You have 2 free stories left this month. Sign up and get an extra one for free.

Face Recognition using OpenFace

Develop a face recognition system using a pre-trained open face model

Satyam Kumar Follow
Mar 27, 2019 · 10 min read ★

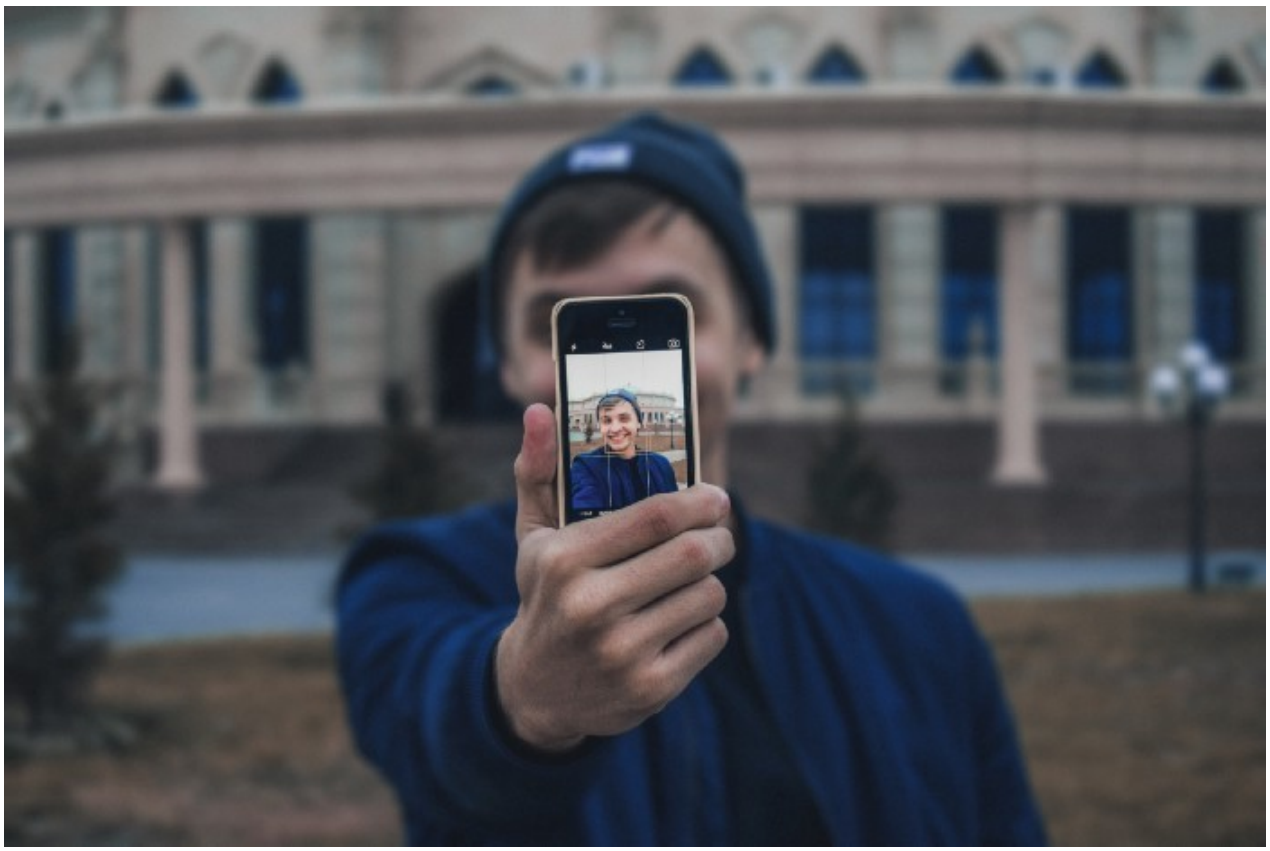


Photo by Dmitriy Tyukov on Unsplash

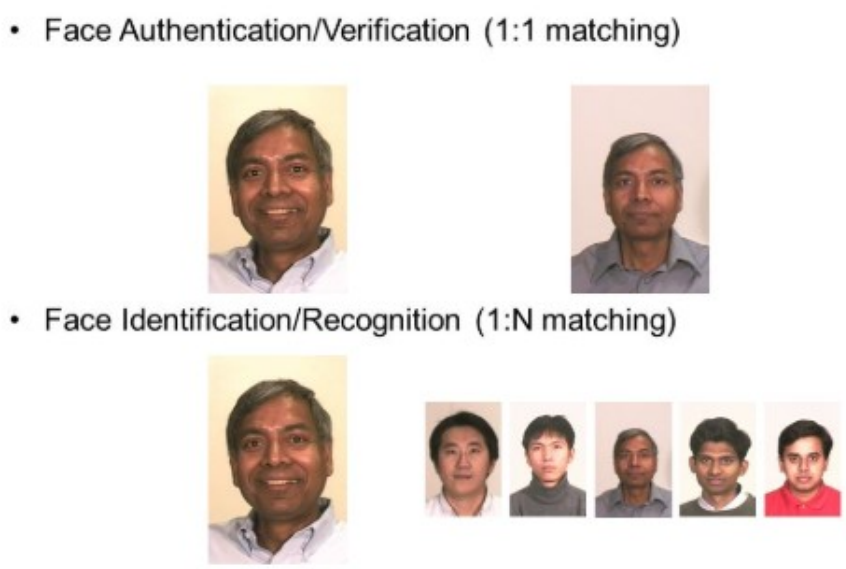
A **facial recognition system** is a technology capable of identifying or

verifying a person from a digital image or a video frame from a video source. There are multiple methods in which facial recognition systems work, but in general, they work by comparing selected facial features from a given image with faces within a database. It is also described as a Biometric Artificial Intelligence based application that can uniquely identify a person by analyzing patterns based on the person's facial textures and shape.

Facial recognition is being used in many businesses. Some of the uses of the face recognition system are to unlock phones, smart advertisement, tagging people on social media platforms, diagnose diseases, track attendance, payment verification, etc.

Difference between Face recognition and Face verification?

Face recognition, verification, and identification are often confused. Face recognition is a general topic that includes both face identification and face verification (also called authentication). On one hand, face verification is concerned with validating a claimed identity based on the image of a face, and either accepting or rejecting the identity claim (one-to-one matching). On the other hand, the goal of face identification is to identify a person based on the image of a face. This face image has to be compared with all the registered persons (one-to-many matching) in the database.



1:1 and 1:n matching

Here we are focusing on face verification.

Steps to follow :

1. Identification of faces from image
2. Projection of face
3. Open face implementation
4. Triplet Loss Function
5. Training and Classification a face recognition model

Identification of faces from image

One of the first steps in facial recognition software is to isolate the actual face from the background of the image along with isolating each face from others found in the image. Face detection algorithms also must be able to deal with bad and inconsistent lighting and various facial positions such as

tilted or rotated faces. For the process the face identification we have different techniques using dlib library and haar cascade classifier.

We can either implement dlib for face detection, which uses a combination of HOG (Histogram of Oriented Gradient) & Support Vector Machine (SVM), or OpenCV's Haar cascade classifier. Both are trained on positive and negative images (meaning some images have faces and ones that don't).

Advantage of Dlib over OpenCV's Haar Cascade Classifier

Dlib along with OpenCV can handle bad and inconsistent lighting and various facial positions such as tilted or rotated faces. Dlib is ahead of the Haar cascade classifier over implementation, speed, and accuracy. There are several benefits to using HOG classifier. First, the training is done using a sliding sub-window on the image so no subsampling and parameter manipulation is required like it is in the Haar classifier. This makes dlib's HOG and SVM face detection easier to use and faster to train. It also means that less data is required. Note that HOG has higher accuracy for face detection than the Haar cascade classifier. It kind of makes using dlib's HOG + SVM a no brainer for face detection!

For embedding for isolated face, we use OpenFace implementation which uses Google's FaceNet architecture which gives better output using the dlib library. We will proceed with dlib library implementation in this blog.

> ***You can find a detailed blog about working and implementation of HAAR CASCADE CLASSIFIER as 2nd part of this blog here***

Haar cascade Face Identification

A facial identification system is a technology capable of identifying a face of a person from a digital image or a...

medium.com

Identification of Face using Dlib library

Dlib for face detection uses a combination of HOG (Histogram of Oriented Gradient) & Support Vector Machine (SVM) which is trained on positive and negative images (meaning some images have faces and ones that don't).

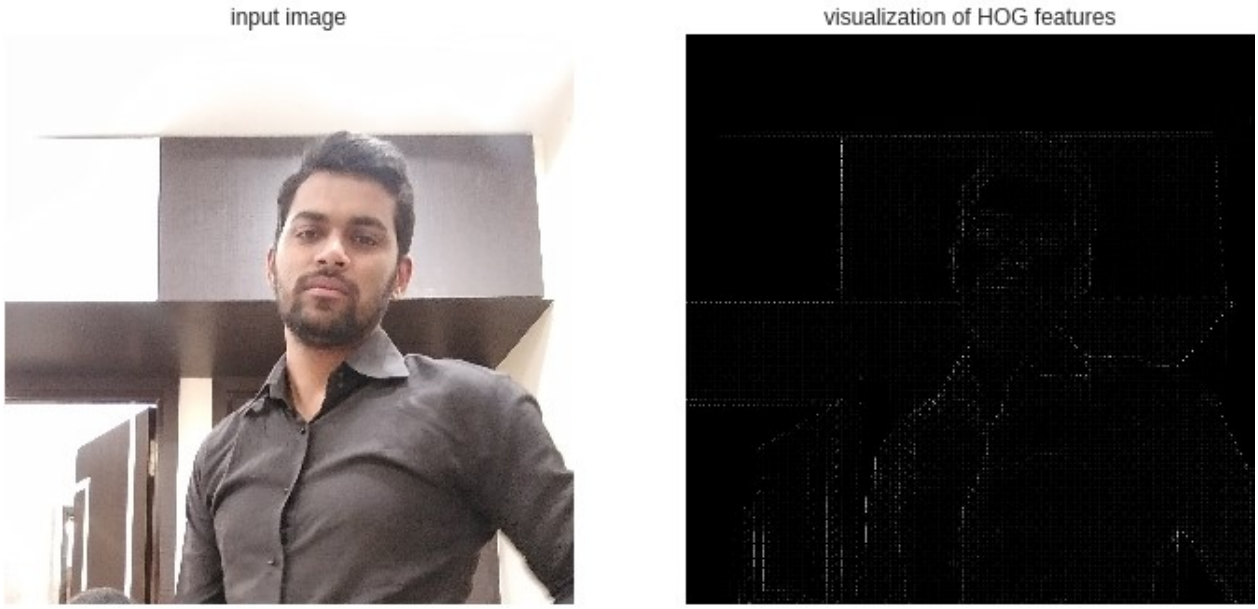
Steps it follows:

- We'll start by making our image black and white.
- For every single pixel, we want to look at the pixels that directly surrounding it.
- Our goal is to figure out how dark the current pixel is compared to the pixels directly surrounding it. Then we want to draw an arrow showing in which direction the image is getting darker
- If you repeat that process for **every single pixel** in the image, you end up with every pixel being replaced by an arrow. These arrows are called *gradients* and they show the flow from light to dark across the entire image

But saving the gradient for every single pixel gives us way too much detail. It would be better if we could just see the basic flow of lightness/darkness at a higher level so we could see the basic pattern of the image.

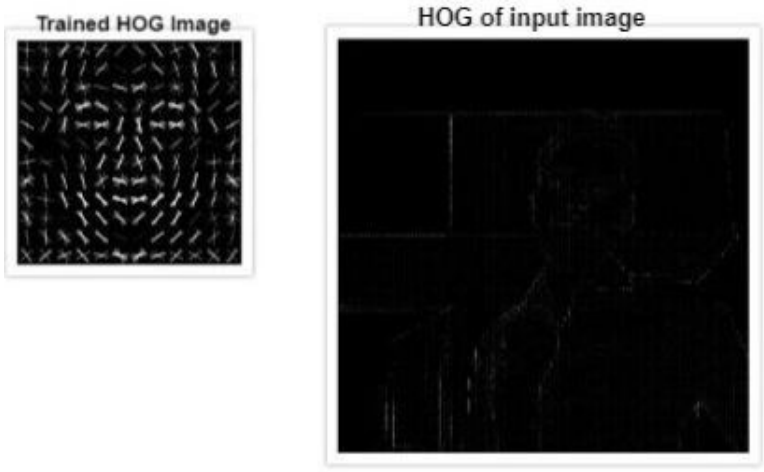
To do this, we'll break up the image into small squares of 16*16 pixels each. In each square, we'll count up how many gradients point in each major direction (how many point up, point up-right, point right, etc…). Then

we'll replace that square in the image with the arrow directions that were the strongest.



Convert input image to HOG visualization

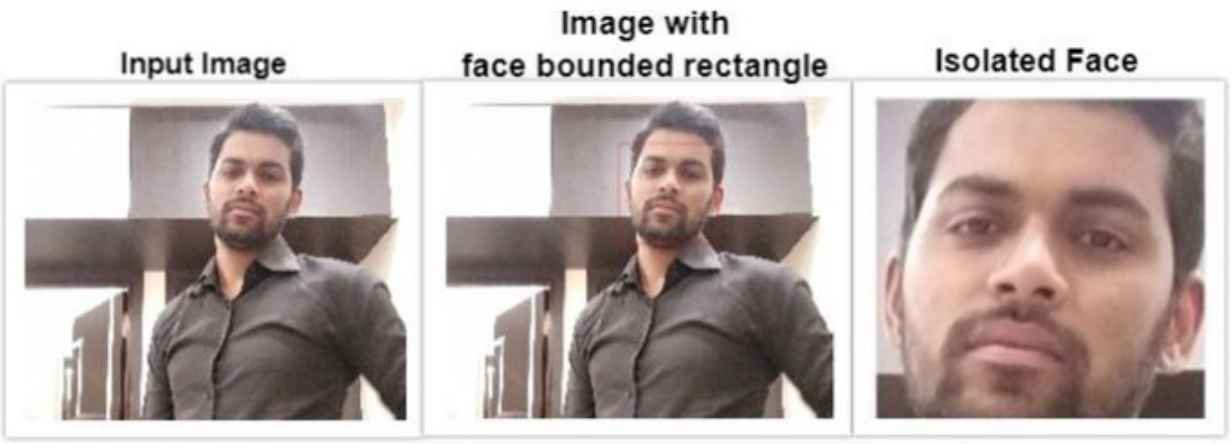
- Now to identify faces we need to find the part of our image that looks the most similar to a known HOG pattern that was extracted from a bunch of other training faces



Searching trained HOG pattern in a image

Identification of face from an image isolating from the background using

HOG recognition. You can see the observation below:



Isolating a face from the background

```
img = cv2.imread("input_image.jpg")
img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
alignment = AlignDlib('landmarks.dat')
face = alignment.getLargestFaceBoundingBox(img)
x = face.left()
y = face.top()
w = face.width()
h = face.height()
img1 = cv2.rectangle(img ,(x,y),(x+w,y+h),(255, 0, 0), 2)
plt.subplot(131)
plt.imshow(img)
plt.subplot(132)
plt.imshow(img1)
plt.subplot(133)
crop_img = img1[y:y+h, x:x+w]
plt.imshow(crop_img)
```

face identification hosted with ❤ by **GitHub** **view raw**

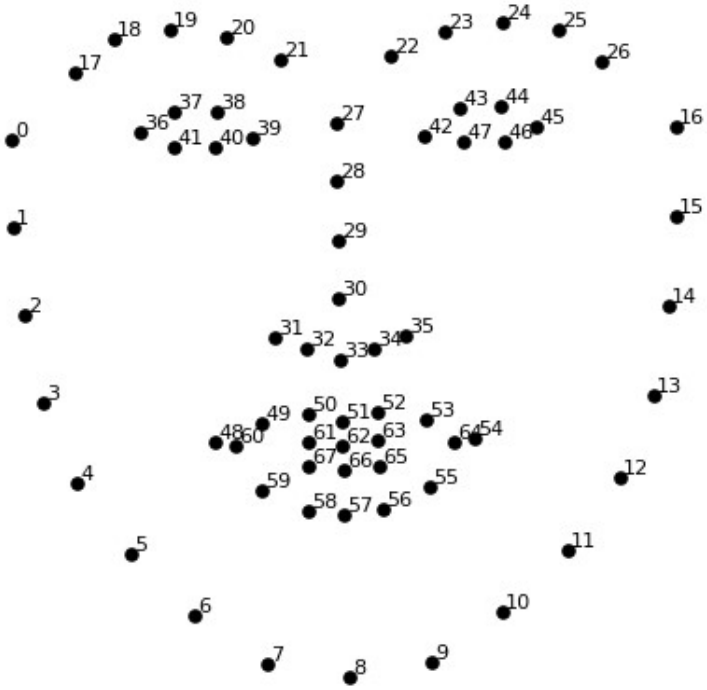
Projection of Face

When we isolate face from the image, the face of a person can be projected in any way or the person may look in any direction. When we create embedding of an isolated face, the embedding of the face of the same

person projected in different ways can vary a lot.

We will try to warp each picture so that the eyes and lips are always in the sample place in the image. This will make it a lot easier for us to compare faces. To do this, we are going to use an algorithm called **face landmark estimation**.

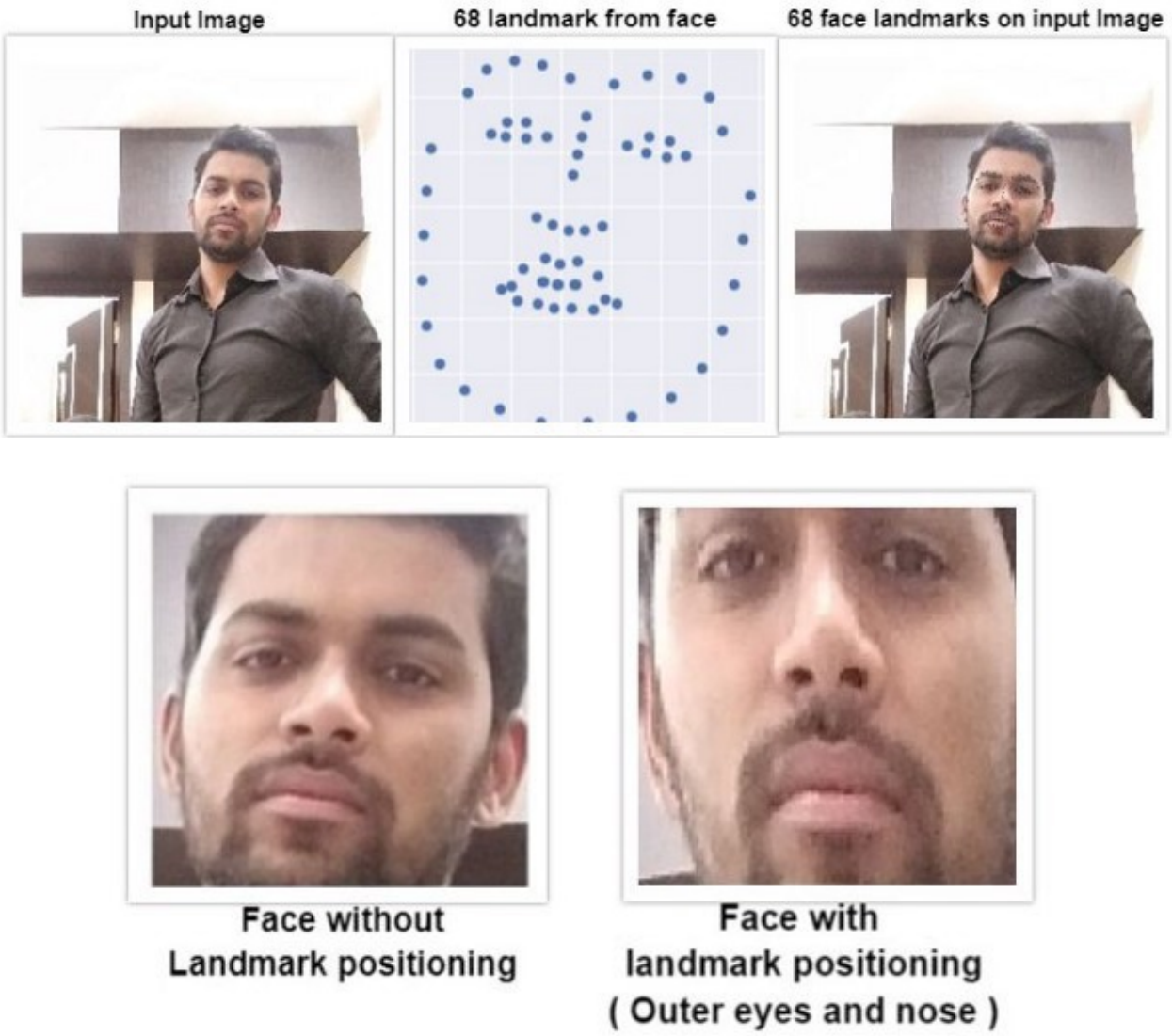
The basic idea is we will come up with 68 specific points (called *landmarks*) that exist on every face. Then we will train a machine-learning algorithm to be able to find these 68 specific points on any face as seen in the figure:



- The *mouth* can be accessed through points *[48, 67]*.
- The *right eyebrow* through points *[17, 21]*.
- The *left eyebrow* through points *[22, 26]*.
- The *right eye* using *[36, 41]*.

- The *left eye* with *[42, 47]*.
- The *nose* using *[27, 34]*.
- And the jaw via *[0, 16]*.

Now we can identify these landmarks in every face we'll simply rotate, scale and shear the image so that the eyes and mouth are centered as best as possible.



> *Now no matter how face is projected the output image having isolated face will always have* outer eyes and nose positioned at same position.

This will make our next step to finding embedding of the face using OpenFace a lot more accurate.

```
face_aligned = alignment.align(96, img,
alignment.getLargestFaceBoundingBox(img),
landmarkIndices=AlignDlib.OUTER_EYES_AND_NOSE)

plt.imshow(face_aligned)
```

OPEN FACE

After we isolate the image from the background and preprocess it using dlib

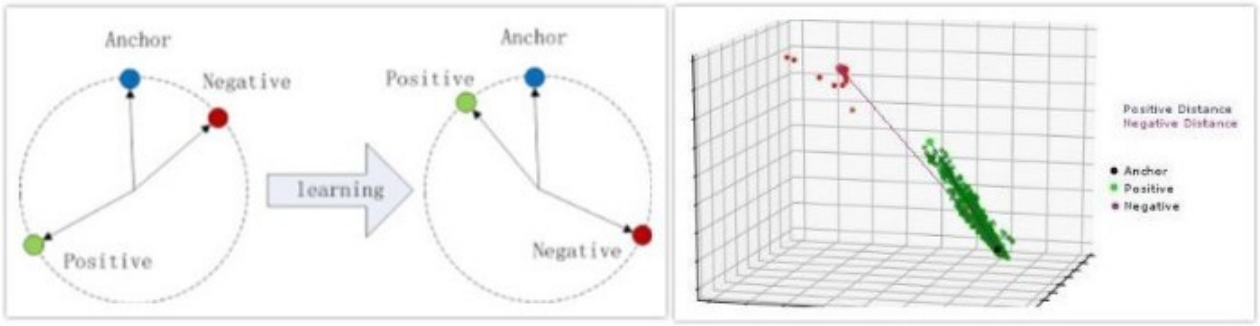
we need to find a way to represent the face in numerical embedding. We can represent it using a pre-trained deep neural network OpenFace.

During the training portion of the OpenFace pipeline, 500,000 images are passed through the neural net. OpenFace trains these images to produce 128 facial embeddings that represent a generic face. OpenFace uses Google's FaceNet architecture for feature extraction and uses a triplet loss function to test how accurate the neural net classifies a face.

Triplet Loss Function:

To learn the parameters of neural network we need to train on every three different sets of images one is known to face image called *anchor image,* another image of the same person as anchor image called a positive image, and 3rd image of a different person than anchor image called negative image.

Neural networks need to be trained in such a way that embedding of anchor image and positive image should be similar and embedding of anchor image and negative image should be much farther apart.



minimizing the distance between embedding of anchor and positive image & maximizing distance between embedding of anchor and negative image

> *Distance b/w anchor embedding & positive embedding* $= ||F(A)-F(P)||^2$
> *Distance b/w anchor embedding & negative embedding* $= ||F(A)-F(N)||^2$

Equation, $||F(A)-F(P)||^2 - ||F(A)-F(N)||^2 \leq 0$

We want our **dist(A, P)** to be less and **dist(A, N)** be higher. To make sure NN does not just make **dist(A, P)** & **dist(A, N)** equate to zero to satisfy the equation so we decrease RHS from 0 to lesser margin value (α).

> *Now equation boils down to ,* $||F(A)-F(P)||^2 - ||F(A)-F(N)||^2 \leq -\alpha$
>
> $||F(A)-F(P)||^2 - ||F(A)-F(N)||^2 \leq -\alpha$

For N set of triplet images,

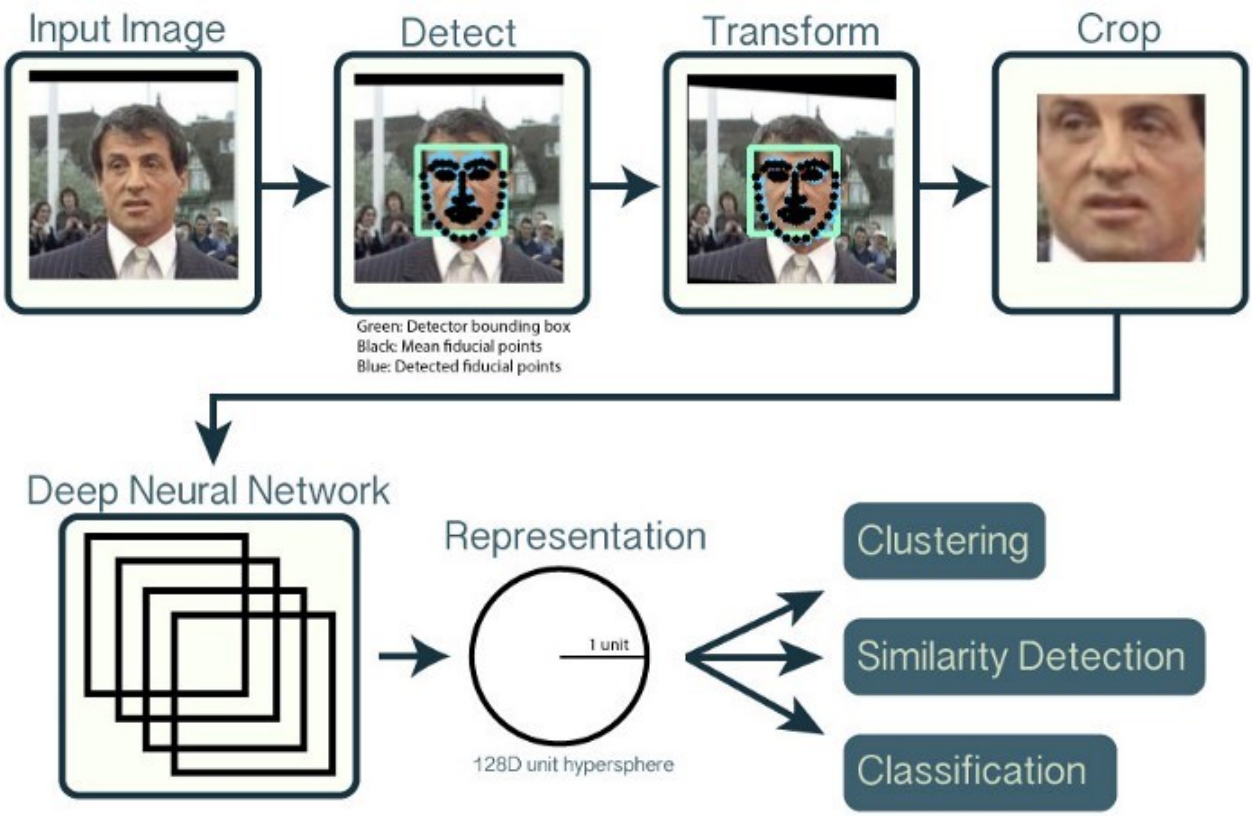
$$Loss = \sum_{i=1}^{N} \left[\| f_i^a - f_i^p \|_2^2 - \| f_i^a - f_i^n \|_2^2 + \alpha \right]_+$$

But once the network has been trained, it can generate measurements for any face, even ones it has never seen before! So this step only needs to be done once. Lucky for us, the fine folks at OpenFace already did this and they published several trained networks that we can directly use. Thanks, Brandon Amos and team!

So all we need to do ourselves is run our face images through their pre-trained open face network to get the 128 measurements for each face. The embedding is a generic representation of anybody's face. Unlike other face representations, this embedding has the nice property that a larger distance between two face embeddings means that the faces are likely not of the same person.

This property makes clustering, similarity detection, and classification tasks easier than other face recognition techniques where the Euclidean distance between features is not meaningful.

The methods for face recognition explained above is represented in an image taken from the blog of creators of the open face.



Source: http://cmusatyalab.github.io/openface/

```
# weights and function code is available in my github
model = create_model()
model.load_weights('open_face.h5')
```

Closest Matching Embedding:

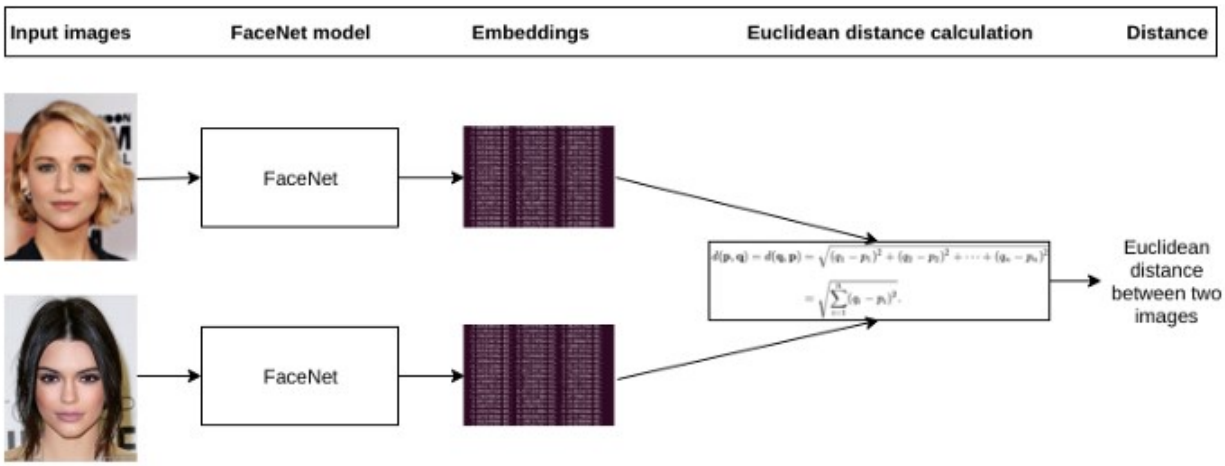
The 128-dimensional embedding returned by the FaceNet model can be used to clusters faces effectively, as discussed above distance of embedding would be closer for similar faces and further away for non-similar faces.

$$d(\mathbf{p}, \mathbf{q}) = d(\mathbf{q}, \mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$

$$= \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}.$$

Euclidean Distance between two points

To compare two images, create the embedding for both images by feeding through the model separately. Then we can use Euclidean distance to find the distance which will be lower value for similar faces and higher value for different faces.



One-Shot Learning using FaceNet

For each query image, we compute the euclidean distance between embedding of the query image and embedding of each of the images present in the dataset. The image having the smallest euclidean distance can be considered as an image of the same person as in query image.

```
# To find the euclidean distance between the two embeddings
def distance(emb1, emb2):
 return np.sum(np.square(emb1 - emb2))
```

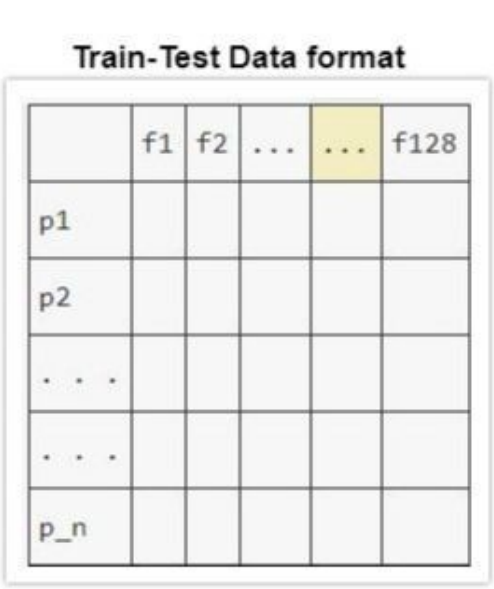
euclidean dist hosted with ❤ by **GitHub** **view raw**

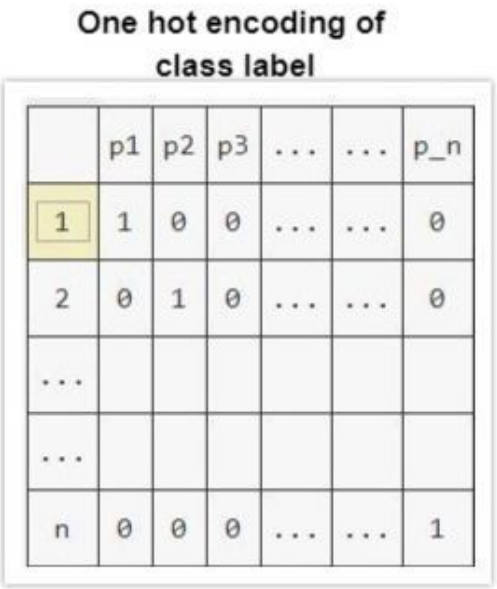
Training a Classifier:

Now we have to embed each of the images in the dataset, we can train a classifier taking face embedding as train data and names as the class labels of train data. Train a machine learning model using different techniques and hyperparameter tune each of the models to get the best accuracy.

Prepare train data

Split the given dataset into train and test data in such a way that the test dataset does not have a row of a unique personality that is not present is train data.





```
#Creating one-hot encoding of names of person

vectorizer = CountVectorizer(tokenizer = lambda x: x.split(), binary='true')
y = vectorizer.fit_transform(names)

# train data: X_train        train label : y_train
# test data : X_test         test label : y_test
```

vectorization hosted with ❤ by GitHub view raw

Define NN architecture

```
output_dim = n
input_dim = 128
batch_size = 128
nb_epoch = 100
```

A sequential NN model is created using “relu” as an activation unit and initializing the weights with He normal initialization.

- Output dimension is the number of unique faces of the person (n=5749 in my case)
- Input is 128-D embedding vector
- Create a 2-Layer NN model adding batch normalization and dropout.
- Train for batch_size of 128 and the number of epochs of 100

Architecture

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_16 (Dense)             (None, 500)               64500
_________________________________________________________________
batch_normalization_3 (Batch (None, 500)               2000
_________________________________________________________________
dropout_3 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_17 (Dense)             (None, 150)               75150
_________________________________________________________________
batch_normalization_4 (Batch (None, 150)               600
_________________________________________________________________
dropout_4 (Dropout)          (None, 150)               0
_________________________________________________________________
dense_18 (Dense)             (None, 5749)              868099
=================================================================
Total params: 1,010,349
Trainable params: 1,009,049
Non-trainable params: 1,300
```

Compile for number of epochs and observe the results

Summary:

1. **Create a raw image directory:**

Create a directory for your raw images so that images from different people are indifferent sub-directories. The names of the labels or images do not matter, and each person can have a different amount of images. The images should be formatted as `jpg` or `png` and have a lowercase extension.

```
$ tree data/mydataset/raw
person-1
├── image-1.jpg
├── image-2.png
...
└── image-p.png

...
```

Machine Learning Face Recognition Deep Learning Data Science Artificial Intelligence

```
├── image-1.png
├── image-2.jpg
...
└── image-q.png
```

Discover Medium

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Make Medium yours

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Become a member

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just $5/month. Upgrade

2. Preprocess the raw images

Preprocessing or raw images resembles to isolate the face from the image. Either using dlib (discussed above) or using haar cascade detector (discussed here)

Isolate each face in the image and align the face using dlib such that every face ha OUTER EYES and NOSE present at the same position.

3. Generate the embedding

Load the saved weights of the pre-trained OpenFace model and predict the 128-D embedding of each of the isolated faces.

4. Training a Classification model

Prepare train data and either train a machine learning model or NN model and sa the model.

Later this model can be deployed for prediction of a face in an image.

You can **find the entire implementation of the project** on my GitHub repository **here**.

> *Thank You for reading*
>
> *Please give 👏 Claps if you like the blog*

References:

- This process is detailed in the paper *FaceNet: A Unified Embedding for Face Recognition and Clustering by Florian Schroff, Dmitry Kalenichenko, and James Philbin.*
- https://cmusatyalab.github.io/openface/
- https://medium.com/@ageitgey/machine-learning-is-fun-part-4-modern-fac recognition-with-deep-learning-c3cffc121d78