Automatic face detection in images/video streams

1. Objectives

This laboratory is focused on the issue of face detection in images/video streams using two popular computer vision methods: (1). Viola-Jones algorithm (i.e., feature-based technique that adopts a cascade classifier) and (2). Multi-task Cascade Convolutional Neural Network algorithm - MTCNN (*i.e.,* a deep learning CNN method). By using the programming language Python and some dedicated machine learning platforms (*i.e.,* Tensorsflow with Keras API and Scipy) the students will discover how to perform face detection by applying classical and deep learning models. The students will implement a face detection framework and will perform the system evaluation with respect to an objective evaluation metric: intersection over union (IoU).

2. Theoretical aspects

Face detection is a problem in computer vision of locating and localizing one or more faces in a photograph. Locating a face in a photograph refers to finding the coordinate of the face in the image, whereas localization refers to demarcating the extent of the face, often via a bounding box around the face.

There are perhaps two main approaches to face recognition: feature-based methods that use hand-crafted filters to search for and detect faces, and image-based methods that learn holistically how to extract faces from the entire image.

2.1. Viola-Jones Face Detection Framework

The Viola-Jones algorithm has four main steps presented below:

- Select the Haar-like features
- Create an integral image
- Run AdaBoost training
- Create classifier cascades

Haar-Like Features: All human faces share some similarities. It can be observed that for most human faces: the eye region is darker than the bridge of the nose or the upper-cheeks, some specific location of eyes, mouth or nose present particular characteristics. These elements can be extracted using Haar-like features.

A Haar-like feature is represented by taking a rectangular part of an image and dividing that rectangle into multiple parts. They are often visualized as black and white adjacent rectangles (Fig. 1):

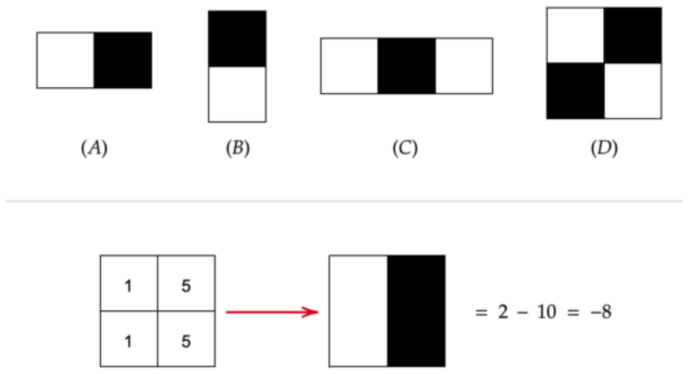


Fig. 1. Haar-like features and how to compute the Haar descriptors

In Fig.1 four Haar-like features are represented: (A). horizontal feature with two rectangles, (B).vertical feature with two rectangles; (C). horizontal feature with three rectangles and (D). diagonal feature with four rectangles. The first two examples are useful for detecting edges. The third one detects lines, and the fourth one is good for finding diagonal features.

The value of the feature is calculated as a single number: the sum of pixel values in the black area minus the sum of pixel values in the white area. For uniform areas like a wall, this number would be close to zero and won't give you any meaningful information.

$$RectangleFeature = \sum(pixels_{blackare}) - \sum(pixels_{whiteare})$$

Then, we apply this rectangle as a convolutional kernel, over our whole image (Fig. 2). In order to be exhaustive, we should apply all possible dimensions and positions of each kernel. A simple 24*24 images would typically result in over 160.000 features, each made of a sum/subtraction of pixels values.

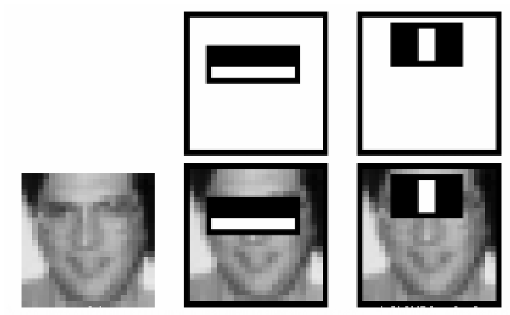


Fig. 2. Haar-like feature applied on the eye region and bridge of the nose

In order to reduce the computational complexity of extracting the Haar based features Viola-Jones propose to use integral images.

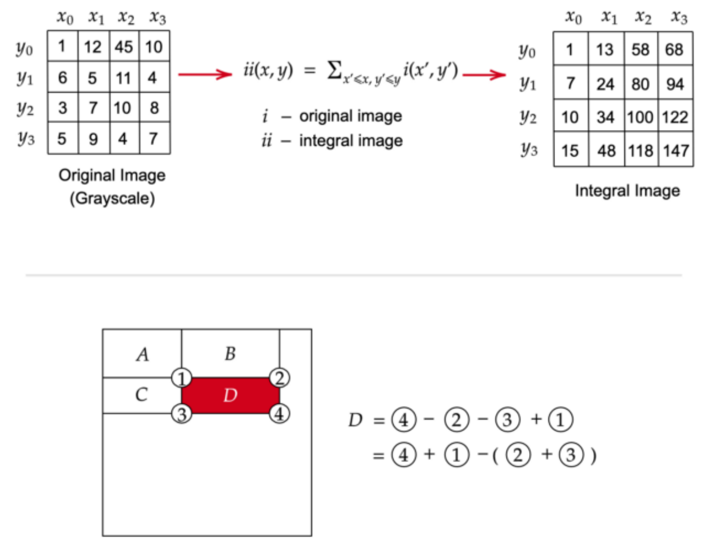
Integral images: An integral image is an intermediate representation of an image where the value for location on the integral image ($ii(x, y)$) equals the sum of the pixels above and to the left (inclusive) of the (x, y) location on the original image ($i(x, y)$).

$$ii(x,y) = \sum_{x' \leq x,\, y' \leq y} i(x', y')$$

This intermediate representation is essential because it allows for fast calculation of rectangular region. To illustrate, Fig. 3 shows that the sum of the red region D can be calculated in constant time instead of having to loop through all the pixels in that region. When you compute the whole integral image, there is a form a recurrence which requires only one pass over the original image. Indeed, we can define the following pair of recurrences:

$$s(x,y) = s(x, y-1) + i(x,y)$$
$$ii(x,y) = ii(x-1, y) + s(x,y)$$

where $s(x,y)$ is the cumulative column sum.



	x_0	x_1	x_2	x_3
y_0	1	12	45	10
y_1	6	5	11	4
y_2	3	7	10	8
y_3	5	9	4	7

Original Image (Grayscale)

$ii(x,y) = \sum_{x' \leq x, y' \leq y} i(x', y')$

i – original image

ii – integral image

	x_0	x_1	x_2	x_3
y_0	1	13	58	68
y_1	7	24	80	94
y_2	10	34	100	122
y_3	15	48	118	147

Integral Image

$$D = ④ - ② - ③ + ①$$
$$= ④ + ① - (② + ③)$$

Fig. 3. Transforming an image to an integral image

Since the process of extracting Haar-like features involves calculating the sum of dark/light rectangular regions, the integral image can be calculated in a single pass over the original image. This reduces summing the pixel intensities within a rectangle into only three operations with four numbers, regardless of rectangle size. The sum of pixels in the rectangle *ABCD* can be derived from the values of points *A*, *B*, *C*, and *D*, using the formula *D* - *B* - *C* + *A*.

AdaBoost training: In face detection, a weak learner can classify a sub-region of an image as a face or not-face only slightly better than random guessing. A strong learner is substantially better at picking faces from non-faces. The power of boosting comes from combining many (thousands) of weak classifiers into a single strong classifier. In the Viola-Jones algorithm, each Haar-like feature represents a weak learner. To decide the type and size of a feature that goes into the final classifier, AdaBoost checks the performance of all classifiers that you supply to it.

To calculate the performance of a classifier, you evaluate it on all sub-regions of all the images used for training. Some sub-regions will produce a strong response in the classifier. Those will be classified as positives, meaning the classifier thinks it contains a human face. Sub-regions that don't produce a strong response don't contain a human face, in the classifiers opinion. They will be classified as negatives.

The classifiers that performed well are given higher importance or **weight**. The final result is a strong classifier, also called a **boosted classifier** that contains the best performing weak classifiers.

Cascade classifiers: Although Adaboost is quite efficient, a major issue remains. In an image, most of the image is a non-face region. Giving equal importance to each region of the image makes no sense, since we should mainly focus on the regions that are most likely to contain faces. An increased detection rate can be achieved, while reducing computation time by using Cascading Classifiers.

When an image sub-region enters the cascade, it is evaluated by the first stage. If that stage evaluates the sub-region as positive, meaning that it thinks it's a face, the output of the stage is *maybe*. If a sub-region gets a *maybe*, it is sent to the next stage of the cascade. If that one gives a positive evaluation, then that's another *maybe*, and the image is sent to the third stage (Fig. 4).

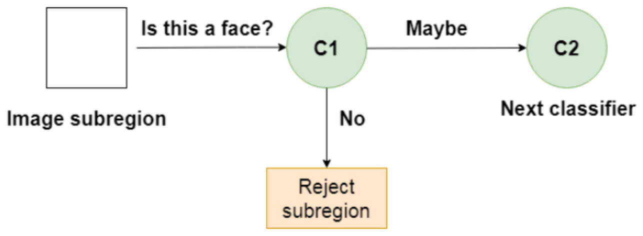


Fig. 4. Weak classifiers in a cascade

As it can be observed these classifiers are simple decision trees. Any negative result at some point leads to a rejection of the sub-window as potentially containing a face. The initial classifier eliminates most negative examples at a low computational cost, and the following classifiers eliminate additional negative examples but require more computational effort.

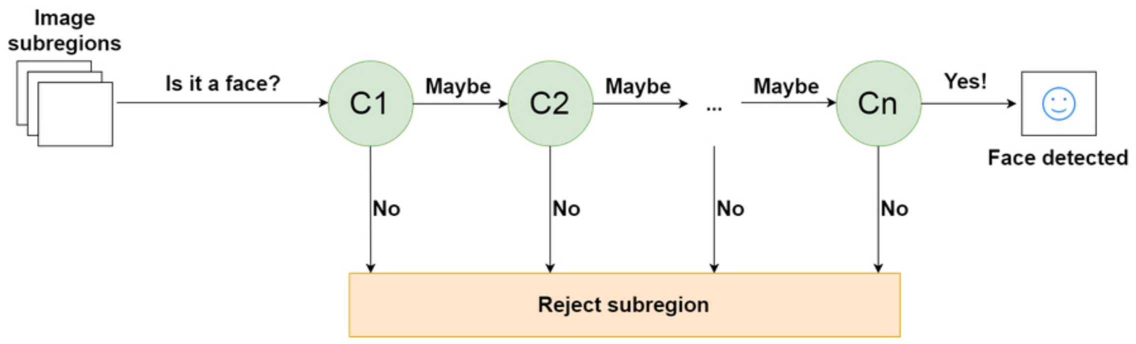


Fig. 5. A cascade of *n* classifiers for face detection

2.2. MTCNN Face Detection Framework

The "Multi-Task Cascaded Convolutional Neural Network" (MTCNN) has been introduced by Zhang in the 2016 in the "Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks" paper. MTCNN is one of the most popular face detection approaches because it achieves state-of-the-art results on a range of benchmark datasets, and because it is capable of also recognizing other facial features such as eyes and mouth, called landmark detection.

First the image is rescaled to a range of different sizes, then the MTCNN breaks down the task into three stages and builds a pipeline as follows:

Stage 1 - **P-Net:** Produces candidate of facial regions by a shallow convolutional network.

Stage 2 - **R-Net**: Filters the bounding boxes and rejects as many non-face windows as possible. The network used here is deeper.

Stage 3 - **O-Net:** This uses an even complex network to further refine the output of R-net and propose the facial landmarks.

Fig. 6 (taken from the paper) provides a helpful summary of the three stages from top-to-bottom and the output of each stage left-to-right. The model is called a multi-task network because each of the three models in the cascades (P-Net, R-Net and O-Net) are trained on three tasks (*e.g.,* make three types of predictions): face detection, bounding box regression, and facial landmark localization.

The three models are not connected directly. The outputs of the previous stage are fed as input to the next stage. This allows additional processing to be performed between stages; for example, non-maximum suppression (NMS) is used to filter the candidate bounding boxes proposed by the first-stage P-Net prior to providing them to the second stage R-Net model.

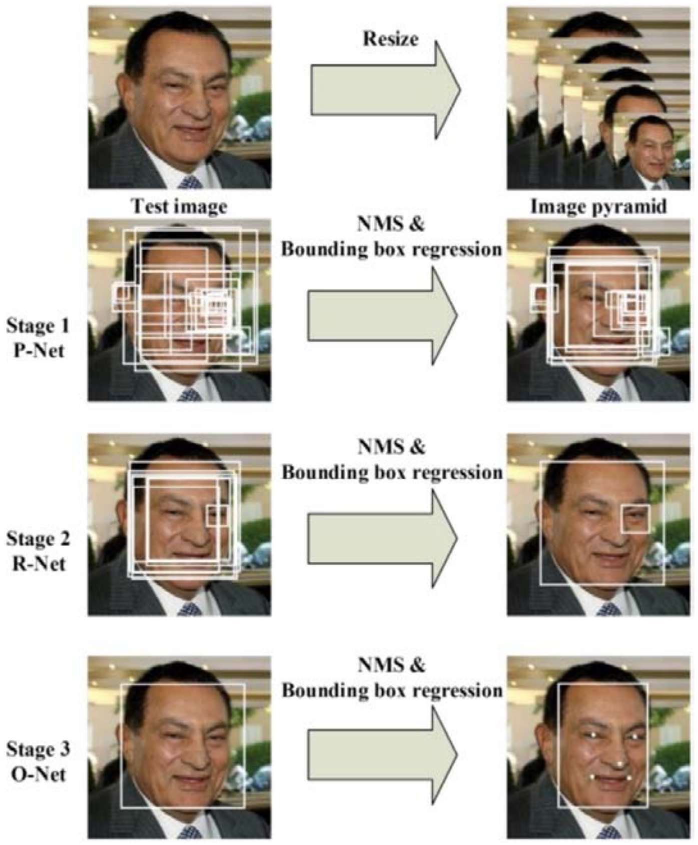


Fig. 6. MTCNN face detection pipeline

3. Practical implementation

The following example aims to evaluate the performances of two popular face detection methods by comparing the results against a ground truth that contains the faces bounding boxes manually annotated by human observers. The ground truth is stored within a textual document that contains for each image existent in the testing dataset the faces bounding boxes.

Pre-requirements: Copy all the documents from the FaceDetectionAndReID_ForStudents directory in the project "ArtificialIntelligenceProject" main directory (developed in the previous laboratory).

Application 1: Consider the AnnotateFacesByHumanObservers.py script, perform the faces annotation for all the images existent in the GT_FaceImages directory. The script will save for each image a text document that contains the faces bounding boxes (manually specified by the human annotator).

In order to specify a face bounding box use the mouse and draw a rectangle around the human face. If the selection is convenient save it by pressing the "**k**" key, otherwise perform another selection in order to better delimitate the face. In order to pass to a new image press the "**n**" key on the keyboard.

Exercise 1: Consider the image folder *“GT_FaceImages”* and the Python script “*FaceDetection.py*”. Write a Python function denoted **faceDetectionViolaJones** that takes as input the query image (from the test directory) and returns a list containing the bounding boxes of all the detected faces existent in the image.

The function should perform the following operations:

- Convert the image from BGR (if OpenCV library is used to read the images) to a grayscale image;

- Load the Haar classifier model into the current working space:

```
    classifier = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')
```

- Perform the face detection using the OpenCV function `classifier.detectMultiScale()`.

- The function `detectMultiScale()`returns the face bounding boxes into the following format: [x_1, y_1, w, h] where x_1, y_1 correspond to the bounding box upper left point, while w, h correspond to the bounding box width and height, respectively. Convert the bounding boxes format into: [x , y_1, x_2, y_2], where x_2, y_2 correspond to the bounding box lower right point.

The `detectMultiScale()` function provides some arguments to help tune the usage of the classifier. The most important parameters that can be adjusted are the `scaleFactor` and `minNeighbors`.

The `scaleFactor` controls how the input image is scaled prior to detection (e.g., it is scaled up or down), which can help to better find the faces in the image. The default value is 1.1 (10% increase), although this can be reduced to values such as 1.05 (5% increase) or increased to 1.6 (60% increase).

The `minNeighbors` determines how robust the detection must be in order to be considered as a good candidate (*e.g.*, the number of candidate rectangles that found the face). The default is 3, but this can be lowered to 1 to detect more faces and will likely increase the false positives, or increase to 6 or more to require a lot more confidence before a face is detected.

The `scaleFactor` and `minNeighbors` often require tuning for a given image or dataset in order to best detect the faces. In the case of the current exercise adopt the default values.

Exercise 2: Write a Python function denoted **drawDetectedFaces** that takes as input the query image and the set of detected bounding boxes (*cf.* Exercise 1, returned by the **faceDetectionViolaJones** function) and displays the image with the highlighted detected faces.

Exercise 3: Write a Python function named **verifyAgainsGT** that takes as input the image file name and the associated faces bounding boxes (*cf.* Exercise 1, returned by the **faceDetectionViolaJones** function) and determines:

- the total number of correctly detected faces (D_{Faces});
- the total number of missed detected faces (MD_{Faces});
- the total number of false alarms (FA) given by the system.

The function should perform the following steps:

Step 1: *Read the ground truth file* - Open the ground truth textual document (associated to the current image) and read it line by line. You may also want to remove whitespace characters like "\n" at the end of each line and the end of the document.

Step 2: *Save the ground truth face bounding boxes* - Save the bounding boxes parsed from the text document into a variable called: `bboxes_GT`.

Step 3: *Compute the IoU (Intersection over Union) score* - In order to verify if two bounding boxes overlap it is necessary to define a novel function `def computeIntersectionOverUnion(box, box_GT)`.The function takes as input two bounding boxes: one given by the detector and the other read from the ground truth document and computes the intersection over union score.

IoU is an evaluation metric used to measure the accuracy of a face detector on a particular dataset. The Intersection over Union metric requires two elements:

- The ground-truth bounding box (*i.e.*, the hand labeled bounding box from the testing set that specify where the face is located – Fig 7.a).
- The predicted bounding box from the model (Fig 7.b).

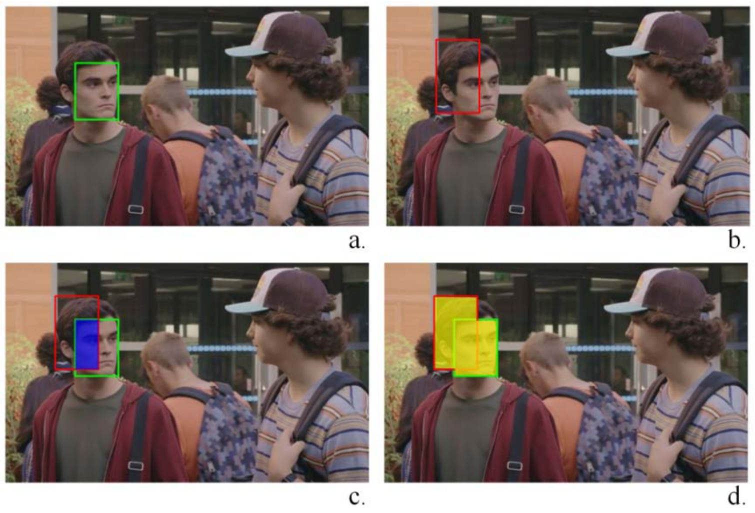


Fig. 7. Intersection over Union example: a. Ground truth face; b. Predicted face; c. Intersection area; d. Union are a
Computing IoU can therefore be determined as:

$$IoU = \frac{|A \cap B|}{|A \cup B|}$$

The numerator computes the ***overlap area*** between the *predicted* bounding box and the *ground-truth* bounding box (Fig 6.c). The denominator determines the ***union area***, between the predicted bounding box and the ground-truth bounding box (Fig 6.d).

The function should perform the following operations:

- Determine the rectangle resulted after the intersection of the two bounding boxes and compute its area (`rectIntersArea`);
- Compute the area of the predicted (`boxArea`) and the ground truth rectangle (`boxGTArea`);
- Compute the area established after the union between the two bounding boxes (`unionArea`):

  ```
  unionArea = boxArea + boxGTArea – rectIntersArea
  ```

- Determine the intersection over union score as:

  ```
  iou = rectIntersArea / unionArea
  ```

Step 4: *Validate the results* – Perform the validation of the faces bounding boxes extracted using the Viola-Jones method (*cf.* Exercise 1) against the ground truth. A face detection will be considered as correct if the intersection over union score is superior to a pre-defined threshold (Th_1 = 0.2). A false alarm will be identified if no matching can be establish between the predicted face and any bounding box from the ground truth document. A missed detected is defined as a face bounding box existent in the ground truth document with no correspondence within the face prediction list.

Exercise 4: Write a Python function denoted **faceDetectionMTCNN** that takes as input the query image (from the test directory) and returns a list containing the bounding boxes of all the detected faces existent in the image.

The function should perform the following operations:

- Convert the image from BGR (if OpenCV library is used to read the images) to a RGB image. The MTCNN detector has been trained on RGB images;
- Create the detector using the default weights (`detector = MTCNN()`);
- Detect the faces in the image by calling the `detect_faces` function (`detector.detect_faces()`);
- Save all the faces bounding boxes in the `bboxes` list.
- The function `detect_faces()` returns the face bounding boxes into the following format: [x_1, y_1, w, h] where x_1, y_1 correspond to the bounding box upper left point, while w, h correspond to the

bounding box width and height, respectively. Convert the bounding boxes format into: [x , y , x_2, y_2], where x_2, y_2 correspond to the bounding box lower right point.

Exercise 5: Consider the video sequence *"videoFaces.mp4"*, write a Python function denoted **faceDetectionInVideos** that takes as input the video stream and performs the face detection on each frame of the video stream using: (1). the Viola-Jones algorithm and (2). the MTCNN method. Display the detected bounding boxes over the corresponding frames and save the video as: *"videoFaces_Detected_MethodName.mp4"*.

Exercise 6: Consider the PersonReidentification.py script, read the images existent in the FaceReID directory and determine the pictures containing the same character.

Step 1: *Read images from directory* – Consider the FaceReID directory, read all the images one by one.

Step 2: *Perform face detection* – Write a Python function denoted **faceDetectionMTCNN** that takes as input the query image and returns a list of image patches containing the cropped faces.

Step 3: *Extract low level features* – Write a Python function denoted **extractCNNFeatures** that takes as input the list of cropped faces returned **faceDetectionMTCNN** method and returns a list of low level features extracted using the VGG16 CNN architecture.

The function should perform the following operations:

- Import the VGG face library into the project (the package `keras-vggface` should be installed):

  ```
  from keras_vggface.vggface import VGGFace
  ```

- Create the VGG face object model. The function will load the VGG16 architecture (`model='vgg16'`), pre-trained on the VGG face dataset and should not include the last layers (dense) of the network. The images input shape can be specified as: `input_shape=(224, 224, 3)`. A global average pooling will be applied to the output of the convolutional layer (`pooling='avg'`) so that feature vector will be a 2D tensor of size *n* x 512, where *n* represents the samples in a batch of images;

- Resize the cropped images to the VGG16 input shape: `(224, 224)` and save all of them in the `resizedFaces` list;

- Convert the resized imaged into a float32 numpy array of size: `(n, 224, 224, 3)`, where *n* represents the number of faces detected in the current image. It can be observed that the VGG16 network can process simultaneously (in parallel) a batch of n images.

- Pre-process the face images to the standard format accepted by the VGG16 architecture using the `preprocess_input()`function.
- Feed forward the images to the VGG network in order to extract the low level features.

Step 4: *Store the face features* – Save the face features returned by the **extractCNNFeatures** method in a dictionary denoted (`featsDict = {}`), where the key represents the image name, while the values are given by the list of face features.

Step 5: *Compare the features* – Write a Python function denoted **personReID** that takes as input the dictionary of facial features generated at Step 4 and prints the images containing the same character. The similarity between two vectors of feature can be established using the `cosine()` function, from the Scipy library. In order to determine if the images contain the same character the similarity score should be compared against a pre-established threshold.