



FACE RECOGNITION

With OpenCV, OpenFace, FaceNet, YOLO, Faced

Comparing between different methods for face recognition

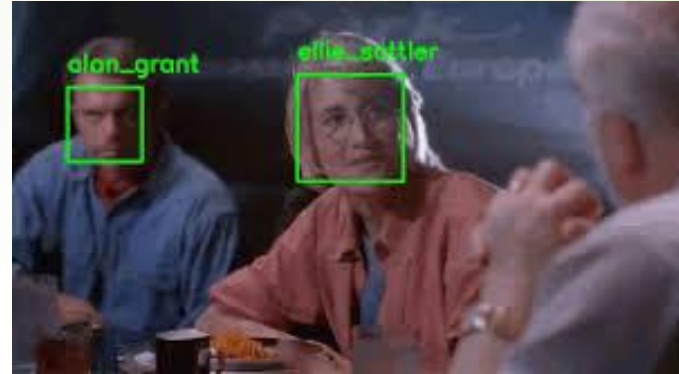
Author: M. Kolaksazov

Sources: Analytics Vidhya: <https://medium.com/analytics-vidhya/how-to-build-a-face-detection-model-in-python-8dc9cecadfe9>
Medium Corporation: <https://medium.com/@sumantrajoshi/face-recognizer-application-using-a-deep-learning-model-python-and-keras-2873e9aa6ab3>
<https://towardsdatascience.com/faced-cpu-real-time-face-detection-using-deep-learning-1488681c1602>

Face detection



face recognition



and

Computer vision is a very popular and important field in the machine learning and deep learning community at the moment. One of the most popular applications of this domain is **face detection** and **face recognition**.

► Difference between face verification and face recognition

Face verification technique is used to verify whether the input image is a face (1 to 1 mapping). In contrast, face recognition is used to recognize whether the input face image is from a set of an authorized group of individuals (1 to M mapping)



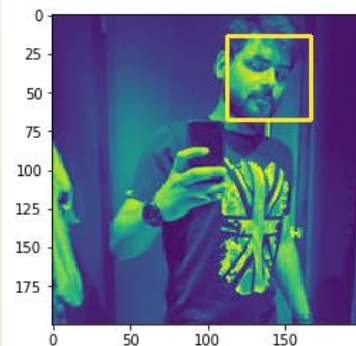
Real-life uses of face recognition

- Access and security
 - Prevent crimes
 - Unlock phones
 - Smarter advertising
 - Find missing persons
 - Help blind people
 - identify illnesses
- 

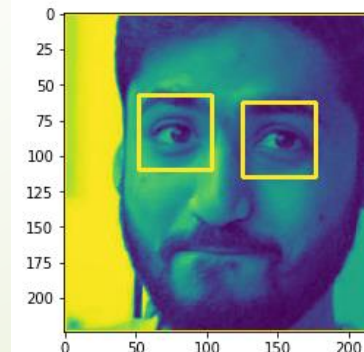
How Face detection works

- While the process is somewhat complex, face detection algorithms often begin by searching for human eyes. Eyes constitute what is known as a valley region and are one of the easiest features to detect. Once eyes are detected, the algorithm might then attempt to detect facial regions including eyebrows, the mouth, nose, nostrils, and the iris. After that, the algorithm confirms that it has detected a facial region, applying additional tests to validate whether it has, in fact, detected a face

```
In [70]: runfile('D:/KJ/Nagesh/Downloads/face_recognition')  
( 'Total number of Faces found : ', 1)
```



```
In [74]: runfile('D:/KJ/Nagesh/Downloads/face_recognition')  
( 'Total number of Faces found : ', 1)
```



OpenCV



- OpenCV (Open Source Computer Vision Library) is an image and video processing library with bindings in C++, C, Python, and Java. OpenCV is used for all sorts of image and video analysis, like facial recognition and detection, license plate reading, photo editing, advanced robotic vision, optical character recognition, and a whole lot more.
- OpenCV has three built-in face recognizers and thanks to its clean coding, you can use any of them just by changing a single line of code. Here are the names of those face recognizers and their OpenCV calls:
- EigenFaces—`cv2.face.createEigenFaceRecognizer()`
FisherFaces—`cv2.face.createFisherFaceRecognizer()`
Local Binary Patterns Histograms (LBPH)—`cv2.face.createLBPHFaceRecognizer()`

Detect faces using OpenCV

➤ There are two main ways to find faces using OpenCV:

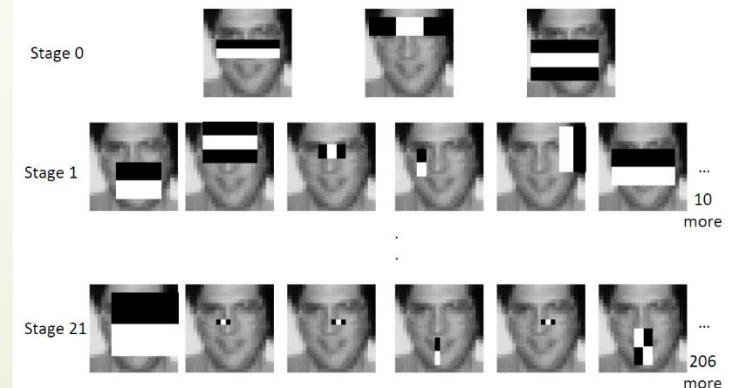
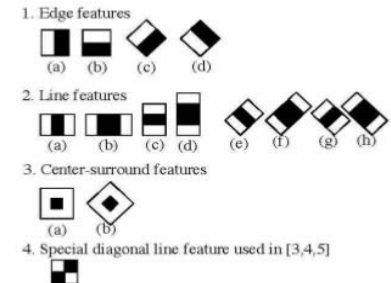
1. Haar Classifier
2. LBP Cascade Classifier

Algorithm	Metrics			
	Detection Accuracy	Computation Complexity	False positives	Robustness to different lighting conditions
HAAR	High	Complex and slow	Low	Less robust
LBP	Low	Simple and fast	High	Highly robust

➤ Most developers use **Haar** because it is more accurate, but it is also much slower than LBP. The OpenCV package has all the data needed to use Haar efficiently. An **XML file** is required, in order to write and read the training face data.

Haar-like features

- The difference of the sum of pixels of areas inside the rectangle
- The values indicate certain characteristics of a particular area of the image.



Cascade CNN for face detection

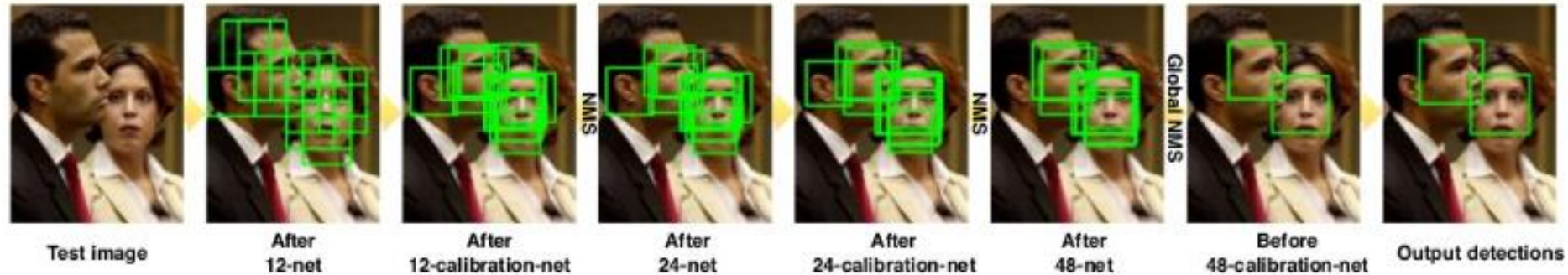
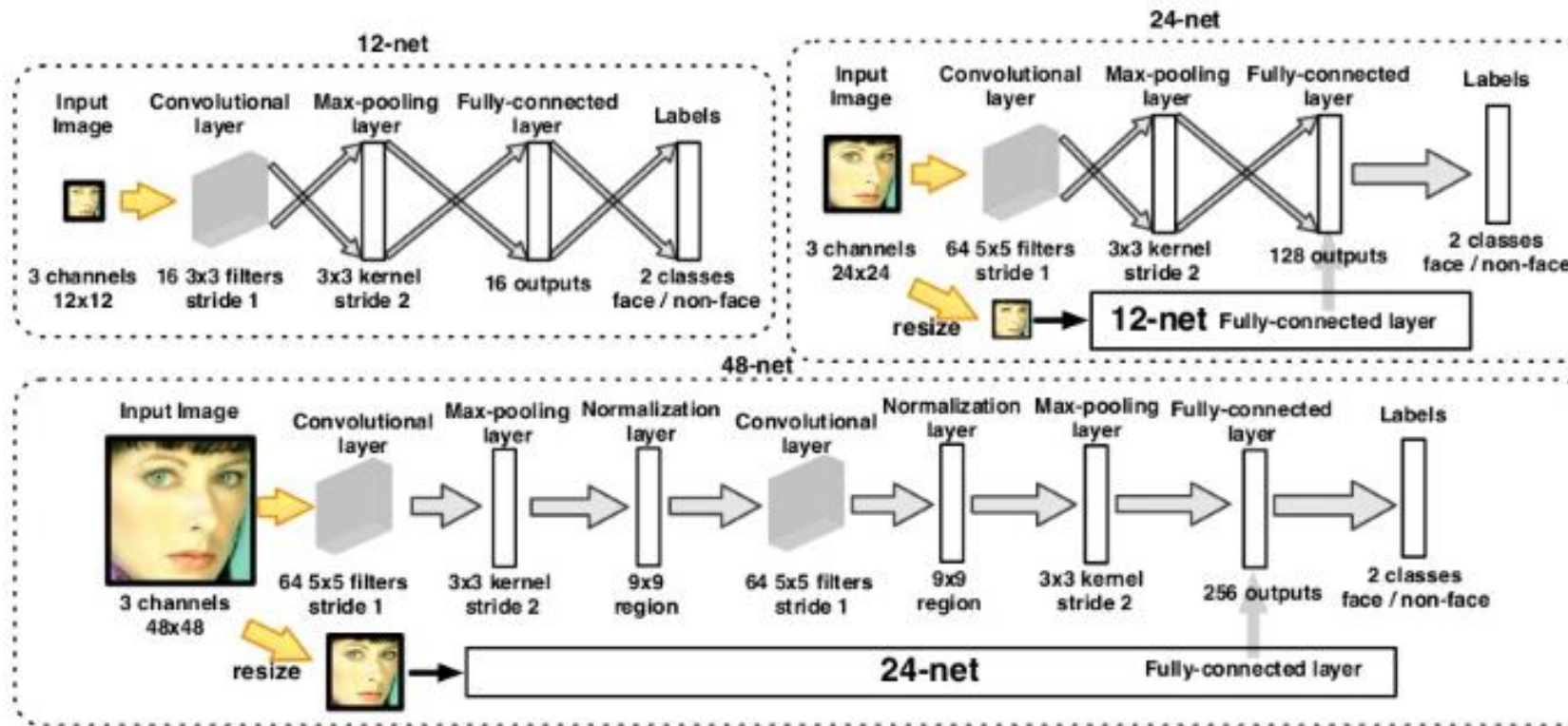
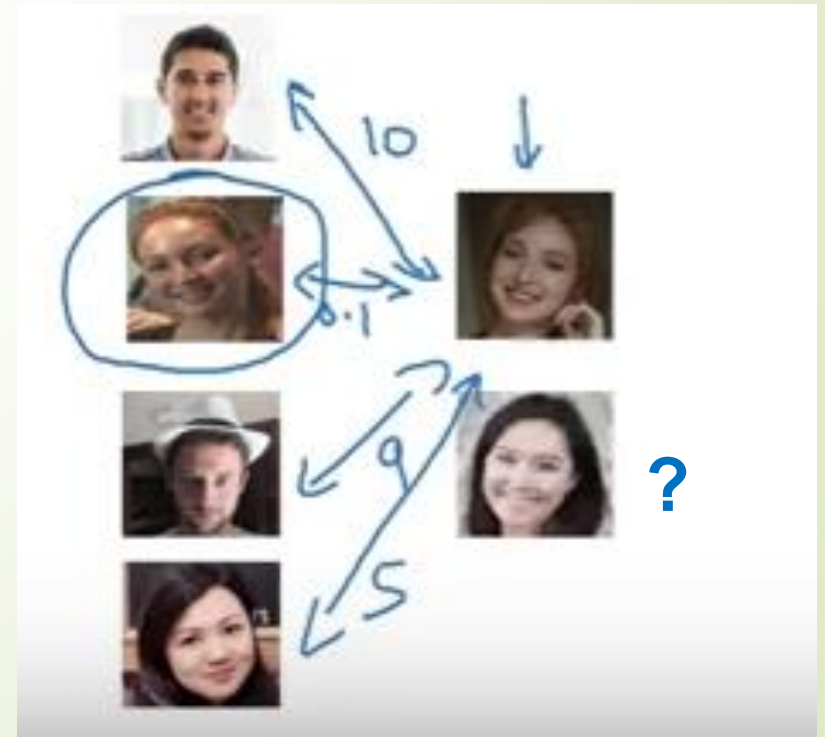
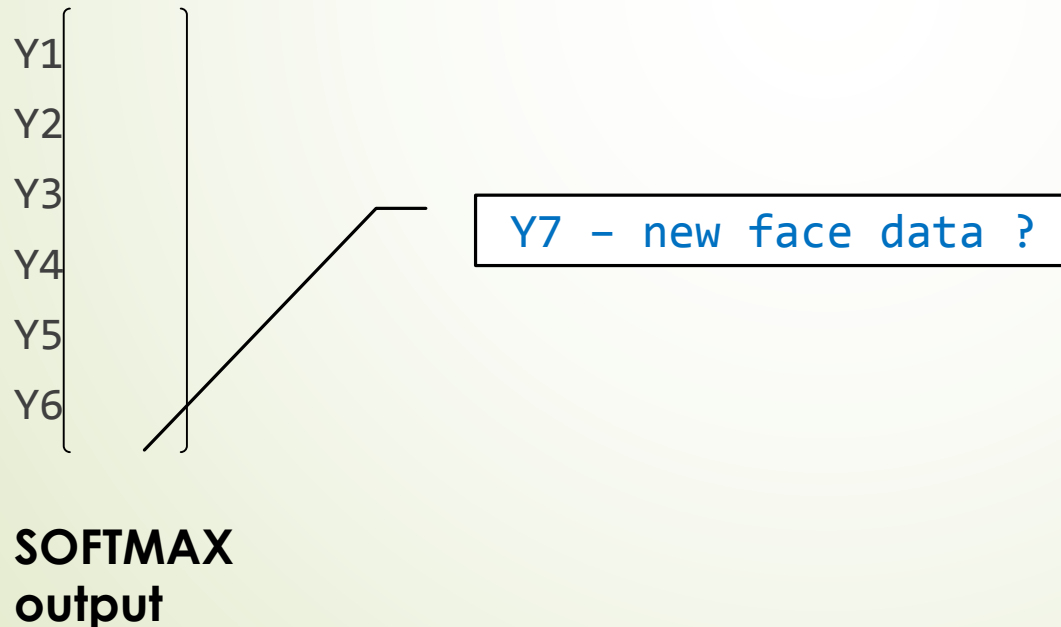


Figure 1: Test pipeline of our detector: from left to right, we show how the detection windows (green squares) are reduced and calibrated from stage to stage in our detector. The detector runs on a single scale for better viewing.



Face recognition: difficulty in using CNN /softmax/ as a classifier

- ▶ Applying **CNN classifier** to face recognition is not a great idea because, as a group of people (like employees of a company) increases or decreases, one has to change the **Softmax** classifier function. A face recognition system can be created by the means of many different ways, and in this application, facial recognition using **one-shot learning** by a **deep neural network** was used.



One shot learning

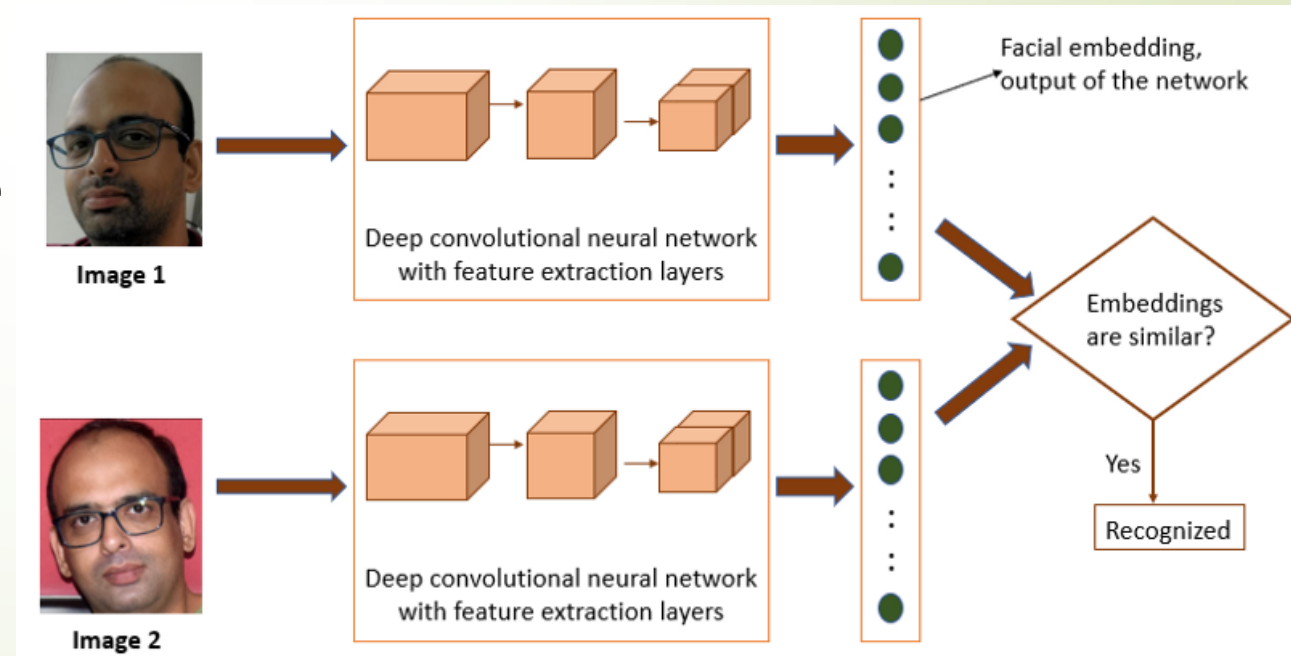
- In **one shot learning**, only one image per person is stored in the database, which is passed through the neural network to generate 128 dimensions embedding vector. This embedding vector is compared with the vector generated for the person who we want to recognize. If similarities between the two vectors exist, the system recognizes that person, otherwise it returns that the current person is not in the database.

“similarity” function

$d(\text{img1}, \text{img2})$ = degree of difference
between images

If $d(\text{img1}, \text{img2}) \leq \tau$ “same”

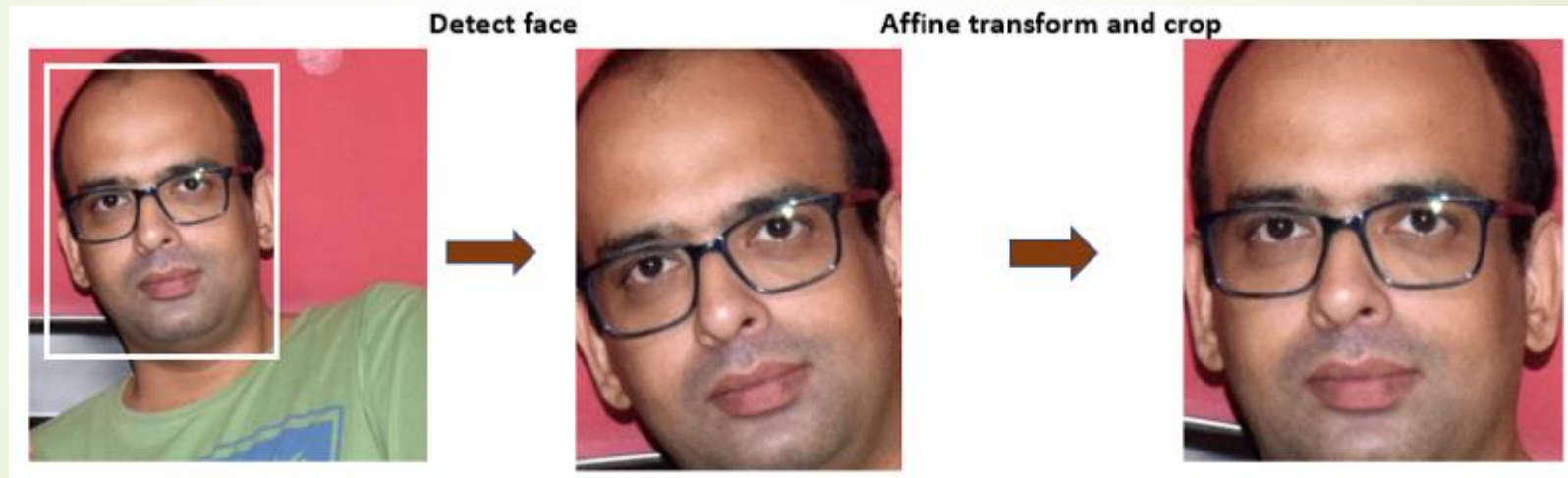
If $d(\text{img1}, \text{img2}) > \tau$ “different”



- Affine transformations could be performed in order to increase the accuracy

Affine transformation

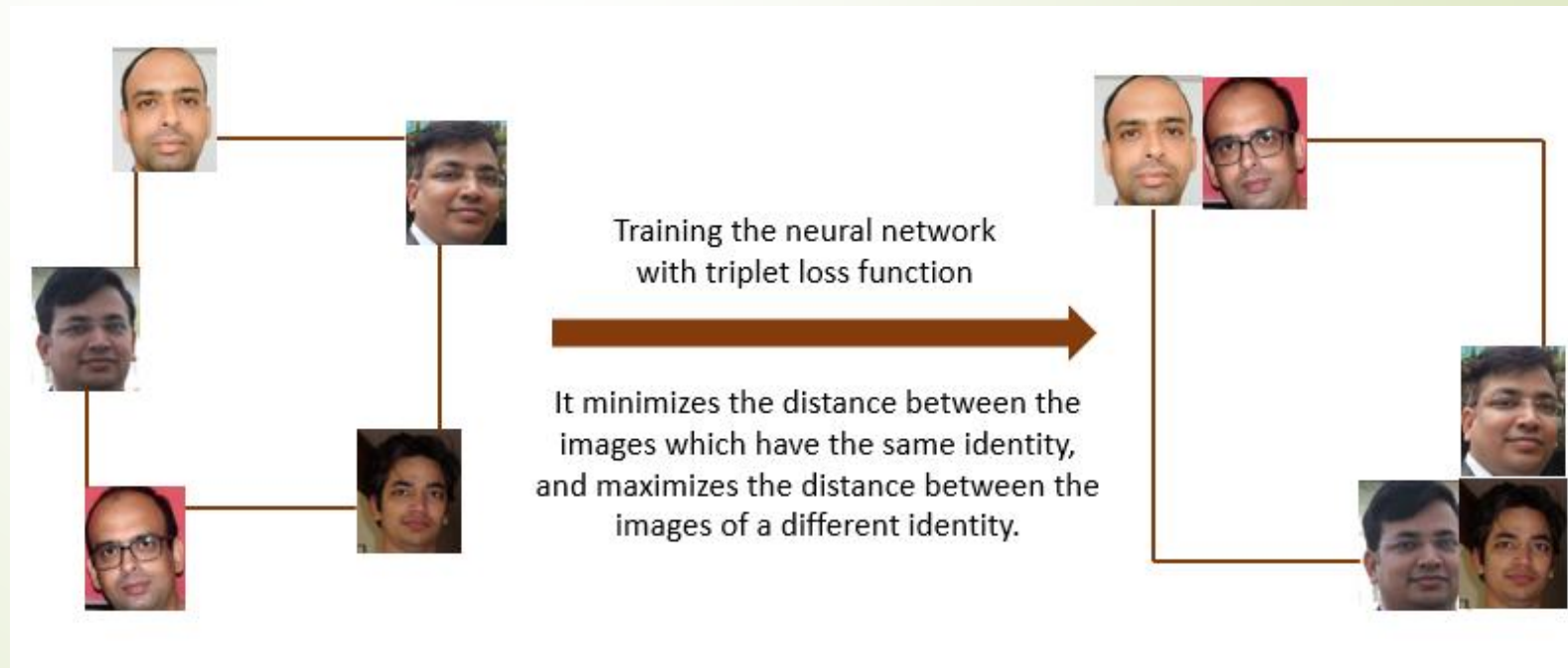
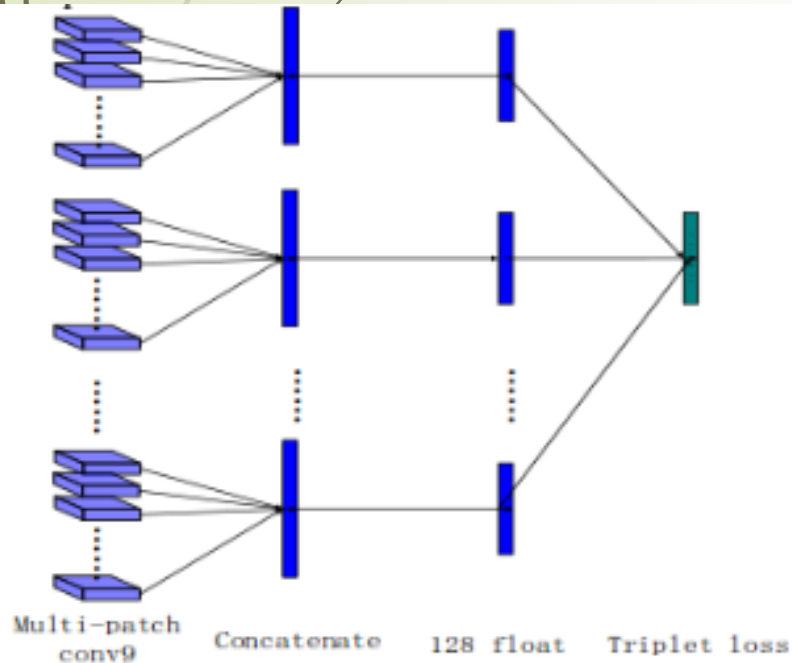
- Pose and illumination are an important challenge in face recognition. A potential bottleneck in the face recognition system is that the faces could be looking in different directions, which can result in generating a different embedding vector each time. We can solve this issue by applying an **Affine transformation** to the image as shown in the diagram.



- An affine transformation rotates the face and makes the position of the eyes, nose, and mouth for each face consistent. Performing an affine transformation ensures the position of eyes, mouth and nose to be fixed, which aids in finding the similarity between both images.

Training neural network for face recognition (triplet loss function)

- Here we are using **OpenFace** pre-trained model for facial recognition. Without going into much details on how this neural network is identifying the same faces, we can say that the model was trained on a large set of face data with a loss function which groups identical images together and separates non-identical faces away from each other. It is also known as “**triplet loss function**”.



The FaceNet Architectures

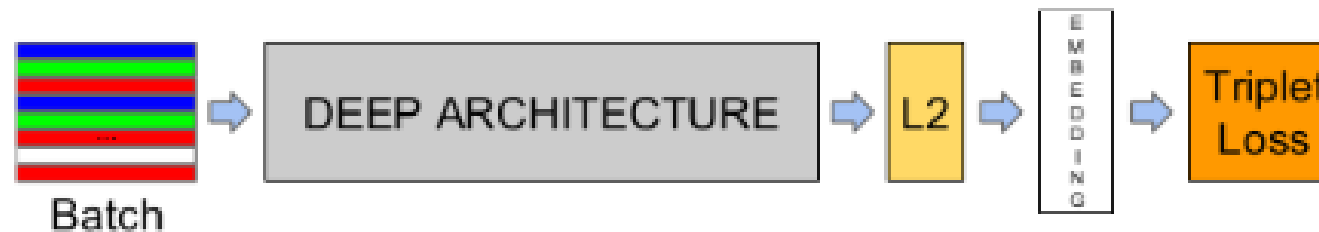
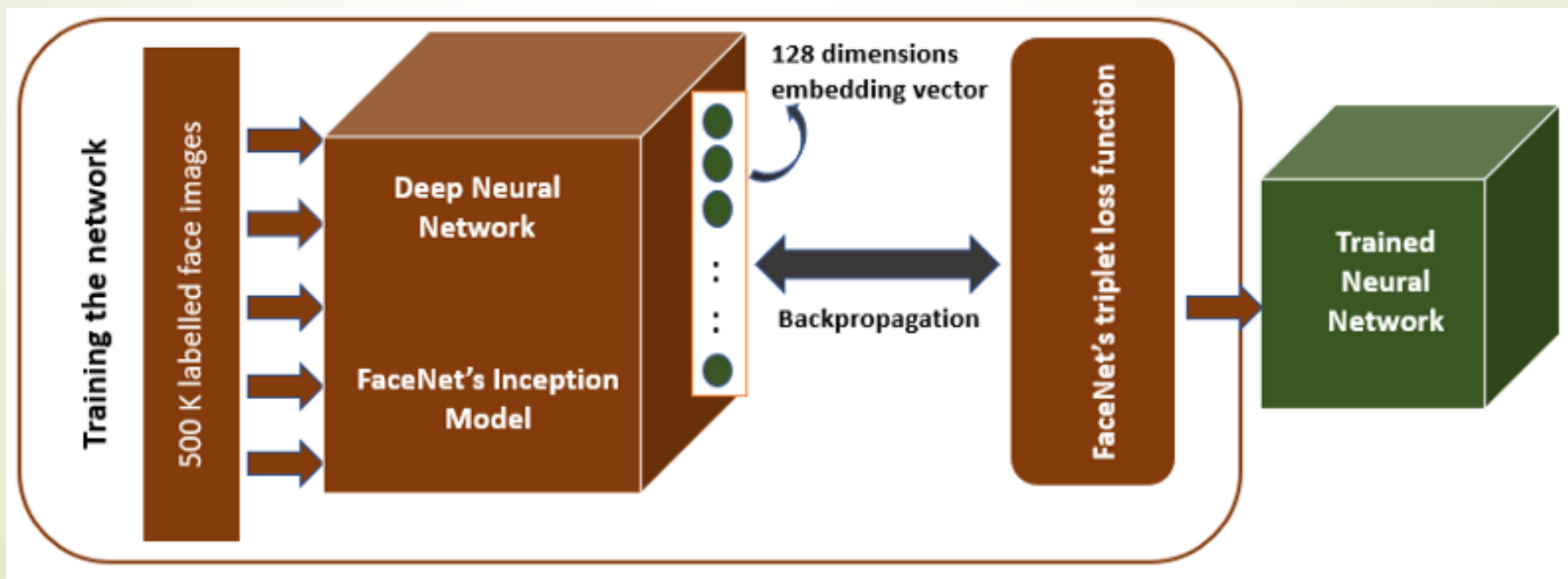
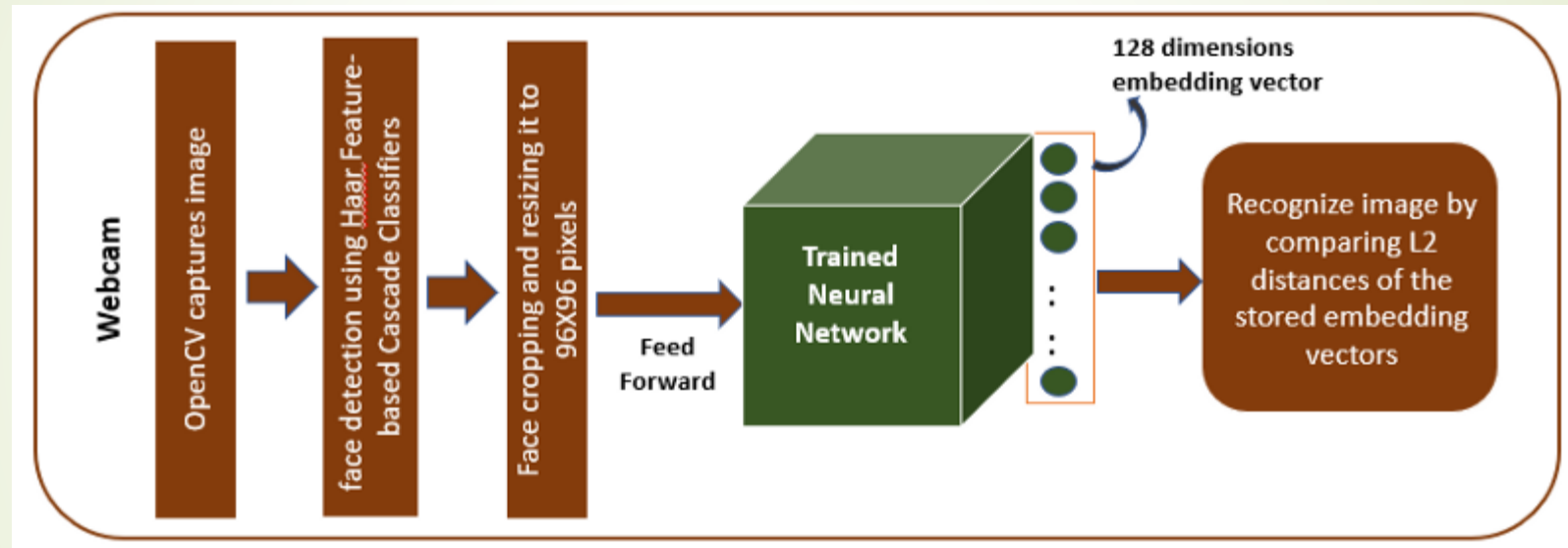


Figure 2. **Model structure.** Our network consists of a batch input layer and a deep CNN followed by L_2 normalization, which results in the face embedding. This is followed by the triplet loss during training.




- this pre-trained network was used to compare the embedding vectors of the images stored in the file system with the embedding vector of the image captured from the webcam.



- if the face captured by webcam has similar **128-bit embedding vector** stored in the database then it can recognize the person. All the images stored in the file system are converted to a dictionary with names as key and embedding vectors as value.
- When processing an image, face detection is done to find bounding boxes around faces. Very often **OpenCV's Haar feature-based Cascade Classifiers** are used for extracting the face area. Before passing the image to the neural network, it is resized to 96x96 pixels as the deep neural network expects the fixed (96x96) input image size.



Face recognition from video



Higher accuracy vs. faster performance: CPU Real Time face detection using Deep Learning

- Is it possible to implement object detection models with real-time performance without GPU?
- faced is a proof of concept that it is possible to build
- a custom object detection model for a single class object (in this case, faces) running in real time on a CPU.
- There are many scenarios where a single class object detection is needed. This means that we want to detect the location of all objects that belong to a specific class in an image. For example, we could be detecting faces for a face identification system or people for pedestrian tracking.
- What is more, most of the time we would like to run these models **in real time**. In order to achieve this, we have a feed of images providing samples at rate x and we need a model to run in less than rate x for each of the samples. **Then, we can process images as soon as they are available.**

- The most accessible and used solution nowadays to solve this task (and many others in computer vision) is to perform **transfer learning** on previously trained models (in general standard models trained on huge datasets like those found in [Tensorflow Hub](#) or in TF **Object Detection API**)
- There are plenty of trained object detection architectures (e.g. FasterRCNN, SSD or YOLO) that achieve impressive accuracy within real-time performance **running on GPUs**.

Method	mAP	FPS
Faster R-CNN (VGG16)	73.2	7
Fast YOLO	52.7	155
YOLO (VGG16)	66.4	21
SSD300	74.3	46
SSD512	76.8	19
SSD300	74.3	59
SSD512	76.8	22

- GPUs are expensive but necessary in the training phase. However, in inference
- having a dedicated GPU to achieve real-time performance is not viable.
- All of the general object detection models (as those mentioned above) fail to run in real time without a GPU.
- Then, how can we revisit the object detection problem for single class objects to achieve real-time performance but on CPU?



Main idea: simpler tasks require less learnable features

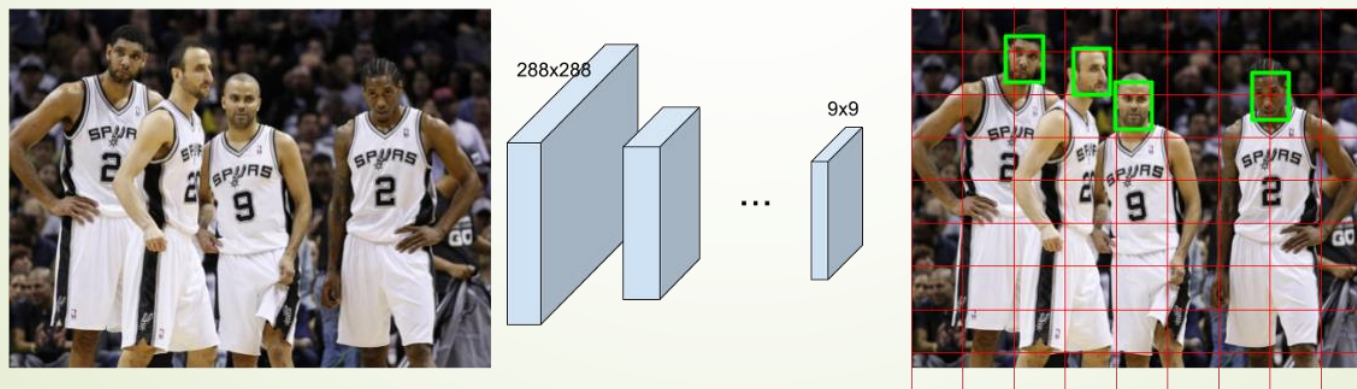
- All of the above mentioned architectures were designed to detect multiple object classes (trained on **COCO** or **PASCAL VOC** datasets). In order to be able to classify each bounding box to its appropriate class, these architectures require a massive amount of feature extraction. This translates to huge amount of learnable parameters, huge amount of filters, huge amount of layers. In other words, **this networks are big**.
- Single class object detection models will need less learnable features. Less parameters mean that the network will be smaller. Smaller networks run faster because it requires less computations. **Then, the question is: how small can we go to achieve real time performance on CPU but keeping accuracy?**
- This is **main concept of “faced”**: building the smallest possible network to run in real time in CPU while maintaining accuracy.

The architecture : “faced”

“faced” is an ensemble of 2 neural networks, both implemented using [Tensorflow](#).

► Main network

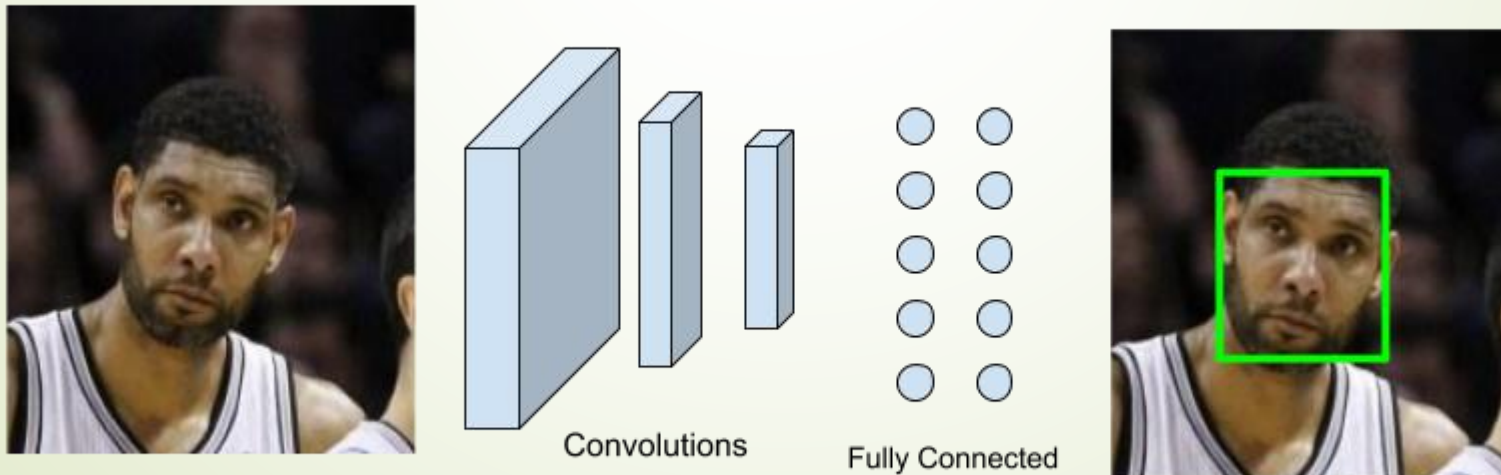
faced main architecture is heavily based on **YOLO’s** architecture. Basically, it’s a Fully Convolutional Network (FCN) that runs a 288x288 input image through a series of convolutional and pooling layers (no other layer types are involved). Convolutional layers are in charge of extracting space-aware features. Pooling layers increase the receptive field of consequent convolutional layers. The architecture’s output is a 9x9 grid (versus 13x13 grid in YOLO). Each grid cell is in charge of predicting whether a face is inside that cell (versus YOLO where each cell can detect up to 5 different object). Each grid cell has 5 associated values. The first one is the probability p of that cell containing the center of a face. The other 4 values are the (x_center, y_center, width, height) of the detected face (relative to the cell).




faced has 6,993,517 parameters. YOLOv2 has 51,000,657 parameters. It's size is 13% of YOLO's size!

Auxiliary network

- $(x_center, y_center, width, height)$ outputs of the main network were not as accurate as expected. Hence, a small CNN network was implemented to take as input a small image containing a face (cropped with the main architecture outputs) and to output a regression on the ground truth bounding box of the face.





Performance “faced” vs. YOLO

faced is able to achieve the following speed on inference:

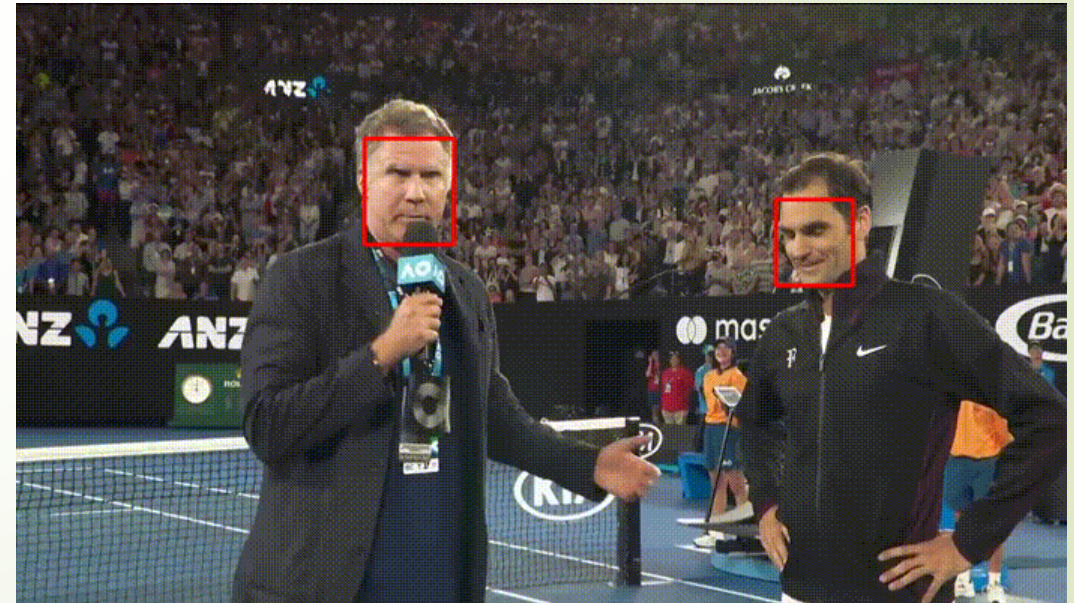
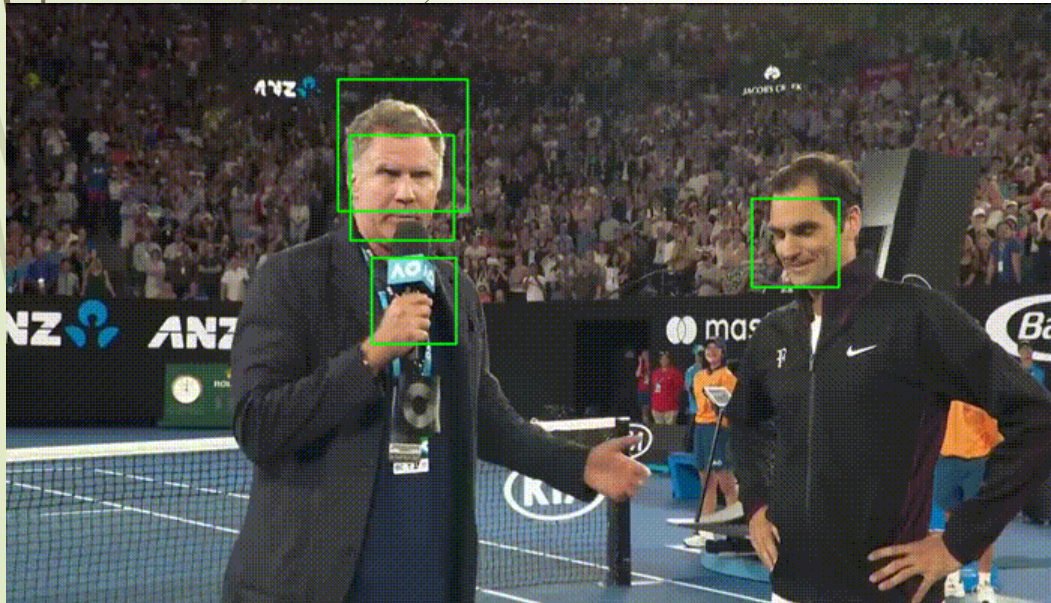
CPU (i5 2015 MBP)	GPU (Nvidia TitanXP)
~5 FPS	> 70 FPS

Performance of YOLOv2 is lower than even 1FPS on an i5 2015 MBP.



Haar vs. “faced”

A comparison between faced and **Haar Cascades**, which is a computer vision traditional approach that does not use **Deep Learning**. Both methods run under similar speed performance. *faced shows significant more accuracy.*



Haar Cascade [left] vs faced [right]