Face Recognition

The goal here is to build a face verification algorithm and then extend it to a face recognition system. Many of the ideas presented here are from FaceNet (https://arxiv.org/pdf/1503.03832.pdf) and DeepFace (https://research.fb.com/wp-content/uploads/2016/11/deepface-closing-the-gap-to-human-level-performance-in-face-verification.pdf).

Face recognition problems commonly fall into two categories:

- **Face Verification** - "is this the claimed person?". For example, at some airports, you can pass through customs by letting a system scan your passport and then verifying that you (the person carrying the passport) are the correct person. A mobile phone that unlocks using your face is also using face verification. This is a 1:1 matching problem.
- **Face Recognition** - "who is this person?". For example, this video shows a face recognition system (https://www.youtube.com/watch?v=wr4rx0Spihs (https://www.youtube.com/watch?v=wr4rx0Spihs)) for Baidu employees entering the office without needing to otherwise identify themselves. This is a 1:K matching problem.

FaceNet learns a neural network that encodes a face image into a vector of 128 numbers. By comparing two such vectors, you can then determine if two pictures are of the same person.

Here, we will be using a pre-trained model which represents ConvNet activations using a "channels first" convention, as opposed to the "channels last" convention. In other words, a batch of images will be of shape (m, n_C, n_H, n_W) instead of (m, n_H, n_W, n_C). Both of these conventions have a reasonable amount of traction among open-source implementations; there isn't a uniform standard yet within the deep learning community.

In [1]:
```
# load the required libraries
from keras.models import Sequential
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
from keras.layers.merge import Concatenate
from keras.layers.core import Lambda, Flatten, Dense
from keras.initializers import glorot_uniform
from keras.engine.topology import Layer
from keras import backend as K
K.set_image_data_format('channels_first')
import cv2
import os
import numpy as np
from numpy import genfromtxt
import pandas as pd
import tensorflow as tf
from fr_utils import *
from inception_blocks_v2 import *

%matplotlib inline
%load_ext autoreload
%autoreload 2

np.set_printoptions(threshold=np.nan)
```

```
Using TensorFlow backend.
```

Naive Face Verification

Here our model will learn an encoding $f(img)$ so that element-wise comparisons of this encoding gives an accurate judgement as to whether two pictures are of the same person.

Encoding face images into a 128-dimensional vector

Using an ConvNet to compute encodings

The FaceNet model takes a lot of data and a long time to train. So following common practice in applied deep learning settings, let's just load weights that someone else has already trained. The network architecture follows the Inception model from Szegedy *et al.* (https://arxiv.org/abs/1409.4842). An inception network implementation is contained within the referenced `inception_blocks.py` python file.

The key items to note about this network:

- This network uses 96x96 dimensional RGB images as its input. Specifically, inputs are a face image (or batch of m face images) as a tensor of shape $(m, n_C, n_H, n_W) = (m, 3, 96, 96)$
- It outputs a matrix of shape $(m, 128)$ that encodes each input face image into a 128-dimensional vector

In [2]:
```
# create the model for encoding the face images
FRmodel = faceRecoModel(input_shape=(3, 96, 96))
```

In [3]:
```
# review the loaded model
FRmodel.summary()
```

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 3, 96, 96)	0	
zero_padding2d_1 (ZeroPadding2D	(None, 3, 102, 102)	0	input_1[0][0]
conv1 (Conv2D)	(None, 64, 48, 48)	9472	zero_padding2d_1[0][0]
bn1 (BatchNormalization)	(None, 64, 48, 48)	256	conv1[0][0]
activation_1 (Activation)	(None, 64, 48, 48)	0	bn1[0][0]
zero_padding2d_2 (ZeroPadding2D	(None, 64, 50, 50)	0	activation_1[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 64, 24, 24)	0	zero_padding2d_2[0][0]
conv2 (Conv2D)	(None, 64, 24, 24)	4160	max_pooling2d_1[0][0]
bn2 (BatchNormalization)	(None, 64, 24, 24)	256	conv2[0][0]
activation_2 (Activation)	(None, 64, 24, 24)	0	bn2[0][0]
zero_padding2d_3 (ZeroPadding2D	(None, 64, 26, 26)	0	activation_2[0][0]
conv3 (Conv2D)	(None, 192, 24, 24)	110784	zero_padding2d_3[0][0]
bn3 (BatchNormalization)	(None, 192, 24, 24)	768	conv3[0][0]
activation_3 (Activation)	(None, 192, 24, 24)	0	bn3[0][0]
zero_padding2d_4 (ZeroPadding2D	(None, 192, 26, 26)	0	activation_3[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 192, 12, 12)	0	zero_padding2d_4[0][0]
inception_3a_3x3_conv1 (Conv2D)	(None, 96, 12, 12)	18528	max_pooling2d_2[0][0]
inception_3a_5x5_conv1 (Conv2D)	(None, 16, 12, 12)	3088	max_pooling2d_2[0][0]
inception_3a_3x3_bn1 (BatchNorm	(None, 96, 12, 12)	384	inception_3a_3x3_conv1[0][0]
inception_3a_5x5_bn1 (BatchNorm	(None, 16, 12, 12)	64	inception_3a_5x5_conv1[0][0]
activation_4 (Activation)	(None, 96, 12, 12)	0	inception_3a_3x3_bn1[0][0]
activation_6 (Activation)	(None, 16, 12, 12)	0	inception_3a_5x5_bn1[0][0]

By using a 128-neuron fully connected layer as its last layer, the model ensures that the output is an encoding vector of size 128. You then use the encodings the compare two face images as follows:

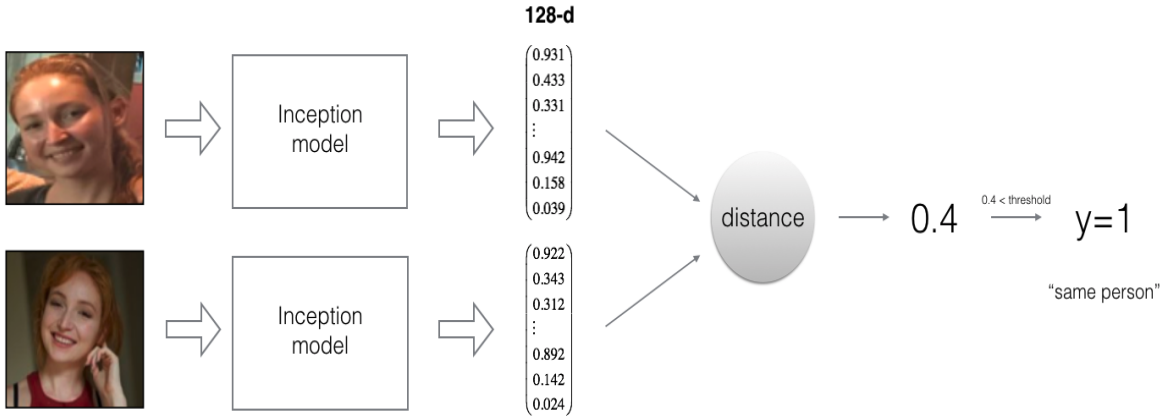


Figure 2:
By computing a distance between two encodings and thresholding, you can determine if the two pictures represent the same person

So, an encoding is a good one if:

- The encodings of two images of the same person are quite similar to each other
- The encodings of two images of different persons are very different

The triplet loss function formalizes this, and tries to "push" the encodings of two images of the same person (Anchor and Positive) closer together, while "pulling" the encodings of two images of different persons (Anchor, Negative) further apart.

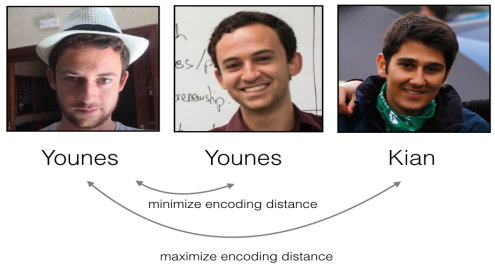
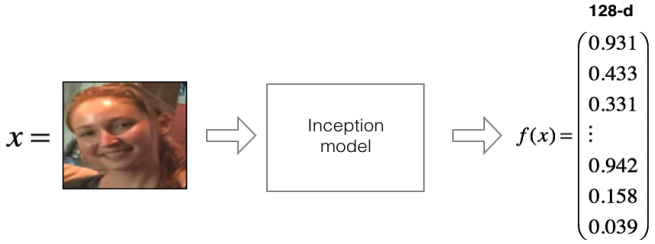


Figure 3:
Going forward, we will call the pictures from left to right: Anchor (A), Positive (P), Negative (N)

The Triplet Loss

For an image x, we denote its encoding $f(x)$, where f is the function computed by the neural network.



Training will use triplets of images (A, P, N):

- A is an "Anchor" image--a picture of a person.
- P is a "Positive" image--a picture of the same person as the Anchor image.
- N is a "Negative" image--a picture of a different person than the Anchor image.

These triplets are picked from our training dataset. We will write $(A^{(i)}, P^{(i)}, N^{(i)})$ to denote the i-th training example.

You'd like to make sure that an image $A^{(i)}$ of an individual is closer to the Positive $P^{(i)}$ than to the Negative image $N^{(i)}$) by at least a margin α:

$$|| f(A^{(i)}) - f(P^{(i)}) ||_2^2 + \alpha < || f(A^{(i)}) - f(N^{(i)}) ||_2^2$$

You would thus like to minimize the following "triplet cost":

$$\mathcal{J} = \sum_{i=1}^{m} [\underbrace{|| f(A^{(i)}) - f(P^{(i)}) ||_2^2}_{(1)} - \underbrace{|| f(A^{(i)}) - f(N^{(i)}) ||_2^2}_{(2)} + \alpha]_+ \tag{3}$$

Here, we are using the notation "$[z]_+$" to denote $max(z, 0)$.

Notes:

- The term (1) is the squared distance between the anchor "A" and the positive "P" for a given triplet; you want this to be small.
- The term (2) is the squared distance between the anchor "A" and the negative "N" for a given triplet, you want this to be relatively large, so it thus makes sense to have a minus sign preceding it.
- α is called the margin. It is a hyperparameter that you should pick manually. We will use $\alpha = 0.2$.

In [4]:
```
# create the triplet loss function

def triplet_loss(y_true, y_pred, alpha = 0.2):
    """
    Implementation of the triplet loss as defined by formula

    Arguments:
    y_true -- true labels, required when you define a loss in Keras, you don't need it in this function.
    y_pred -- python list containing three objects:
            anchor -- the encodings for the anchor images, of shape (None, 128)
            positive -- the encodings for the positive images, of shape (None, 128)
            negative -- the encodings for the negative images, of shape (None, 128)

    Returns:
    loss -- value of the loss as a real number
    """

    anchor, positive, negative = y_pred[0], y_pred[1], y_pred[2]
    # Note: here we use tf.reduce_sum instead of np.sum because numpy upcasts uint8 and int32 to int64
    # while tensorflow returns the same dtype as the input tensor.

    # Compute the (encoding) distance between the anchor and the positive, sum over axis=-1
    pos_dist = tf.reduce_sum(tf.square(anchor - positive), -1)
    # Compute the (encoding) distance between the anchor and the negative, sum over axis=-1
    neg_dist = tf.reduce_sum(tf.square(anchor - negative), -1)
    # subtract the two previous distances and add alpha.
    basic_loss = pos_dist - neg_dist + alpha
    # take the maximum of basic_loss and 0.0. Sum over the training examples.
    loss = tf.reduce_sum(tf.maximum(basic_loss, 0.0))

    return loss
```

Loading the trained model

FaceNet is trained by minimizing the triplet loss. But since training requires a lot of data and a lot of computation, we won't train it from scratch here. Instead, we load a previously trained model.

In [5]:
```
# compile the model using the triplet loss function and load the pre-trained weights
FRmodel.compile(optimizer = 'adam', loss = triplet_loss, metrics = ['accuracy'])
load_weights_from_FaceNet(FRmodel)
```

Here're some examples of distances between the encodings between three individuals:

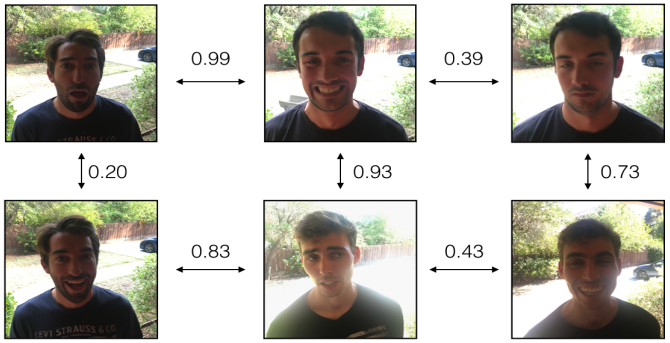


Figure 4:
Example of distance outputs between three individuals' encodings

Applying the model

Let's now use this model to perform face verification and face recognition.

Here we build a **Face verification** system so as to only let people from a specified list come in the door. To get admitted, each person has to swipe an ID card (identification card) to identify themselves at the door. The face verification system then checks that they are who they claim to be.

Face Verification

First we create a database containing one encoding vector for each person allowed to enter the building. To generate the encoding we use `img_path_to_encoding(image_path, model)` function which basically runs forward propagation of the model on the specified image.

Here, the database is represented as a python dictionary. This database maps each person's name to a 128-dimensional encoding of their face.

In [6]:
```
# create python dictionary database
database = {}
database["danielle"] = img_path_to_encoding("images/danielle.png", FRmodel)
database["younes"] = img_path_to_encoding("images/younes.jpg", FRmodel)
database["tian"] = img_path_to_encoding("images/tian.jpg", FRmodel)
database["andrew"] = img_path_to_encoding("images/andrew.jpg", FRmodel)
database["kian"] = img_path_to_encoding("images/kian.jpg", FRmodel)
database["dan"] = img_path_to_encoding("images/dan.jpg", FRmodel)
database["sebastiano"] = img_path_to_encoding("images/sebastiano.jpg", FRmodel)
database["bertrand"] = img_path_to_encoding("images/bertrand.jpg", FRmodel)
database["kevin"] = img_path_to_encoding("images/kevin.jpg", FRmodel)
database["felix"] = img_path_to_encoding("images/felix.jpg", FRmodel)
database["arnaud"] = img_path_to_encoding("images/arnaud.jpg", FRmodel)
```

In [7]:
```
# review one of the 128 dimensional encoding vectors
print(database["tian"])
```

```
[[ 0.0959584   0.04985306  0.10651767  0.11571032  0.13681366  0.18608071
   0.14005502 -0.10926028  0.01872359 -0.06331692 -0.06204369  0.05466765
   0.02865271  0.00248118  0.11283264 -0.05270756  0.05291301  0.07012399
  -0.10958154  0.01465024  0.01171816  0.05346058  0.04002672  0.18946882
   0.04128189 -0.20998392 -0.16843951 -0.0595573  -0.04289399  0.07861047
   0.07125293  0.13135675 -0.0974834   0.12935266  0.03672201 -0.01074999
   0.10086963  0.00816261 -0.04230201 -0.03034977  0.13557258  0.01277895
   0.05787134 -0.19190623 -0.12180469 -0.07851403  0.00319197  0.02082619
   0.0227944   0.06520662  0.10927535 -0.14072703  0.08944922  0.00400114
   0.07338614  0.0458586  -0.02479612  0.11365233  0.00805301 -0.00955212
  -0.0858904   0.12223675 -0.05246716 -0.31436285  0.10579244  0.09632063
   0.05157845 -0.07394308 -0.26784107  0.05522593 -0.00180449  0.1176901
  -0.0119062   0.06515745 -0.00638415  0.08521026  0.08004487 -0.08223655
   0.04531537 -0.03005511  0.01512369  0.06601492  0.01755629 -0.02023394
   0.07192096 -0.03858086  0.00286676 -0.01929288 -0.13027473  0.10597104
   0.07494818 -0.11774056  0.09238428  0.00793594 -0.06812259  0.02498502
  -0.01524812 -0.00616478  0.05843063  0.0910916   0.07100721  0.00154545
  -0.04580434  0.12697351 -0.06535149 -0.00428132 -0.02517836 -0.0920529
  -0.07862609  0.0683303  -0.00937787  0.06804062  0.07454986 -0.00070209
  -0.01900145 -0.00035591 -0.08317938  0.10539121 -0.12647772  0.10591158
   0.05556659  0.02230788 -0.04768839  0.01567735  0.01917292  0.10276749
  -0.06845059 -0.1271911 ]]
```

Now, when someone shows up at the front door and swipes their ID card (thus giving you their name), you can look up their encoding in the database and use it to check if the person standing at the front door matches the name on the ID. The below verify() function checks if the front-door camera picture (`image_path`) is actually the person called "identity".

Note: In this implementation, we compare the L2 distance, not the square of the L2 distance, to a threshold of 0.7.

In [8]:
```
# create the image verfication function

def verify(image_path, identity, database, model):
    """
    Function that verifies if the person on the "image_path" image is "identity".

    Arguments:
    image_path -- path to an image
    identity -- string, name of the person you'd like to verify the identity. Has to be a resident of the Happ
y house.
    database -- python dictionary mapping names of allowed people's names (strings) to their encodings (vector
s).
    model -- your Inception model instance in Keras

    Returns:
    dist -- distance between the image_path and the image of "identity" in the database.
    door_open -- True, if the door should open. False otherwise.
    """

    # compute the encoding for the image
    encoding = img_path_to_encoding(image_path, model)

    # compute the distance from the identity's image
    # Note: using the L2 distance (np.linalg.norm)
    dist = np.linalg.norm(database[identity] - encoding)

    # verification check - open the door if dist < 0.7, else don't open
    if dist < 0.7:
        print("It's " + str(identity) + ", welcome!")
        door_open = True
    else:
        print("You are not " + str(identity) + ", please go away!")
        door_open = False

    return dist, door_open
```

Younes is trying to enter the building and the camera takes a picture of him ("images/camera_0.jpg"). Let's run the verification algorithm on this picture:

```
In [9]: verify("images/camera_0.jpg", "younes", database, FRmodel)
```

```
It's younes, welcome!
```

```
Out[9]: (0.67100716, True)
```

Here is an image of someone not in our database. He stole Kian's ID card and came back to the building to try to present himself as Kian. The front-door camera took a picture ("images/camera_2.jpg) and ran the verification algorithm to check if he is indeed Kian.

```
In [10]: verify("images/camera_2.jpg", "kian", database, FRmodel)
```

```
You are not kian, please go away!
```

```
Out[10]: (0.8580017, False)
```

Face Recognition

Now extend the face verification system to a face recognition system. This way, no one has to carry an ID card anymore. An authorized person can just walk up to the building and the front door will unlock for them!

The face recognition system takes an image as input and figures out if it is one of the authorized persons (and if so, who). Unlike the previous face verification system, we will no longer get a person's name as another input.

In [11]:
```
# create the face recognition function

def who_is_it(image_path, database, model):
    """
    Implements face recognition by finding who is the person on the image_path image.

    Arguments:
    image_path -- path to an image
    database -- database containing image encodings along with the name of the person on the image
    model -- your Inception model instance in Keras

    Returns:
    min_dist -- the minimum distance between image_path encoding and the encodings from the database
    identity -- string, the name prediction for the person on image_path
    """

    # compute the target "encoding" for the image
    encoding = img_path_to_encoding(image_path, model)

    # find the closest encoding; we initialize "min_dist" to a large value, say 100
    min_dist = 100

    # loop over the database dictionary's names and encodings.
    for (name, db_enc) in database.items():

        # compute the L2 distance between the target "encoding" and the current "emb" from the database
        dist = np.linalg.norm(db_enc - encoding)

        # if this distance is less than the min_dist, then set min_dist to dist, and identity to name. (≈ 3 li
nes)
        if dist < min_dist:
            min_dist = dist
            identity = name

    # recognition check
    if min_dist > 0.7:
        print("You are not in our allowed database.")
    else:
        print ("It's " + str(identity) + ", and the image the distance is " + str(min_dist))

    return min_dist, identity
```

A person is at the front-door and the camera takes a picture of them ("images/camera_0.jpg"). Let's see if the who_it_is() algorithm identifies the person.

In [12]:
```
who_is_it("images/camera_0.jpg", database, FRmodel)
```

```
It's younes, and the image the distance is 0.67100716
```

Out[12]:
```
(0.67100716, 'younes')
```

Key Notes::

- Face verification solves an easier 1:1 matching problem; face recognition addresses a harder 1:K matching problem.
- The triplet loss is an effective loss function for training a neural network to learn an encoding of a face image.
- The same encoding can be used for verification and recognition. Measuring distances between two images' encodings allows you to determine whether they are pictures of the same person.
- You could crop the images to just contain the face, and less of the "border" region around the face. This preprocessing removes some of the irrelevant pixels around the face, and also makes the algorithm more robust

References:

- Florian Schroff, Dmitry Kalenichenko, James Philbin (2015). FaceNet: A Unified Embedding for Face Recognition and Clustering (https://arxiv.org/pdf/1503.03832.pdf)
- Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, Lior Wolf (2014). DeepFace: Closing the gap to human-level performance in face verification (https://research.fb.com/wp-content/uploads/2016/11/deepface-closing-the-gap-to-human-level-performance-in-face-verification.pdf)
- The pretrained model was inspired by Victor Sy Wang's implementation and was loaded using his code: https://github.com/iwantooxxoox/Keras-OpenFace (https://github.com/iwantooxxoox/Keras-OpenFace).
- This implementation also took a lot of inspiration from the official FaceNet github repository: https://github.com/davidsandberg/facenet (https://github.com/davidsandberg/facenet)