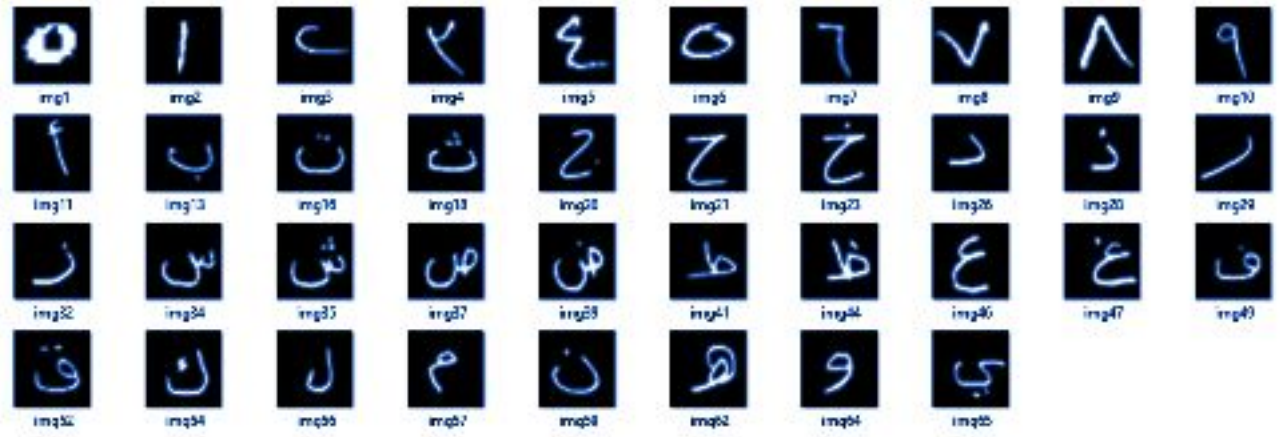
Arabic Handwritten Recognition



Machine Learning Nanodegree Capstone Project

Amr Hendy | September 21, 2018

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 and digits recognition that face several challenges, including the unlimited variation in human handwriting and the large public databases.
In this project we will use the public datasets **Arabic Digits** and **Arabic Letters** for arabic letters and 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 or digit. 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 **here**
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.

Metrics

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.

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

5. **Log loss** - Logarithmic loss measures the performance of a classification model where the prediction input is a probability value between 0 and 1. The goal of our machine learning models is to minimize this value. A perfect model would have a log loss of 0. Log loss increases as the predicted probability diverges from the actual label. We will use it to take into account the uncertainty of your prediction based on how much it varies from the actual label. This gives us a more nuanced view into the performance of our model.
   It is calculated as following:

$$F = -\frac{1}{N}\sum_{i}^{N}\sum_{j}^{M} y_{ij} \cdot Ln(p_{ij})) = \sum_{j}^{M}\left(-\frac{1}{N}\sum_{i}^{N} y_{ij} \cdot Ln(p_{ij}))\right)$$

where N is the number of instances, M is the number of different labels, y_{ij} is the binary variable with the expected labels and p_{ij} is the classificiation probability output by the classifier for the i-instance and the j-label.

Analysis

Data Exploration

I. Datasets and Inputs

We will use these datasets **Arabic Digits** and **Arabic Letters** for arabic letters and arabic digits respectively.
All the datasets are csv files representing the image pixels values and their corresponding label.

Here are some more details about the datasets:

1. Arabic Digits Dataset represents **MADBase** (modified Arabic handwritten digits database) which contains **60,000 training images**, and **10,000 test images**. MADBase was **written by 700 writers**. Each writer wrote each digit (from 0 -9) ten times. To ensure including different writing styles, the database was gathered from different institutions: Colleges of Engineering and Law, School of Medicine, the Open University (whose students span a wide range of ages), a high school, and a governmental institution.
   MADBase is available for free and can be downloaded from **here**.

2. 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 digit images of 64x64 pixels and 3360 testing arabic digit images of 64x64 pixels.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**...**	**4086**	**4087**	**4088**	**4089**	**4090**	**4091**	**4092**	**4093**	**4094**	**4095**	
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
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

III. Loading Arabic Digits Dataset

There are 60000 training arabic digit images of 64x64 pixels and 10000 testing arabic digit images of 64x64 pixels.

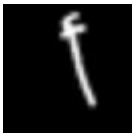
0	1	2	3	4	5	6	7	8	9	...	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095	
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

Let’s visualize some images to understand the inputs we will deal with.

Let’s discuss some descriptive statistics about the dataset. We will do that by studying mean/std. deviation of the pixel intensities for some classes (‘Alef’ , ‘Beh’, ‘Dal’)

Here is the examples i have explored

Alef



Beh

Dal

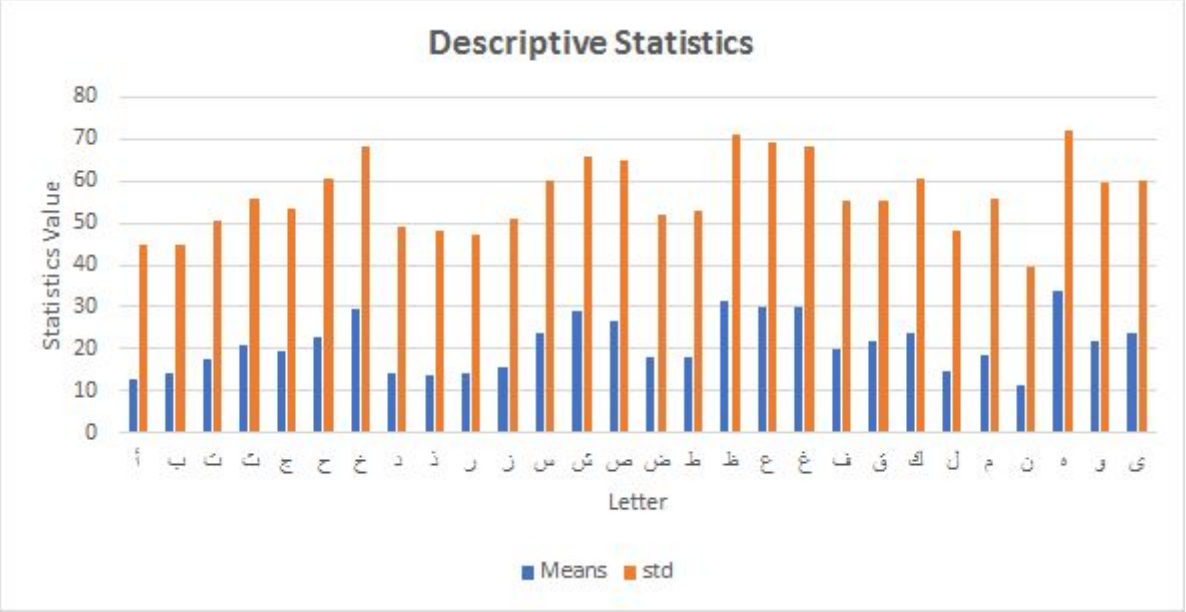
The means and std values represent the blue, green, and red channels, respectively.

```
import cv2
# alef image statistics
image = cv2.imread("alef.png")
(means, stds) = cv2.meanStdDev(image)
print("Alef Image: ")
print("Means = {}".format(means))
print("Std = {}".format(stds))
print("========================")
# beh image statistics
image = cv2.imread("beh.png")
(means, stds) = cv2.meanStdDev(image)
print("Beh Image: ")
print("Means = {}".format(means))
print("Std = {}".format(stds))
print("========================")
# dal image statistics
image = cv2.imread("dal.png")
(means, stds) = cv2.meanStdDev(image)
print("Dal Image: ")
print("Means = {}".format(means))
print("Std = {}".format(stds))
print("========================")
```

```
Alef Image:
Means = [[12.94824219]
 [12.94824219]
 [12.94824219]]
Std = [[44.87789703]
 [44.87789703]
 [44.87789703]]
========================
Beh Image:
Means = [[13.97363281]
 [13.97363281]
 [13.97363281]]
Std = [[44.63214763]
 [44.63214763]
 [44.63214763]]
========================
Dal Image:
Means = [[14.36230469]
 [14.36230469]
 [14.36230469]]
Std = [[49.14393207]
 [49.14393207]
 [49.14393207]]
========================
```

Exploratory Visualization

Here we will discuss the descriptive statistics of the dataset by visualizing Means and std values of the pixel intensities in all letter classes.
Remember that our dataset contains gray images only so means values of blue, green and red channels will be equal as the image is not colored.



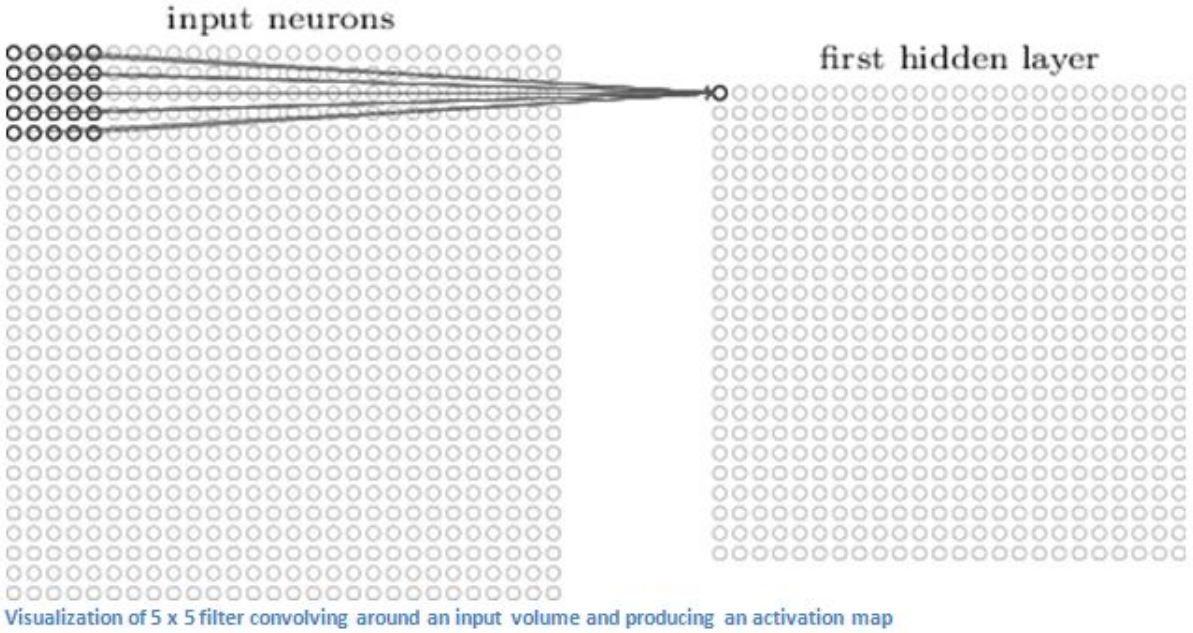
Algorithms and Techniques

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.

Benchmark

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 16 filters and window size of 3 to capture the basic patterns like edges from the input images.
- 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 38 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 20 (to reduce training time and overfitting) and finally using 5 epochs as we want a simple model just to capture the basic patterns.

We got test accuracy of 32.37% as shown below in the figure.

```
baseline_model = Sequential()
baseline_model.add(Conv2D(filters=16, kernel_size=3, padding='same', input_shape=(64, 64, 1), activation='relu')) # Input layer
baseline_model.add(GlobalAveragePooling2D())
baseline_model.add(Dense(38, activation = 'softmax')) # Output layer => output dimension = 38 as it is multi-class

# Compile the baseline model
baseline_model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='Adam')

# Fit the baseline model with training dataset
epochs = 5
batch_size = 20

baseline_model.fit(training_data_images, training_data_labels,
                   validation_data=(testing_data_images, testing_data_labels),
                   epochs=epochs, batch_size=batch_size, verbose=1)

# Test the baseline model
baseline_metrics = baseline_model.evaluate(testing_data_images, testing_data_labels, verbose=1)
print("Baseline Model Test Accuracy: {}".format(baseline_metrics[1]))
print("Baseline Model Test Loss: {}".format(baseline_metrics[0]))
```

```
Train on 73440 samples, validate on 13360 samples
Epoch 1/5
73440/73440 [==============================] - 45s 619us/step - loss: 2.7163 - acc: 0.1814 - val_loss: 2.7079 - val_acc: 0.2183
Epoch 2/5
73440/73440 [==============================] - 40s 550us/step - loss: 2.4709 - acc: 0.2514 - val_loss: 2.6120 - val_acc: 0.2372
Epoch 3/5
73440/73440 [==============================] - 40s 549us/step - loss: 2.3945 - acc: 0.2754 - val_loss: 2.5453 - val_acc: 0.2608
Epoch 4/5
73440/73440 [==============================] - 41s 552us/step - loss: 2.3237 - acc: 0.3069 - val_loss: 2.4661 - val_acc: 0.2959
Epoch 5/5
73440/73440 [==============================] - 40s 550us/step - loss: 2.2357 - acc: 0.3444 - val_loss: 2.3761 - val_acc: 0.3237
13360/13360 [==============================] - 2s 175us/step
Baseline Model Test Accuracy: 0.32372754491017963
Baseline Model Test Loss: 2.376121672327647
```

Methodology

Data Preprocessing

1. Image Normalization

We rescaled the images by dividing every pixel in the image by 255 to make them into range [0, 1] to be normalized.

2. Encoding Categorical Labels

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:

- Digits from 0 to 9 have categories numbers from 0 to 9
- Letters from ’alef’ to ’yeh’ have categories numbers from 10 to 37

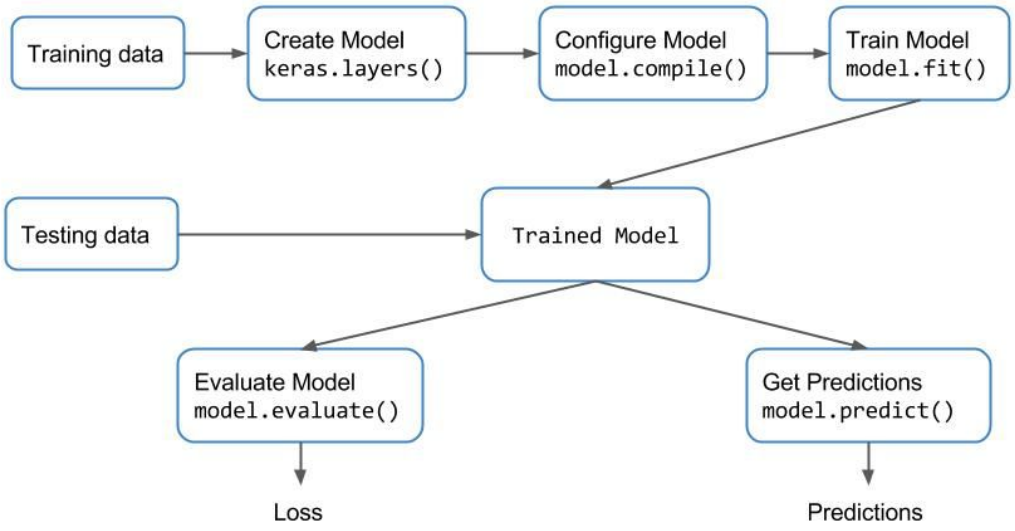
Here we will encode these categories values using One Hot Encoding with keras. One-hot encoding transforms integer to a binary matrix where the array contains only one ‘1’ and the rest elements are ‘0’.

3. 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, 64, 64 ,1) as we use grayscale images of 64x64 pixels.

Implementation

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



Now, let's discuss each part in details.

1. Design Model Architecture

In this section we will discuss our CNN Model.

- The first hidden layer is a convolutional layer. The layer has 16 feature maps, which with the size of 3×3 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 20% of neurons in the layer in order to reduce overfitting.

- Another hidden layer with 32 feature maps with the size of 3×3 and a relu activation function to capture more features from the image.

- Other hidden layers with 64 and 128 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 38 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.

I 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.

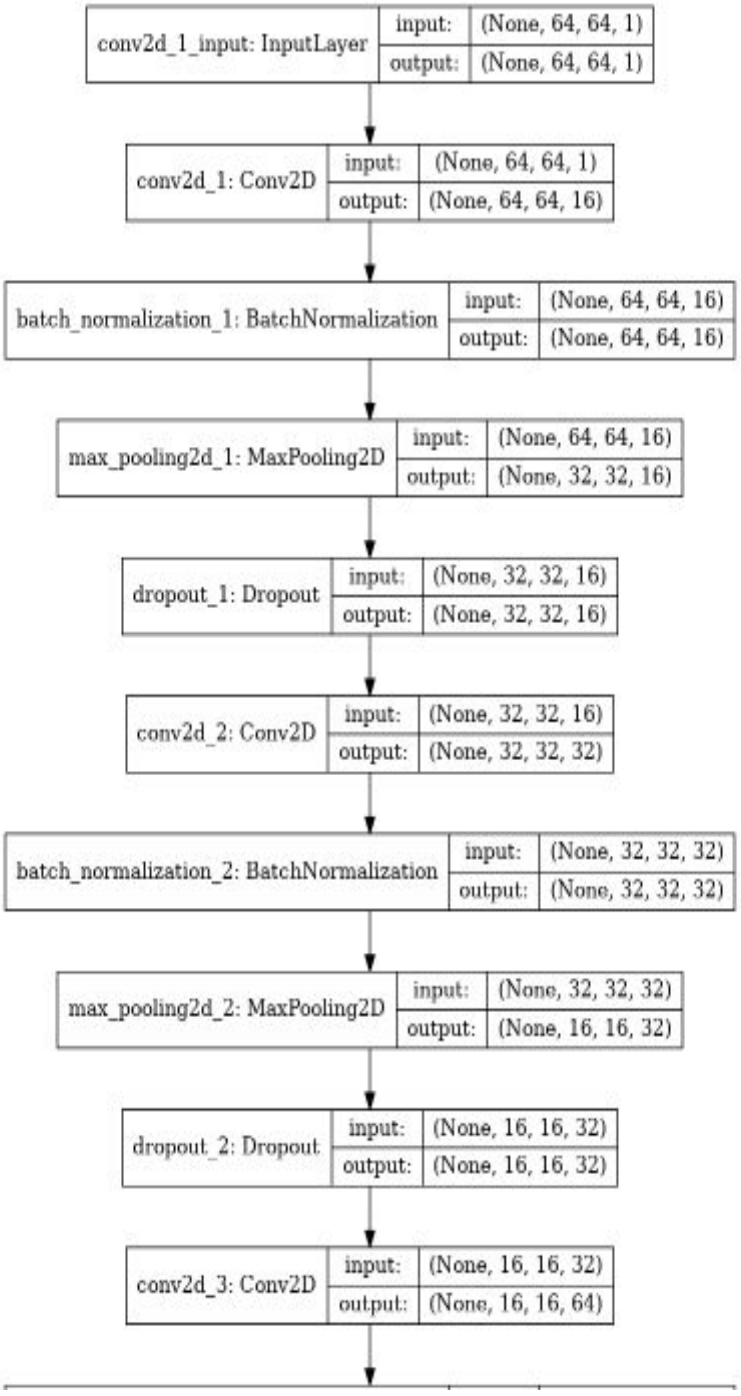
Notice the hyperparameters used for the model:

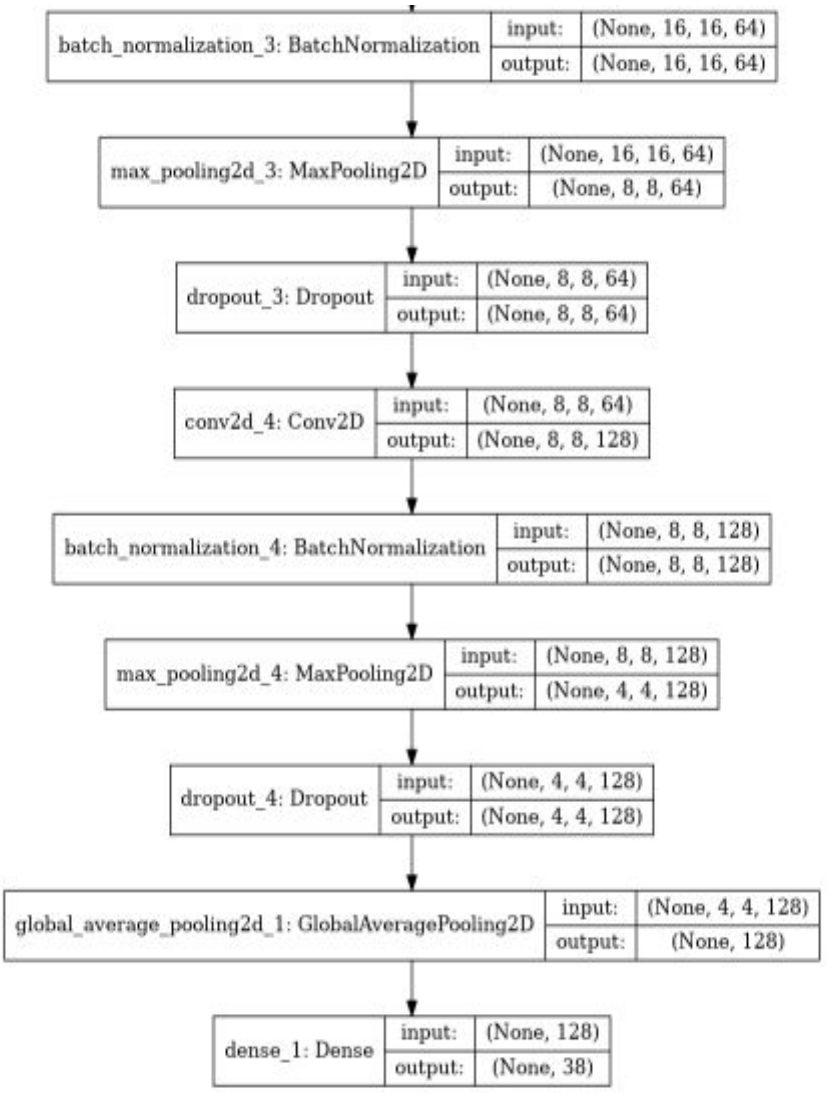
- **Dropout rate: 20% of the layer nodes.**
- **Epochs: 10 then we will fit the model incrementally on 20 more epochs**.
- **Batch size: 20 as it is enough amount and divisible by the size of total training dataset and also the size of total testing dataset.**
- **Optimizer: After the refinement section we will see the best optimizer we will use is Adam.**
- **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 uniform.**

2. Model Summary

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 64, 64, 16)	160
batch_normalization_1 (Batch	(None, 64, 64, 16)	64
max_pooling2d_1 (MaxPooling2	(None, 32, 32, 16)	0
dropout_1 (Dropout)	(None, 32, 32, 16)	0
conv2d_2 (Conv2D)	(None, 32, 32, 32)	4640
batch_normalization_2 (Batch	(None, 32, 32, 32)	128
max_pooling2d_2 (MaxPooling2	(None, 16, 16, 32)	0
dropout_2 (Dropout)	(None, 16, 16, 32)	0
conv2d_3 (Conv2D)	(None, 16, 16, 64)	18496
batch_normalization_3 (Batch	(None, 16, 16, 64)	256
max_pooling2d_3 (MaxPooling2	(None, 8, 8, 64)	0
dropout_3 (Dropout)	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73856
batch_normalization_4 (Batch	(None, 8, 8, 128)	512
max_pooling2d_4 (MaxPooling2	(None, 4, 4, 128)	0
dropout_4 (Dropout)	(None, 4, 4, 128)	0
global_average_pooling2d_1 (	(None, 128)	0
dense_1 (Dense)	(None, 38)	4902

Total params: 103,014
Trainable params: 102,534
Non-trainable params: 480


conv2d_1_input: InputLayer
input: (None, 64, 64, 1)
output: (None, 64, 64, 1)
conv2d_1: Conv2D
input: (None, 64, 64, 1)
output: (None, 64, 64, 16)
batch_normalization_1: BatchNormalization
input: (None, 64, 64, 16)
output: (None, 64, 64, 16)
max_pooling2d_1: MaxPooling2D
input: (None, 64, 64, 16)
output: (None, 32, 32, 16)
dropout_1: Dropout
input: (None, 32, 32, 16)
output: (None, 32, 32, 16)
conv2d_2: Conv2D
input: (None, 32, 32, 16)
output: (None, 32, 32, 32)
batch_normalization_2: BatchNormalization
input: (None, 32, 32, 32)
output: (None, 32, 32, 32)
max_pooling2d_2: MaxPooling2D
input: (None, 32, 32, 32)
output: (None, 16, 16, 32)
dropout_2: Dropout
input: (None, 16, 16, 32)
output: (None, 16, 16, 32)
conv2d_3: Conv2D
input: (None, 16, 16, 32)
output: (None, 16, 16, 64)


batch_normalization_3: BatchNormalization
input: (None, 16, 16, 64)
output: (None, 16, 16, 64)
max_pooling2d_3: MaxPooling2D
input: (None, 16, 16, 64)
output: (None, 8, 8, 64)
dropout_3: Dropout
input: (None, 8, 8, 64)
output: (None, 8, 8, 64)
conv2d_4: Conv2D
input: (None, 8, 8, 64)
output: (None, 8, 8, 128)
batch_normalization_4: BatchNormalization
input: (None, 8, 8, 128)
output: (None, 8, 8, 128)
max_pooling2d_4: MaxPooling2D
input: (None, 8, 8, 128)
output: (None, 4, 4, 128)
dropout_4: Dropout
input: (None, 4, 4, 128)
output: (None, 4, 4, 128)
global_average_pooling2d_1: GlobalAveragePooling2D
input: (None, 4, 4, 128)
output: (None, 128)
dense_1: Dense
input: (None, 128)
output: (None, 38)

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:

```
{'optimizer': 'RMSprop', 'kernel_initializer': 'normal', 'activation': 'relu'}
Train on 73440 samples, validate on 13360 samples
Epoch 1/5
73440/73440 [==============================] - 84s 1ms/step - loss: 0.3349 - acc: 0.9089 - val_loss: 2.4386 - val_acc: 0.5037
Epoch 2/5
73440/73440 [==============================] - 82s 1ms/step - loss: 0.1106 - acc: 0.9670 - val_loss: 0.1605 - val_acc: 0.9507
Epoch 3/5
73440/73440 [==============================] - 81s 1ms/step - loss: 0.0865 - acc: 0.9738 - val_loss: 0.1138 - val_acc: 0.9665
Epoch 4/5
73440/73440 [==============================] - 81s 1ms/step - loss: 0.0788 - acc: 0.9772 - val_loss: 2.5557 - val_acc: 0.5233
Epoch 5/5
73440/73440 [==============================] - 81s 1ms/step - loss: 0.0710 - acc: 0.9788 - val_loss: 0.1519 - val_acc: 0.9559
================================================================================
{'optimizer': 'RMSprop', 'kernel_initializer': 'uniform', 'activation': 'relu'}
Train on 73440 samples, validate on 13360 samples
Epoch 1/5
73440/73440 [==============================] - 83s 1ms/step - loss: 0.3012 - acc: 0.9181 - val_loss: 0.3428 - val_acc: 0.8807
Epoch 2/5
73440/73440 [==============================] - 82s 1ms/step - loss: 0.1075 - acc: 0.9672 - val_loss: 0.1243 - val_acc: 0.9611
Epoch 3/5
73440/73440 [==============================] - 82s 1ms/step - loss: 0.0875 - acc: 0.9734 - val_loss: 0.0698 - val_acc: 0.9793
Epoch 4/5
73440/73440 [==============================] - 81s 1ms/step - loss: 0.0793 - acc: 0.9761 - val_loss: 7.7491 - val_acc: 0.3073
Epoch 5/5
73440/73440 [==============================] - 81s 1ms/step - loss: 0.0727 - acc: 0.9780 - val_loss: 3.6094 - val_acc: 0.5719
================================================================================
{'optimizer': 'RMSprop', 'kernel_initializer': 'normal', 'activation': 'linear'}
Train on 73440 samples, validate on 13360 samples
Epoch 1/5
73440/73440 [==============================] - 81s 1ms/step - loss: 0.6358 - acc: 0.8246 - val_loss: 0.5407 - val_acc: 0.8210
Epoch 2/5
73440/73440 [==============================] - 80s 1ms/step - loss: 0.2895 - acc: 0.9135 - val_loss: 0.3693 - val_acc: 0.8883
Epoch 3/5
73440/73440 [==============================] - 80s 1ms/step - loss: 0.2099 - acc: 0.9368 - val_loss: 0.1959 - val_acc: 0.9409
Epoch 4/5
73440/73440 [==============================] - 80s 1ms/step - loss: 0.1700 - acc: 0.9478 - val_loss: 2.2486 - val_acc: 0.7231
Epoch 5/5
73440/73440 [==============================] - 80s 1ms/step - loss: 0.1494 - acc: 0.9535 - val_loss: 1.5786 - val_acc: 0.6550
================================================================================
```

```
{'optimizer': 'Adam', 'kernel_initializer': 'normal', 'activation': 'relu'}
Train on 73440 samples, validate on 13360 samples
Epoch 1/5
73440/73440 [==============================] - 90s 1ms/step - loss: 0.3496 - acc: 0.9075 - val_loss: 6.2671 - val_acc: 0.2249
Epoch 2/5
73440/73440 [==============================] - 89s 1ms/step - loss: 0.1066 - acc: 0.9679 - val_loss: 0.2868 - val_acc: 0.9138
Epoch 3/5
73440/73440 [==============================] - 89s 1ms/step - loss: 0.0800 - acc: 0.9755 - val_loss: 0.1065 - val_acc: 0.9688
Epoch 4/5
73440/73440 [==============================] - 89s 1ms/step - loss: 0.0692 - acc: 0.9782 - val_loss: 0.0776 - val_acc: 0.9775
Epoch 5/5
73440/73440 [==============================] - 89s 1ms/step - loss: 0.0581 - acc: 0.9820 - val_loss: 0.2886 - val_acc: 0.9206
================================================================================
{'optimizer': 'Adam', 'kernel_initializer': 'uniform', 'activation': 'relu'}
Train on 73440 samples, validate on 13360 samples
Epoch 1/5
73440/73440 [==============================] - 92s 1ms/step - loss: 0.3284 - acc: 0.9116 - val_loss: 1.2440 - val_acc: 0.7041
Epoch 2/5
73440/73440 [==============================] - 90s 1ms/step - loss: 0.1081 - acc: 0.9666 - val_loss: 1.4073 - val_acc: 0.7209
Epoch 3/5
73440/73440 [==============================] - 90s 1ms/step - loss: 0.0826 - acc: 0.9748 - val_loss: 0.5529 - val_acc: 0.8179
Epoch 4/5
73440/73440 [==============================] - 90s 1ms/step - loss: 0.0669 - acc: 0.9794 - val_loss: 0.0744 - val_acc: 0.9786
Epoch 5/5
73440/73440 [==============================] - 90s 1ms/step - loss: 0.0618 - acc: 0.9812 - val_loss: 0.0576 - val_acc: 0.9831
================================================================================
{'optimizer': 'Adam', 'kernel_initializer': 'normal', 'activation': 'linear'}
Train on 73440 samples, validate on 13360 samples
Epoch 1/5
73440/73440 [==============================] - 91s 1ms/step - loss: 0.6370 - acc: 0.8235 - val_loss: 2.8734 - val_acc: 0.6292
Epoch 2/5
73440/73440 [==============================] - 88s 1ms/step - loss: 0.2778 - acc: 0.9138 - val_loss: 0.3445 - val_acc: 0.8853
Epoch 3/5
73440/73440 [==============================] - 88s 1ms/step - loss: 0.2002 - acc: 0.9372 - val_loss: 0.2580 - val_acc: 0.9189
Epoch 4/5
73440/73440 [==============================] - 89s 1ms/step - loss: 0.1637 - acc: 0.9491 - val_loss: 2.0659 - val_acc: 0.4595
Epoch 5/5
73440/73440 [==============================] - 88s 1ms/step - loss: 0.1404 - acc: 0.9550 - val_loss: 0.1368 - val_acc: 0.9581
```

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

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

Which will produce train accuracy of 98.12% and test accuracy of 98.31%

Results

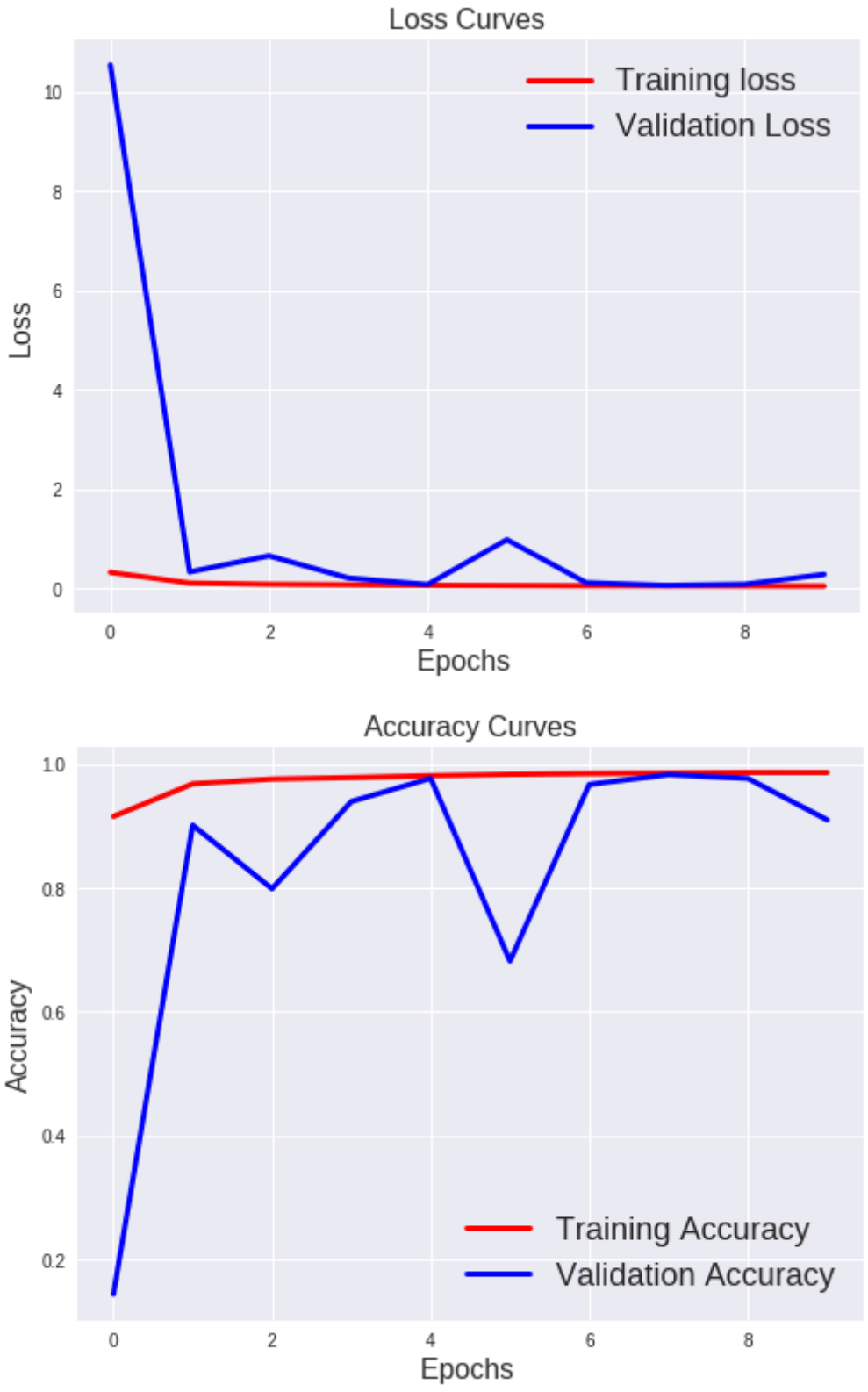
Model Evaluation and Validation

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 10 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 10 epochs.

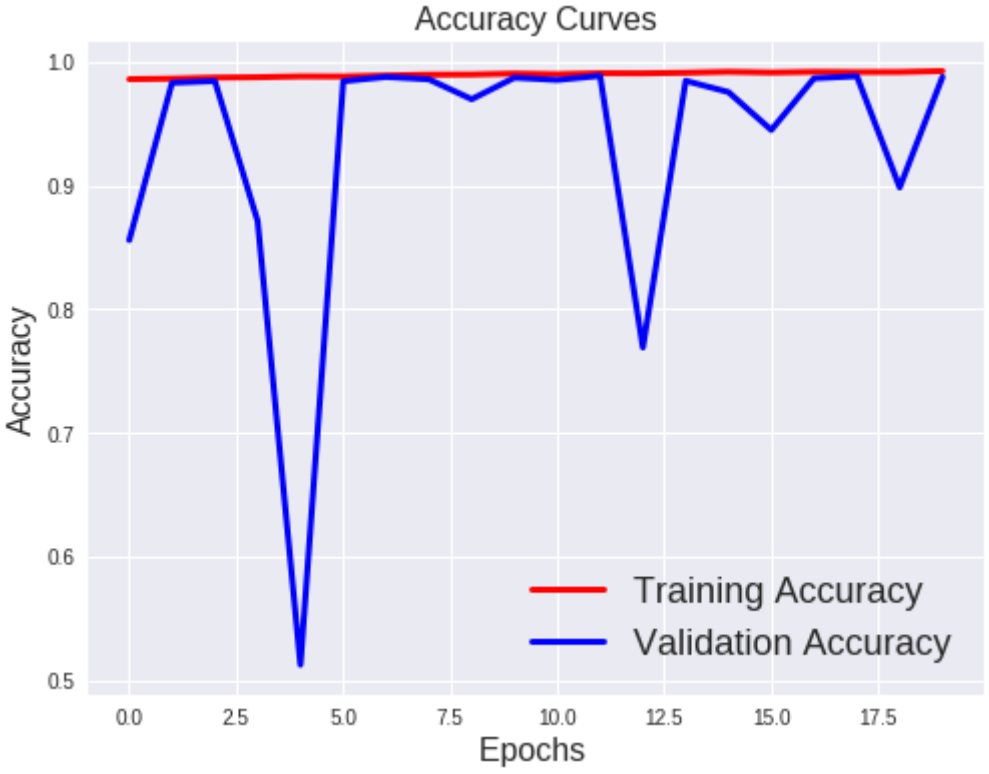
```
Train on 73440 samples, validate on 13360 samples
Epoch 1/10
73440/73440 [==============================] - 94s 1ms/step - loss: 0.3189 - acc: 0.9153 - val_loss: 10.5339 - val_acc: 0.1437

Epoch 00001: val_loss improved from inf to 10.53390, saving model to weights.hdf5
Epoch 2/10
73440/73440 [==============================] - 89s 1ms/step - loss: 0.1049 - acc: 0.9681 - val_loss: 0.3283 - val_acc: 0.9012

Epoch 00002: val_loss improved from 10.53390 to 0.32826, saving model to weights.hdf5
Epoch 3/10
73440/73440 [==============================] - 90s 1ms/step - loss: 0.0797 - acc: 0.9757 - val_loss: 0.6523 - val_acc: 0.7985

Epoch 00003: val_loss did not improve from 0.32826
Epoch 4/10
73440/73440 [==============================] - 89s 1ms/step - loss: 0.0702 - acc: 0.9782 - val_loss: 0.2071 - val_acc: 0.9394

Epoch 00004: val_loss improved from 0.32826 to 0.20706, saving model to weights.hdf5
Epoch 5/10
73440/73440 [==============================] - 89s 1ms/step - loss: 0.0592 - acc: 0.9811 - val_loss: 0.0752 - val_acc: 0.9769

Epoch 00005: val_loss improved from 0.20706 to 0.07522, saving model to weights.hdf5
Epoch 6/10
73440/73440 [==============================] - 89s 1ms/step - loss: 0.0545 - acc: 0.9833 - val_loss: 0.9768 - val_acc: 0.6818

Epoch 00006: val_loss did not improve from 0.07522
Epoch 7/10
73440/73440 [==============================] - 88s 1ms/step - loss: 0.0497 - acc: 0.9845 - val_loss: 0.1118 - val_acc: 0.9671

Epoch 00007: val_loss did not improve from 0.07522
Epoch 8/10
73440/73440 [==============================] - 89s 1ms/step - loss: 0.0473 - acc: 0.9854 - val_loss: 0.0576 - val_acc: 0.9829

Epoch 00008: val_loss improved from 0.07522 to 0.05756, saving model to weights.hdf5
Epoch 9/10
73440/73440 [==============================] - 89s 1ms/step - loss: 0.0453 - acc: 0.9863 - val_loss: 0.0800 - val_acc: 0.9767

Epoch 00009: val_loss did not improve from 0.05756
Epoch 10/10
73440/73440 [==============================] - 89s 1ms/step - loss: 0.0432 - acc: 0.9862 - val_loss: 0.2793 - val_acc: 0.9097

Epoch 00010: val_loss did not improve from 0.05756
```

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



From the above plots we can see that we are still safe from overfitting as both training accuracy and validation accuracy are increasing with increasing in epochs.

Let’s try increasing number of epochs and see what will be the effect.
We will train the same model on 20 more epochs.
We will get the following plots:



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 testing dataset (**we got test accuracy of 98.286%**).

Justification

In this section we will compare the results obtained from the benchmark model with the results of the final model.

Metric	Final Model	Benchmark Model
Accuracy	98.286%	32.37%
Loss value	0.0419	2.376
Precision	99%	24%
Recall	99%	32%
F1-score	0.99	0.26

The above results represent the average per class.
For more details about each single class, Find below detailed table results.

The following Results Table will show the metrics comparison between the final model and the benchmark model for each class of the 38 classes.

Final Model

	precision	recall	f1-score
0	0.99	0.98	0.99
1	0.99	1.00	0.99
2	0.99	0.99	0.99
3	1.00	0.99	0.99
4	1.00	0.99	1.00
5	0.99	0.99	0.99
6	1.00	0.99	1.00
7	1.00	0.99	1.00
8	1.00	1.00	1.00
9	1.00	1.00	1.00
10	1.00	1.00	1.00
11	1.00	1.00	1.00
12	0.96	0.96	0.96
13	0.98	0.99	0.98
14	0.99	0.97	0.98
15	0.97	0.99	0.98
16	0.99	0.97	0.98
17	0.96	0.94	0.95
18	0.95	0.92	0.93
19	0.89	0.99	0.94
20	0.94	0.92	0.93
21	0.98	1.00	0.99
22	0.99	1.00	1.00
23	0.96	0.99	0.98
24	1.00	0.95	0.97
25	0.96	1.00	0.98
26	1.00	0.96	0.98
27	0.98	0.98	0.98
28	1.00	0.99	1.00
29	0.93	0.98	0.96
30	0.98	0.93	0.95
31	1.00	0.98	0.99
32	1.00	0.99	1.00
33	0.98	1.00	0.99
34	0.97	0.93	0.95
35	0.99	0.96	0.97
36	0.94	0.97	0.96
37	1.00	0.99	1.00
avg / total	0.99	0.99	0.99

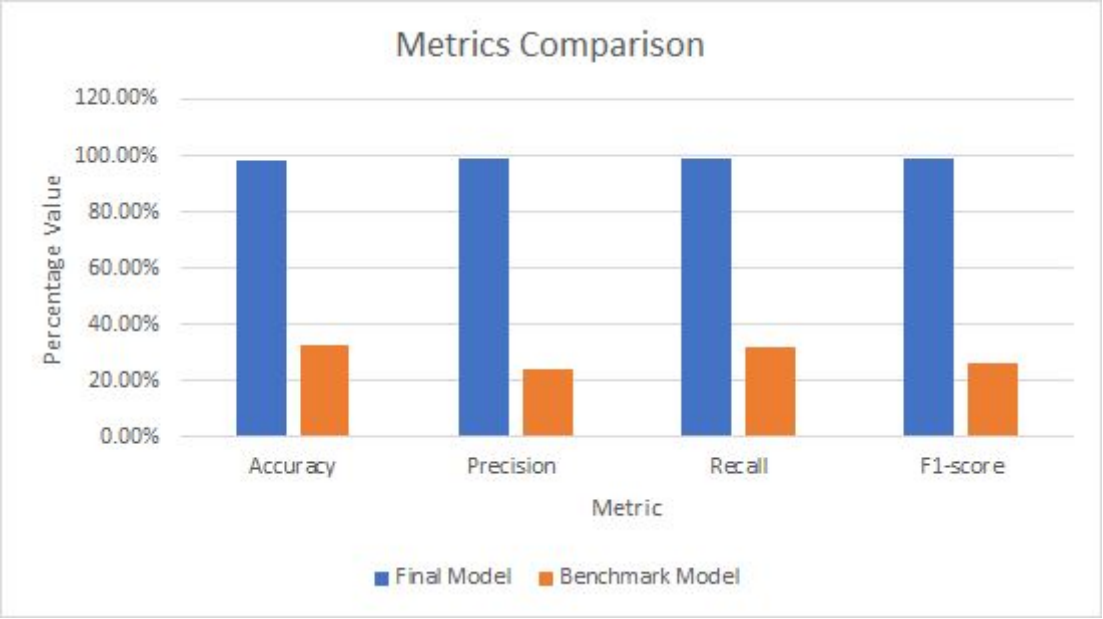
Benchmark Model

	precision	recall	f1-score
0	0.95	0.84	0.89
1	0.25	0.90	0.39
2	0.29	0.23	0.26
3	0.24	0.28	0.25
4	0.31	0.29	0.30
5	0.52	0.73	0.61
6	0.18	0.37	0.24
7	0.28	0.64	0.39
8	0.00	0.00	0.00
9	0.16	0.03	0.04
10	0.00	0.00	0.00
11	0.00	0.00	0.00
12	0.00	0.00	0.00
13	0.00	0.00	0.00
14	0.00	0.00	0.00
15	0.00	0.00	0.00
16	0.00	0.00	0.00
17	0.48	0.18	0.27
18	0.00	0.00	0.00
19	0.00	0.00	0.00
20	0.00	0.00	0.00
21	0.00	0.00	0.00
22	0.00	0.00	0.00
23	0.00	0.00	0.00
24	0.00	0.00	0.00
25	0.00	0.00	0.00
26	0.00	0.00	0.00
27	0.00	0.00	0.00
28	0.00	0.00	0.00
29	0.00	0.00	0.00
30	0.00	0.00	0.00
31	0.00	0.00	0.00
32	0.00	0.00	0.00
33	0.00	0.00	0.00
34	0.00	0.00	0.00
35	0.00	0.00	0.00
36	0.00	0.00	0.00
37	0.00	0.00	0.00
avg / total	0.24	0.32	0.26

Conclusion

Free-Form Visualization

In this project I built a CNN model which can classify the arabic handwritten images into digits and letters. We tested the model on more than 13000 image with all possible classes and got very high accuracy of 98.86% which is much better than the benchmark model.
See the following comparison charts to see the clear improvement.



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 98.86% 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.

References

[1] **HMM Based Approach for Handwritten Arabic Word Recognition**
[2] **An Arabic handwriting synthesis system**
[3] **Advanced Convolutional Neural Networks**
[4] **Normalization-Cooperated Gradient Feature Extraction for Handwritten Character Recognition**