

# Blnk Arabic OCR Task

Prepared by Khaled Khalifa

<b>Introduction</b>	<b>3</b>
Thought Process	3
<b>Training Process</b>	<b>3</b>
Data inspection	3
Data Preprocessing	4
Image Preprocessing	4
Text Preprocessing	4
Model Architecture	5
Loss Function	6
Training	6
Inference	6
<b>Django App Integration</b>	<b>6</b>
Running the server	7

# Introduction

The main task is to create an Arabic OCR API that takes an image as input and returns the predicted name.

## Thought Process

I'll first need to inspect the data, preprocess it if necessary, create the model architecture, build and train the model, then build a Django API interface after making sure the model is working properly.

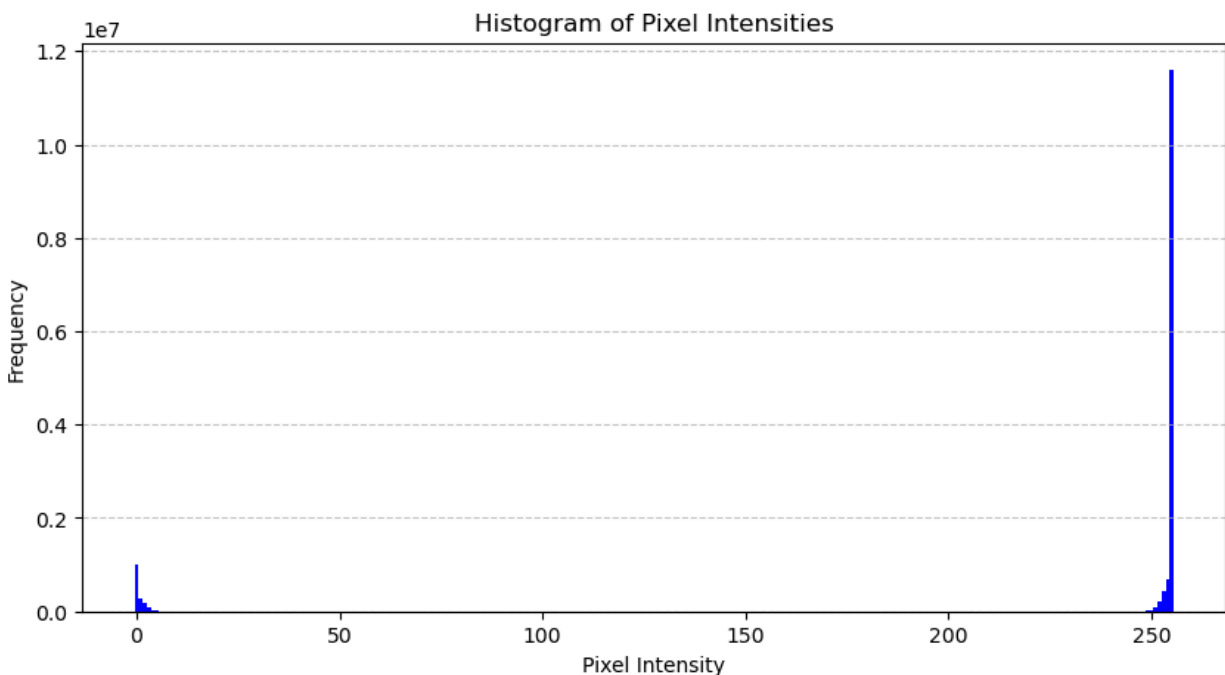
# Training Process

## Data inspection

First I needed to inspect the data to see what I'm working with, after manual inspection I found that the data is split into two types, images containing names, and associated text files containing the names in these images as a form of supervised learning.

Then I loaded the data into the notebook to start inspecting pixel-wise details such as the min and max intensities which will allow us to threshold the images to remove unwanted noise to the model.

After collecting these intensities, I plotted a histogram of all intensities found in the dataset to get this figure.



# Data Preprocessing

## Image Preprocessing

It's clear that the images mostly consisted of white pixels, hence the massive numbers of pixels that have an intensity of around 245 and above, and some pixels which represent the writing font itself are located around the 0-10 intensity mark.

I decided to threshold the image to set any pixel above 245 as 255 (white), and anything else as 0 (black).

Then I proceeded to normalize the image and set all values to either 0 or 1 by dividing by 255 to make it easier to calculate the weights later on.

I also resized the images to 140 x 40 to unify the input dimensions.

## Text Preprocessing

After this I created a set of all the characters inside the text data so see what the training data consists of, which then I embedded using LabelEncoder to convert the strings to integers for the model to be able to calculate them.

I then counted the number of characters in each label respectively. After that, I calculated the maximum number of characters in a label which was 10, then I padded each label to the maximum length to unify the input dimensions to the model, with **value = length(character\_set) + 1** to make sure that it won't affect the inference.

After these steps, I created a Dataframe with the following columns:

**[Image, label, encoded\_labels, padded\_labels, input\_length, label\_length]**

	image	label	encoded_labels	padded_labels	input_length	label_length
0	[[[1.0], [1.0], [1.0], [1.0], [1.0], [1.0], [1...	ءارسا	[1, 6, 15, 17, 6]	[1, 6, 15, 17, 6, 35, 35, 35, 35, 35]	32	5
1	[[[1.0], [1.0], [1.0], [1.0], [1.0], [1.0], [1...	فيرش	[25, 34, 15, 18]	[25, 34, 15, 18, 35, 35, 35, 35, 35, 35]	32	4
2	[[[1.0], [1.0], [1.0], [1.0], [1.0], [1.0], [1...	ءامسا	[1, 6, 29, 17, 6]	[1, 6, 29, 17, 6, 35, 35, 35, 35, 35]	32	5

## Model Architecture

I found the model architecture itself after digging around some papers and github repos.

I altered the dimensions of the input and output layers to make it fit our data by changing the input dimensions to (40,140,1) and the output layer to size (len(character\_set) + 1).

The model consisted of seven convolution layers with sizes varying from 64 to 512.

Then two max-pooling layers are added with size (2,2) and another two of size (2,1) to extract features with a larger width to predict long names.

Then it used batch normalization layers after the fifth and sixth convolution layers which accelerates the training process.

Then it used a Lambda layer to squeeze the output from the convolution layer to fit the LSTM layer.

Lastly, two Bidirectional LSTM layers, each consists of 128 units. This layer gives the output of size (batch\_size, 34, 36) where 36 is the total number of output classes.

Layer (type)	Output Shape	Param #
input_7 (InputLayer)	[(None, 40, 140, 1)]	0
conv2d_42 (Conv2D)	(None, 40, 140, 64)	640
max_pooling2d_24 (MaxPooling2D)	(None, 20, 70, 64)	0
conv2d_43 (Conv2D)	(None, 20, 70, 128)	73856
max_pooling2d_25 (MaxPooling2D)	(None, 10, 35, 128)	0
conv2d_44 (Conv2D)	(None, 10, 35, 256)	295168
conv2d_45 (Conv2D)	(None, 10, 35, 256)	590080
max_pooling2d_26 (MaxPooling2D)	(None, 5, 35, 256)	0
conv2d_46 (Conv2D)	(None, 5, 35, 512)	1180160
batch_normalization_12 (BatchNormalization)	(None, 5, 35, 512)	2048
conv2d_47 (Conv2D)	(None, 5, 35, 512)	2359808
batch_normalization_13 (BatchNormalization)	(None, 5, 35, 512)	2048
max_pooling2d_27 (MaxPooling2D)	(None, 2, 35, 512)	0
conv2d_48 (Conv2D)	(None, 1, 34, 512)	1049088
lambda_6 (Lambda)	(None, 34, 512)	0
bidirectional_12 (Bidirectional)	(None, 34, 256)	656384
bidirectional_13 (Bidirectional)	(None, 34, 256)	394240
dense_6 (Dense)	(None, 34, 36)	9252

## Loss Function

I used a CTC loss function as it is mostly used with text recognition models, I had to define a lambda function to be able to use it during model training. The function takes four arguments, predicted outputs, actual labels, input sequence length to the LSTM, and actual label length. I set the input sequence length to a fixed value of **33** (it had to be lower than **36** which is the total number of output classes) because of the CTC function.

## Training

I trained the model for **10** epochs with a batch size of **256** and achieved a validation loss of **13.64**.

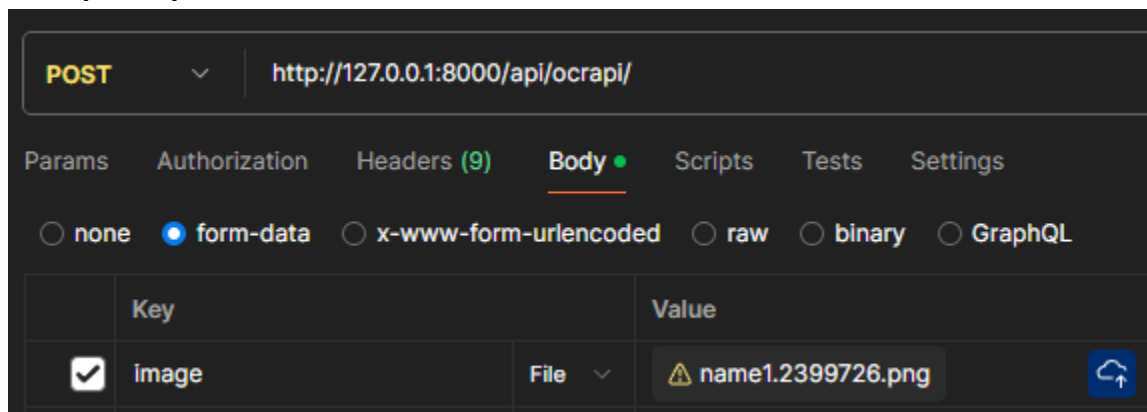
## Inference

To run inference on input images, I loaded the trained weights into a model without the CTC loss function. The outcome is then passed to a ctc\_decoder which is then passed to the LabelEncoder model we fit to the dataset earlier to get the resulting characters. The result is then appended to form a word instead of an array of characters.

## Django App Integration

When the app first starts, the model gets loaded once so as to avoid multiple loadings to save resources. The API itself accepts attached files with key "image", preprocesses it, passes it to the model, then returns the result.


**Example request:**



POST ▼ <http://127.0.0.1:8000/api/ocrapi/>

Params Authorization Headers (9) **Body** ● Scripts Tests Settings

☐ none ☒ form-data ☐ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

	Key	Value
<input checked="" type="checkbox"/>	image	File <span>▼</span> <span>⚠ name1.2399726.png</span> 

**Example response:**

```
{
  "prediction": "لُل"
}
```

## Running the server

To run the server locally, cd into the djangoapp folder, then run

**python manage.py runserver**

Or you can build the Dockerfile inside the djangoapp by running:

1. **docker build -t <app-name> .**
2. **docker run -d -p 8000:8000 <app-name>**

The API local url is <http://127.0.0.1:8000/api/ocrapi/>

To send a request, open Postman, go under Body, add key “**image**” then upload the image you’d like.

An example postman request is located in the solution files.