SOME NOTES
ON A TOPIC

My personal notes on

List Of Topics

By

0xLeo (github.com/0xleo)

September 30, 2019

DRAFT X.YY
Missing: . . .

Contents

1 Viola Jones detection

1.1 Problem statement and idea

Detecting faces is complicates since they have so many features, such as forehead, eyes, lips etc. Faces can also be in any size and position through the image, they can be non-uniformly illuminated, have multiple expressions or be occluded. Viola Jones takes advantage of the fact that every face has certain characteristics, that such the nose usually brighter is darker than the surrounding cheek area, the eye pupil is darker than the sclera, the nose, lip and eye position relative to each other etc. Viola Jones works in real time by combining the following methods:

1. Integral image. This speeds up the feature computation time.
2. Haar features. These are simple rectangular masks that resemble with Haar wavelets.
3. Boosting for feature selection ("AdaBoosting"). Iteratively classifies good and bad features.
4. Cascade of weak feature classifiers to quickly reject windows without faces (Attentional cascade).

Viola Jones scans a window that searches features over the image, incrementing pixel by pixel. The window size remains constant – the authors of VJ suggest 24×24 window the image is considered at various scales – the authors suggest a scaling factor of 1.25. All the scalings along with the original image are called `image pyramid`. Each level in the pyramid is obtain by downsampling the previous. For example, downsampling of 1.25 implies that every 5 pixels we save 4 and reject the 5th.

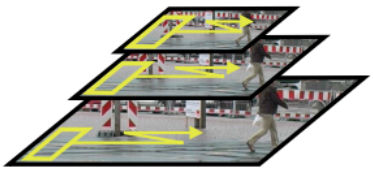


Fig. 1. Same window scans different image scales.

The window contains a different feature detector each time, e.g. one that might detect an eye or one that might detect a nose. *Haar features* determine what we want to compute each time, e.g. an eye, a forehead etc. One might think that the larger the window, the larger the number of operations done to compute the feature inside it. We will see that the number of computations is always constant thanks to the idea of *integral image*, whose computation is the first step of the algorithm. Why integral image is useful will be shown later, next section explains how it's computed. To summarise, the whole algorithm is explained by the following high-level pseudocode.

```
for number of scales in image pyramid do
    downsample image by one scale
    compute integral image for current scale
    for each shift step of the sliding detection window do
        for each stage in the cascade classifier do
            for each filter in the stage do
                filter the detection window
            end
            accumulate filter outputs within this stage
            if accumulation fails to pass per-stage threshold do
                break the for loop and reject this window as a face
            end
        end
        if this detection window passes all per-stage thresholds do
            accept this window as a face
        else
            reject this window as a face
        end
    end
end
```

1.2 Integral image

Given the source image $ii(x, y)$ where we want to detect faces, a new, integral image $i(x, y)$ is created such that each new pixel is the sum of the pixels above and to the left of it, including the original pixel itself:

$$ii(x, y) = \sum_{x' \leq x,\ y' \leq y} i(x', y') \tag{1.1}$$

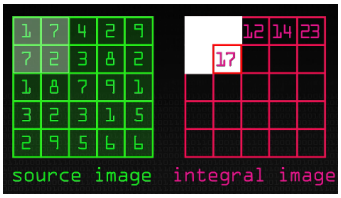


Fig. 2. Computing a single pixel of the integral image.

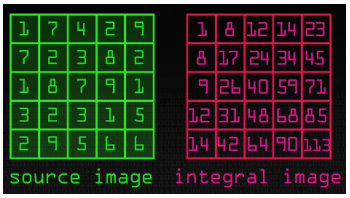
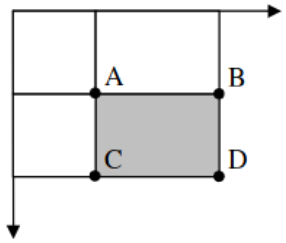


Fig. 3. The whole integral image computed.

This technique allows constant time ($\mathcal{O}(1)$) calculation of the sum of all pixels within a rectangle.



Sum of grey rectangle = D - (B + C) + A

Fig. 4. Rectangle sum calculation. The rectangles from the origin to A, B, C, D are pre-computed by the integral image. A is counted in both B and C so in the end we subtract it.

Sums of pixels are computed in order to find the Haar features described in the next sections. Since we have tons of Haar features, pre-computing the integral image is necessary.

1.3 Haar features

But what exactly are the Haar features and how do they look like? They are simple rectangular masks and each one is supposed to roughly resemble with a certain face feature, e.g. the left end of the lip. They contain certain combinations of zeros and ones. Ones should match bright areas of the feature and zeros dark. Haar features are always located in the sliding window and they are used similarly to convolution kernels.

Haar features are supposed to be similar to simple face features found in the nose, eyes, etc.

Fig. 5. Some Haar features for line and edge detection.

For example, the edge feature in the above figure could roughly match an eyebrow and the vertical line feature could match the now bridge, as shown below. Of course, line features can also be diagonal to detect rotated faces.

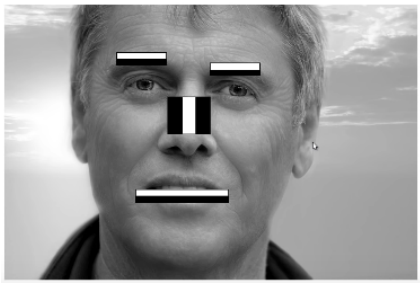


Fig. 6. Some Haar features that match face features.

To measure the quality of a feature, i.e. its similarity to the overlapping Haar cascade, we simply subtract the pixels under the black ones from those under the whites and divide the result by the size of the Haar feature. The higher the result Δ, the more likely a feature to belong to a face. This method is also called "difference of adjacent boxes". Below, n of the number of pixels of the Haar feature.

$$\Delta = \text{dark} - \text{white} = \frac{1}{n} \sum_{\substack{(x,y) \\ \text{under white}}} i(x,y) - \frac{1}{n} \sum_{\substack{(x,y) \\ \text{under black}}} i(x,y) \tag{1.2}$$

The two figures below explain the Δ calculations. For the feature in Fig. 8, we have [1]

$$\Delta = \frac{1}{16}(209+214+207+199+183+196+200+212-181-77-105-67-68-101-183-196) = 40.125$$

The closer the value of Δ to 255, the better the feature. A region of random pixels would have $\Delta \approx 0$. This calculation seems quite fast but it needs to be repeated for a huge number of times. When the Haar rectangle is large, it adds up a lot of computation time.

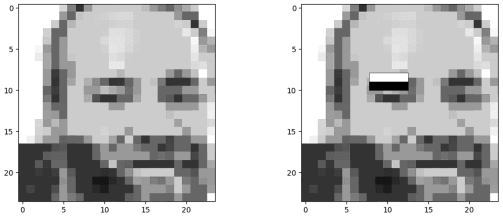


Fig. 7. The sliding 24×24 window captures a downsampled face. A line feature is overlaid the eyebrow.

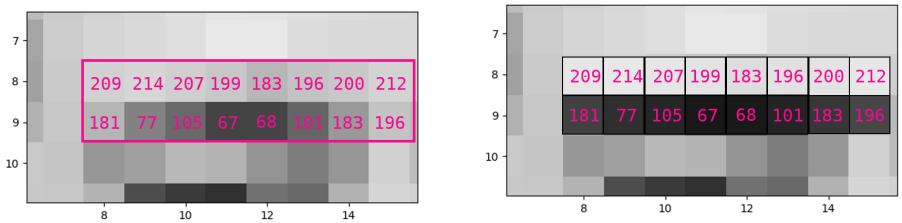


Fig. 8. A closer look at the intensity values of the eyebrow. Pixels under the black region will be subtracted and those under the white region will be added.

This is where integral image discussed in the previous section comes in handy. Instead of calculating the sum naively, for Fig. 8 we would find the sum inside the white and black rectangles using the integral image. Therefore each black or white sub-rectangle only requires 4 additions.

Integral image speeds up the Haar feature computation.

[1] Images are represented as 8-bit integers so a negative result would be clipped to 0 and one that cannot be represented by 8 bits to 255.

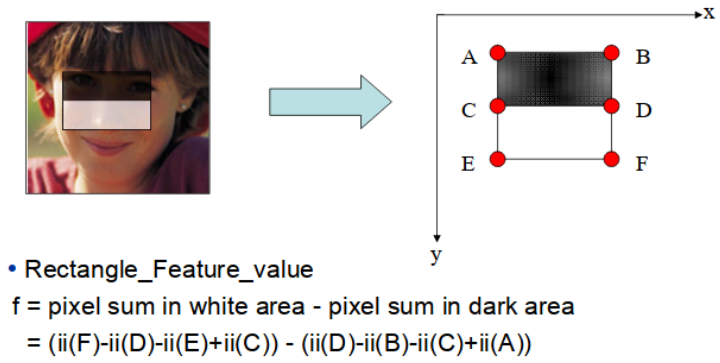


Fig. 9. Computing an example Haar feature using the integral image ii.

By comparing each Δ value to a threshold, e.g. 30 we can tell if the region under the window is likely to be a face, although this procedure will give lots of *false positives*. If the result is less than 30, then that feature in that window position will not be considered. If it's greater, more Haar classifiers will be applied. That's called a *weak classifier* and the algorithm combines multiple of them. "Weak essentially means better than random classifier, so in this case as long as the classifier outputs a value roughly greater than zero, then the window is likely to contain a face:

$$y(\Delta_{\boldsymbol{\theta}}) = \begin{cases} 1, & \Delta_{\boldsymbol{\theta}} > T \\ 0, & \text{else} \end{cases} \tag{1.3}$$

, where $\boldsymbol{\theta}$ refers to a set of parameters such as the number of rectangles, their positions, etc. The problem with using Haar features inside the window is that for a 24×24 window there are about $160,000$ of them, so computing them all is inefficient. The algorithm needs to select and combine in advance some good Haar features. This is done by a machine learning algorithm called "AdaBoost".

1.4 Adaboost training

The solution to the fact that the sliding window contains over $160,000$ features is to select a few of them – the most reliable ones that can tell whether the sub-image contains a face. This is done by AdaBoost (adaptive boost), which is a topic on its own, so it won't be discussed in too much detail but this section includes the overview.

For a certain position of the sliding window and given its features $f_1, f_2, \ldots$ such as position of black boxes, coordinates, threshold etc, the idea is that we have a set of data and outputs $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$, $\mathbf{x} \in \mathbb{R}^d$, $y \in \{-1, 1\}$ (-1: is not face, 1: is face) and we want to train perceptrons to correctly classify the output points, i.e. we want to linearly classify them. The goal is to find the features that best separate the positive and negative classes.

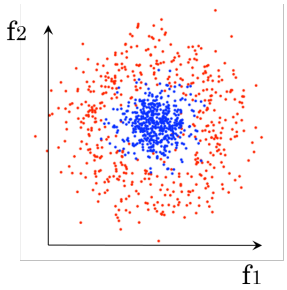


Fig. 10. Various data points $((x_{i1}, x_{i2}), y_i)$
in the 2D feature space f_1, f_2. Each class is represented by a colour.

Adaboost aims to separate the data in two classes, minimising some error function. It constructs a "strong" linear classifier as a combination of T simple "weak" simple linear classifiers $h^{(1)}(x), \ldots, h^{(T)}(x)$:

$$f(x) = \sum_{t=1}^{T} \alpha^{(t)} h^{(t)}(x) \tag{1.4}$$

, where:

- $h_t(x)$ is a "weak" or base classifier – can be a perceptron, or decision tree etc, as long as it can perform linear binary classification.
- $H(x) = \text{sgn}(f(x))$ is the final, strong, classifier.
- $\alpha^{(t)}$ is a coefficient we set at each iteration depending on how good the base classifier $h^{(t)}$ is.

AdaBoost essentially applies a series of linear classifiers, increasing the weights of the errors and decreasing the correct ones each iteration.

The algorithm maintains a weight distribution $\mathbf{w}^{(t)} = (w_1^{(t)}, \ldots, w_m^{(t)})$ over the data points [2]. The weights $w^{(t)}$ are initialised uniformly, and are updated in each iteration, keeping their sum constant, e.g. normalised. The goal of the base learner is to minimise the *weighted (base) error*

$$\epsilon^{(t)} = \sum_{i=0}^{m} w_i^{(t)} I\left\{h^{(t)}(\mathbf{x}_i) \neq y_i\right\} \tag{1.5}$$

, where $I(true) = 1$, $I(false) = 0$ so $\epsilon^{(t)}$ can be considered as the sum of the weights of the wrong examples [3], i.e. $\sum_{h^{(t)}(\mathbf{x}_i) \neq y_i} w_i^{(t)}$.

Having found the total error, the weak classifier weight $\alpha^{(t)}$ is set as

$$\alpha^{(t)} = \frac{1}{2} \ln \frac{1 - \epsilon^{(t)}}{\epsilon^{(t)}} \tag{1.6}$$

So the larger the error $\epsilon^{(t)}$, the smaller the weight $\alpha^{(t)}$. The final pseudocode is shown below. We denote the training set as $\mathcal{D}_m = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$

Algorithm 1 AdaBoost algorithm outline.

procedure ADABOOST($\mathcal{D}_m$, Base(.,.), T)
  $w^{(1)} \leftarrow (1/m, \ldots, 1/m)$ ▷ initial weights
  for $t \leftarrow 1, \ldots, T$ **do**
    $h^{(t)} \leftarrow Base(\mathcal{D}_m, \mathbf{w}^{(t)})$ ▷ apply base classifier; e.g. a simple perceptron
    $\epsilon^{(t)} \leftarrow \sum_{i=0}^{m} w_i^{(t)} I\left\{h^{(t)}(\mathbf{x}_i) \neq y_i\right\}$ ▷ weighted (total) error of base classifier t
    $\alpha^{(t)} \leftarrow \frac{1}{2} \ln \frac{1-\epsilon^{(t)}}{\epsilon^{(t)}}$
    for $i \leftarrow 1, \ldots, m$ **do** ▷ re-weight each training point
      if $h^{(t)}(t) \neq y_i$ **then** ▷ error
        $w_i^{(t+1)} \leftarrow \frac{w_i^{(t)}}{2\epsilon^{(t)}}$ ▷ increase the weight
      else ▷ correct classification
        $w_i^{(t+1)} \leftarrow \frac{w_i^{(t)}}{2(1-\epsilon^{(t)})}$ ▷ decrease the weight
  return $\text{sgn}(f(\mathbf{x})) = \text{sgn}\left(\sum_{t=1}^{T} \alpha^{(t)} h^{(t)}(\mathbf{x})\right)$ ▷ weighted "vote" of best (a.k.a. base) classifiers

It's important to remember the weights are normalised for each iteration of the algorithm, i.e.

$$\sum_{i=1}^{m} w_i^t = 1 \tag{1.7}$$

[2] Weights $\mathbf{w}^{(t)}$ and $\alpha_1, \ldots, \alpha_m$ are different.

[3] There are better error functions than this sum but we don't discuss them here for simplicity.

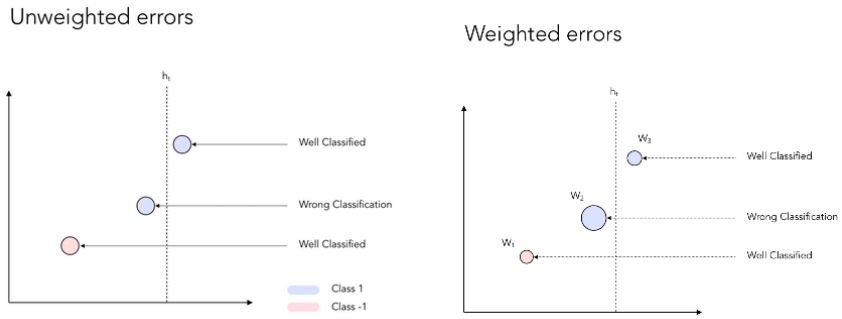


Fig. 11. Weights of wrong examples are increased and weights of correct ones are decreased.

The figure below shows how the weights are updated at every iteration.

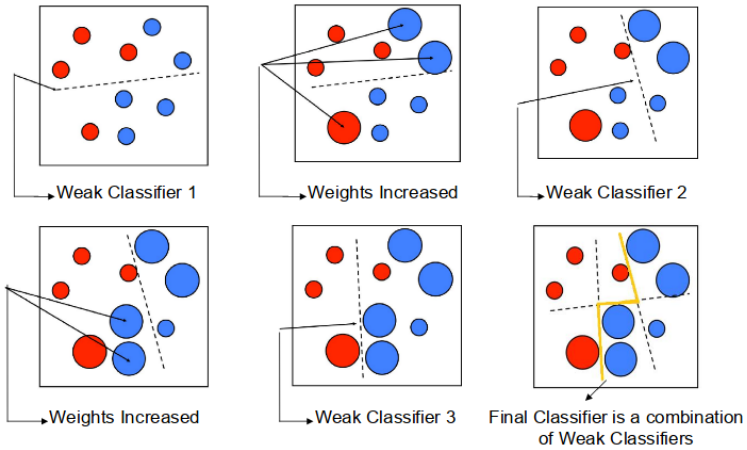


Fig. 12. At every iteration t the weights of false classifications are increased and the correct ones are decreased. Then the base classifier $h^{(t)}$ is applied again.

1.5 Classifier cascade

So far, we applied various Haar masks of different sizes, formations, and locations, etc (there are the features $\mathbf{x}^{(i)} \in \mathbb{R}^d$, where d the total number of sizes, formations, and locations etc), compared the summation result of masking calculated using the integral image to a threshold, and depending on the comparison result we classified the output $y^{(i)} \in \{-1, 1\}$ as "is in face" (1) or "is not in face" (-1). Then we separate all positive and negative results in two groups using a strong classifier calculated by AdaBoost. We had been dealing with only a fixed image size (no scales).

Each part of the cascade operates on a different input scale.

For a large image, and considering the image will have to be scaled and rescaled to account for differently sized faces, this amounts to using Viola-Jones a large number of times. If the system needs to detect faces in real-time, the computation needs to be fast. It is for this purpose that Viola and Jones introduced the attentional cascade.

The attentional cascade uses a series of strong classifiers (see previous section), each progressively having more features, to classify an image. An image is only put through by the nth classifier (larger image size) if the $n-1$th classifier (smaller size) classifies it as a positive example. If at any point a classifier does not think the image is a positive example, the cascade stops and the region is rejected as a face. The reason behind it is that in an image we will have only a few faces and thousands of non-faces. Being able to quickly discard non-faces saves time.

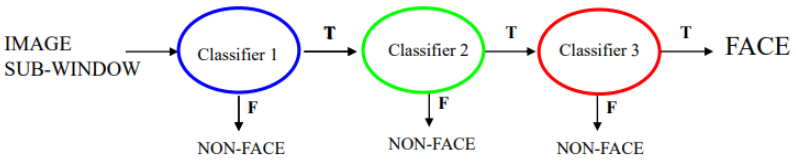


Fig. 13. How multiple strong classifiers are chained to form a cascade one. The first cascale belongs to large scale (smallest image), the second to smaller scale, etc. Classifier 2 is activated by 1 and 3 by 2.

We study how the length of the cascade improves its quality. Let f_i be the False Positive Rate (FPR) of each classifier and d_i is detection rate (True Positive Rate – TPR[4]). Assuming the stages are independent, then the *overall* false positive rate F and the overall detection rate D is

$$F = \prod_{i=1}^{K} f_i \tag{1.8}$$

$$D = \prod_{i=1}^{K} d_i \tag{1.9}$$

Assume $K = 10$. If every stage (assume they all preform roughly the same) has TPR $d_i = 99\%$ and FPR $f_i = 40\%$, then the cascade has $D = 0.99^{10} = 90\%$ and $F = 0.4^{10} = 0.001\%$.

High FPR does not really affect the cascade's quality. However, TPR needs to be high.

1.5.1 Training the cascaded classifier

Therefore having high FPR at each stage is not a bad thing but having low TPR is undesirable. Hence, training the whole cascade is summarised as follows.

1. Set target detection and false positive rates for each stage.
2. Keep adding features to the current stage until its target rates have been met.
   (a) Need to lower AdaBoost threshold to maximise detection (as opposed to minimising total classification error).
   (b) Test on a validation set.
3. If the overall false positive rate is not low enough, then add another stage.
4. Use false positives from current stage as the negative training examples for the next stage.

Training could, for instance, include $5 \cdot 10^3$ positives, $350 \cdot 10^6$ negatives, and could result in a real-time cascaded detector with 38 stages and 6061 features along all layers. The figure below how individual classifiers, each with its own Haar features, could look like.

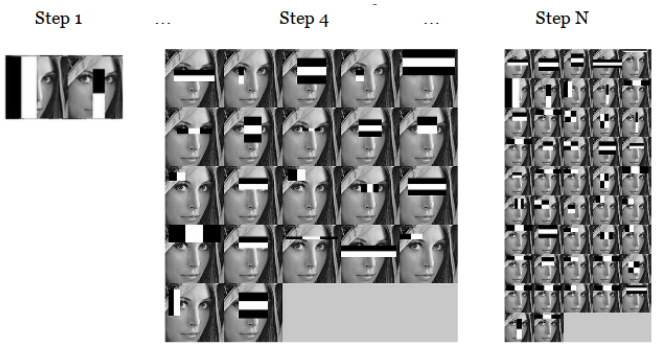


Fig. 14. The first weak classifier has in this case 2 simple features , outputting lots of FP. The rest have progressively more features.

[4]FPR and TPR are defined as follows. P are the total positive and N the total negative examples. $TPR = \frac{TP}{P} = \frac{TP}{TP+FN}$, $FPR = \frac{FP}{N} = \frac{FP}{FP+TN}$.

1.6 Face and object detection in OpenCV

As mentioned before, the cascade Viola Jones classifier can work in real time thanks to the cascade of strong classifiers, which are applied sequentially. Although we could have thousands of features, they are not applied at the same time. The first classifier contains the fewest features, the second more etc. If at any stage the input image does not pass one of the classifiers, it's rejected. This saves huge amounts of computation time.

Object, not only face, detection in OpenCV is very straightforward. First, we initialise a `cv2.CascadeClassifier` object:

```
cv2.CascadeClassifier(xml_file) -> cascade_detector
```

- `xml_file` is the path to an .xml file that contains training data, i.e. the Haar features and threshold of each classifier stage. It contains data in a special format. Some samples can be found in OpenCV's repository. Note that the `.xml` files contain also features trained to detect face profiles, eyes, bodies, etc.
- `cascade_detector` is the object that will actually run the Viola-Jones algorithm. More on it next.

```
cv2.CascadeClassifier.detectMultiScale(image[, scaleFactor[, minNeighbors[, flags[,
minSize[, maxSize]]]]]) -> objects
```

- `image`: Greyscale image that contains objects to be detected.
- `scaleFactor`: Specifies how much the image size is reduced at each image scale. Default value is `1.2`.
- `minNeighbors`: Specifyies how many neighbors each candidate rectangle should have to retain it. Default value is `3`.
- `flags`: Parameter with the same meaning for an old cascade as in the function cv2.HaarDetectObjects. It is not used for a new cascade.
- `minSize`: A tuple. Minimum possible object size. Objects smaller than that are ignored.
- `maxSize`: A tuple. Maximum possible object size. Objects larger than that are ignored. If `maxSize == minSize` model is evaluated on single scale.
- `objects`: An array of rectangles. Each rectangle is, in turn, an array `x0, y0, width, height`.

If the input image has low constrast, OpenCV suggests histogram pre-processing before applying the face detector.

`scaleFactor` may need to be calibrated depending on how large the face to detect is relative to the image size. Recall that scale factor defines how much the image is downsampled from one pyramid level to another. For example, if the input image has resolution 1280×800 and `scaleFactor = 1.2`, then the pyramid base level has size 1280×800 and the next has 20% of it, so 1168×730 and so forth. Reducing the scale factor increases the chance of finding a face but also increases computation time and also false positives.

Decreasing `minNeighbors` increases the number of detection rectangles that can overlap each other.

The code is listed in A.1 and a result is shown below.

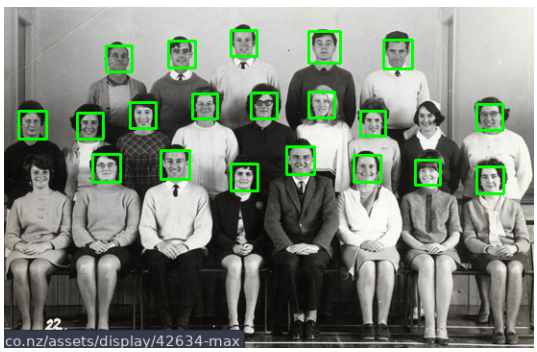


Fig. 15. Applying Viola-Jones for `scalingFactor = 1.1` and `minNeighbors = 5`.

1.7 How to train your own Haar cascade classifier in OpenCV

TODO!

A Appendices

A.1 Viola Jones in OpenCV

Listing 1: Python implementation (src/viola_jones_cv.py).

```
import cv2
import numpy as np

# download the xml from repo is search it in your installation
faceCascade = cv2.CascadeClassifier('./haarcascade_frontalface_default.xml')

im = cv2.imread('screenshot.png')
grey = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)

face_rects = faceCascade.detectMultiScale(
        grey,
        scaleFactor = 1.1,
        minNeighbors = 1)
import pdb; pdb.set_trace()

for (x,y,w,h) in face_rects:
     cv2.rectangle(im, (x, y), (x+w, y+h), (0, 255, 0), 2)
cv2.imshow("detected", im)
cv2.waitKey(0)
cv2.destroyAllWindows()
```