**DATE:10-05-2020**

**NAME: AVULA.ROHITHA**

**REG NO:17BLC1007**

**----------------------------------------------------------------------------------------------------------------**

**QUESTION:**

**Submit a case study report (maximum of 10 pages) on Face recognition application. Your report may include the following**

**• Block schematic of the application**

**• Image processing algorithms**

**• Feature extraction algorithms**

**• Feature matching**

**• Code (MATLAB/PYTHON)**

**ANSWER:**

**INTRODUCTION:**

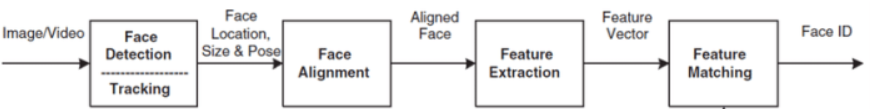
With the rapid growth of computational powers and accessibility of modern intellect, analysis and rendering tools and technologies, computers are becoming more and more intellectual. Face detection is the step stone to all facial analysis algorithms, including face alignment, face modelling, face relighting, face recognition, face verification/authentication, head pose tracking, facial expression tracking/recognition, gender/age recognition, etc. Image processing techniques in face recognition are used to enhance raw images received from cameras/sensors placed on satellites, space probes and aircrafts or pictures taken in normal day-to-day life for various applications. Most of the image processing techniques developed so far are mainly for enhancing images obtained from unmanned spacecrafts, space probes and military reconnaissance flights. Image Processing systems are becoming more popular due to the easy availability of powerful personnel computers, large size memory devices, graphics software etc. Given an arbitrary image, the aim of face detection is to find out whether or not there are any faces in the image and, if present, return the image location and scope of each face. The obscurity associated with face detection can be attributed to many variations in scale, location, orientation (in-plane rotation), pose (out-of-plane rotation), facial expression, lighting conditions, and occlusions.

**TENSORFLOW/KERAS:**

TensorFlow is an open source library created for Python by the Google Brain team. TensorFlow compiles many different algorithms and models together, enabling the user to implement deep neural networks for use in tasks like image recognition/classification and [natural language processing](https://stackabuse.com/what-is-natural-language-processing/). TensorFlow is a powerful framework that functions by implementing a series of processing nodes, each node representing a mathematical operation, with the entire series of nodes being called a "graph".

In terms of Keras, it is a high-level *API* (application programming interface) that can use TensorFlow's functions underneath (as well as other ML libraries like Theano). Keras was designed with user-friendliness and modularity as its guiding principles. In practical terms, Keras makes implementing the many powerful but often complex functions of TensorFlow as simple as possible, and it's configured to work with Python without any major modifications or configuration.

**BLOCK SCHEMATIC OF THE APPLICATION:**



**IMAGE PROCESSING ALGORITHMS:**

Image processing techniques in face recognition can be used to enhance raw images.

The various stages in image processing includes

♦ Image scanning

♦ Storing

♦ Enhancing

♦ Interpretation

**IMAGE EMBEDDING ALGORITHM:**

**Arguments:**

* input\_dim: int > 0. Size of the vocabulary, i.e. maximum integer index + 1.
* output\_dim: int >= 0. Dimension of the dense embedding.
* embeddings\_initializer: Initializer for the embeddings matrix.
* embeddings\_regularizer: Regularizer function applied to the embeddings matrix.
* embeddings\_constraint: Constraint function applied to the embeddings matrix.
* mask\_zero: Whether or not the input value 0 is a special "padding" value that should be masked out. This is useful when using recurrent layers which may take variable length input. If this is True then all subsequent layers in the model need to support masking or an exception will be raised. If mask\_zero is set to True, as a consequence, index 0 cannot be used in the vocabulary (input\_dim should equal size of vocabulary + 1).
* input\_length: Length of input sequences, when it is constant. This argument is required if you are going to connect Flatten then Dense layers upstream (without it, the shape of the dense outputs cannot be computed).

**Input shape:**

2D tensor with shape: (batch\_size, input\_length).

**Output shape:**

3D tensor with shape: (batch\_size, input\_length, output\_dim).

**CONCATENATE ALGORITHM:**

**Arguments:**

* inputs: A list of input tensors (at least 2).
* axis: Concatenation axis.
* \*\*kwargs: Standard layer keyword arguments.

**Returns:**

A tensor, the concatenation of the inputs alongside axis axis

**FEATURE EXTRACTION ALGORITHMS:**

**FEATURE EXTRACTION:**

In order to carry out image recognition/classification, the neural network must carry out [feature extraction](https://en.wikipedia.org/wiki/Feature_extraction). Features are the elements of the data that you care about which will be fed through the network. In the specific case of image recognition, the features are the groups of pixels, like edges and points, of an object that the network will analyze for patterns.

Feature recognition (or feature extraction) is the process of pulling the relevant features out from an input image so that these features can be analyzed. Many images contain annotations or metadata about the image that helps the network find the relevant features.

**CONVOLUTION:**

The first layer of a neural network takes in all the pixels within an image. After all the data has been fed into the network, different filters are applied to the image, which forms representations of different parts of the image. This is feature extraction and it creates "feature maps".

This process of extracting features from an image is accomplished with a "convolutional layer", and convolution is simply forming a representation of part of an image. It is from this convolution concept that we get the term [Convolutional Neural Network](https://en.wikipedia.org/wiki/Convolutional_neural_network) (CNN), the type of neural network most commonly used in image classification/recognition.

If you want to visualize how creating feature maps works, think about shining a flashlight over a picture in a dark room. As you slide the beam over the picture you are learning about features of the image. A filter is what the network uses to form a representation of the image, and in this metaphor, the light from the flashlight is the filter.

The width of your flashlight's beam controls how much of the image you examine at one time, and neural networks have a similar parameter, the filter size. Filter size affects how much of the image, how many pixels, are being examined at one time. A common filter size used in CNNs is 3, and this covers both height and width, so the filter examines a 3 x 3 area of pixels.

Digital images are rendered as height, width, and some *RGB value* that defines the pixel's colors, so the "depth" that is being tracked is the number of color channels the image has. Grayscale (non-color) images only have 1 color channel while color images have 3 depth channels.

All of this means that for a filter of size 3 applied to a full-color image, the dimensions of that filter will be 3 x 3 x 3. For every pixel covered by that filter, the network multiplies the filter values with the values in the pixels themselves to get a numerical representation of that pixel. This process is then done for the entire image to achieve a complete representation. The filter is moved across the rest of the image according to a parameter called "stride", which defines how many pixels the filter is to be moved by after it calculates the value in its current position. A conventional stride size for a CNN is 2.

The end result of all this calculation is a feature map. This process is typically done with more than one filter, which helps preserve the complexity of the image.

**CONV 2D:**

**Arguments:**

* filters: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
* kernel\_size: An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
* strides: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the height and width. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any dilation\_rate value != 1.
* padding: one of "valid" or "same" (case-insensitive).
* data\_format: A string, one of channels\_last (default) or channels\_first. The ordering of the dimensions in the inputs. channels\_last corresponds to inputs with shape (batch\_size, height, width, channels) while channels\_first corresponds to inputs with shape (batch\_size, channels, height, width). It defaults to the image\_data\_format value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "channels\_last".
* dilation\_rate: an integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any dilation\_rate value != 1 is incompatible with specifying any stride value != 1.
* activation: Activation function to use. If you don't specify anything, no activation is applied ( see keras.activations).
* use\_bias: Boolean, whether the layer uses a bias vector.
* kernel\_initializer: Initializer for the kernel weights matrix ( see keras.initializers).
* bias\_initializer: Initializer for the bias vector ( see keras.initializers).
* kernel\_regularizer: Regularizer function applied to the kernel weights matrix (see keras.regularizers).
* bias\_regularizer: Regularizer function applied to the bias vector ( see keras.regularizers).
* activity\_regularizer: Regularizer function applied to the output of the layer (its "activation") ( see keras.regularizers).
* kernel\_constraint: Constraint function applied to the kernel matrix ( see keras.constraints).
* bias\_constraint: Constraint function applied to the bias vector ( see keras.constraints).

**Input shape:**

4D tensor with shape: (batch\_size, channels, rows, cols) if data\_format='channels\_first' or 4D tensor with shape: (batch\_size, rows, cols, channels) if data\_format='channels\_last'.

**Output shape:**

4D tensor with shape: (batch\_size, filters, new\_rows, new\_cols) if data\_format='channels\_first' or 4D tensor with shape: (batch\_size, new\_rows, new\_cols, filters) if data\_format='channels\_last'. rows and cols values might have changed due to padding.

**Returns:**

A tensor of rank 4 representing activation (conv2d(inputs, kernel) + bias).

**Raises:**

* ValueError: if padding is "causal".
* ValueError: when both strides > 1 and dilation\_rate > 1.

**BATCH NORMALIZATION:**

Normalize the activations of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.

Batch normalization differs from other layers in several key aspects:

1) Adding BatchNormalization with training=True to a model causes the result of one example to depend on the contents of all other examples in a minibatch. Be careful when padding batches or masking examples, as these can change the minibatch statistics and affect other examples.

2) Updates to the weights (moving statistics) are based on the forward pass of a model rather than the result of gradient computations.

3) When performing inference using a model containing batch normalization, it is generally (though not always) desirable to use accumulated statistics rather than mini-batch statistics. This is accomplished by passing training=False when calling the model, or using model.predict.

**Arguments:**

* axis: Integer, the axis that should be normalized (typically the features axis). For instance, after a Conv2D layer with data\_format="channels\_first", set axis=1 in BatchNormalization.
* momentum: Momentum for the moving average.
* epsilon: Small float added to variance to avoid dividing by zero.
* center: If True, add offset of beta to normalized tensor. If False, beta is ignored.
* scale: If True, multiply by gamma. If False, gamma is not used. When the next layer is linear (also e.g. nn.relu), this can be disabled since the scaling will be done by the next layer.
* beta\_initializer: Initializer for the beta weight.
* gamma\_initializer: Initializer for the gamma weight.
* moving\_mean\_initializer: Initializer for the moving mean.
* moving\_variance\_initializer: Initializer for the moving variance.
* beta\_regularizer: Optional regularizer for the beta weight.
* gamma\_regularizer: Optional regularizer for the gamma weight.
* beta\_constraint: Optional constraint for the beta weight.
* gamma\_constraint: Optional constraint for the gamma weight.
* renorm:This adds extra variables during training. The inference is the same for either value of this parameter.
* renorm\_clipping: A dictionary that may map keys 'rmax', 'rmin', 'dmax' to scalar Tensors used to clip the renorm correction. The correction (r, d) is used as corrected\_value = normalized\_value \* r + d, with r clipped to [rmin, rmax], and d to [-dmax, dmax]. Missing rmax, rmin, dmax are set to inf, 0, inf, respectively.
* renorm\_momentum: Momentum used to update the moving means and standard deviations with renorm. Unlike momentum, this affects training and should be neither too small (which would add noise) nor too large (which would give stale estimates). Note that momentum is still applied to get the means and variances for inference.
* fused: if True, use a faster, fused implementation, or raise a ValueError if the fused implementation cannot be used. If None, use the faster implementation if possible. If False, do not used the fused implementation.
* trainable: Boolean, if True the variables will be marked as trainable.
* virtual\_batch\_size: An int. By default, virtual\_batch\_size is None, which means batch normalization is performed across the whole batch. When virtual\_batch\_size is not None, instead perform "Ghost Batch Normalization", which creates virtual sub-batches which are each normalized separately (with shared gamma, beta, and moving statistics). Must divide the actual batch size during execution.
* adjustment: A function taking the Tensor containing the (dynamic) shape of the input tensor and returning a pair (scale, bias) to apply to the normalized values (before gamma and beta), only during training. For example, if axis==-1, adjustment = lambda shape: ( tf.random.uniform(shape[-1:], 0.93, 1.07), tf.random.uniform(shape[-1:], -0.1, 0.1)) will scale the normalized value by up to 7% up or down, then shift the result by up to 0.1 (with independent scaling and bias for each feature but shared across all examples), and finally apply gamma and/or beta. If None, no adjustment is applied. Cannot be specified if virtual\_batch\_size is specified.

**Call arguments:**

* inputs: Input tensor (of any rank).
* training: Python boolean indicating whether the layer should behave in training mode or in inference mode.
* training=True: The layer will normalize its inputs using the mean and variance of the current batch of inputs.
* training=False: The layer will normalize its inputs using the mean and variance of its moving statistics, learned during training.

**Input shape:**

Arbitrary. Use the keyword argument input\_shape (tuple of integers, does not include the samples axis) when using this layer as the first layer in a model.

**Output shape:**

Same shape as input.

About setting layer.trainable = False on a `BatchNormalization layer:

The meaning of setting layer.trainable = False is to freeze the layer, i.e. its internal state will not change during training: its trainable weights will not be updated during fit() or train\_on\_batch(), and its state updates will not be run.

Usually, this does not necessarily mean that the layer is run in inference mode (which is normally controlled by the training argument that can be passed when calling a layer). "Frozen state" and "inference mode" are two separate concepts.

However, in the case of the BatchNormalization layer, setting trainable = False on the layer means that the layer will be subsequently run in inference mode (meaning that it will use the moving mean and the moving variance to normalize the current batch, rather than using the mean and variance of the current batch).

This behavior has been introduced in TensorFlow 2.0, in order to enable layer.trainable = False to produce the most commonly expected behavior in the convnet fine-tuning use case.

**ACTIVATION FUNCTION:**

After the feature map of the image has been created, the values that represent the image are passed through an activation function or activation layer. The activation function takes values that represent the image, which are in a linear form (i.e. just a list of numbers) thanks to the convolutional layer, and increases their non-linearity since images themselves are non-linear.

The typical activation function used to accomplish this is a **Rectified Linear Unit** (ReLU), although there are some other activation functions that are occasionally used.

**Arguments:**

* x: Input tensor or variable.
* alpha: A float that governs the slope for values lower than the threshold.
* max\_value: A float that sets the saturation threshold (the largest value the function will return).
* threshold: A float giving the threshold value of the activation function below which values will be damped or set to zero.

**Returns:**

A Tensor representing the input tensor, transformed by the relu activation function. Tensor will be of the same shape and dtype of input x.

**POOLING LAYERS:**

After the data is activated, it is sent through a [pooling layer](http://deeplearning.stanford.edu/tutorial/supervised/Pooling/). Pooling "downsamples" an image, meaning that it takes the information which represents the image and compresses it, making it smaller. The pooling process makes the network more flexible and more adept at recognizing objects/images based on the relevant features.

When we look at an image, we typically aren't concerned with all the information in the background of the image, only the features we care about, such as people or animals.

Similarly, a pooling layer in a CNN will abstract away the unnecessary parts of the image, keeping only the parts of the image it thinks are relevant, as controlled by the specified size of the pooling layer.

Because it has to make decisions about the most relevant parts of the image, the hope is that the network will learn only the parts of the image that truly represent the object in question. This helps prevent [overfitting](https://en.wikipedia.org/wiki/Overfitting), where the network learns aspects of the training case too well and fails to generalize to new data.

There are various ways to pool values, but [max pooling](https://computersciencewiki.org/index.php/Max-pooling_/_Pooling) is most commonly used. Max pooling obtains the maximum value of the pixels within a single filter (within a single spot in the image). This drops 3/4ths of information, assuming 2 x 2 filters are being used.

The maximum values of the pixels are used in order to account for possible image distortions, and the parameters/size of the image are reduced in order to control for overfitting. There are other pooling types such as average pooling or sum pooling, but these aren't used as frequently because max pooling tends to yield better accuracy.

**AVERAGE POOLING 2D:**

**Arguments:**

* pool\_size: integer or tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.
* strides: Integer, tuple of 2 integers, or None. Strides values. If None, it will default to pool\_size.
* padding: One of "valid" or "same" (case-insensitive).
* data\_format: A string, one of channels\_last (default) or channels\_first. The ordering of the dimensions in the inputs. channels\_last corresponds to inputs with shape (batch, height, width, channels) while channels\_first corresponds to inputs with shape (batch, channels, height, width). It defaults to the image\_data\_format value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "channels\_last".

**Input shape:**

* If data\_format='channels\_last': 4D tensor with shape (batch\_size, rows, cols, channels).
* If data\_format='channels\_first': 4D tensor with shape (batch\_size, channels, rows, cols).

**Output shape:**

* If data\_format='channels\_last': 4D tensor with shape (batch\_size, pooled\_rows, pooled\_cols, channels).
* If data\_format='channels\_first': 4D tensor with shape (batch\_size, channels, pooled\_rows, pooled\_cols).

**MAXPOOLING 2D:**

**Arguments:**

* pool\_size: integer or tuple of 2 integers, window size over which to take the maximum. (2, 2) will take the max value over a 2x2 pooling window. If only one integer is specified, the same window length will be used for both dimensions.
* strides: Integer, tuple of 2 integers, or None. Strides values. Specifies how far the pooling window moves for each pooling step. If None, it will default to pool\_size.
* padding: One of "valid" or "same" (case-insensitive). "valid" adds no zero padding. "same" adds padding such that if the stride is 1, the output shape is the same as input shape.
* data\_format: A string, one of channels\_last (default) or channels\_first. The ordering of the dimensions in the inputs. channels\_last corresponds to inputs with shape (batch, height, width, channels) while channels\_first corresponds to inputs with shape (batch, channels, height, width). It defaults to the image\_data\_format value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "channels\_last".

**Input shape:**

* If data\_format='channels\_last': 4D tensor with shape (batch\_size, rows, cols, channels).
* If data\_format='channels\_first': 4D tensor with shape (batch\_size, channels, rows, cols).

**Output shape:**

* If data\_format='channels\_last': 4D tensor with shape (batch\_size, pooled\_rows, pooled\_cols, channels).
* If data\_format='channels\_first': 4D tensor with shape (batch\_size, channels, pooled\_rows, pooled\_cols).

**Returns:**

A tensor of rank 4 representing the maximum pooled values. See above for output shape.

**FLATTENING:**

The final layers of our CNN, the densely connected layers, require that the data is in the form of a vector to be processed. For this reason, the data must be "flattened". The values are compressed into a long vector or a column of sequentially ordered numbers.

**FULLY CONNECTED LAYERS:**

The final layers of the CNN are densely connected layers, or an artificial neural network (ANN). The primary function of the ANN is to analyze the input features and combine them into different attributes that will assist in classification. These layers are essentially forming collections of neurons that represent different parts of the object in question, and a collection of neurons may represent the floppy ears of a dog or the redness of an apple. When enough of these neurons are activated in response to an input image, the image will be classified as an object.

The error, or the difference between the computed values and the expected value in the training set, is calculated by the ANN. The network then undergoes [backpropagation](https://en.wikipedia.org/wiki/Backpropagation), where the influence of a given neuron on a neuron in the next layer is calculated and its influence adjusted. This is done to optimize the performance of the model. This process is then repeated over and over. This is how the network trains on data and learns associations between input features and output classes.

The neurons in the middle fully connected layers will output binary values relating to the possible classes. If you have four different classes (let's say a dog, a car, a house, and a person), the neuron will have a "1" value for the class it believes the image represents and a "0" value for the other classes.

The final fully connected layer will receive the output of the layer before it and deliver a probability for each of the classes, summing to one. If there is a 0.75 value in the "dog" category, it represents a 75% certainty that the image is a dog.

The image classifier has now been trained, and images can be passed into the CNN, which will now output a guess about the content of that image.

**ZERO PADDING 2D:**

**Arguments:**

* padding: Int, or tuple of 2 ints, or tuple of 2 tuples of 2 ints.
* If int: the same symmetric padding is applied to height and width.
* If tuple of 2 ints: interpreted as two different symmetric padding values for height and width: (symmetric\_height\_pad, symmetric\_width\_pad).
* If tuple of 2 tuples of 2 ints: interpreted as ((top\_pad, bottom\_pad), (left\_pad, right\_pad))
* data\_format: A string, one of channels\_last (default) or channels\_first. The ordering of the dimensions in the inputs. channels\_last corresponds to inputs with shape (batch\_size, height, width, channels) while channels\_first corresponds to inputs with shape (batch\_size, channels, height, width). It defaults to the image\_data\_format value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "channels\_last".

**Input shape:**

4D tensor with shape:

* If data\_format is "channels\_last": (batch\_size, rows, cols, channels)
* If data\_format is "channels\_first": (batch\_size, channels, rows, cols)

**Output shape:**

4D tensor with shape:

* If data\_format is "channels\_last": (batch\_size, padded\_rows, padded\_cols, channels)
* If data\_format is "channels\_first": (batch\_size, channels, padded\_rows, padded\_cols)

**fr\_utils.py**

import tensorflow as tf

import numpy as np

import os

import cv2

from numpy import genfromtxt

from keras.layers import Conv2D, ZeroPadding2D, Activation, Input, concatenate

from keras.models import Model

from keras.layers.normalization import BatchNormalization

from keras.layers.pooling import MaxPooling2D, AveragePooling2D

import h5py

import matplotlib.pyplot as plt

\_FLOATX = 'float32'

def variable(value, dtype=\_FLOATX, name=None):

v = tf.Variable(np.asarray(value, dtype=dtype), name=name)

\_get\_session().run(v.initializer)

return v

def shape(x):

return x.get\_shape()

def square(x):

return tf.square(x)

def zeros(shape, dtype=\_FLOATX, name=None):

return variable(np.zeros(shape), dtype, name)

def concatenate(tensors, axis=-1):

if axis < 0:

axis = axis % len(tensors[0].get\_shape())

return tf.concat(axis, tensors)

def LRN2D(x):

return tf.nn.lrn(x, alpha=1e-4, beta=0.75)

def conv2d\_bn(x,

layer=None,

cv1\_out=None,

cv1\_filter=(1, 1),

cv1\_strides=(1, 1),

cv2\_out=None,

cv2\_filter=(3, 3),

cv2\_strides=(1, 1),

padding=None):

num = '' if cv2\_out == None else '1'

tensor = Conv2D(cv1\_out, cv1\_filter, strides=cv1\_strides, data\_format='channels\_first', name=layer+'\_conv'+num)(x)

tensor = BatchNormalization(axis=1, epsilon=0.00001, name=layer+'\_bn'+num)(tensor)

tensor = Activation('relu')(tensor)

if padding == None:

return tensor

tensor = ZeroPadding2D(padding=padding, data\_format='channels\_first')(tensor)

if cv2\_out == None:

return tensor

tensor = Conv2D(cv2\_out, cv2\_filter, strides=cv2\_strides, data\_format='channels\_first', name=layer+'\_conv'+'2')(tensor)

tensor = BatchNormalization(axis=1, epsilon=0.00001, name=layer+'\_bn'+'2')(tensor)

tensor = Activation('relu')(tensor)

return tensor

WEIGHTS = [

'conv1', 'bn1', 'conv2', 'bn2', 'conv3', 'bn3',

'inception\_3a\_1x1\_conv', 'inception\_3a\_1x1\_bn',

'inception\_3a\_pool\_conv', 'inception\_3a\_pool\_bn',

'inception\_3a\_5x5\_conv1', 'inception\_3a\_5x5\_conv2', 'inception\_3a\_5x5\_bn1', 'inception\_3a\_5x5\_bn2',

'inception\_3a\_3x3\_conv1', 'inception\_3a\_3x3\_conv2', 'inception\_3a\_3x3\_bn1', 'inception\_3a\_3x3\_bn2',

'inception\_3b\_3x3\_conv1', 'inception\_3b\_3x3\_conv2', 'inception\_3b\_3x3\_bn1', 'inception\_3b\_3x3\_bn2',

'inception\_3b\_5x5\_conv1', 'inception\_3b\_5x5\_conv2', 'inception\_3b\_5x5\_bn1', 'inception\_3b\_5x5\_bn2',

'inception\_3b\_pool\_conv', 'inception\_3b\_pool\_bn',

'inception\_3b\_1x1\_conv', 'inception\_3b\_1x1\_bn',

'inception\_3c\_3x3\_conv1', 'inception\_3c\_3x3\_conv2', 'inception\_3c\_3x3\_bn1', 'inception\_3c\_3x3\_bn2',

'inception\_3c\_5x5\_conv1', 'inception\_3c\_5x5\_conv2', 'inception\_3c\_5x5\_bn1', 'inception\_3c\_5x5\_bn2',

'inception\_4a\_3x3\_conv1', 'inception\_4a\_3x3\_conv2', 'inception\_4a\_3x3\_bn1', 'inception\_4a\_3x3\_bn2',

'inception\_4a\_5x5\_conv1', 'inception\_4a\_5x5\_conv2', 'inception\_4a\_5x5\_bn1', 'inception\_4a\_5x5\_bn2',

'inception\_4a\_pool\_conv', 'inception\_4a\_pool\_bn',

'inception\_4a\_1x1\_conv', 'inception\_4a\_1x1\_bn',

'inception\_4e\_3x3\_conv1', 'inception\_4e\_3x3\_conv2', 'inception\_4e\_3x3\_bn1', 'inception\_4e\_3x3\_bn2',

'inception\_4e\_5x5\_conv1', 'inception\_4e\_5x5\_conv2', 'inception\_4e\_5x5\_bn1', 'inception\_4e\_5x5\_bn2',

'inception\_5a\_3x3\_conv1', 'inception\_5a\_3x3\_conv2', 'inception\_5a\_3x3\_bn1', 'inception\_5a\_3x3\_bn2',

'inception\_5a\_pool\_conv', 'inception\_5a\_pool\_bn',

'inception\_5a\_1x1\_conv', 'inception\_5a\_1x1\_bn',

'inception\_5b\_3x3\_conv1', 'inception\_5b\_3x3\_conv2', 'inception\_5b\_3x3\_bn1', 'inception\_5b\_3x3\_bn2',

'inception\_5b\_pool\_conv', 'inception\_5b\_pool\_bn',

'inception\_5b\_1x1\_conv', 'inception\_5b\_1x1\_bn',

'dense\_layer'

]

conv\_shape = {

'conv1': [64, 3, 7, 7],

'conv2': [64, 64, 1, 1],

'conv3': [192, 64, 3, 3],

'inception\_3a\_1x1\_conv': [64, 192, 1, 1],

'inception\_3a\_pool\_conv': [32, 192, 1, 1],

'inception\_3a\_5x5\_conv1': [16, 192, 1, 1],

'inception\_3a\_5x5\_conv2': [32, 16, 5, 5],

'inception\_3a\_3x3\_conv1': [96, 192, 1, 1],

'inception\_3a\_3x3\_conv2': [128, 96, 3, 3],

'inception\_3b\_3x3\_conv1': [96, 256, 1, 1],

'inception\_3b\_3x3\_conv2': [128, 96, 3, 3],

'inception\_3b\_5x5\_conv1': [32, 256, 1, 1],

'inception\_3b\_5x5\_conv2': [64, 32, 5, 5],

'inception\_3b\_pool\_conv': [64, 256, 1, 1],

'inception\_3b\_1x1\_conv': [64, 256, 1, 1],

'inception\_3c\_3x3\_conv1': [128, 320, 1, 1],

'inception\_3c\_3x3\_conv2': [256, 128, 3, 3],

'inception\_3c\_5x5\_conv1': [32, 320, 1, 1],

'inception\_3c\_5x5\_conv2': [64, 32, 5, 5],

'inception\_4a\_3x3\_conv1': [96, 640, 1, 1],

'inception\_4a\_3x3\_conv2': [192, 96, 3, 3],

'inception\_4a\_5x5\_conv1': [32, 640, 1, 1,],

'inception\_4a\_5x5\_conv2': [64, 32, 5, 5],

'inception\_4a\_pool\_conv': [128, 640, 1, 1],

'inception\_4a\_1x1\_conv': [256, 640, 1, 1],

'inception\_4e\_3x3\_conv1': [160, 640, 1, 1],

'inception\_4e\_3x3\_conv2': [256, 160, 3, 3],

'inception\_4e\_5x5\_conv1': [64, 640, 1, 1],

'inception\_4e\_5x5\_conv2': [128, 64, 5, 5],

'inception\_5a\_3x3\_conv1': [96, 1024, 1, 1],

'inception\_5a\_3x3\_conv2': [384, 96, 3, 3],

'inception\_5a\_pool\_conv': [96, 1024, 1, 1],

'inception\_5a\_1x1\_conv': [256, 1024, 1, 1],

'inception\_5b\_3x3\_conv1': [96, 736, 1, 1],

'inception\_5b\_3x3\_conv2': [384, 96, 3, 3],

'inception\_5b\_pool\_conv': [96, 736, 1, 1],

'inception\_5b\_1x1\_conv': [256, 736, 1, 1],

}

def load\_weights\_from\_FaceNet(FRmodel):

# Load weights from csv files (which was exported from Openface torch model)

weights = WEIGHTS

weights\_dict = load\_weights()

# Set layer weights of the model

for name in weights:

if FRmodel.get\_layer(name) != None:

FRmodel.get\_layer(name).set\_weights(weights\_dict[name])

elif model.get\_layer(name) != None:

model.get\_layer(name).set\_weights(weights\_dict[name])

def load\_weights():

# Set weights path

dirPath = './weights'

fileNames = filter(lambda f: not f.startswith('.'), os.listdir(dirPath))

paths = {}

weights\_dict = {}

for n in fileNames:

paths[n.replace('.csv', '')] = dirPath + '/' + n

for name in WEIGHTS:

if 'conv' in name:

conv\_w = genfromtxt(paths[name + '\_w'], delimiter=',', dtype=None)

conv\_w = np.reshape(conv\_w, conv\_shape[name])

conv\_w = np.transpose(conv\_w, (2, 3, 1, 0))

conv\_b = genfromtxt(paths[name + '\_b'], delimiter=',', dtype=None)

weights\_dict[name] = [conv\_w, conv\_b]

elif 'bn' in name:

bn\_w = genfromtxt(paths[name + '\_w'], delimiter=',', dtype=None)

bn\_b = genfromtxt(paths[name + '\_b'], delimiter=',', dtype=None)

bn\_m = genfromtxt(paths[name + '\_m'], delimiter=',', dtype=None)

bn\_v = genfromtxt(paths[name + '\_v'], delimiter=',', dtype=None)

weights\_dict[name] = [bn\_w, bn\_b, bn\_m, bn\_v]

elif 'dense' in name:

dense\_w = genfromtxt(dirPath+'/dense\_w.csv', delimiter=',', dtype=None)

dense\_w = np.reshape(dense\_w, (128, 736))

dense\_w = np.transpose(dense\_w, (1, 0))

dense\_b = genfromtxt(dirPath+'/dense\_b.csv', delimiter=',', dtype=None)

weights\_dict[name] = [dense\_w, dense\_b]

return weights\_dict

def load\_dataset():

train\_dataset = h5py.File('datasets/train\_happy.h5', "r")

train\_set\_x\_orig = np.array(train\_dataset["train\_set\_x"][:]) # your train set features

train\_set\_y\_orig = np.array(train\_dataset["train\_set\_y"][:]) # your train set labels

test\_dataset = h5py.File('datasets/test\_happy.h5', "r")

test\_set\_x\_orig = np.array(test\_dataset["test\_set\_x"][:]) # your test set features

test\_set\_y\_orig = np.array(test\_dataset["test\_set\_y"][:]) # your test set labels

classes = np.array(test\_dataset["list\_classes"][:]) # the list of classes

train\_set\_y\_orig = train\_set\_y\_orig.reshape((1, train\_set\_y\_orig.shape[0]))

test\_set\_y\_orig = test\_set\_y\_orig.reshape((1, test\_set\_y\_orig.shape[0]))

return train\_set\_x\_orig, train\_set\_y\_orig, test\_set\_x\_orig, test\_set\_y\_orig, classes

#Function for resizing an image

def pre\_process\_image(img, image\_size):

"""

Resizes an image into given image\_size (height, width, channel)

Arguments:

img -- original image, array

image\_size -- tuple containing width, height, channel of the image (h, w, c)

Returns:

img -- resized image

"""

height, width, channels = image\_size

img = cv2.resize(img, dsize=(height, width))

return img

def img\_to\_encoding(image\_path, model):

img1 = cv2.imread(image\_path, 1)

# resize the image

image\_size = (96, 96, 3)

img1 = pre\_process\_image(img1, image\_size)

img = img1[...,::-1]

img = np.around(np.transpose(img, (2,0,1))/255.0, decimals=12)

x\_train = np.array([img])

embedding = model.predict\_on\_batch(x\_train)

return embedding

**inception\_blocks.py**

import tensorflow as tf

import numpy as np

import os

from numpy import genfromtxt

from keras import backend as K

from keras.layers import Conv2D, ZeroPadding2D, Activation, Input, concatenate

from keras.models import Model

from keras.layers.normalization import BatchNormalization

from keras.layers.pooling import MaxPooling2D, AveragePooling2D

import fr\_utils

from keras.layers.core import Lambda, Flatten, Dense

def inception\_block\_1a(X):

"""

Implementation of an inception block

"""

X\_3x3 = Conv2D(96, (1, 1), data\_format='channels\_first', name ='inception\_3a\_3x3\_conv1')(X)

X\_3x3 = BatchNormalization(axis=1, epsilon=0.00001, name = 'inception\_3a\_3x3\_bn1')(X\_3x3)

X\_3x3 = Activation('relu')(X\_3x3)

X\_3x3 = ZeroPadding2D(padding=(1, 1), data\_format='channels\_first')(X\_3x3)

X\_3x3 = Conv2D(128, (3, 3), data\_format='channels\_first', name='inception\_3a\_3x3\_conv2')(X\_3x3)

X\_3x3 = BatchNormalization(axis=1, epsilon=0.00001, name='inception\_3a\_3x3\_bn2')(X\_3x3)

X\_3x3 = Activation('relu')(X\_3x3)

X\_5x5 = Conv2D(16, (1, 1), data\_format='channels\_first', name='inception\_3a\_5x5\_conv1')(X)

X\_5x5 = BatchNormalization(axis=1, epsilon=0.00001, name='inception\_3a\_5x5\_bn1')(X\_5x5)

X\_5x5 = Activation('relu')(X\_5x5)

X\_5x5 = ZeroPadding2D(padding=(2, 2), data\_format='channels\_first')(X\_5x5)

X\_5x5 = Conv2D(32, (5, 5), data\_format='channels\_first', name='inception\_3a\_5x5\_conv2')(X\_5x5)

X\_5x5 = BatchNormalization(axis=1, epsilon=0.00001, name='inception\_3a\_5x5\_bn2')(X\_5x5)

X\_5x5 = Activation('relu')(X\_5x5)

X\_pool = MaxPooling2D(pool\_size=3, strides=2, data\_format='channels\_first')(X)

X\_pool = Conv2D(32, (1, 1), data\_format='channels\_first', name='inception\_3a\_pool\_conv')(X\_pool)

X\_pool = BatchNormalization(axis=1, epsilon=0.00001, name='inception\_3a\_pool\_bn')(X\_pool)

X\_pool = Activation('relu')(X\_pool)

X\_pool = ZeroPadding2D(padding=((3, 4), (3, 4)), data\_format='channels\_first')(X\_pool)

X\_1x1 = Conv2D(64, (1, 1), data\_format='channels\_first', name='inception\_3a\_1x1\_conv')(X)

X\_1x1 = BatchNormalization(axis=1, epsilon=0.00001, name='inception\_3a\_1x1\_bn')(X\_1x1)

X\_1x1 = Activation('relu')(X\_1x1)

# CONCAT

inception = concatenate([X\_3x3, X\_5x5, X\_pool, X\_1x1], axis=1)

return inception

def inception\_block\_1b(X):

X\_3x3 = Conv2D(96, (1, 1), data\_format='channels\_first', name='inception\_3b\_3x3\_conv1')(X)

X\_3x3 = BatchNormalization(axis=1, epsilon=0.00001, name='inception\_3b\_3x3\_bn1')(X\_3x3)

X\_3x3 = Activation('relu')(X\_3x3)

X\_3x3 = ZeroPadding2D(padding=(1, 1), data\_format='channels\_first')(X\_3x3)

X\_3x3 = Conv2D(128, (3, 3), data\_format='channels\_first', name='inception\_3b\_3x3\_conv2')(X\_3x3)

X\_3x3 = BatchNormalization(axis=1, epsilon=0.00001, name='inception\_3b\_3x3\_bn2')(X\_3x3)

X\_3x3 = Activation('relu')(X\_3x3)

X\_5x5 = Conv2D(32, (1, 1), data\_format='channels\_first', name='inception\_3b\_5x5\_conv1')(X)

X\_5x5 = BatchNormalization(axis=1, epsilon=0.00001, name='inception\_3b\_5x5\_bn1')(X\_5x5)

X\_5x5 = Activation('relu')(X\_5x5)

X\_5x5 = ZeroPadding2D(padding=(2, 2), data\_format='channels\_first')(X\_5x5)

X\_5x5 = Conv2D(64, (5, 5), data\_format='channels\_first', name='inception\_3b\_5x5\_conv2')(X\_5x5)

X\_5x5 = BatchNormalization(axis=1, epsilon=0.00001, name='inception\_3b\_5x5\_bn2')(X\_5x5)

X\_5x5 = Activation('relu')(X\_5x5)

X\_pool = AveragePooling2D(pool\_size=(3, 3), strides=(3, 3), data\_format='channels\_first')(X)

X\_pool = Conv2D(64, (1, 1), data\_format='channels\_first', name='inception\_3b\_pool\_conv')(X\_pool)

X\_pool = BatchNormalization(axis=1, epsilon=0.00001, name='inception\_3b\_pool\_bn')(X\_pool)

X\_pool = Activation('relu')(X\_pool)

X\_pool = ZeroPadding2D(padding=(4, 4), data\_format='channels\_first')(X\_pool)

X\_1x1 = Conv2D(64, (1, 1), data\_format='channels\_first', name='inception\_3b\_1x1\_conv')(X)

X\_1x1 = BatchNormalization(axis=1, epsilon=0.00001, name='inception\_3b\_1x1\_bn')(X\_1x1)

X\_1x1 = Activation('relu')(X\_1x1)

inception = concatenate([X\_3x3, X\_5x5, X\_pool, X\_1x1], axis=1)

return inception

def inception\_block\_1c(X):

X\_3x3 = fr\_utils.conv2d\_bn(X,

layer='inception\_3c\_3x3',

cv1\_out=128,

cv1\_filter=(1, 1),

cv2\_out=256,

cv2\_filter=(3, 3),

cv2\_strides=(2, 2),

padding=(1, 1))

X\_5x5 = fr\_utils.conv2d\_bn(X,

layer='inception\_3c\_5x5',

cv1\_out=32,

cv1\_filter=(1, 1),

cv2\_out=64,

cv2\_filter=(5, 5),

cv2\_strides=(2, 2),

padding=(2, 2))

X\_pool = MaxPooling2D(pool\_size=3, strides=2, data\_format='channels\_first')(X)

X\_pool = ZeroPadding2D(padding=((0, 1), (0, 1)), data\_format='channels\_first')(X\_pool)

inception = concatenate([X\_3x3, X\_5x5, X\_pool], axis=1)

return inception

def inception\_block\_2a(X):

X\_3x3 = fr\_utils.conv2d\_bn(X,

layer='inception\_4a\_3x3',

cv1\_out=96,

cv1\_filter=(1, 1),

cv2\_out=192,

cv2\_filter=(3, 3),

cv2\_strides=(1, 1),

padding=(1, 1))

X\_5x5 = fr\_utils.conv2d\_bn(X,

layer='inception\_4a\_5x5',

cv1\_out=32,

cv1\_filter=(1, 1),

cv2\_out=64,

cv2\_filter=(5, 5),

cv2\_strides=(1, 1),

padding=(2, 2))

X\_pool = AveragePooling2D(pool\_size=(3, 3), strides=(3, 3), data\_format='channels\_first')(X)

X\_pool = fr\_utils.conv2d\_bn(X\_pool,

layer='inception\_4a\_pool',

cv1\_out=128,

cv1\_filter=(1, 1),

padding=(2, 2))

X\_1x1 = fr\_utils.conv2d\_bn(X,

layer='inception\_4a\_1x1',

cv1\_out=256,

cv1\_filter=(1, 1))

inception = concatenate([X\_3x3, X\_5x5, X\_pool, X\_1x1], axis=1)

return inception

def inception\_block\_2b(X):

#inception4e

X\_3x3 = fr\_utils.conv2d\_bn(X,

layer='inception\_4e\_3x3',

cv1\_out=160,

cv1\_filter=(1, 1),

cv2\_out=256,

cv2\_filter=(3, 3),

cv2\_strides=(2, 2),

padding=(1, 1))

X\_5x5 = fr\_utils.conv2d\_bn(X,

layer='inception\_4e\_5x5',

cv1\_out=64,

cv1\_filter=(1, 1),

cv2\_out=128,

cv2\_filter=(5, 5),

cv2\_strides=(2, 2),

padding=(2, 2))

X\_pool = MaxPooling2D(pool\_size=3, strides=2, data\_format='channels\_first')(X)

X\_pool = ZeroPadding2D(padding=((0, 1), (0, 1)), data\_format='channels\_first')(X\_pool)

inception = concatenate([X\_3x3, X\_5x5, X\_pool], axis=1)

return inception

def inception\_block\_3a(X):

X\_3x3 = fr\_utils.conv2d\_bn(X,

layer='inception\_5a\_3x3',

cv1\_out=96,

cv1\_filter=(1, 1),

cv2\_out=384,

cv2\_filter=(3, 3),

cv2\_strides=(1, 1),

padding=(1, 1))

X\_pool = AveragePooling2D(pool\_size=(3, 3), strides=(3, 3), data\_format='channels\_first')(X)

X\_pool = fr\_utils.conv2d\_bn(X\_pool,

layer='inception\_5a\_pool',

cv1\_out=96,

cv1\_filter=(1, 1),

padding=(1, 1))

X\_1x1 = fr\_utils.conv2d\_bn(X,

layer='inception\_5a\_1x1',

cv1\_out=256,

cv1\_filter=(1, 1))

inception = concatenate([X\_3x3, X\_pool, X\_1x1], axis=1)

return inception

def inception\_block\_3b(X):

X\_3x3 = fr\_utils.conv2d\_bn(X,

layer='inception\_5b\_3x3',

cv1\_out=96,

cv1\_filter=(1, 1),

cv2\_out=384,

cv2\_filter=(3, 3),

cv2\_strides=(1, 1),

padding=(1, 1))

X\_pool = MaxPooling2D(pool\_size=3, strides=2, data\_format='channels\_first')(X)

X\_pool = fr\_utils.conv2d\_bn(X\_pool,

layer='inception\_5b\_pool',

cv1\_out=96,

cv1\_filter=(1, 1))

X\_pool = ZeroPadding2D(padding=(1, 1), data\_format='channels\_first')(X\_pool)

X\_1x1 = fr\_utils.conv2d\_bn(X,

layer='inception\_5b\_1x1',

cv1\_out=256,

cv1\_filter=(1, 1))

inception = concatenate([X\_3x3, X\_pool, X\_1x1], axis=1)

return inception

def faceRecoModel(input\_shape):

"""

Implementation of the Inception model used for FaceNet

Arguments:

input\_shape -- shape of the images of the dataset

Returns:

model -- a Model() instance in Keras

"""

# Define the input as a tensor with shape input\_shape

X\_input = Input(input\_shape)

# Zero-Padding

X = ZeroPadding2D((3, 3))(X\_input)

# First Block

X = Conv2D(64, (7, 7), strides = (2, 2), name = 'conv1')(X)

X = BatchNormalization(axis = 1, name = 'bn1')(X)

X = Activation('relu')(X)

# Zero-Padding + MAXPOOL

X = ZeroPadding2D((1, 1))(X)

X = MaxPooling2D((3, 3), strides = 2)(X)

# Second Block

X = Conv2D(64, (1, 1), strides = (1, 1), name = 'conv2')(X)

X = BatchNormalization(axis = 1, epsilon=0.00001, name = 'bn2')(X)

X = Activation('relu')(X)

# Zero-Padding + MAXPOOL

X = ZeroPadding2D((1, 1))(X)

# Second Block

X = Conv2D(192, (3, 3), strides = (1, 1), name = 'conv3')(X)

X = BatchNormalization(axis = 1, epsilon=0.00001, name = 'bn3')(X)

X = Activation('relu')(X)

# Zero-Padding + MAXPOOL

X = ZeroPadding2D((1, 1))(X)

X = MaxPooling2D(pool\_size = 3, strides = 2)(X)

# Inception 1: a/b/c

X = inception\_block\_1a(X)

X = inception\_block\_1b(X)

X = inception\_block\_1c(X)

# Inception 2: a/b

X = inception\_block\_2a(X)

X = inception\_block\_2b(X)

# Inception 3: a/b

X = inception\_block\_3a(X)

X = inception\_block\_3b(X)

# Top layer

X = AveragePooling2D(pool\_size=(3, 3), strides=(1, 1), data\_format='channels\_first')(X)

X = Flatten()(X)

X = Dense(128, name='dense\_layer')(X)

# L2 normalization

X = Lambda(lambda x: K.l2\_normalize(x,axis=1))(X)

# Create model instance

model = Model(inputs = X\_input, outputs = X, name='FaceRecoModel')

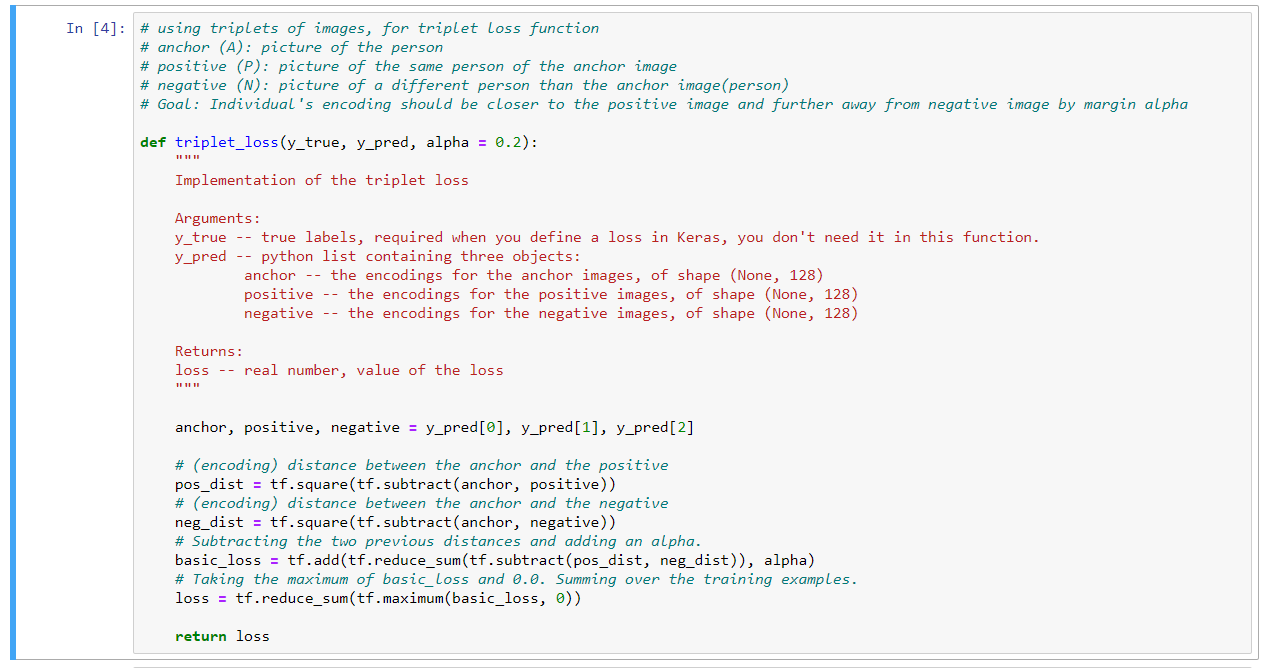
return model

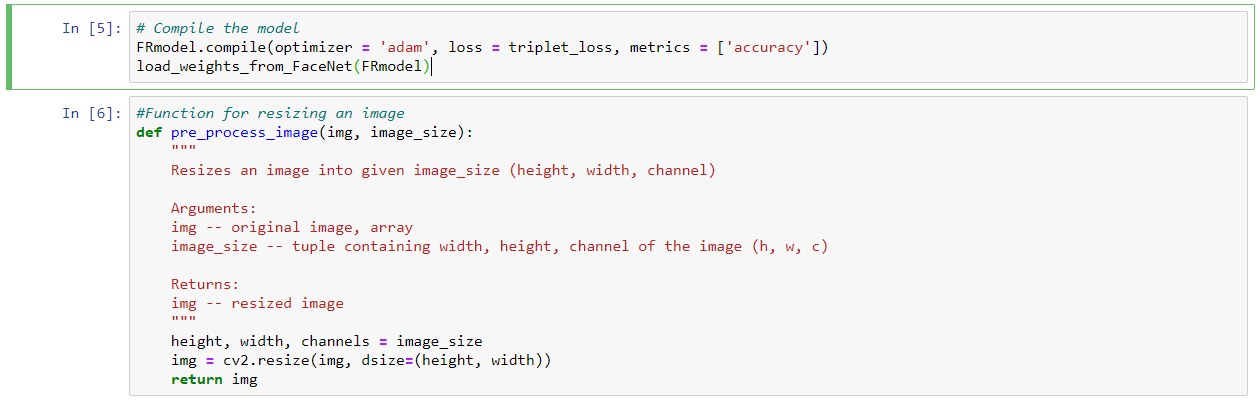
**FACE RECOGNITION IPYTHON NOTEBOOK:**

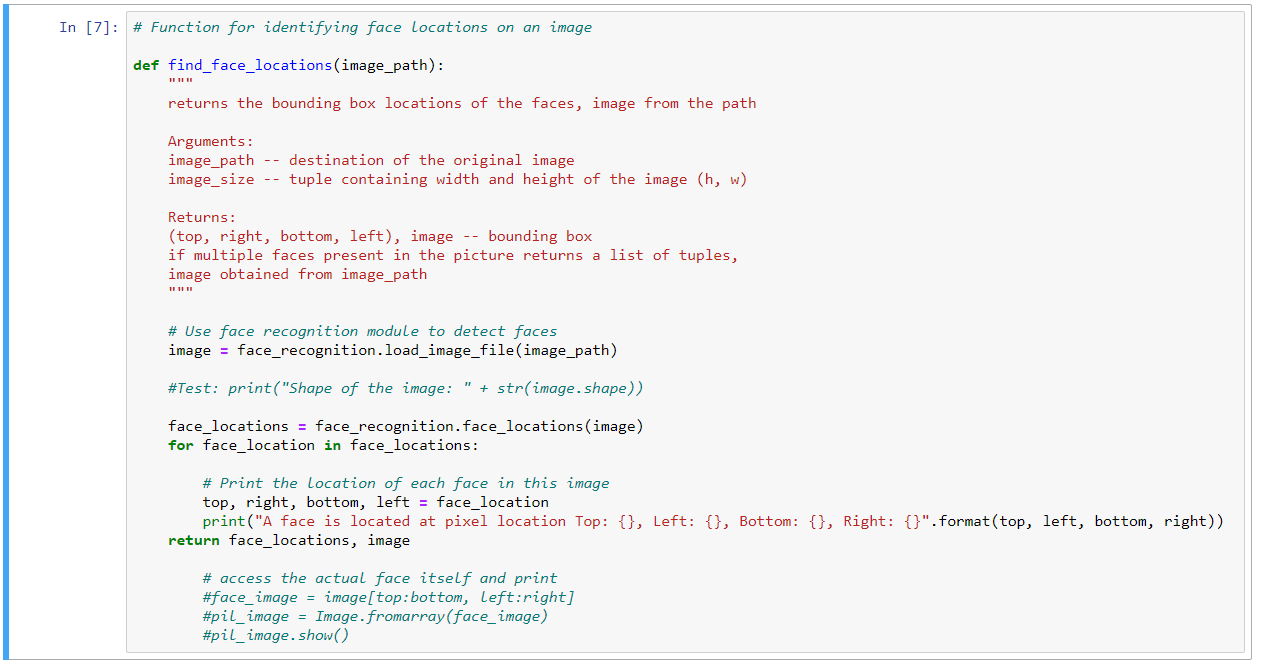
In this notebook, I implemented Face recognition system by using pre-trained model to map face images into 128 dimensional encodings.





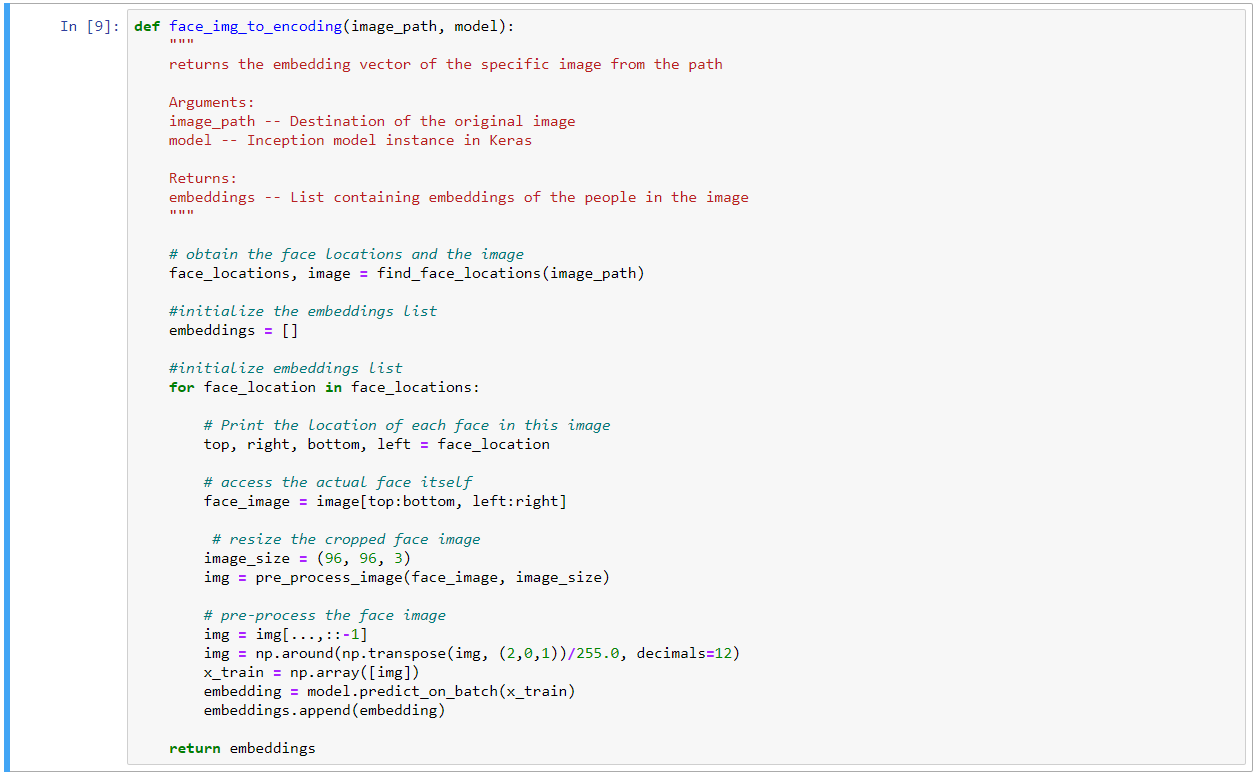




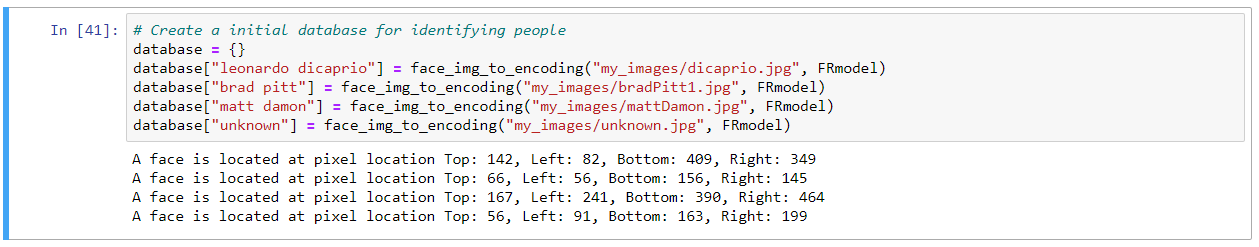


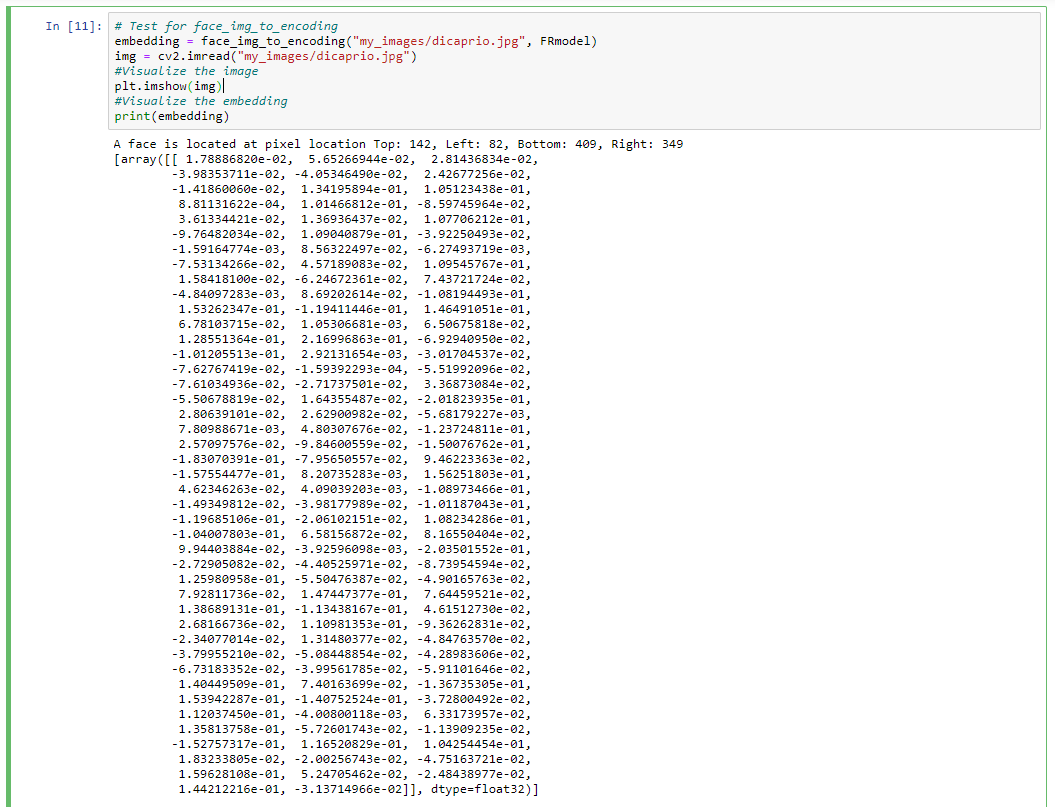
**IMAGE TO EMBEDDING:**

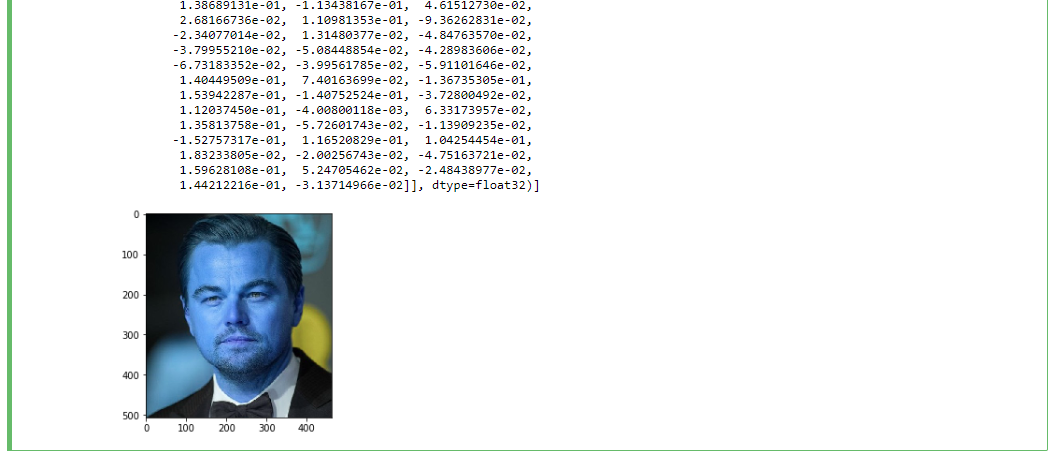
face\_img\_to\_encoding(image\_path, model) : basically runs the forward propagation of the model on the specified image.



**CREATING DATABASE:**







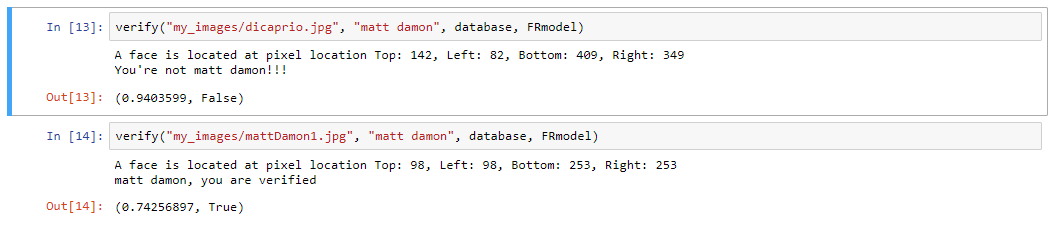
**FACE MATCHING:**

Face Verification is a 1:1 matching problem given identity of a person program identifies if the picture of a person matches with identity

- verify() function below implements simple face-verification functionality



**Let's see if we can verify Matt Damon:**

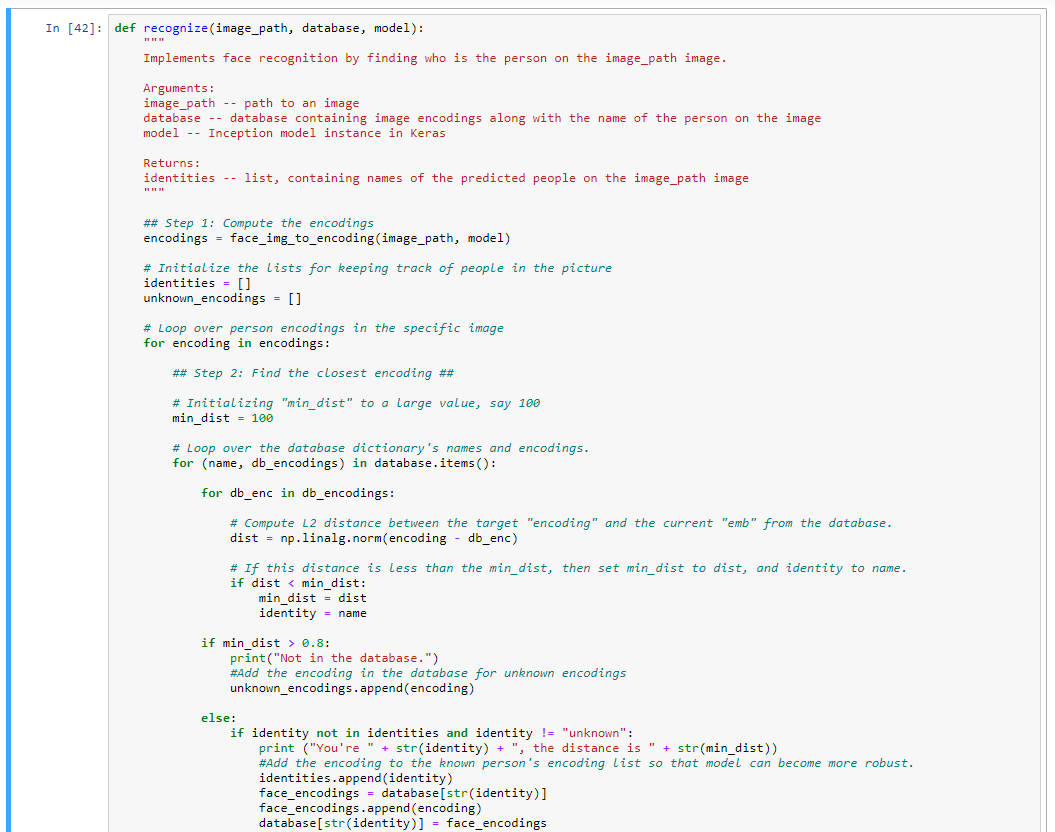


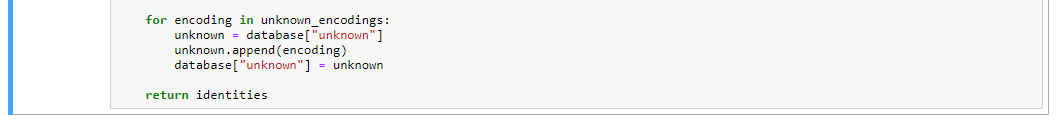
**FACE RECOGNITION:**

Identifies the person without needing to provide an identity. This is a 1:K matching problem.

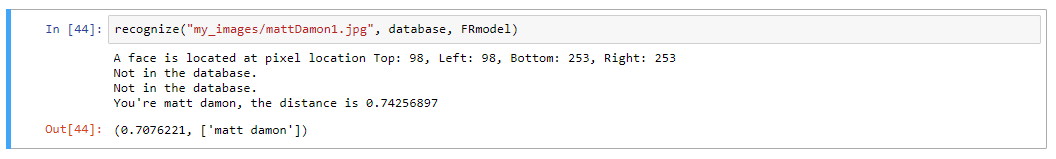
Steps:

1. Compute the target encoding of the image from image\_path
2. Find the encoding from the database that has smallest distance with the target encoding.





**OUTPUT:**



**RESULT:**

Hence Successfully built a face recognition application using python.