Pattern Recognition and Machine Learning
The Viola-Jones face detector

Émilien FACHE Vincent HACHIN Martin SCHNEIDER

November 2020

In this lab, we will be experimenting face detection in images using a sliding window approach. For this, we will use the OpenCV version of the Viola-Jones detector, and we will focus on the evaluation of its performances over various datasets.

1 Viola-Jones as a face detector

We first want to evaluate the ability of the detector to correctly recognize one or more faces in a given image. This evaluation is made possible by the annotation files that go in hand with most visual datasets. These files give us a ground truth knowledge of where faces should be found in each image. Thus, if we perform a face detection on one image, we can measure a number of True Positives (TP), False Positives (FP) and False Negatives (FN). If a detected face sufficiently overlaps a ground truth face, it is counted as TP. If not it is counted as FP. And if a ground truth face is not associated to any detected face, it is counted as FN. Doing this over the whole dataset, we obtain total numbers of TPs, FPs and FNs, allowing us to compute a precision, a recall and a F1-score.

In order to know if a detected face sufficiently overlaps a ground truth one, we use the Intersection over Union (IOU) criterion : the area of the intersection of both bounding boxes is divided by the area of their union. If the result is higher than 0.5, we conclude that both boxes represent the same face.

In OpenCV, Viola-Jones can be used with the function `detectMultiScale`. Among its parameters, we will look at the influence of two of them on the quality of the detection : `scaleFactor` and `minNeighbors`. The former specifies how much the image size is reduced at each image scale, and the latter indicates how many neighbors each candidate rectangle should have to retain it. Performing the evaluation through a range of these parameters, we obtain heatmaps for precision, recall, F1-score and execution time. Below are those obtained using Viola-Jones on the FDDB dataset. Note that the annotations for this dataset are only ellipses, so we had to convert these into rectangles using the OpenCV function `ellipse2Poly`.

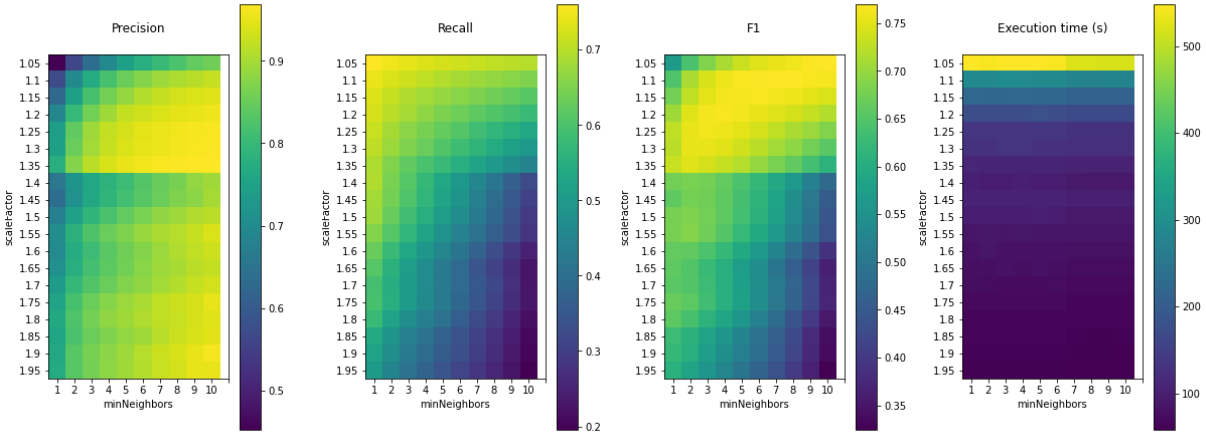

FIGURE 1 – Performances of the detector on FDDB

We can see that there is a discontinuity of the precision with respect to the scale factor around 1.4, and as F1 depends on precision, it has the same discontinuity. This could explain why it is often recommended on Internet not to go over this value for `scaleFactor` (e.g. : https ://stackoverflow.com/questions/20801015/recommended-values-for-opencv-detectmultiscale-parameters).

If we only focus on the values of `scaleFactor` lower than 1.4, it is clear that this parameter has a positive impact on precision, but a negative impact on recall. At the same time, `minNeighbors` has a positive impact on precision, but a negative impact on recall. As the harmonic mean of these two metrics, the F1-score is a good way to find a trade-off between both of them.

Regarding execution time, we can notice that it dramatically increases when `scaleFactor` is close to 1, and it becomes almost stationary for larger values. The parameter `minNeighbors` has only very little effect on it.

Trying to understand what form FPs and FNs can take, we sometimes find frustrating results. On the image below, one of the ground truth bounding boxes is represented in pink, and the blue rectangle is one of the detected faces. It seems to match decently, but the IOU is only 0.41. As a consequence, these rectangles are respectively counted as FN and FP. This shows that the IOU criterion is only conventional, it does not perfectly separate matches and mismatches.

FIGURE 2 – Ambiguous example

We then tried to confirm the tendencies we observed on FDDB. Thus we also performed the evaluation over the WIDER (validation) dataset. However, as this dataset contains many more faces, computation time has become much more problematic. We have decided not to compute our metrics on a whole grid of parameters. Instead, we just made one parameter vary while the other one was fixed. Consequently, we only plotted curves in place of heatmaps.

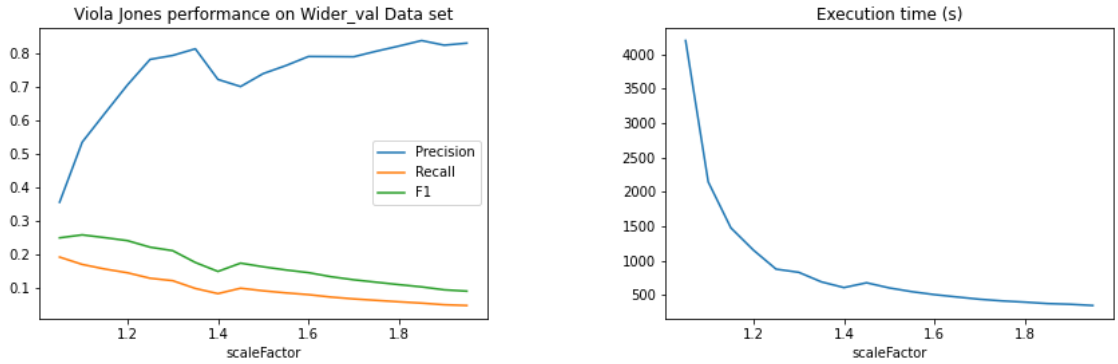

FIGURE 3 – Performances as a function of `scaleFactor`, `minNeighbors` = 3

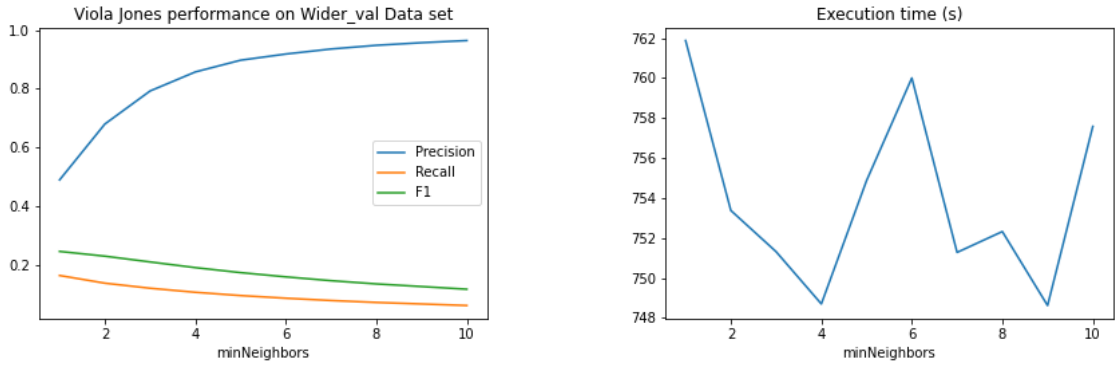

FIGURE 4 – Performances as a function of `minNeighbors`, `scaleFactor` = 1.3

We observe the same kind of evolution, although the performances seem to be much worse on recall. This can be explained by the fact that WIDER presents images with high numbers of ground truth faces that are not always well detected. This leads to a high number of FNs, thus to a low recall. For example, on the image below, red rectangles represent ground truth faces and blue ones detected faces. Real faces are numerous and too small to be detected.

FIGURE 5 – Bad detection example

2 Viola-Jones as a classifier

Now, we want to use the Viola-Jones detector as a classifier, so we can answer the question : "does an image contain a face or not ?". The first step to do this is to build a balanced dataset of face imagettes. We proceed as follows.

For each image in the FDDB dataset, we look for each ground truth face, resize it to the standard dimensions 64×64 and save it as a positive example. Each time we do that, we also look for another imagette in the same image at a random position and with the same dimensions. We ensure that it does not overlap any of the ground truth rectangles too much using a criterion on the IOU ; if it fails, we just pick a new one until we get a suitable one. We resize it to the dimensions 64×64 too and we save it as a negative example.

In practice, it is not always possible to find a negative example in an image because a real face can take too much place. Consequently, we skipped images where at least one face's area was higher than one half of the total image area.

We also realized that even with the IOU criterion, negative imagettes sometimes contained a big part of a face. Thus we have selected a threshold on the IOU that was the lowest possible to try to avoid this situation, but not too low so it remained possible to find negative examples in an image. We ultimately chose the value 0.45.

Once this dataset is built, we can evaluate Viola-Jones as a classifier. But as it is intended to be a detector, we had to make a little simplification in order to work with positive and negative classifications. We considered that the decision was positive when the output of `detectMultiScale` was non-empty (i.e. when at least one face was detected), negative otherwise.

This allows us to look at classification metrics such as the accuracy and the confusion matrix. For example, if we apply Viola-Jones with `scaleFactor = 1.2` and `minNeighbors = 4` (values that gave us a good F1-score on FDDB in the previous part), we obtain an accuracy of 0.697, and the following confusion matrix :

Predicted class \ Real class	positive	negative
positive	1991	5
negative	3048	5034

As we can see, there are almost no FPs, but we also have a very high number of FNs, which is the main source of error explaining that the accuracy is not close to 1.

Initially, our main objective was to display a ROC curve for the classification task. The idea was to use the function `detectMultiScale3` (an alternative version of `detectMultiScale`) to output the cascade depth at which an imagette was rejected during the detection process, and use it as a classification score. We could then have moved a bias between the minimum and the maximum of these values. Each time, we would have counted every prediction as positive if its score was above this bias, negative otherwise. We would have obtained True Positive Rates (TPR) and False Positive Rates (FPR) for each value of this bias, allowing us to eventually plot a ROC curve. Unfortunately, `detectMultiScale3` only outputs a cascade depth for a positive detection, involving that we would not have had a score for our negative predictions.

However, we finally found another solution to plot a ROC curve. If we look at the `.xml` file of OpenCV's pre-trained face classifier, we can notice that it describes each committee of the cascade classifier one after the other, for a total of 25 committees. If we remove one or more committees from this file (starting from the end), we obtain a similar model but with a lower acceptance level. Thus, we manually wrote new classifiers just by removing committees from the original one. We then looped on these classifiers (including the original one) and performed a new classification over our dataset each time. This allowed us to have a set of (FPR, TPR) measures, and thus to plot the following ROC curve :

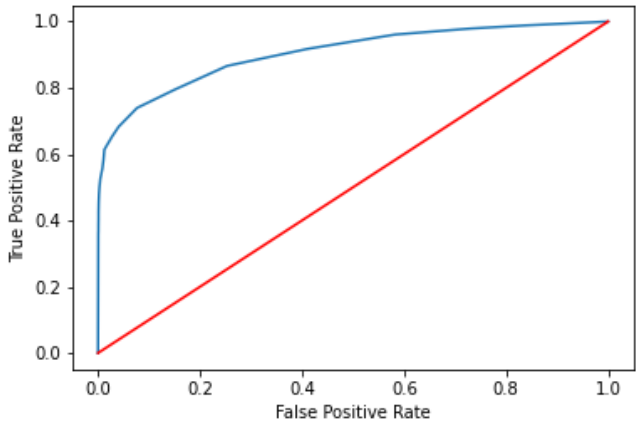

FIGURE 6 – ROC curve

The curve is clearly above the line $y = x$, showing a good result for Viola-Jones as classifier. It sticks to the y-axis for high numbers of committees, which confirms the very low number of FPs we got in the previous confusion matrix with the original classifier.

3 To go further

3.1 Eye detector

With the face classifier comes a lot of other pre-trained models in OpenCV. There are for example several models for eye detector or body detector. These detectors work exactly like the face detector we used in this lab.

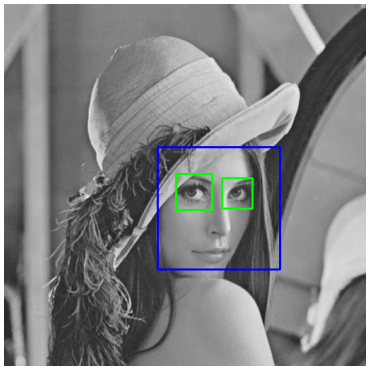

FIGURE 7 – Face and eye detection

3.2 Real time detection using the webcam

The OpenCV function `VideoCapture` allows us to directly get an image from the computer's camera. With this function, it is pretty easy to visualize any cascade detector. Thanks to this kind of visualisation we saw that faces that are oriented more than 45° are not recognized. Again, the detector only works with front photos. Profile faces are not recognized. Apart from these observations, the Viola-Jones Cascade Face detector works pretty good.

We will illustrate the use of the webcam directly through another application in the next part.

3.3 Building a new detector

We found out that there are a few interesting binaries that come with OpenCV 3 (that are not in OpenCV 4). These binaries are very helpful for creating new datasets and therefore we decided to create a new dataset in order to train a new Cascade Classifier.

As a relevant detector must be trained with a lot of data and for a long period, we built a detector for easily recognizable objects : smartphones. A smartphone detector does not need as much data as a face detector to be relevant, so we made the assumption that only a few hundreds of images would be enough.

We proceeded as follows :

1. **Have some pictures.** We wrote a simple Python script that can save a picture in a folder `positive` or `negative` by clicking on a different key, depending on the presence or not of a smartphone in the picture. After taking some pictures with this script, we got about 100 photos in each folder. For a classifier to be trained, we needed to specify where the smartphones were located in our photos.
2. **Annotate the pictures.** To do so, we used the executable from OpenCV 3 : `opencv_annotation`. This script will loop through every picture and let us easily draw

rectangles for each smartphone. After drawing 100 rectangles, we have a `.txt` file that must be converted into a `.vec` file with the `opencv_createsamples` executable.

3. **Train the classifier.** Again, there is an executable that lets us train a model with specific parameters : `opencv_traincascade`. We played a bit with the parameters before finally coming up with a relevant enough classifier !
4. **Test the detector.** This script produces a `.xml` file that can easily be loaded inside a Python file. Thanks to this file, we were able to manually test the model by applying the detector to the computer's camera. For each frame, we load the input from the camera and we can apply the MultiScale detector of our new trained model.

You can find attached a nice animation showing our results ! Please note that the classifier is not perfect because it would have taken a lot more time to increase the size of our database, but the process would have been the same !

This method has been inspired from :
https ://github.com/learncodebygaming/opencv_tutorials (section 8 – cascade classifier).