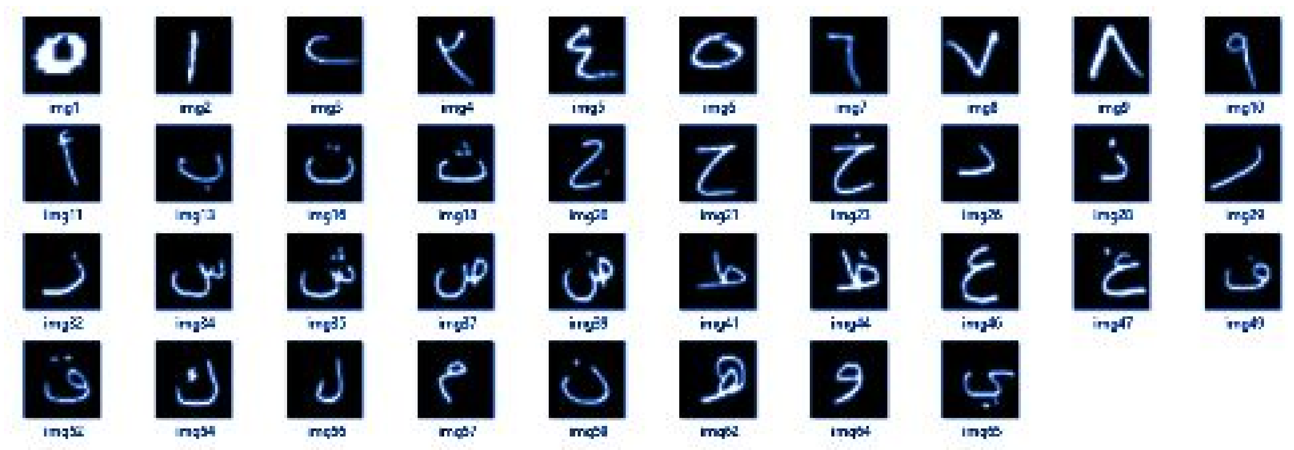
Arabic Handwritten Recognition



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.
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.

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

Given that there aren't any class balance issues, I think that all of following metrics (Accuracy, Precision, Recall and F1-score) should be appropriate. Log loss might also be a practical metric to be used when we tune/refine our model solution, So we will add Log loss to the chosen metrics also.

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

Exploratory Visualization

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

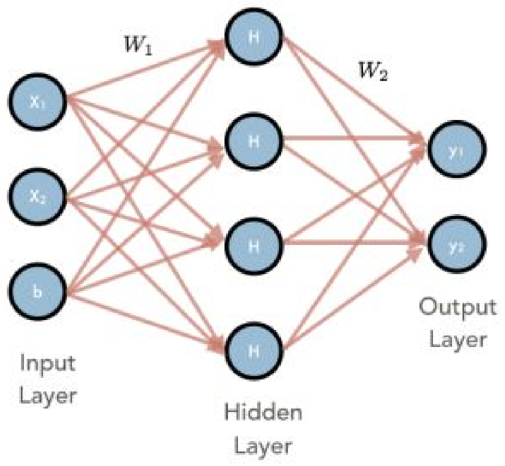
Algorithms and Techniques

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.

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. Then Compare the results between the vanilla model and our complex model.

Here is our vanilla model



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

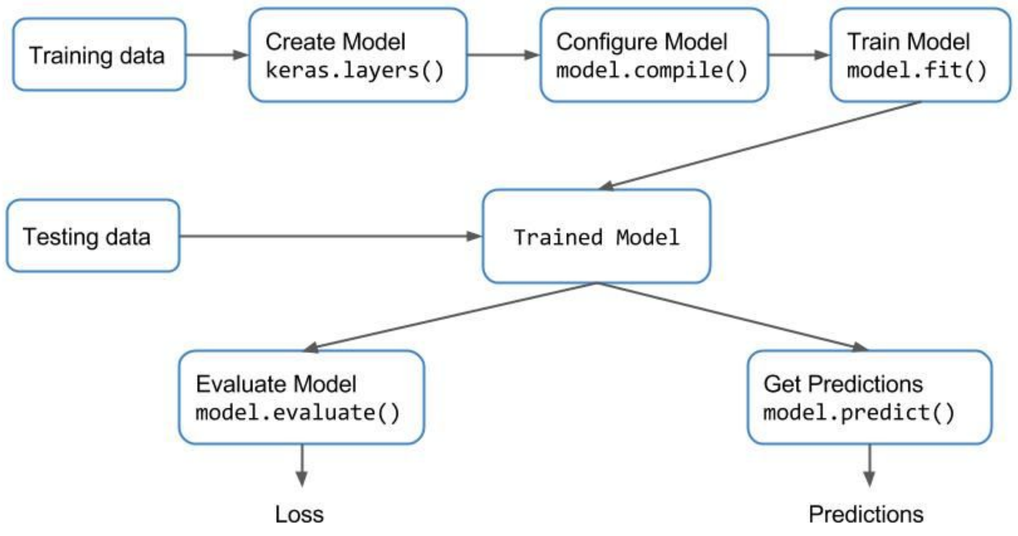
Here we will encode these categories values using One Hot Encoding with keras. One-hot encoding transforms integer to a binary matrix where the array contains only one ‘1’ and the rest elements are ‘0’.

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.

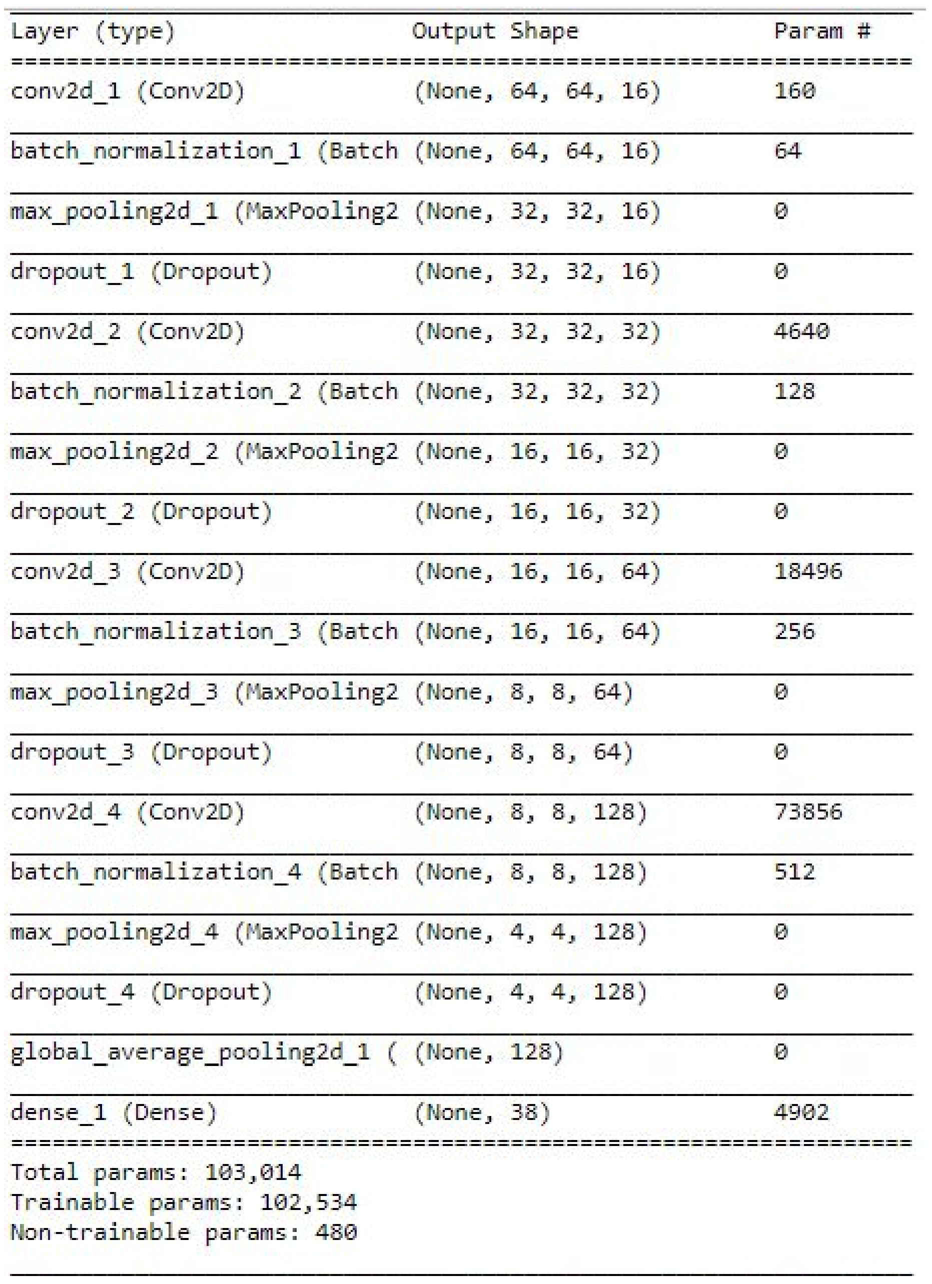
1. Design Model Architecture

In this section we will discuss our CNN Model.

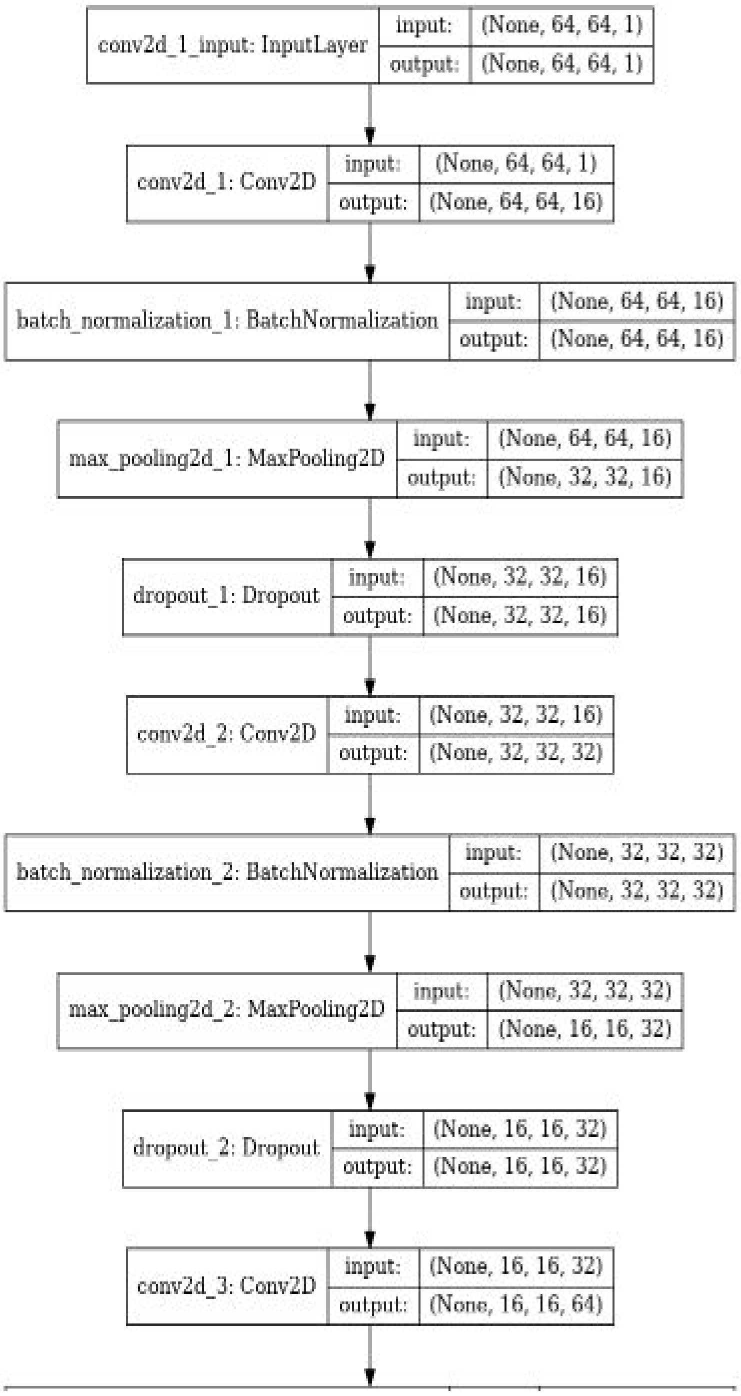
- The first hidden layer is a convolutional layer. The layer has 16 feature maps, which with the size of 3×3 and an activation function which is relu. This is the input layer, expecting images with the structure outlined above.
- The second layer is Batch Normalization which solves having distributions 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 10 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.

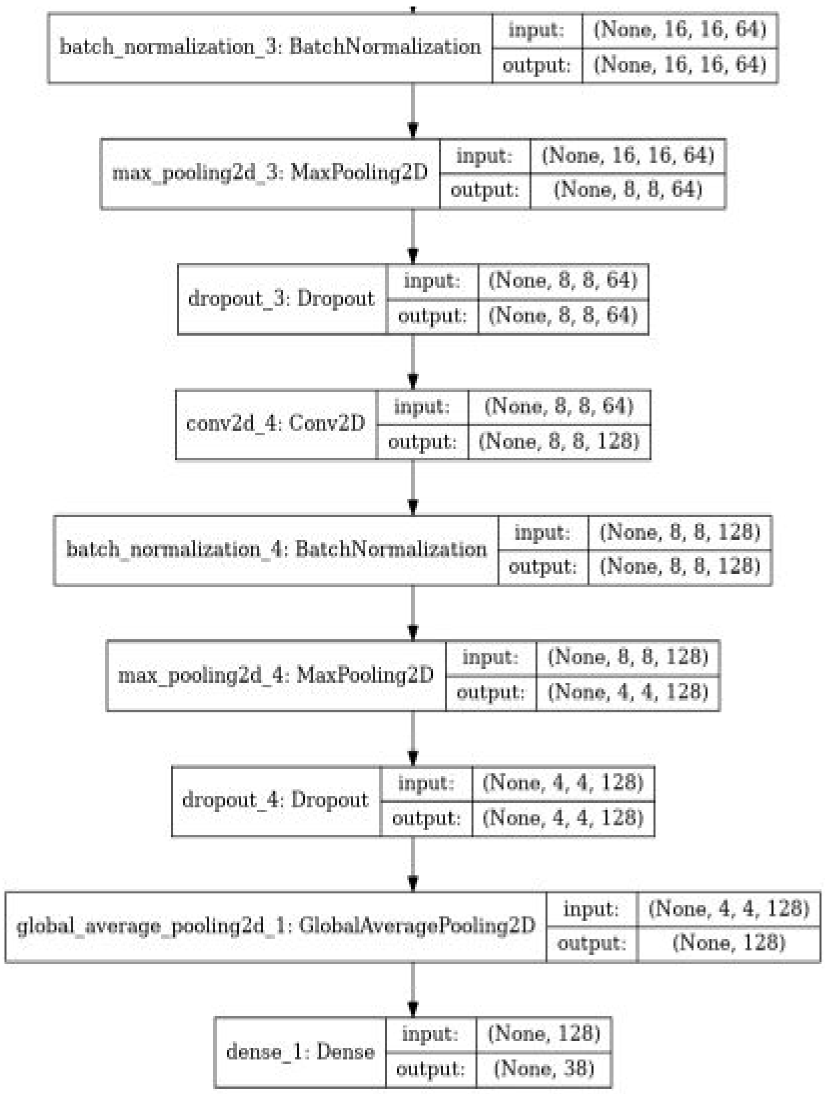
I used categorical_crossentropy as a loss function because its a multi-class classification problem. I also used accuracy as metrics to improve the performance of our neural network.

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
_________________________________________________________________
```

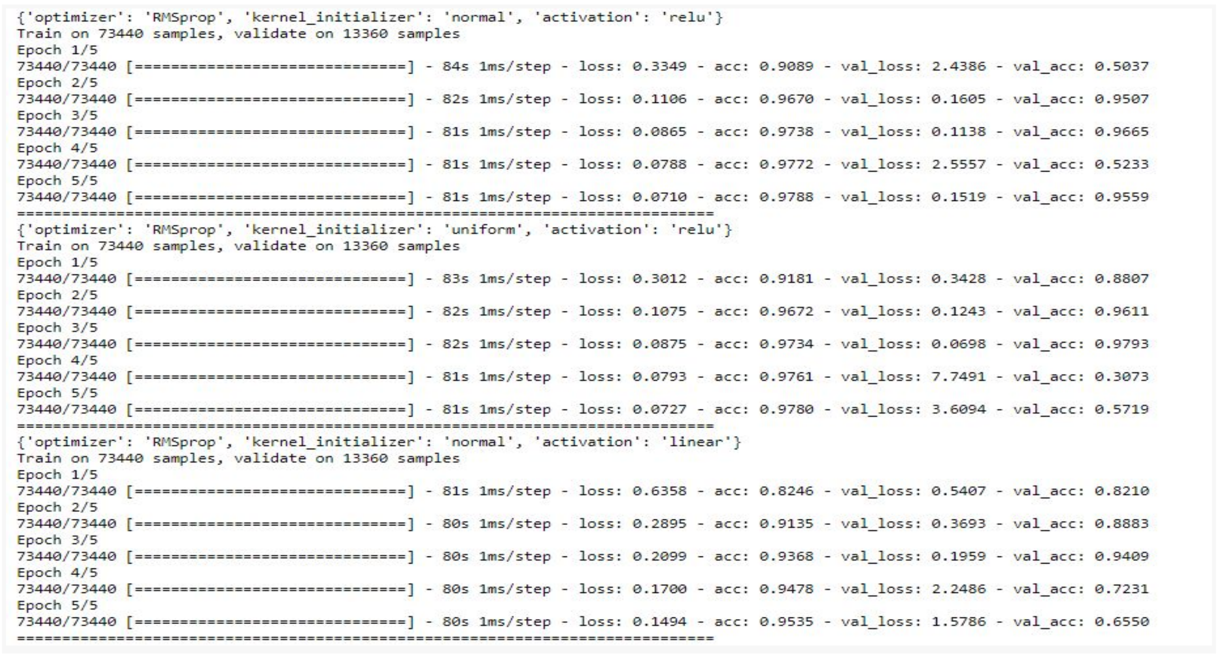

conv2d_1_input: InputLayer
input: (None, 64, 64, 1)
output: (None, 64, 64, 1)
conv2d_1: Conv2D
input: (None, 64, 64, 1)
output: (None, 64, 64, 16)
batch_normalization_1: BatchNormalization
input: (None, 64, 64, 16)
output: (None, 64, 64, 16)
max_pooling2d_1: MaxPooling2D
input: (None, 64, 64, 16)
output: (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)

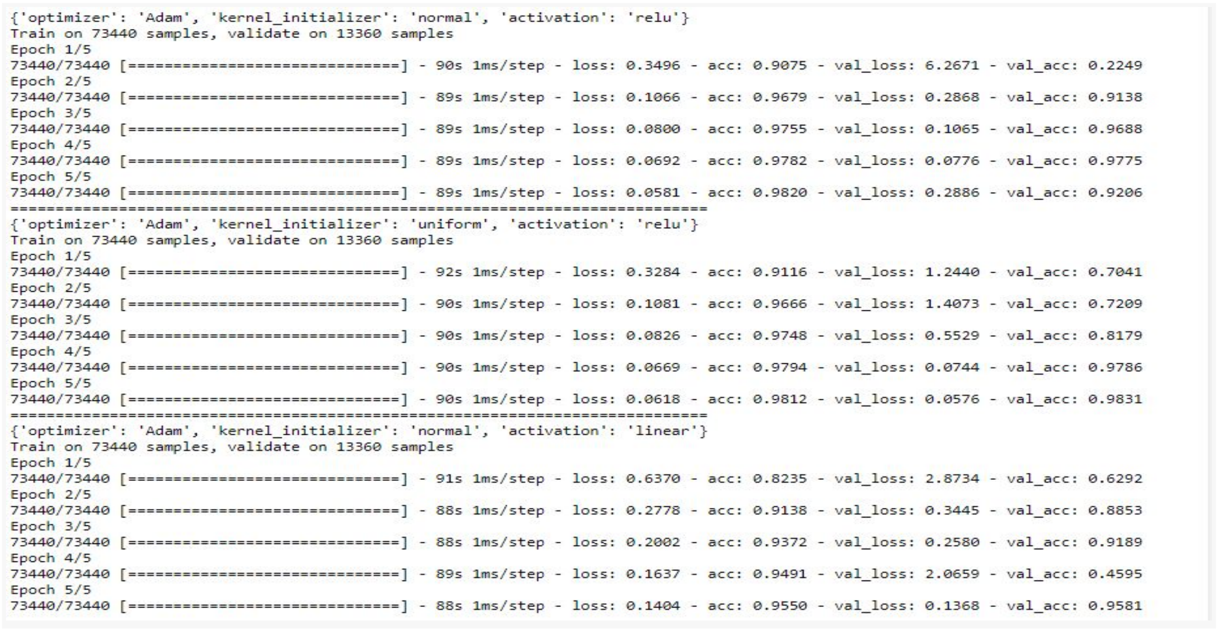
Refinement

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**

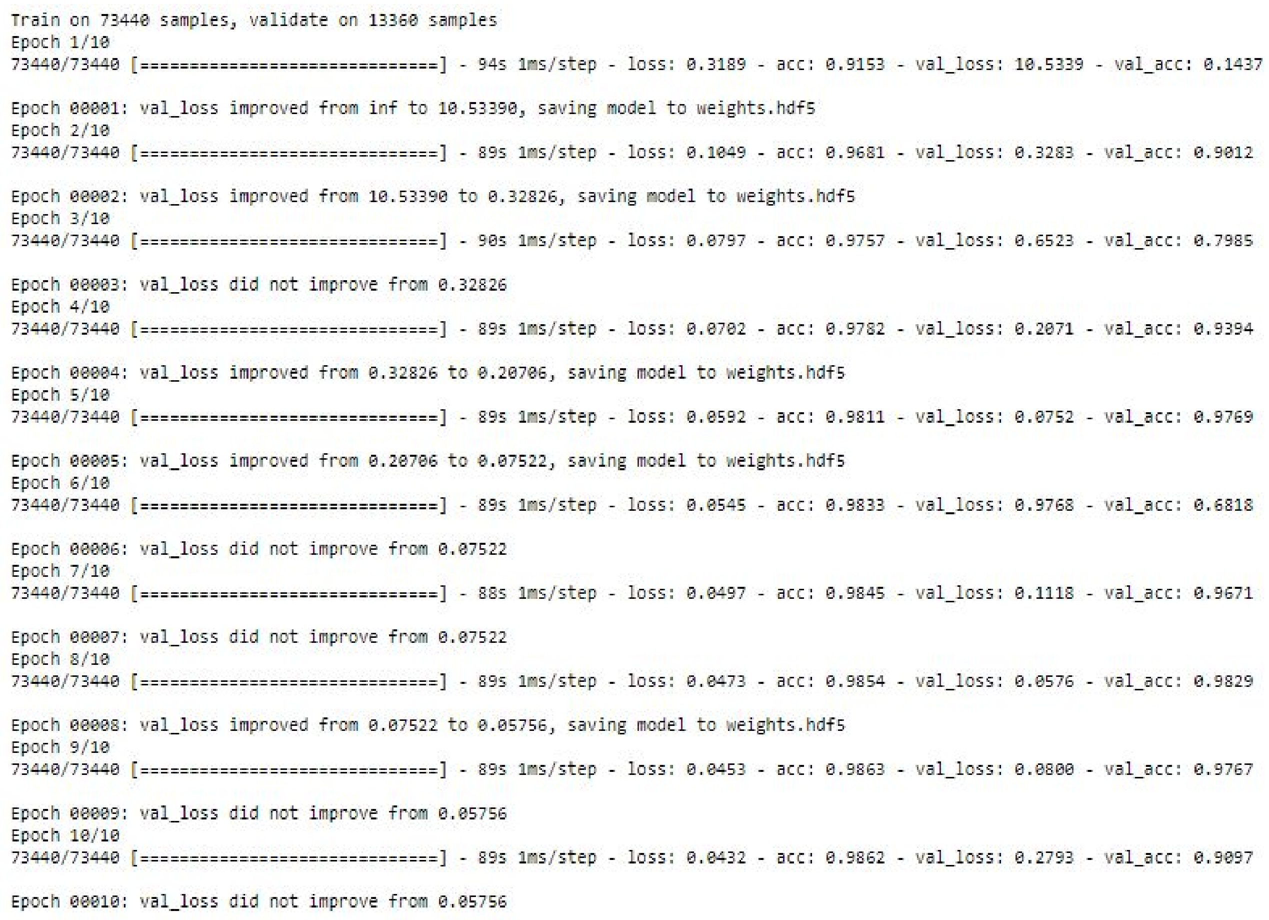
Which will produce train accuracy of 98.12% and test accuracy of 98.31%

Results

Model Evaluation and Validation

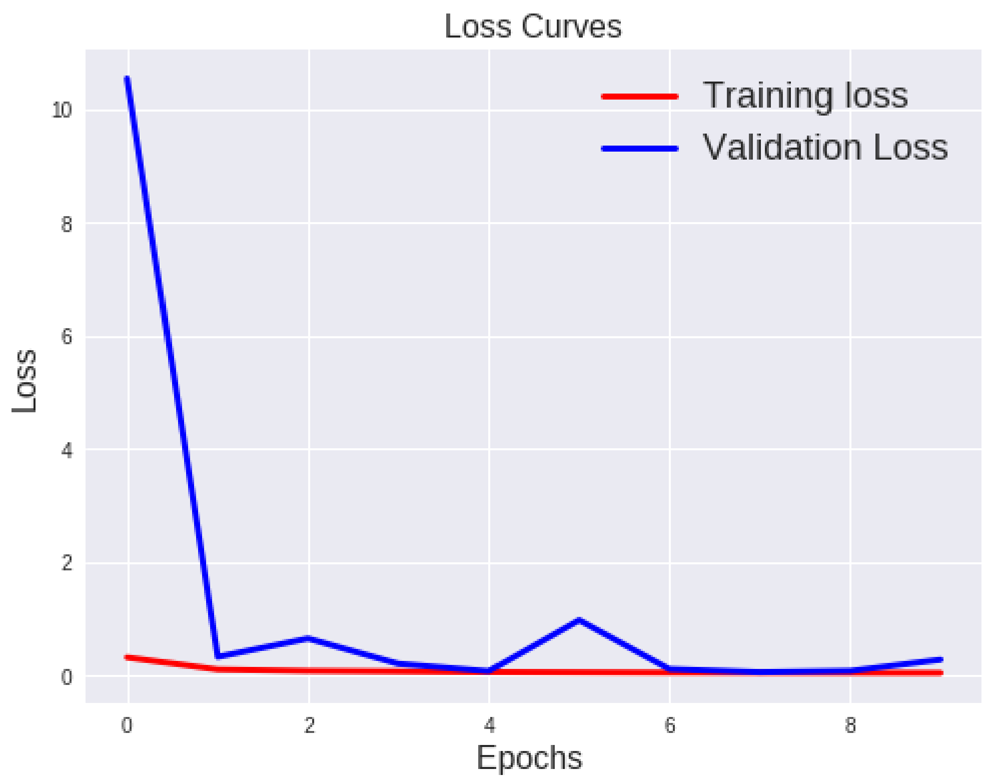
We will Train the model using batch_size=20 to reduce used memory and make the training more quick. We will train the model first on 10 epochs to see the accuracy that we will obtain then we will increase number of epochs to be trained on to improve the accuracy.

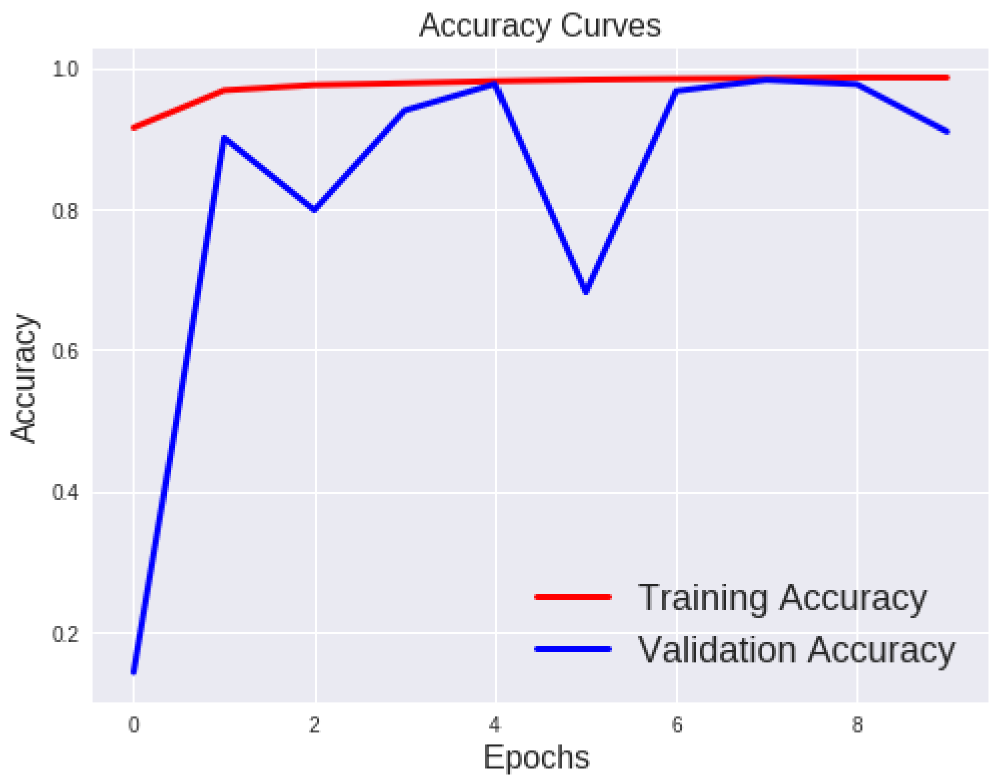
Here are the results after 10 epochs.



```
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
```

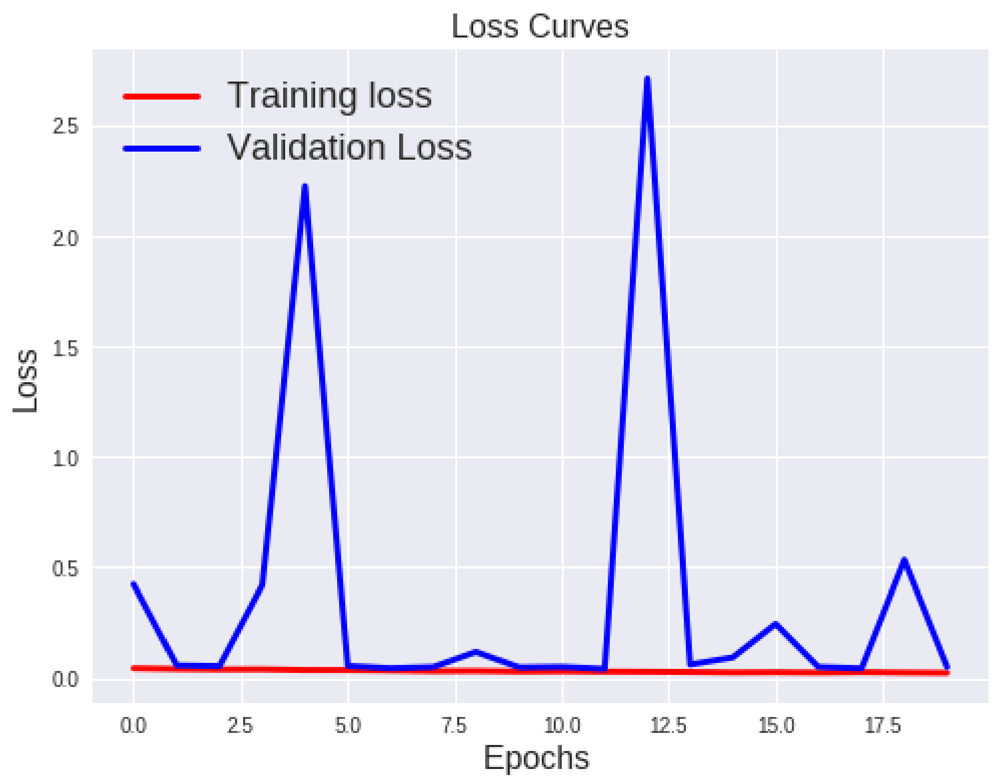
Let’s Plot loss and accuracy curves with epochs to see the changes easily.

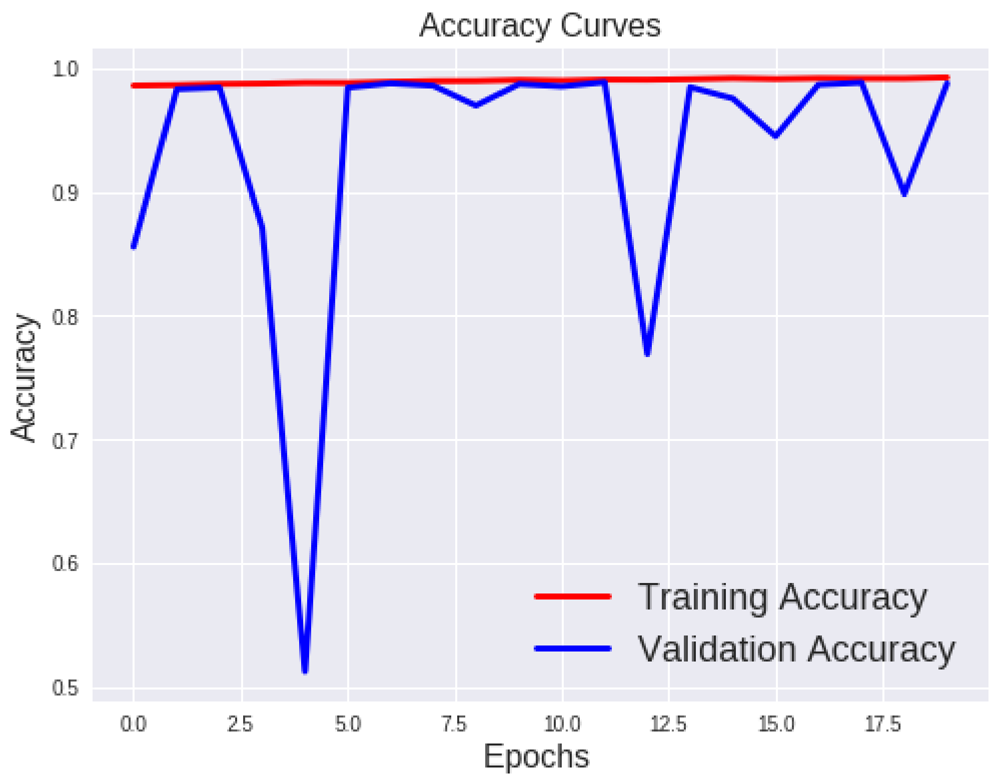




From the above plots we can see that we are still safe from overfitting as both training accuracy and validation accuracy are increasing with increasing in epochs.

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:

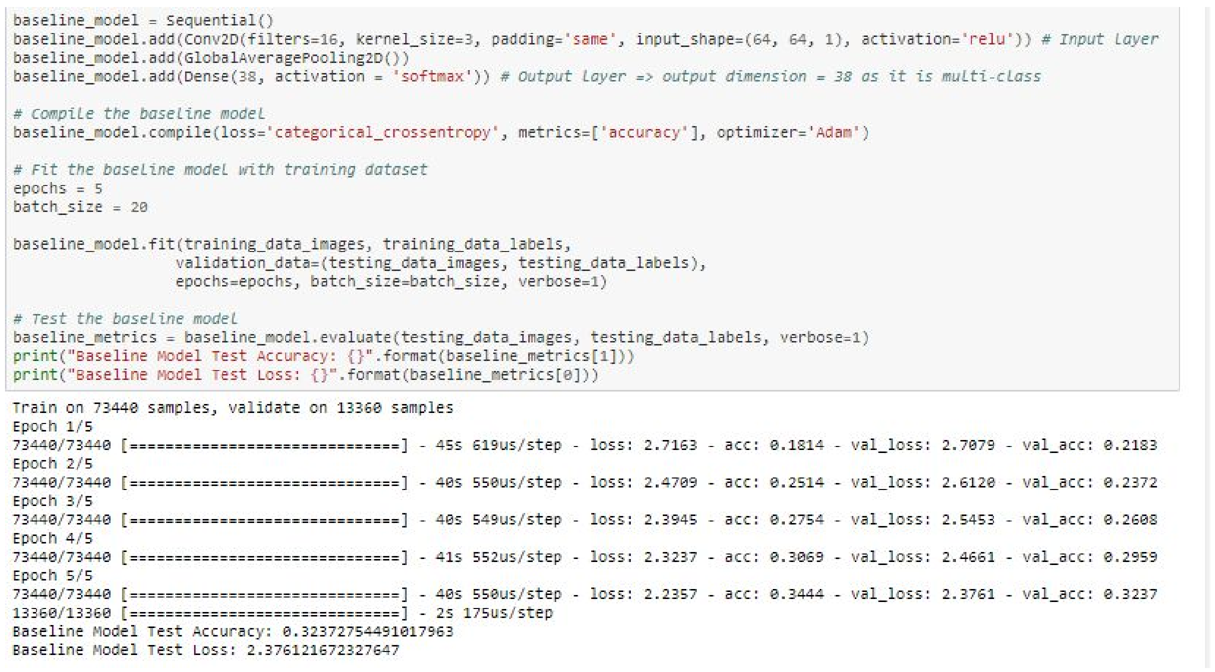




We can see that we may encounter overfitting if we continue to increase the number of epochs. So we will stop fitting the model at that point as we have already got very satisfied score on the testing dataset (**we got test accuracy of 98.286%**).

Justification

In this section we will compare the results obtained from the benchmark model (which is shown in the next figure) with the results of the final model.



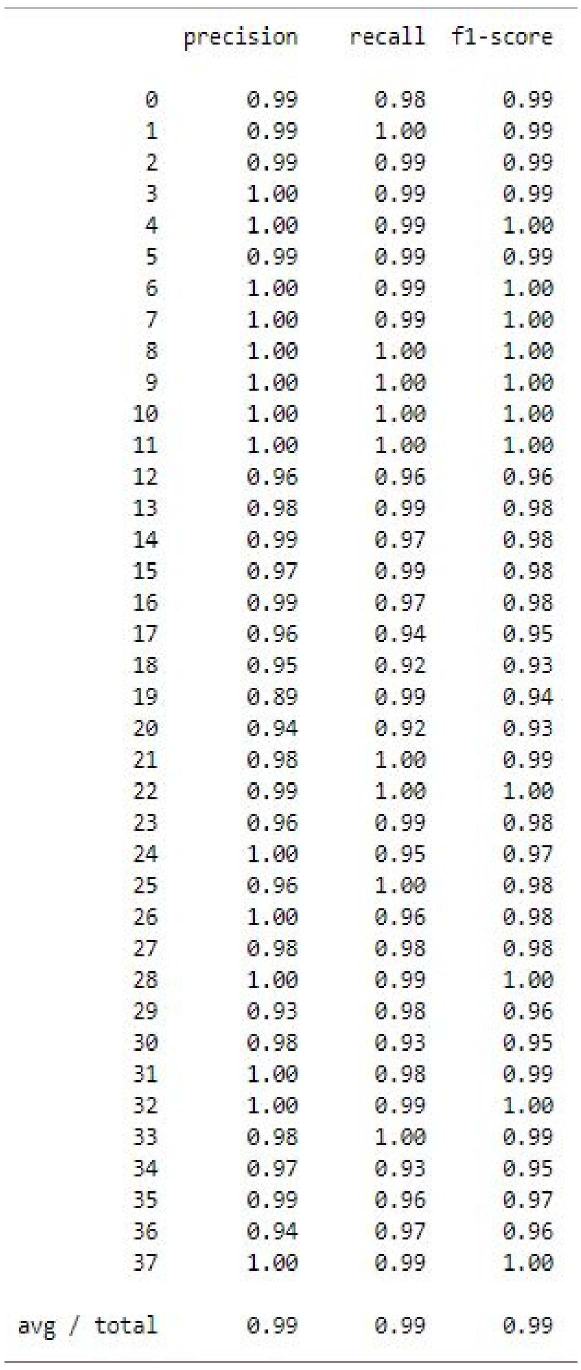
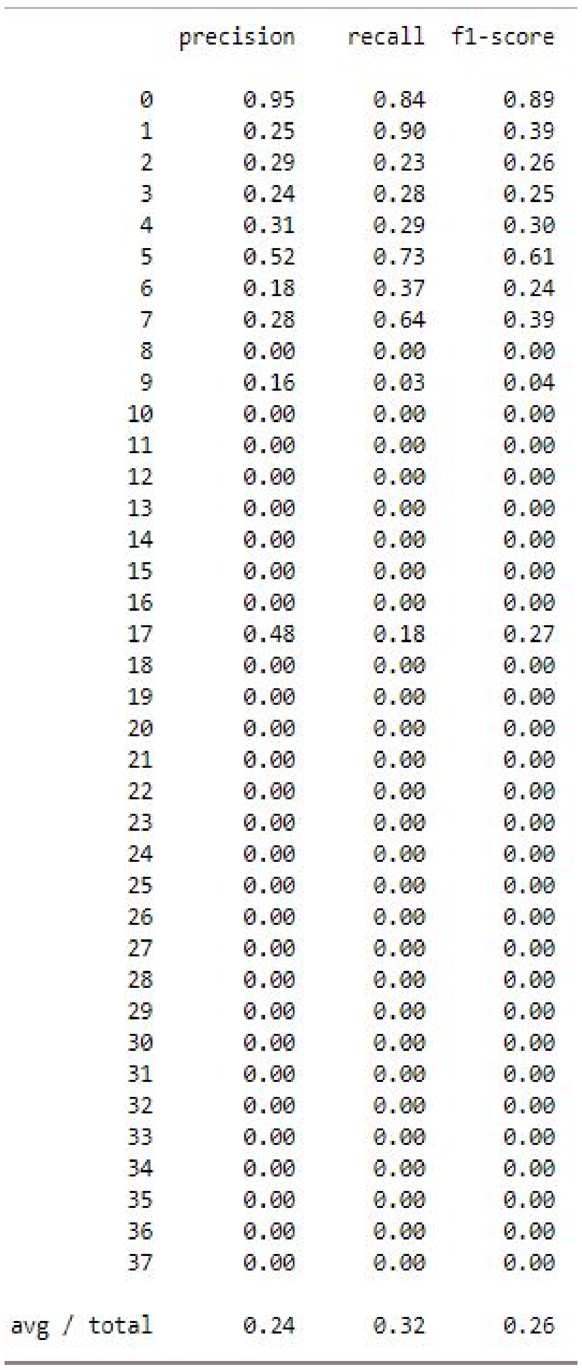
```
baseline_model = Sequential()
baseline_model.add(Conv2D(filters=16, kernel_size=3, padding='same', input_shape=(64, 64, 1), activation='relu')) # Input Layer
baseline_model.add(GlobalAveragePooling2D())
baseline_model.add(Dense(38, activation = 'softmax')) # Output Layer => output dimension = 38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

As we can see from the above results we can compare between our final model and the benchmark model in the following table:

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 results.

Final Model	**Benchmark Model**
	

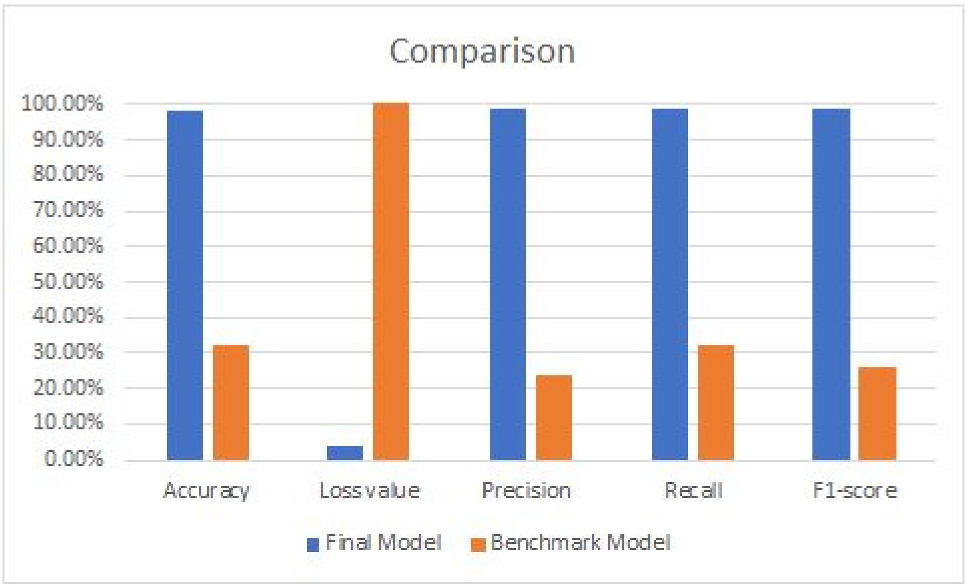
```
             precision    recall  f1-score

          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
```

```
             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 chart 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 including the unlimited variation in human handwriting so we took care of it using large variations of dataset.

As shown in the results section we got test accuracy of 98.86% which is much better than my expectation.

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