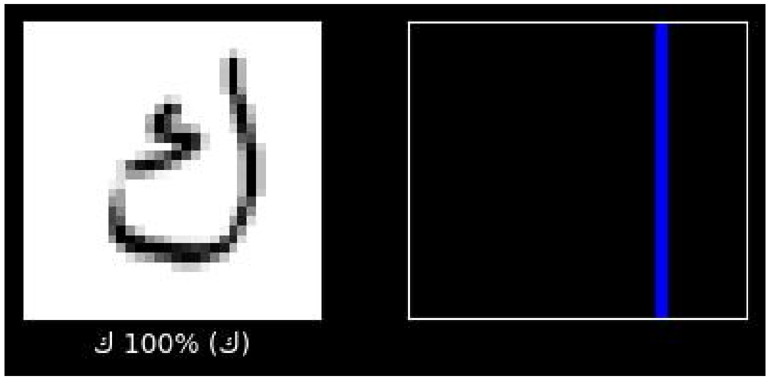
```
In [480]: fpr_keras, tpr_keras, thresholds = roc_curve(ground.ravel(), pred.ravel(),pos_label=28)
          auc_keras = auc(fpr_keras, tpr_keras)
          auc_keras

Out[480]: 0.9998456790123457
```

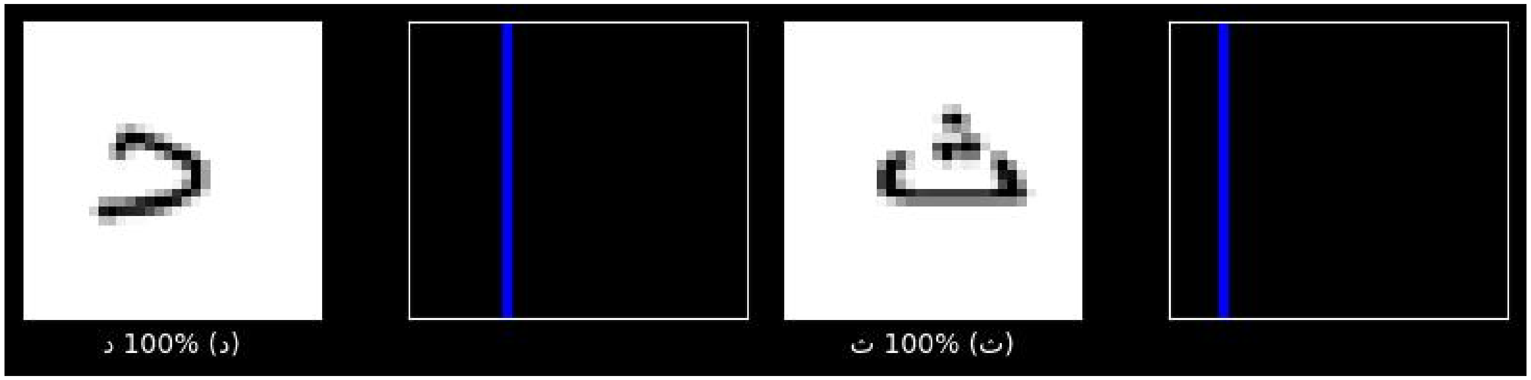
ROC_Curve

```
In [482]: plot_roc_curve(fpr_keras, tpr_keras)
```



Let's try the model to see some pictures and how to expect them





Reflection

As we discussed in the problem statement section we are trying to solve the Arabic handwritten image recognition. The domain of handwriting in the Arabic script presents unique technical challenges and has been addressed more recently than other domains. Many different methods have been proposed and applied to various types of images but here we will solve the problem from another side using CNN.

We face several challenges here starting from necessary including many variations in human handwriting so we took care of it using large variations dataset, then preprocessing the images of different size by reshaping them followed by image normalization to make all the values almost of equal importance. Then the big challenges of choosing an appropriate model to be able to capture the complex patterns from the images to classify them correctly with the minimum misclassification errors. Last but not least we tune the model hyperparameters which can affect the model significantly so we did our best to choose the best hyperparameters values to make our model optimal with the best performance.

As a result of that hard work I got very satisfied test accuracy of 97.7% which is much better than my expectation and crushed the benchmark model with large difference.

Improvement

As we see the final model captured almost all the dataset, So if we want to improve it we should include more datasets which have more variations in the handwritten style although i think we already included much variations in our dataset. As another improvement I think we can use that model as benchmark to solve more complex which is recognise Arabic handwritten words instead of single character.