PRACTICAL 6

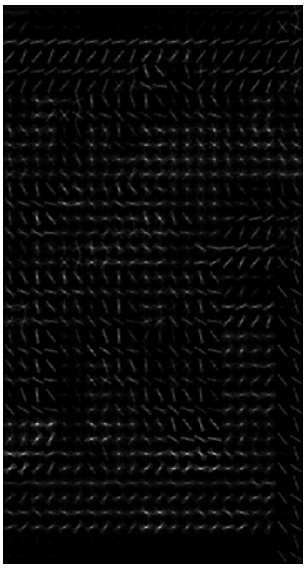
Feature extraction and recognition

In this practical, we are going to explore different techniques for doing feature extraction and detection. As application example, we will explore human detection. In particular, face detection and pedestrian detection.

Download from Canvas the provided functions to assist you during the practical as well as the clips and image sequences to be used.

TASK 1: PEDESTRIAN CLASSIFICATION

In this task we are going to make use of HOG features to detect human beings. First, we will explore how the generated features are. Then we will be using them for training and testing a pedestrian detector.



In order to assist you in the process, you have some matlab functions available as well as positives and negatives cropped examples of pedestrians. There are also 2 text files *.cdataset containing the labelling of those images as well as the split of the images on training and testing.

STEP 1: Create and script where you will implement your classifier. Load all the **training** images by using the function `loadPedestrianDatabase('*.cdataset')`. Be sure you understand the format of the output variables

STEP 2: Extract one of the images from the full database and apply the HOG feature extraction to it. To do so, make use of the provided function `hog_feature_vector()`.

This function will divide first the image into cells and blocks of 8 x8 pixels and calculate the gradients within them. By grouping this gradients into 9-bin histograms (20 degrees resolution 0-180), we are registering the HOG features. Read the code through to have a better understanding of the processing.

Please remember that the two hour duration of a practical class will probably not be enough time to complete all sections. There is an expectation that you will work outside of class as an independent learner.

STEP 3: Display in a figure the image and its corresponding HOG feature vector. To transform the feature vector in something understandable, you may want to make use of the function `showHog()`. Observe the HOG image. What conclusion can you extract?

Now that we are confident about extracting Hog features, we can proceed to process the full database and train a classifier. For this task, we will be using SVM as in the previous practical.

STEP 4: Process every image in the training set using HOG and store all the resulting features vectors into a matrix.

STEP 5: Use the previous matrix and its corresponding label array (given by `loadPedestrianDatabase`) to train a SVM classifier.

STEP 6 (OPTIONAL): Display in a new figure the learned model. To do so, calculated the weighted average of the model's support vectors multiplied by their weights. Afterwards, you can use the function `showHog` again for visualisation.

Once a classification model has been obtained, we can proceed to the testing stage.

STEP 7: Load all the testing images. For each of them you will first extract the HOG features and then predict a label given by SVM and the previously trained model.

STEP 8: Calculate the accuracy of your classifier. Reflect about the result. Is it good or bad?

STEP 9: Play with the SVM parameters and possible kernels to see if you manage to improve the accuracy.

As you can see, you have managed to obtain a quite competent pedestrian detector by combining feature extraction and machine learning. However, how good is this combination? Are all the components needed?

To give you a better insight in these points, we propose you an experiment:

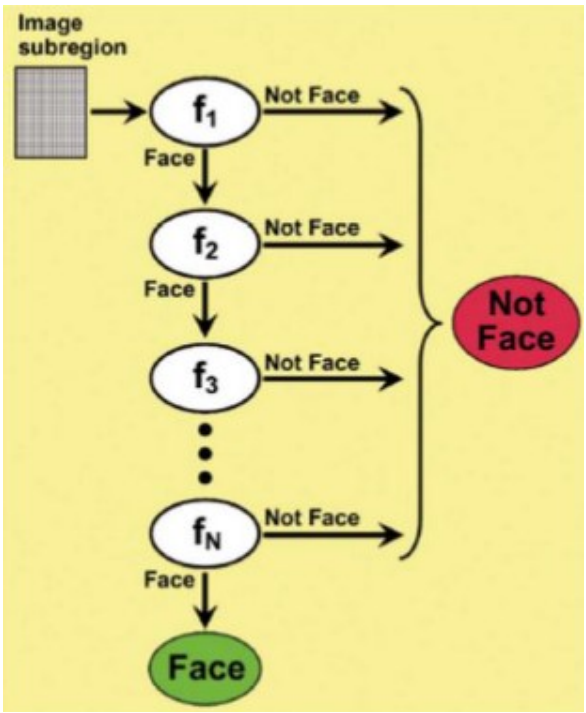
STEP 10: Save the previous script with a new name. Remove the Hog feature extraction so you will be feeding all the pixels values straight into the SVM classifier without further processing (similar to the OCR in prac 3). Recalculate the accuracy and observe its new value. What conclusion can you extract regarding accuracy? And regarding the computational time (both in training and testing)?

TASK 2: FACE DETECTION

In this task we are going to implement a frontal face detector based on Haar features (rectangular filters) and a cascade of classifiers.

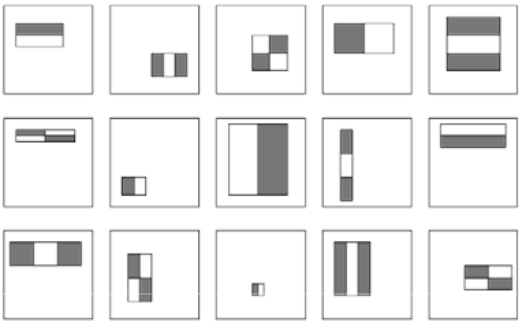
The function `ObjectDetection` will run both the feature extraction and the classification. The classifier is depicted in the following figure, where we can observe how an image region need to pass through a set of stages, being accepted in all of them to be classified as a face. This process has the name of cascade of classifiers, and allows speeding up the classification by having increasingly refined and more complex stages. If a region is clearly not a face, it will be discarded in the first stages, no wasting valuable processing time.

For this task, the classifier has been already pre-trained, therefore the function Object detection will only perform the testing stage. The classification model has been store in the file `haarcascade_frontalface_alt.mat`



Our first step will consist on visualising the mask to be able to understand how feature extraction is performed. The function `GetHaarCasade()` creates a structure containing every mask for every stage and stores is in the structure variable `HaarCasade`. Please note that every stage uses a different subset of masks, from very few ones in the first stage to hundreds in the very last ones)

STEP 1: Open and read through the function `ShowHaarCasade`. Be sure you understand how the mask and each of the rectangles composing the mask are stored before continuing. Using the variable `LeafImage` display every mask. To do so, you will need to complete the function by given a value 255 to each pixel inside a rectangle with a positive weight (white) and a value of -255 to each pixel inside a rectangle with a negative weight (white).



STEP 2: Call the function you just created in the main function so you can display all the masks for all the stages first, and for stages 1 to 5 later.

The next step in the algorithm consists of calculating the integral image, so each individual featured generated by previous masks is quickly calculated.

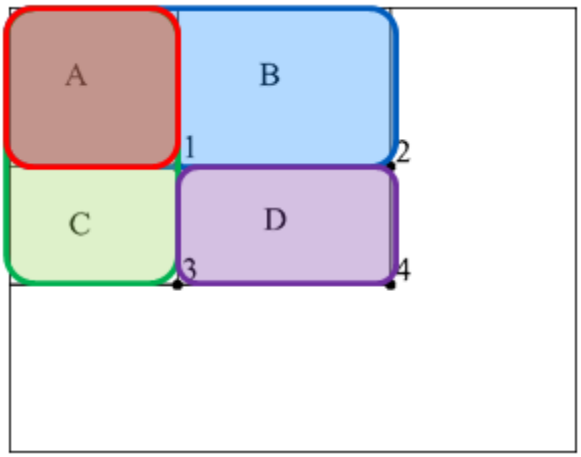
STEP 3: Complete the function `IntegralImageCalculation` (which is called by `GetIntergralImages`), so it calculates the integral images, where each pixel has the value corresponding to the sum of all previous pixels to the left and above:

$$IH(x,y) = \sum_{x' \le x, y' \le y,} I(x',y')$$

STEP 4: Draw, in the main function `ObjectDetection`, the 2 integral images returned within the structure `IntegralImages`.

With all the elements ready we can finally apply the feature extraction and the classification testing stage. To do so, one of the key components is to evaluate each of the mask on the integral image. This is done as a sum of four values

STEP 5: Open the function `GetSumRect` and analyse the operations performed there. Complete the function to obtain the value corresponding to the sum of all the pixel values within the given rectangle.



STEP 6: Run the main function for the first one of the 6 provided images. Extract your own conclusion about how well is working.

```
ObjectDetection ('1.jpg','haarcascade_frontalface_alt.mat');
```

Each detected face (i.e. each rectangle) has a different colour depending on how confident is the detector. You can see the confidence equation in the function `OneScaleObjectDetection`. The colour code from low to high confidence is: yellow-magenta-cyan-blue.

As you can see, the detector is working fine, however many detections overlap the same face. This will increase our count of false positives since only one of all these overlapping detections counts as the true positive and all the rest are false positives.

To avoid this, we can implement an extra algorithm called non-maxima suppression. This algorithm will reduce the number of redundant detections.

STEP 7: Create a function `simpleNMS` to implement the non-maxima suppression

```
function Objects = simpleNMS(Objects,threshold)
```

The algorithm inside this function should implement the following pseudocode

for every detected_bounding_box:

calculate the intersection_area with any other detected_bounding_box

if (intersection_area / bounding_box_area) is > threshold

remove one of the intersecting bounding box (the one with smaller confidence)

end

end

[Hint: for the intersection area, you can use the Matlab function `rectint()`]

STEP 8: Call the new NMS function within the main object detection function and look at the new resulting objects. Is the final result better than before?